1 SCAPI Documentation 3
1.1 Introduction ................................................. 3
1.2 Installation .................................................. 4
1.3 Quickstart ..................................................... 6
1.4 The Communication Layer .................................... 9
1.5 Layer 1: Basic Primitives .................................... 33
1.6 Layer 2: Non Interactive Protocols .......................... 54
1.7 Layer 3: Interactive Protocols .............................. 84
1.8 SCAPI in C++ (Beta) ........................................ 104
1.9 License ....................................................... 107
SCAPI is an open-source Java 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 c++ version - currently in beta. SCAPI is free and is licensed under an adaptation of the MIT license, you can read more about the license here.
SCAPI Documentation

Introduction

SCAPI is an open-source general library tailored for Secure Computation implementations. SCAPI 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 SCAPI?

- **SCAPI provides uniformity.** As of today, different research groups are using different implementations. It is hard to compare different results, and implementations carried out by one group cannot be used by others. SCAPI is trying to solve this problem by offering a modular codebase to be used as the standard library for Secure Computation.

- **SCAPI is flexible.** SCAPI’s lower-level primitives inherit from modular interfaces, so that primitives can be replaced easily. SCAPI 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.

- **SCAPI is efficient.** Most of SCAPI’s low level code is built upon native C/C++ libraries using JNI (the java native interface) in order to run more efficiently. For example, elliptic curve operations in SCAPI are implemented using the extremely efficient Miracl library written in C.

- **SCAPI is built to please.** SCAPI 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

SCAPI is composed of the following three 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.

In addition to these three main layers, there is an orthogonal communication layer that is used for setting up communication channels and sending messages.
Installation

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

Prerequisites on Linux

There are a few prerequisites you must install before being able to compile scapi on your machine.

1. Install git
2. Install java, junit4, and ant
3. Install the gcc compiler environment: gcc, make, ar, ld, etc. Under Ubuntu you can simply run `sudo apt-get install build-essential`.

On Ubuntu environment is should look like:

```
$ sudo apt-get update
$ sudo apt-get install git
$ sudo apt-get install default-jre
$ sudo apt-get install default-jdk
$ sudo apt-get install build-essential
$ sudo apt-get install ant
$ sudo apt-get install junit4
```

**Note:** Notice that scapi should compile with gcc 4.7 or higher (if you also compile the c++ version - see specific instruction in c++ section)

Prerequisites on Mac OS X

On Mac OS X, git is usually preinstalled, and so are java and the gcc compiler environment. However, ant is not preinstalled, and you must install it via homebrew:
Prerequisites OS X 10.9 (Mavericks) and above

Starting in OS X Mavericks, apple has switched the default compiler from gcc to clang. Since Scapi’s buildsystem is based on gcc, you must install gcc manually using homebrew:

```
$ brew tap homebrew/versions
$ brew install gcc49
```

**Note:** Homebrew compiles gcc from source, and it can take a lot of time (usually around 20-50 minutes). **DO NOT** stop the shell process even though it seems to be stuck, it’s not stuck.

Installing Scapi from Source (On UNIX-based Operating Systems)

In order to install scapi:

```
$ git clone git://github.com/cryptobiu/scapi.git
$ cd scapi
$ git submodule init
$ git submodule update
$ sudo make prefix=/usr
$ sudo make install prefix=/usr
```

In case you want to install the latest java build as well and not use the pre-compiled header:

```
$ sudo make compile-scapi
```

**Note:** Seems that JDK 1.8 introduced a change of behavior regarding JNI include file locations. Details on the problem and workaround can be found in the following link: [jdk-1-8-on-linux](https://github.com/cryptobiu/scapi/wiki/jdk-1-8-on-linux)

Instructions for Windows

We currently do not have a makefile for windows, but we intend to add one in the near future.

In order to use SCAPI on windows, we included precompiled assets on the **assets/** dir.

Needed Files

1. ScapiWin-{version}.jar (Scapi)
2. bcprov-{version}.jar (Bouncy Castle)
3. commons-exec-{version}.jar (Apache Commons utilities)
4. activemq-all-{version}.jar (Apache ActiveMQ™ is an open source messaging and Integration Patterns server)
5. Precompiled DLLs under win32Dlls (for 32-bit windows) or under x64Dlls (for 64-bit windows)
6. msvcp100.dll, msvcr100.dll, msvcp120.dll and msvcr120.dll (Microsoft DLLs used by the native code)
In order to install SCAPI

On Eclipse:
1. Configure build path: go to Libraries tab, and add external JARS.
   (a) Add ScapiWin-{version}.jar.
   (b) Add bcprov-{version}.jar.
   (c) Add commons-exec-{version}.jar.
   (d) Add activemq-all-{version}.jar
2. Configure build path: go to Source tab and locate the Native Library Location section.
   (a) Add the lib folder where you have the precompiled DLLs (assets/win32Dlls or assets/x64Dlls).
3. Place the msvcp100.dll and msvcr100.dll in [C:]WindowsSystem32 folder if they are missing there.

Quickstart

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

Your First Scapi Application

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

```java
import java.io.IOException;
import java.math.BigInteger;
import java.security.SecureRandom;
import org.bouncycastle.util.BigIntegers;
import edu.biu.scapi.primitives.dlog.DlogGroup;
import edu.biu.scapi.primitives.dlog.GroupElement;
import edu.biu.scapi.primitives.dlog.openSSL.OpenSSLDlogECF2m;

public class DlogExample {
    public static void main(String[] args) throws IOException {
        // initiate a discrete log group
        // (in this case the OpenSSL implementation of the elliptic curve group K-233)
        DlogGroup dlog = new OpenSSLDlogECF2m("K-233");
        SecureRandom random = new SecureRandom();

        // get the group generator and order
        GroupElement g = dlog.getGenerator();
        BigInteger q = dlog.getOrder();
        BigInteger qMinusOne = q.subtract(BigInteger.ONE);

        // create a random exponent r
        BigInteger r = BigIntegers.createRandomInRange(BigInteger.ZERO, qMinusOne, random);

        // exponentiate g in r to receive a new group element
        GroupElement g1 = dlog.exponentiate(g, r);
        // create a random group element
    }
}
```
Pay attention to the definition of the discrete log group. In Scapi 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 first imported the needed classes that are built-in in java. Scapi uses heavily the SecureRandom class. This class provides a cryptographically strong random number generator (RNG). We also use the BigInteger type to handle big numbers. Since java has such class we do not need to re-implement it in Scapi.

```java
import java.math.BigInteger;
import java.security.SecureRandom;
```

We import the BouncyCastle utility class BigIntegers that provide a very convenient function to generate a random big integer in a given range.

```java
import org.bouncycastle.util.BigIntegers;
```

We import the Scapi generic primitives DlogGroup (implements a discrete log group) and GroupElement (a group member). We then import the OpenSSLDlogECF2m class. This is a wrapper class to a native implementation of an elliptic curve group in the OpenSSL library. Since GroupElement and DlogGroup are interfaces, we can easily choose a different group without changing a single line of code except the one in emphasis.

```java
import edu.biu.scapi.primitives.dlog.DlogGroup;
import edu.biu.scapi.primitives.dlog.GroupElement;
import edu.biu.scapi.primitives.dlog.openSSL.OpenSSLDlogECF2m;
```

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

```java
public static void main(String[] args) throws IOException {
    // initiate a discrete log group
    // (in this case the OpenSSL implementation of the elliptic curve group K-233)
    DlogGroup dlog = new OpenSSLDlogECF2m("K-233");
    SecureRandom random = new SecureRandom();

    // get the group generator and order
    GroupElement g = dlog.getGenerator();
    BigInteger q = dlog.getOrder();
    BigInteger qMinusOne = q.subtract(BigInteger.ONE);

    // create a random exponent r
    BigInteger r = BigIntegers.createRandomInRange(BigInteger.ZERO, qMinusOne, random);
}
```

We then choose a random exponent, and exponentiate the generator in this exponent.
// exponentiate g in r to receive a new group element
GroupElement g1 = dlog.exponentiate(g, r);

We then select another group element randomly.
// create a random group element
GroupElement h = dlog.createRandomElement();

Finally, we demonstrate how to multiply group elements.
// multiply elements
GroupElement gMult = dlog.multiplyGroupElements(g1, h);

Compiling and Running the Scapi Code

Save this example to a file called DlogExample.java. In order to compile this file, type in the terminal:
$ scapic DlogExample.java

The scapic command is created during the installation of scapi, and is used instead of the javac command. In reality, scapic is actually a shortcut to javac with the Scapi jar files appended to the java classpath.

A file called DlogExample.class should be created as a result. In order to run this file, type in the terminal:
$ scapi DlogExample

Like scapic, scapi replaces the java command, and defines the java classpath correctly as well as import the scapi jni interface shared libraries.

Establishing Secure Communication

The first thing that needs to be done to obtain communication services is to setup the connections between the different parties. Each party needs to run the setup process at the end of which the established connections are obtained. The established connections are called channels.

The CommunicationSetup Classes are responsible for establishing secure communication to other parties. An application requesting from CommunicationSetup to prepare for communication needs to call the CommunicationSetup::prepareForCommunication() function:

Map<String, Channel> prepareForCommunication(List<PartyData> listOfParties, long timeOut)

Parameters

• listOfParties (List<PartyData>) – The list of parties to connect to. As a convention, we will set the first party in the list to be the requesting party, that is, the party represented by the application.

• timeOut (long) – A time-out (in milliseconds) specifying how long to wait for connections to be established and secured.

Returns a map of the established channels.

Let’s add the following method to the DlogExample class:

```java
import java.net.InetSocketAddress;
import java.util.List;
import java.util.Map;
import java.util.concurrent.TimeoutException;
```

8 Chapter 1. SCAPI Documentation
import edu.biu.scapi.comm.Channel;
import edu.biu.scapi.comm.twoPartyComm.LoadSocketParties;
import edu.biu.scapi.comm.twoPartyComm.NativeSocketCommunicationSetup;
import edu.biu.scapi.comm.twoPartyComm.PartyData;
import edu.biu.scapi.exceptions.DuplicatePartyException;

private static Channel setCommunication() throws TimeoutException, DuplicatePartyException {
    // Prepare the parties list.
    LoadSocketParties loadParties = new LoadSocketParties("Parties0.properties");
    List<PartyData> listOfParties = loadParties.getPartiesList();

    // Create the communication setup.
    NativeSocketCommunicationSetup commSetup = new NativeSocketCommunicationSetup(listOfParties.get(0), listOfParties.get(1));

    long timeoutInMs = 60000; // The maximum amount of time we are willing to wait to set a connection.
    int numberOfChannels = 1; // The number of required channels.

    Map<String, Channel> connections = commSetup.prepareForCommunication(numberOfChannels, timeoutInMs);
    // prepareForCommunication() returns a map with all the established channels, we return only the first one since this code assumes the two-party case.
    return connections.values().iterator().next();
}

In this example, the list of parties is read from a properties file called Parties0.properties:

# A configuration file for the parties

NumOfParties = 2
IP0 = 127.0.0.1
IP1 = 127.0.0.1
Port0 = 8001
Port1 = 8000

A Channel represents an established connection between two parties. A channel can have Plain, Encrypted or Authenticated security level, depending on the requirements of the application. In all cases the channel has two main functions:

public void send (Serializable data)
    Sends a message msg to the other party, msg must be a Serializable object.

public Serializable receive ()
    Receives a message from the channel. Conversion to the right type is the responsibility of the caller.

This means that from the applications point of view, once it obtains the channels and sets their Security Level it can completely forget about it and just send and receive messages knowing that all the encryption or authentication work is done automatically.

The Communication Layer
Communication Design

The communication layer provides communication services for any interactive cryptographic protocol. The basic communication channel is plain (unauthenticated and unencrypted). Secure channels can be obtained using TLS or by using a pre-shared symmetric key. This layer is heavily used by the interactive protocols in SCAPI's third layer and by MPC protocols. It can also be used by any other cryptographic protocol that requires communication. The communication layer is comprised of two basic communication types: a two-party communication channel and a multiparty layer that arranges communication between multiple parties.

Two-Party communication

The two party communication layer provides a way to setup a two-party channel and communicate between them. This implementation improves on the previous SCAPI two-party channel in the following aspects:

- The user can choose between three different channel types: a socket-based channel, a queue-based channel, and a TLS channel (for each of the socket and queue options). Each option has its own advantages and disadvantages,
and the user should analyze which channel is most appropriate. In general, a queue is a more robust channel, but is less efficient than a socket channel.

- The queue channels avoid, or more accurately automatically recover from, communication failures. Thus, an application that needs to be robust and recover from such failures should use the queue channel.
- TLS/SSL is used for providing secure communication. (In many cases, this is more convenient than the previous SCAPI channel which used SCAPI encryption and authentication and thus required preshared keys.)
- Any number of channels can be established between a single pair of parties. Each party provides the number of channels that are desired, and the communication setup function returns a map containing this number of channels. This is very important for multithreading (e.g., SCAPI oblivious transfer protocols receive a channel in their constructor; in order to run many OTs in parallel, it is necessary to generate a different channel for each thread). We stress that only one port is needed, even if many channels are created. The different channels have unique internal port numbers, but use only a single external port number.

All the classes in the two-party communication are in the `edu.biu.scapi.twoPartyComm` package.

### Plain communication

This type of channel delivers every message as is, without encryption or authentication. It is possible to encrypt and/or authenticate the channel by wrapping the channel with SCAPI's encrypted and/or authenticated channel at scapi.edu.biu.comm. (Note that this requires pre-shared symmetric keys.) Alternatively, the SSL communication can be used instead, as described below.

There are two different implementations of the plain communication channel: socket communication and queue communication.

**Socket communication** This implementation of a plain communication channel uses the `Socket` and `ServerSocket` of the `java.net` package.

Internally, there are two sockets for each channel: one socket for sending messages (sendSocket) and another socket to receive messages (receiveSocket). This mechanism makes the communication more convenient and easy to understand.

The class that implements this communication type is called `SocketCommunicationSetup`.

**Queue communication** This implementation of a plain communication channel uses the JMS API for sending and receiving messages.

JMS enables distributed communication that is loosely coupled. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver only need to know which message format and which destination to use. In this respect, messaging differs from tightly coupled technologies, like Remote Method Invocation (RMI), which requires an application to know a remote application’s methods. Moreover, the JMS API knows how to automatically recover from communication failures; in case a connection falls during the communication, it is automatically reconnected. In addition, messages cannot get lost in the communication. A queue is therefore a far more robust method of communication.

In SCAPI's implementation, the server manages two queues between each pair of parties P1 and P2: one of them is used for P1 to send messages and for P2 to receive them, and the other is used for P2 to send messages and for P1 to receive them.

The class that implements this communication type is called `QueueCommunicationSetup`. This class gets a `ConnectionFactory` in the constructor and uses it to create the JMS connection. This allows us to deal with every JMS implementation. In addition, we provide a concrete implementation that uses ActiveMQ implementation of JMS.
that creates the factory inside the constructor. Thus, the user can use this class instead of dealing with the factory construction.

Note: In order to use Queue-based communication, a queue server needs to be configured, and up and running. We remark, however, that the queue server can be run by one of the parties if desired and so no additional machines are actually needed.

SSL communication

In this type of channel, the establishment of the secure channel, and the encryption and authentication are carried out by the TLS protocol. The implementation uses mutual (client and server) authentication and so both parties need certificates. The protocol version used is TLS v1.2 and forward-secure cipher suites are used.

Note: TLS v1.2 is supported from Java 7 only. In order to use the SSL channel, you need to make sure that you have at least Java 7 installed.

The security of SSL relies on the ability of each party to validate that it has received the authentic certificate of the other party. We support two ways to validate the other party’s certificate. The first is to use a CA-signed certificate and carry out the validation using the CA certificate in the party’s existing certificate store. The second is to use a self-signed certificate and carry out the validation using a method called “certificate pinning” which just means that it is assumed that each party already has the other party’s certificate and trusts it. We now describe these two methods:

• CA-signed certificate

With this method, it is assumed that the parties have certificates that were signed by a trusted CA. In order to validate the authenticity of the certificate, the protocol takes the CA key from the trustStore and verifies that the certificate is indeed signed by the CA and is therefore valid.

The steps that should be taken in order to work with a CA certificate are as follows:

1. Open cmd and go to your JAVA_HOME path. For example:
   ```
   >> cd C:\Program Files\Java\jre6\bin
   ```

2. Generate a key store:
   ```
   >> keytool -genkey -alias {your_domain} -keyalg RSA -keysize 2048 -keypass changeit -keystore scapiKeystore.jks
   ```

3. Create a certificate request to send to the CA:
   ```
   >> keytool -certreq -alias {your_domain} -keystore scapiKeystore.jks -file scapiCert.csr
   ```

4. The Certificate Signing Request that you generated can be submitted to a CA to create a certificate signed by the CA.

   Note: You must obtain the signed certificate from the CA before carrying out the following steps.

5. Install the CA root and any intermediate certificates into the keystore:
   ```
   >> keytool -import -trustcacerts -alias {root_certificate_alias} -file root.crt -keystore scapiKeystore.jks
   ```

6. Install the generated server certificate into the keystore:
   ```
   >> keytool -import -trustcacerts -alias <server_certificate_alias> -file scapiCert.crt -keystore scapiKeystore.jks
   ```

7. Install the CA root and any intermediate certificates into the truststore:
8. After you have the scapiKeystore.jks and scapiCacerts.jks files, put them in your project root directory.

After the CA certificate has been installed, the parties can use any certificate signed by that CA without any further manual setup.

- Self-signed certificate and certificate pinning

With this method, the users sign the certificates themselves and send them to the other parties in some out-of-band communication before running the protocol. It is assumed that the parties manually validate the authenticity of the certificates (e.g., by comparing their fingerprints over the phone). Each party has two certificates. The first is the certificate that the party generated itself; this should be installed in the keyStore. The second is the certificate that it received from the other party; this certificate should be installed in the trustStore, and declared as “trusted”. During the SSL handshake, each party receives the certificate of the other party. Since this certificate was already declared as “trusted”, SSL accepts the certificate as valid. Each party is responsible to generate its own self-signed certificate, put it in its keystore and send it to the other party. Moreover, each party must receive the self-signed certificate of the other party and put it in its truststore.

To help with the certificate generation process, we describe here the exact steps that should be taken:

1. Open cmd and go to your JAVA_HOME path. For example:
   >> cd C:\Program Files\Java\jre6\bin

2. Generate a self signed certificate and put it in the key store:
   >> keytool -genkey -alias {your_domain} -keyalg RSA -keysize 2048 -keypass changeit -keystore scapiKeystore.jks

3. Get the certificate file from the key store in order to send it to the other party:
   >> keytool -export -alias {your_domain} -storepass changeit -file myCert.cer -keystore scapiKeystore.jks

4. When receiving the other party's certificate:
   >> keytool -import -v -trustcacerts -alias {other_party_domain} -file otherCert.cer -keystore scapiCacerts.jks -keypass changeit

5. After you have the scapiKeystore.jks and scapiCacerts.jks files, put them in your project root directory.

There are two different implementations of the SSL communication channel: SSL socket communication and SSL queue communication.

**SSL socket communication** This is a special case of socket communication that uses an SSL socket instead of a plain one. This implementation uses the SSLSocket and SSLServerSocket of javax.net.ssl package.

This implementation loads the scapiKeystore.jks and scapiCacerts.jks mentioned above. The names of the files are hardcoded and thus should not be changed. Make sure to put these files in the project directory so that they can be found.

The class that implements this communication type is called SSLSocketCommunicationSetup.

**SSL queue communication** This is a special case of Queue communication that uses the SSL protocol during the communication with the JMS broker (server).

The way to construct an SSL queue differs from the way to construct an SSL socket. Unlike a socket construction, where there are unique classes for SSL sockets, in the JMS implementation the classes are the same. The only thing that determines the communication type is the URI given in the ConnectionFactory constructor. To create a plain and insecure communication use tcp://localhost:port uri; to create a secure connection that uses SSL protocol use ssl://localhost:port uri. In SCAPI's QueueCommunicationSetup class the connectionFactory is given as an argument.
to the constructor, when the factory is already initialized with the URI. As a result, the choice of whether or not to use the SSL protocol is the user’s responsibility.

We provide a concrete implementation of SSL queue communication that uses the ActiveMQ implementation, called SSLActiveMQCommunicationSetup. Like plain queue communication, the SSLActiveMQCommunicationSetup creates the factory inside the constructor and this way the user can avoid the factory construction. If a different SSL queue implementation is used, then the factory needs to be used, and the client and server certificates need to be loaded into the key store and trust store.

**Note:** In the SSL queue implementation, the other party of the SSL protocol is the JMS broker. Thus, the certificate that needs to be placed in the trust store is the certificate of the broker. In addition, this means that the broker server must either be trusted, or it must run on the same machine as one of the parties. Otherwise, the broker itself can run an man-in-the-middle attack.

SCAPI’s SSLActiveMQCommunicationSetup implementation loads the scapiKeystore.jks and scapiCacerts.jks files mentioned above. It is the user’s responsibility to put these files in the project library so that they can be found. On the ActiveMQ server side, there is a file called activemq.xml that manages the broker properties. In order to use the broker in SSL protocols one should add the following lines to this file:

```xml
<sslContext>
  <sslContext keyStore="{path_to_broker_keystore}/{name_of_broker_keystore}.jks" keyStorePassword="{broker_keystore_password}"
    trustStore="{path_to_broker_truststore}/{name_of_broker_truststore}.jks" trustStorePassword="{broker_truststore_password}"/>
</sslContext>

<transportConnectors>
  <transportConnector name="ssl" uri="ssl://0.0.0.0:61617?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/>
</transportConnectors>
```

**Note:** In order to use ActiveMQ with the SSL protocol use the port 61617. This is unlike with plain queue communication where the port number is 61616.

We have specified the enabled SSL protocol to be TLSv1.2, and the enabled cipher suites to be TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 and TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256. Moreover, we have specified the broker to use client authentication, and in addition to not use Nagle’s algorithm. If you wish to enable Nagle’s algorithm, then change the SSL tcpNoDelay property to false.

**Multiparty communication**

This is the communication layer for multiparty protocols, that provides a way to setup channels between all parties in a protocol and communicate between them.

Like the two party communication, this implementation improves the previous SCAPI multiparty communication in some aspects, including letting the user choose the channel type and the number of channels between each pair of parties, option to recover from communication failures and secure channels.

All the classes in the multiparty communication are in the edu.biu.scapi.multiPartyComm package.

The previous multiparty implementation in the edu.biu.scapi.comm package is from now on deprecated. We recommend not to use it since it will not be supported anymore.

The new multiparty implementation has two main parts:
• The first part is a communication that uses ActiveMQ queues in order to communicate between the parties. It uses a TwoPartyCommunicationSetup instance between each pair of parties. Thus, the classes that implement the queue multiparty communication are very slim since they delegate all functionality to the underlying two party instances.

• The second part is a communication that uses sockets in order to communicate between the parties. In this implementation, using the two party communication will be less effective. A Two party object listens to incoming connection on a given fixed port. If the multiparty communication will use two party instances, every two party instance will have the same port number (given to the multiparty comm) so all the objects will listen on the same port. This cannot be done in parallel since only one socket can be bound to each port at the same time. So the communication can be done only serially which is not efficient. In light of this, we chose to have a different, but very similar, implementation that listens to incoming connections from all parties. When receiving a connection, the listening thread will determine which party is calling and set the created socket to this party. The creation of the channels and the connecting step will be identical to the two party communication.

Both queues’ and sockets’ multiparty implementation has a plain communication and a secure communication channel. All the instructions (regarding the SSL protocol, ActiveMQ usage, etc) are the same as in the TwoPartyCommunicationSetup classes.

**Setting up communication**

There are several steps involved in setting up a communication channel between parties. Each one of them will be explained below:

**Fetch the list of parties from a properties file**

The first step towards obtaining communication services is to setup the connections between the different parties. Each party needs to run the setup process, at the end of which the established connections are obtained. The established connections are called channels. The list of parties and their addresses are usually obtained from a properties file. The format of the properties file depends on the concrete communication type.

The format of a socket properties file is as follows:

```
NumOfParties = 2
IP0 = <ip address of this application>
IP1 = <ip address of the other party>
Port0 = <port number of this application>
Port1 = <port number of party>
```

The format of a queue properties file is as follows:

```
URL = <URL of the JMS broker>
NumOfParties = 2
ID0 = <ID of this party>
ID1 = <ID of the other party>
```

**Note:** The property files and the classes that load them are not a necessary part of the communication. This is merely a more decoupled way to construct the PartyData objects that are needed in the communication setup phase; an application can also just construct these objects directly.

An example of the properties file used in socket communication (including SSL socket) called `SocketParties0.properties`, is as follows:

```
# A configuration file for the parties
```

1.4. The Communication Layer
An example of the properties file used in queue communication called \textit{JMSParties0.properties} is as follows:

```
# A configuration file for the parties
URL = 132.71.122.117:61616
NumOfParties = 2
ID0 = 0
ID1 = 1
```

The socket and queue LoadParties classes are used for reading the properties file for socket and queue communication respectively:

```java
import edu.biu.scapi.twoPartyComm.LoadSocketParties;
import edu.biu.scapi.twoPartyComm.SocketPartyData;

LoadSocketParties loadParties = new LoadSocketParties("SocketParties1.properties");
List<PartyData> listOfParties = loadParties.getPartiesList();
```

or

```java
import edu.biu.scapi.twoPartyComm.LoadQueueParties;
import edu.biu.scapi.twoPartyComm.QueuePartyData;

LoadQueueParties loadParties = new LoadQueueParties("JmsParties1.properties");
List<PartyData> listOfParties = loadParties.getPartiesList();
```

Each party is represented by an instance of the PartyData class. A List<PartyData> object is required in the communication setup phase.

### Setting up the actual communication

This step is different for two-party communication and for multiparty communication.

#### Two-Party communication

The TwoPartyCommunicationSetup interface is responsible for establishing secure communication to the other party. An application requesting from TwoPartyCommunicationSetup to prepare for communication needs to create the required concrete communicationSetup class: SocketCommunicationSetup, SSLSocketCommunicationSetup and QueueCommunicationSetup:

```
public class SocketCommunicationSetup implements TwoPartyCommunicationSetup, TimeoutObserver
```

```
public class SSLSocketCommunicationSetup extends SocketCommunicationSetup
```

```
public class QueueCommunicationSetup implements TwoPartyCommunicationSetup, TimeoutObserver
```

There is no specific class for SSL Queue communication because QueueCommunicationSetup can be used for SSL too. The actual communication protocol is determined in the ConnectionFactory constructor. The connectionFactory
is given in the QueueCommunicationSetup’s constructor when it is already initialized. Thus, if SSL is to be used, then
this needs to be specified in the factory creation, before calling the QueueCommunicationSetup constructor. As we
have explained above, we have implemented a concrete class that uses the ActiveMQ implementation of JMS with
SSL. It is called SSLActiveMQCommunicationSetup and will be explained later. The advantage of using this class is
that the factory is not needed.

All concrete classes implement the org.apache.commons.exec.TimeoutObserver interface. This interface supplies a
mechanism for notifying classes that a timeout has occurred.

In order to setup the actual communication, one of the following constructors is called (using the PartyData objects
obtained from the LoadParties method previously used).

public void SocketCommunicationSetup (PartyData me, PartyData party)

Parameters

• me (PartyData) – Data of the current application.
• party (PartyData) – Data of the other application to communicate with.

public void SSLSocketCommunicationSetup (PartyData me, PartyData party, String storePassword)

Parameters

• me (PartyData) – Data of the current application.
• party (PartyData) – Data of the other application to communicate with.
• storePassword (String) – The password of the keystore and truststore.

public void SSLSocketCommunicationSetup (PartyData me, PartyData party, String keyStoreName,
                                        String trustStoreName, String storePass)

Parameters

• me (PartyData) – Data of the current application.
• party (PartyData) – Data of the other application to communicate with.
• keyStoreName (String) – Name of the key store file of this party.
• trustStoreName (String) – Name of the trust store file of this party.
• storePassword (String) – The password of the keystore and truststore.

public void QueueCommunicationSetup (ConnectionFactory factory, DestroyDestinationUtil destroyer,
                                      PartyData me, PartyData party)

Parameters

• factory (ConnectionFactory) – The class used to create the JMS connection. We get it from
  the user in order to be able to work with all types of connections.
• destroyer (DestroyDestinationUtil) – The class that delete the created destinations. Should
  match to the given factory.
• me (PartyData) – Data of the current application.
• party (PartyData) – Data of the other application to communicate with.

All constructors receive the data of the current and the other application. Note that the party data is different for socket
and queue communication.

The SSLSocketCommunicationSetup constructor also receive the password of the keyStore and trustStore
where the certificates are placed. This is needed for accessing the party’s own private key. Also, there are constructors
that get the key store and trust store files’ names if the user does not want to use the default files.
The QueueCommunicationSetup constructor also receives the JMS factory and destroyer as parameters. We implement derived classes that uses the ActiveMQ implementation of JMS, called ActiveMQCommunicationSetup (for plain communication) and SSLActiveMQCommunicationSetup (for SSL communication). The constructors of these classes receive the parties’ data and the ActiveMQ broker’s URL and create both the factory and the DestroyDestinationUtil. Thus, the user can use this class instead of dealing with the factory and destroyer construction, i.e. instead of using QueueCommunicationSetup described above, one can call:

```java
public void ActiveMQCommunicationSetup(String url, PartyData me, PartyData party)
```

**Parameters**
- `url (String)` – URL of the ActiveMQ broker.
- `me (PartyData)` – Data of the current application.
- `party (PartyData)` – Data of the other application to communicate with.

```java
public void SSLActiveMQCommunicationSetup(String url, PartyData me, PartyData party, String storePass)
```

**Parameters**
- `url (String)` – URL of the ActiveMQ broker.
- `me (PartyData)` – Data of the current application.
- `party (PartyData)` – Data of the other application to communicate with.
- `storePass (String)` – The password of the keystore and truststore.

```java
public void SSLActiveMQCommunicationSetup(String url, PartyData me, PartyData party, String keyStoreName, String trustStoreName, String storePass)
```

**Parameters**
- `url (String)` – URL of the ActiveMQ broker.
- `me (PartyData)` – Data of the current application.
- `party (PartyData)` – Data of the other application to communicate with.
- `keyStoreName (String)` – Name of the key store file of this party.
- `trustStoreName (String)` – Name of the trust store file of this party.
- `storePass (String)` – The password of the keystore and truststore.

After calling the constructor of the communication setup class, the application should call one of the TwoPartyCommunicationSetup::prepareForCommunication functions in order to establish connections:

```java
public Map<String, Channel> prepareForCommunication(String[] connectionsIds, long timeOut)
```

**Parameters**
- `connectionsIds (String[])` – The names of the required connections.
- `timeOut (long)` – A time-out (in milliseconds) specifying how long to wait for connections to be established.

**Returns** a map of the established channels.

```java
public Map<String, Channel> prepareForCommunication(int connectionsNum, long timeOut)
```

**Parameters**
- `connectionsNum (int)` – The number of requested connections. The IDs of the created connection will be set with defaults values.
• **timeOut** (long) – A time-out (in milliseconds) specifying how long to wait for connections to be established.

**Returns** a map of the established channels.

In both of the above functions, the user can generate one or more connections between the parties. The channels are connected using a **single port** for each application, specified in the PartyData objects given in the constructor. The first function is used when the user wishes to provide the name of each connection. The second function is used if the user wishes these “names” to be generated automatically. In this case, the name of a channel is actually the index of the channel. That is, the first created channel is named “1”, the second is “2” and so on. These functions can be called several times. The class internally stores the number of created channels so that the next index can be given, when using the second function.

By default, Nagle’s algorithm is disabled since this has much better performance for cryptographic algorithms. In order to change the default value, call the **enableNagle()** function in the socket implementations. In the queue implementations usage of Nagle’s algorithm can be changed only on construction time so we’ve added a constructor with **boolean enableNagle** to let the user determine if Nagle’s algorithm should be used or not.

Here is an example on how to use the **SocketCommunicationSetup** class:

```java
import java.util.List;
import java.util.Map;
import edu.biu.scapi.exceptions.DuplicatePartyException;
import edu.biu.scapi.twoPartyComm.LoadSocketParties;
import edu.biu.scapi.twoPartyComm.PartyData;
import edu.biu.scapi.twoPartyComm.SocketCommunicationSetup;
import edu.biu.scapi.twoPartyComm.SocketPartyData;
import edu.biu.scapi.twoPartyComm.TwoPartyCommunicationSetup;

//Prepare the parties list.
LoadSocketParties loadParties = new LoadSocketParties("SocketParties1.properties");
List<PartyData> listOfParties = loadParties.getPartiesList();

TwoPartyCommunicationSetup commSetup = new SocketCommunicationSetup(listOfParties.get(0), listOfParties.get(1));

//Call the prepareForCommunication function to establish one connection within 2000000 milliseconds.
Map<String, Channel> connections = commSetup.prepareForCommunication(1, 2000000);

//Return the channel to the calling application. There is only one created channel.
return (Channel) connections.values().toArray()[0];
```

In order to use the **SSLSocketCommunicationSetup** class one should add the password parameter to the constructor:

```java
import java.util.List;
import java.util.Map;
import edu.biu.scapi.exceptions.DuplicatePartyException;
import edu.biu.scapi.twoPartyComm.LoadSocketParties;
import edu.biu.scapi.twoPartyComm.PartyData;
import edu.biu.scapi.twoPartyComm.SSLSocketCommunicationSetup;
import edu.biu.scapi.twoPartyComm.TwoPartyCommunicationSetup;

//Prepare the parties list.
LoadSocketParties loadParties = new LoadSocketParties("SocketParties1.properties");
List<PartyData> listOfParties = loadParties.getPartiesList();

TwoPartyCommunicationSetup commSetup = new SSLSocketCommunicationSetup(listOfParties.get(0), listOfParties.get(1), "changeit");
```
//Call the prepareForCommunication function to establish one connection within 2000000 milliseconds.
Map<String, Channel> connections = commSetup.prepareForCommunication(1, 2000000);

//Return the channel with the other party. There was only one channel created.
return (Channel) connections.values().toArray()[0];

Here is an example of how to use the ActiveMQCommunicationSetup class:

```java
import java.util.List;
import java.util.Map;
import edu.biu.scapi.exceptions.DuplicatePartyException;
import edu.biu.scapi.twoPartyComm.LoadQueueParties;
import edu.biu.scapi.twoPartyComm.PartyData;
import edu.biu.scapi.twoPartyComm.ActiveMQCommunicationSetup;
import edu.biu.scapi.twoPartyComm.TwoPartyCommunicationSetup;

//Prepare the parties list.
LoadQueueParties loadParties = new LoadQueueParties("JmsParties1.properties");
List<PartyData> listOfParties = loadParties.getPartiesList();

TwoPartyCommunicationSetup commSetup = new ActiveMQCommunicationSetup(loadParties.getURL(), listOfParties.get(0), listOfParties.get(1));

//Call the prepareForCommunication function to establish two connections within 2000000 milliseconds.
Map<String, Channel> connections = commSetup.prepareForCommunication(2, 2000000);

//Return the channels to the calling application.
return connections.values().toArray();
```

And an example to SSLActiveMQCommunicationSetup class:

```java
import java.util.List;
import java.util.Map;
import edu.biu.scapi.exceptions.DuplicatePartyException;
import edu.biu.scapi.twoPartyComm.LoadQueueParties;
import edu.biu.scapi.twoPartyComm.PartyData;
import edu.biu.scapi.twoPartyComm.SSLActiveMQCommunicationSetup;
import edu.biu.scapi.twoPartyComm.TwoPartyCommunicationSetup;

//Prepare the parties list.
LoadQueueParties loadParties = new LoadQueueParties("JmsParties1.properties");
List<PartyData> listOfParties = loadParties.getPartiesList();

TwoPartyCommunicationSetup commSetup = new SSLActiveMQCommunicationSetup(loadParties.getURL(), listOfParties.get(0), listOfParties.get(1), "changeit");

Map<String, Channel> connections = commSetup.prepareForCommunication(1, 2000000);

//Return the channels to the calling application.
return connections.values().toArray();
```

Multiparty communication

The MultipartyCommunicationSetup interface is responsible for establishing secure communication between the current running application to each other party in the protocol. An application requesting from MultipartyCommunicationSetup to prepare for communication needs to create the required concrete multiparty communicationSetup class: SocketMultipartyCommunicationSetup.
SSL_SOCKET_MultipartyCommunicationSetup, ActiveMQ_MultipartyCommunicationSetup and
SSL_ActiveMQ_MultipartyCommunicationSetup:

public class Socket_MultipartyCommunicationSetup implements MultipartyCommunicationSetup,
TimeoutObserver
public class SSL_Socket_MultipartyCommunicationSetup extends Socket_MultipartyCommunicationSetup
public class ActiveMQ_MultipartyCommunicationSetup implements MultipartyCommunicationSetup
public class SSL_ActiveMQ_MultipartyCommunicationSetup extends ActiveMQ_MultipartyCommunicationSetup

Here, unlike the two party implementation there is no abstract class for queue communication. As mentioned above,
the queue multiparty communication uses two party instances between the current running application and any other
party in the protocol. In order to create the underlying two party objects, one should know exactly which specific class
he should create. This specific two party object fixed the multiparty implementation and that is the reason it cannot be
abstract.

The socket implementations implement the org.apache.commons.exec.TimeoutObserver interface. This interface sup-
plies a mechanism for notifying classes that a timeout has occurred. The queue implementations should not implemen-
tat this interface since they use the two party queue implementation that implements the TimeoutObserver interface.

In order to setup the actual communication, one of the following constructors is called (using the PartyData list
obtained from the LoadParties method previously used).

public void Socket_MultipartyCommunicationSetup (List<PartyData> parties)
Parameters
- parties (List<PartyData>) – Data of the current application.

public void SSL_Socket_MultipartyCommunicationSetup (List<PartyData> parties, String storePass)
Parameters
- parties (List<PartyData>) – Data of the current application.
- storePass (String) – The password of the keystore and truststore.

public void SSL_Socket_MultipartyCommunicationSetup (List<PartyData> parties, String keyStore-
Name, String trustStoreName, String storePass)
Parameters
- parties (List<PartyData>) – Data of the current application.
- keyStoreName (String) – Name of the key store file of this party.
- trustStoreName (String) – Name of the trust store file of this party.
- storePass (String) – The password of the keystore and truststore.

public void ActiveMQ_MultipartyCommunicationSetup (String url, List<PartyData> parties)
Parameters
- url (String) – URL of the ActiveMQ broker.
- parties (List<PartyData>) – Data of the current application.

public void SSL_ActiveMQ_MultipartyCommunicationSetup (String url, List<PartyData> parties,
String storePass)
Parameters
- url (String) – URL of the ActiveMQ broker.
- parties (List<PartyData>) – Data of the current application.
SSLActiveMQMultipartyCommunicationSetup(String url, List<PartyData> parties, String keyStoreName, String trustStoreName, String storePass)

Parameters

- **url** (String) – URL of the ActiveMQ broker.
- **parties** (List<PartyData>) – Data of the current application.
- **keyStoreName** (String) – Name of the key store file of this party.
- **trustStoreName** (String) – Name of the trust store file of this party.
- **storePass** (String) – The password of the keystore and truststore.

All constructors receive the data of all the parties participate in the protocol, while the first party in the list represents the current running party. Note that the party data objects are different for socket and queue communication.

As in the two parties communication, the secure multiparty classes also receive in the constructor the password of the keyStore and trustStore where the certificates are placed, and there are constructors that get the key store and trust store files’ names.

After calling the constructor of the communication setup class, the application should call the MultipartyCommunicationSetup::prepareForCommunication function in order to establish connections:

prepareForCommunication(Map<PartyData, Object> connectionsPerParty, long timeOut)

Parameters

- **connectionsPerParty** (Map<PartyData, Object>) – provides the amount of connections or the names of connections that should be created between the current application and other parties in the protocol.
- **timeOut** (long) – A time-out (in milliseconds) specifying how long to wait for connections to be established.

Returns a map contains the connected channels. The key to the map is a PartyData object (represents a party participates in the protocol) and the value is the created connections to this party.

Notice that in the two party implementation there are two prepareForCommunication functions. One gets the number of requested connections and one gets the names of the requested connections. Here, there is only one function that can get both things, since the parameter type is Map<PartyData, Object> and object can be anything. This is done that way because of technical reasons, since for java Map<PartyData, Integer> and Map<PartyData, string[]> are the same thing and therefore java does not allow to split that into two different functions.

As in the two party communication, the user can generate one or more connections between the parties. The channels are connected using a single port for each application, specified in the first PartyData object in the given parties list.

Here is an example on how to use the SocketMultipartyCommunicationSetup class:

```java
import java.util.List;
import java.util.Map;
import edu.biu.scapi.comm.Channel;
import edu.biu.scapi.comm.twoPartyComm.LoadSocketParties;
import edu.biu.scapi.comm.twoPartyComm.PartyData;
```
//Prepare the parties list.
LoadSocketParties loadParties = new LoadSocketParties("MultiPartySocketParties1.properties");
List<PartyData> listOfParties = loadParties.getPartiesList();

//Create the communication setup class.
MultipartyCommunicationSetup commSetup = new SocketMultipartyCommunicationSetup(listOfParties);

//Request two channels between me and each other party.
HashMap<PartyData, Object> connectionsPerParty = new HashMap<PartyData, Object>();
connectionsPerParty.put(listOfParties.get(1), 2);
connectionsPerParty.put(listOfParties.get(2), 2);

//Call the prepareForCommunication function to establish the connections within 2000000 milliseconds.
Map<PartyData, Map<String, Channel>> connections = commSetup.prepareForCommunication(connectionsPerParty, 2000000);

//Returns the channels to the other parties.
return connections;

In order to use the SSLSocketMultipartyCommunicationSetup class one should add the key store name, trust store name and password to the constructor:

import java.util.List;
import java.util.Map;

import edu.biu.scapi.comm.Channel;
import edu.biu.scapi.comm.multiPartyComm.SSLSocketMultipartyCommunicationSetup;
import edu.biu.scapi.comm.twoPartyComm.LoadSocketParties;
import edu.biu.scapi.comm.twoPartyComm.PartyData;

//Prepare the parties list.
LoadSocketParties loadParties = new LoadSocketParties("SocketParties1.properties");
List<PartyData> listOfParties = loadParties.getPartiesList();

//Create the communication setup class.
MultipartyCommunicationSetup commSetup = new SSLSocketMultipartyCommunicationSetup(listOfParties, "p1Keystore.jks", "p1Cacerts.jks", "changeit");

//Request two channels between me and each other party.
HashMap<PartyData, Object> connectionsPerParty = new HashMap<PartyData, Object>();
connectionsPerParty.put(listOfParties.get(1), 2);
connectionsPerParty.put(listOfParties.get(2), 2);

//Call the prepareForCommunication function to establish the connections within 2000000 milliseconds.
Map<PartyData, Map<String, Channel>> connections = commSetup.prepareForCommunication(connectionsPerParty, 2000000);

//Returns the channels to the other parties.
return connections;

Here is an example of how to use the ActiveMQMultipartyCommunicationSetup class:

import java.util.List;
import java.util.Map;

import edu.biu.scapi.comm.Channel;
import edu.biu.scapi.exceptions.DuplicatePartyException;
import edu.biu.scapi.comm.twoPartyComm.LoadQueueParties;

1.4. The Communication Layer
import edu.biu.scapi.twoPartyComm.PartyData;

//Prepare the parties list.
LoadQueueParties loadParties = new LoadQueueParties("MultiPartyQueueParties1.properties");
List<PartyData> listOfParties = loadParties.getPartiesList();

//Create the communication setup class.
MultipartyCommunicationSetup commSetup = new ActiveMQMultipartyCommunicationSetup(loadParties.getURL(), listOfParties);

//Request two channels between me and each other party.
HashMap<PartyData, Object> connectionsPerParty = new HashMap<PartyData, Object>();
connectionsPerParty.put(listOfParties.get(1), 2);
connectionsPerParty.put(listOfParties.get(2), 2);

//Call the prepareForCommunication function to establish the connections within 2000000 milliseconds.
Map<PartyData, Map<String, Channel>> connections = commSetup.prepareForCommunication(connectionsPerParty, 2000000);

//Returns the channels to the other parties.
return connections;

And an example to SSLActiveMQMultipartyCommunicationSetup class:

    import java.util.List;
    import java.util.Map;

    import edu.biu.scapi.comm.Channel;
    import edu.biu.scapi.exceptions.DuplicatePartyException;
    import edu.biu.scapi.twoPartyComm.LoadQueueParties;
    import edu.biu.scapi.twoPartyComm.PartyData;

    //Prepare the parties list.
    LoadQueueParties loadParties = new LoadQueueParties("MultiPartySSLQueueParties1.properties");
    List<PartyData> listOfParties = loadParties.getPartiesList();

    //Create the communication setup class.
    MultipartyCommunicationSetup commSetup = new SSLActiveMQMultipartyCommunicationSetup(loadParties.getURL(), listOfParties, "changeit");

    //Request two channels between me and each other party.
    HashMap<PartyData, Object> connectionsPerParty = new HashMap<PartyData, Object>();
    connectionsPerParty.put(listOfParties.get(1), 2);
    connectionsPerParty.put(listOfParties.get(2), 2);

    //Call the prepareForCommunication function to establish the connections within 2000000 milliseconds.
    Map<PartyData, Map<String, Channel>> connections = commSetup.prepareForCommunication(connectionsPerParty, 2000000);

    //Returns the channels to the other parties.
    return connections;

Verifying that the connections were established

In two-party protocols, success means that all requested channels have been established between the parties. The output from the prepareForCommunication function is a map containing the established channels.

In multi-party protocols, different computations may require different types of success when checking the connections.
between all the parties that were supposed to participate. Some protocols may need to make sure that absolutely all parties participating in it have established connections one with another; other protocols may need only a certain percentage of connections to have succeeded. In the current multiparty communication setup implementation, we implement success as all requested channels have been established. In the future we may support different levels of success.

In case a timeout has occurred before all requested channels have been connected, all connected channels will be closed and a ScapiRuntimeException will be thrown.

**Closing the connection**

The application is responsible for closing the communicationSetup class that creates the channels. This is because this class may contain some members that need to be closed. For example, the QueueCommunicationSetup has the JMS Connection object as a class member, and this must be closed at the end of the setup.

Needless to say, the application must also close each created channel when it is no longer needed.

**Using an established connection**

A connection is represented by the Channel interface. Once a channel is established, we can send() and receive() data between parties.

```java
public interface Channel {
    public void send(Serializable data)
        Sends a message msg to the other party, msg must be a Serializable object.
    public Serializable receive()
        Receives a message from the channel.

        Returns Returns the received message as Serializable. Conversion to the right type is the responsibility of the caller.
    public void close()
        Closes the connection.
    public boolean isClosed()
        Returns true if the connection is closed, false otherwise.
}
```

**Security of the connection**

**Note:** This section is relevant for all channel types except the SSLChannel. SSL channels do the encryption and authentication in the SSL protocol and therefore do not need to be wrapped with SCAPI’s encrypted and/or authenticated channels. The methods described here are useful for anyone who does not wish to setup certificates and would rather work with pre-shared secrets.

A channel can have Plain, Encrypted or Authenticated security level, depending on the requirements of the application. The type of security set by CommunicationSetup classes is Plain security, and is represented by the classes PlainTCPChannel, PlainTCP_socketChannel and QueueChannel. In case a higher security standard is needed, the user must set it manually, by using the decorator classes AuthenticatedChannel and EncryptedChannel.
Plain Channel

Plain security is the default type of security set by the CommunicationSetup classes. The `PlainTCPChannel`, `PlainTCPsocketChannel` and `QueueChannel` classes are plain channels by default and so do not provide authentication or encryption. The plain channel types are as follows:

```java
public class PlainTCPChannel extends Channel
public class PlainTCPsocketChannel extends Channel
public class QueueChannel extends Channel
```

AuthenticatedChannel

```java
public class AuthenticatedChannel extends ChannelDecorator
```

This channel ensures `UnlimitedTimes` security level, meaning that there is no a priori bound on the number of messages that can be MACed. The owner of the channel is responsible for setting the MAC algorithm to use and making sure that the MAC is initialized with a suitable key. Then, every message sent via this channel is authenticated using the underlying MAC algorithm and every message received is verified by it.

The user needs not worry about any of the authentication and verification tasks as they are carried out automatically by the channel. Note that plain objects are passed to the channel and received from the channel and the processes of MACing and verifying the MAC are carried out inside the channel, invisible to the user.

```java
public AuthenticatedChannel (Channel channel, Mac mac)
```

This public constructor can be used by anyone holding a channel that is connected. Such a channel can be obtained by running the prepareForCommunication function of `CommunicationSetup` which returns a set of already connected channels.

**Parameters**

- `channel` – an already connected channel
- `mac` – the MAC algorithm required to authenticate the messages sent by this channel

**Throws**

- `SecurityLevelException` – if the MAC algorithm passed is not UnlimitedTimes-secure

```java
public void setKey (SecretKey key)
```

Sets the key of the underlying MAC algorithm. This function must be called before sending or receiving messages if the MAC algorithm passed to this channel had not been set with a key yet. The key can be set indefinite number of times depending on the needs of the application.

**Parameters**

- `key` – a suitable SecretKey

**Throws**

- `InvalidKeyException` – if the given key does not match the underlying MAC algorithm.

Example of Usage:

We assume in this example that `ch` is an already established channel as we have already shown how to setup a channel using `CommunicationSetup`. We stress that this is the code for one party, but both parties must decorate their respective channels with `AuthenticatedChannel` in order for it to work.
import java.security.InvalidKeyException;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import edu.biu.scapi.comm.*;
import edu.biu.scapi.midLayer.symmetricCrypto.mac.Mac;
import edu.biu.scapi.tools.Factories.*;
import edu.biu.scapi.exceptions.*;

public AuthenticatedChannel createAuthenticatedChannel(Channel ch) {
    Mac mac = null;
    
    // You could generate the key here and then somehow send it to the other party so the other party
    // can use the same secret key.
    //SecretKey macKey = SecretKeyGeneratorUtil.generateKey("AES");
    // Instead, we use a secretKey that has already been agreed upon by both parties:
    byte[] aesFixedKey = new byte[]{-61, -19, 106, -97, 106, 40, 52, -64, -115, -19, -87, -67, 98, 102, 16, 21};
    SecretKey key = new SecretKeySpec(aesFixedKey, "AES");
    try {
        mac.setKey(key);
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    }
    try {
        authenChannel = new AuthenticatedChannel(ch, mac);
    } catch (SecurityLevelException e) {
        // This exception will not happen since we chose a Mac that meets the Security Level requirements.
        e.printStackTrace();
    }
    return authenChannel;
}

After converting the channel to an authenticated channel, we can simply call send() and receive() again in the
same manner as before, only this time the messages are authenticated for us.

EncryptedChannel

public class EncryptedChannel extends ChannelDecorator
    This channel ensures CPA security level (security in the presence of chosen-plaintext attacks). The owner of the
channel is responsible for setting the encryption scheme to use and making sure that the encryption scheme is
initialized with a suitable key. Then, every message sent via this channel is encrypted and decrypted using the
underlying encryption scheme. As with an authenticated channel, the encryption and decryption are carried out invisibly to the user (who sends and receives plain objects).

We remark that in the setting of secure computation, encrypted but not authenticated channels should typically
not be used.

public EncryptedChannel (Channel channel, SymmetricEnc encScheme)
    This public constructor can be used by anyone holding a channel that is connected. Such a channel can be
obtained by running the prepareForCommunications function of CommunicationSetup which returns a set
of already connected channels.

The function creates a new EncryptedChannel that wraps the already connected channel mentioned above. The encryption scheme must be CPA-secure, otherwise an exception is thrown. The encryption scheme does not need to be initialized with a key at this moment (even though it can be), but before sending or receiving a message over this channel the relevant secret key must be set with setKey().

Parameters

- **channel** – an already connected channel
- **encScheme** – a symmetric encryption scheme that is CPA-secure.

Throws

- **SecurityLevelException** – if the encryption scheme is not CPA-secure

```java
public void setKey(SecretKey key)
```

Sets the key of the underlying encryption scheme. This function must be called before sending or receiving messages if the encryption scheme passed to this channel had not been set with a key yet. The key can be set indefinite number of times depending on the needs of the application.

Parameters

- **key** – a suitable SecretKey

Throws

- **InvalidKeyException** – if the given key does not match the underlying MAC algorithm.

Example of Usage

This example is very similar to the previous one. As before we only show how to decorate the established channel after CommunicationSetup is called.

```java
import java.io.IOException;
import java.security.InvalidKeyException;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import edu.biu.scapi.comm.Channel;
import edu.biu.scapi.comm.EncryptedChannel;
import edu.biu.scapi.exceptions.SecurityLevelException;
import edu.biu.scapi.midLayer.symmetricCrypto.encryption.ScCTREncRandomIV;
import edu.biu.scapi.primitives.prf.AES;
import edu.biu.scapi.primitives.prf.bc.BcAES;

public EncryptedChannel createEncryptedChannel(Channel ch) {
    ScCTREncRandomIV enc = null;
    try {
        // first we generate the secret key for the PRP that is used by the encryption object.
        // You could generate the key here and then somehow send it to the other party so the other party can use the same key.
        // SecretKey encKey = SecretKeyGeneratorUtil.generateKey("AES");
        // Instead, we use a secretKey that has already been agreed upon by both parties:
        byte[] aesFixedKey = new byte[]{-61, -19, 106, -97, 106, 40, 52, -64, -115, -19, -87, -67, 98, 102, 16, 21};
        SecretKey encKey = new SecretKeySpec(aesFixedKey, "AES");

        // now, we initialize the PRP, set the key, and then initialize the encryption object
        AES aes = new BcAES();
```
Encrypted and Authenticated Channel

We now provide an example of both encrypted and authenticated communication. This example is very similar to the previous ones. When using encryption and authentication in the correct order (encrypt-then-authenticate), authenticated encryption security is obtained (which is in particular CCA secure).

```java
import java.io.IOException;
import java.security.InvalidKeyException;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import edu.biu.scapi.comm.Channel;
import edu.biu.scapi.comm.EncryptedChannel;
import edu.biu.scapi.exceptions.SecurityLevelException;
import edu.biu.scapi.midLayer.symmetricCrypto.encryption.ScCTREncRandomIV;
import edu.biu.scapi.midLayer.symmetricCrypto.encryption.ScEncryptThenMac;
import edu.biu.scapi.midLayer.symmetricCrypto.mac.ScCbcMacPrepending;
import edu.biu.scapi.primitives.prf.AES;
import edu.biu.scapi.primitives.prf.bc.BcAES;

public EncryptedChannel createSecureChannel(Channel ch) {
    ScCTREncRandomIV enc = null;
    ScCbcMacPrepending cbcMac = null;
    try {
        // first, we set the encryption object
        // You could generate the key here and then somehow send it to the other party so the other party uses the same secret key
        // SecretKey encKey = SecretKeyGeneratorUtil.generateKey("AES");
        // Instead, we use a secretKey that has already been agreed upon by both parties.
        byte[] aesFixedKey = new byte[]{-61, -19, 106, -97, 106, 40, 52, -64, -115, -19, -87, -67, 98, 102, 16, 21};
        SecretKey aesKey = new SecretKeySpec(aesFixedKey, "AES");
        AES encryptAes = new BcAES();
        encryptAes.setKey(aesKey);
        // create encryption object from PRP
        enc = new ScCTREncRandomIV(encryptAes);
    }
}
```
// second, we create the mac object
AES macAes = new BcAES();

macAes.setKey(aesKey);
// create Mac object from PRP
cbcMac = new ScCbcMacPrepending(macAes);
}
} catch (InvalidKeyException e) {
 e.printStackTrace();
}

//Create the encrypt-then-mac object using encryption and authentication objects.
ScEncryptThenMac encThenMac = null;
encThenMac = new ScEncryptThenMac(enc, cbcMac);

//Decorate the Plain TCP Channel with the authentication
EncryptedChannel secureChannel = null;
try {
 secureChannel = new EncryptedChannel(ch, encThenMac);
} catch (SecurityLevelException e) {
 // This exception will not happen since we chose a Mac that meets the Security level requirements
 e.printStackTrace();
}

return secureChannel;


Multiparty communication

The multiparty communication layer will be updated soon to be based on the two-party communication layer. Meanwhile, the description below is for the old implementation which will soon be deprecated.

Fetch the list of parties from a properties file

The first thing that needs to be done to obtain communication services is to setup the connections between the different parties. Each party needs to run the setup process at the end of which the established connections are obtained. The established connections are called channels. The list of parties and their addresses are usually obtained from a Properties file. For example, here is a properties file called Parties0.properties:

# A configuration file for the parties

NumOfParties = 2
IP0 = 127.0.0.1
IP1 = 127.0.0.1
Port0 = 8001
Port1 = 8000

In order to read this file, we can use the LoadParties class:

import edu.biu.scapi.comm.Party;
import edu.biu.scapi.comm.LoadParties;
LoadParties loadParties = new LoadParties("Parties0.properties");
List<Party> listOfParties = loadParties.getPartiesList();

Each party is represented by an instance of the Party class. A List<Party> object is required in the multiParty communication setup phase.

Setup communication to other parties

The CommunicationSetup Class is responsible for establishing secure communication to other parties. An application requesting from CommunicationSetup to prepare for communication needs to call the CommunicationSetup::prepareForCommunication() function:

```java
public class CommunicationSetup implements TimeoutObserver

CommunicationSetup implements the org.apache.commons.exec.TimeoutObserver interface. This interface supplies a mechanism for notifying classes that a timeout has arrived.

Map<InetSocketAddress, Channel> prepareForCommunication(List<Party> listOfParties, ConnectivitySuccessVerifier successLevel, long timeOut, boolean enableNagle)

Parameters

- listOfParties (List<Party>) – The list of parties to connect to. As a convention, we will set the first party in the list to be the requesting party, that is, the party represented by the application.
- successLevel (ConnectivitySuccessVerifier) – The type of multi party connecting success required.
- timeOut (long) – A time-out (in milliseconds) specifying how long to wait for connections to be established and secured.
- enableNagle (boolean) – Whether or not Nagle’s algorithm <http://en.wikipedia.org/wiki/Nagle’s_algorithm> can be enabled.

Returns a map of the established channels.

Here is an example on how to use the CommunicationSetup class, we leave the discussion about the ConnectivitySuccessVerifier instance to the next section.
```

```java
import java.net.InetSocketAddress;
import java.util.List;
import java.util.Map;
import edu.biu.scapi.comm.Party;
import edu.biu.scapi.comm.LoadParties;
import edu.biu.scapi.comm.Channel;
import edu.biu.scapi.comm.CommunicationSetup;
import edu.biu.scapi.comm.ConnectivitySuccessVerifier;
import edu.biu.scapi.comm.NaiveSuccess;

//Prepare the parties list.
LoadParties loadParties = new LoadParties("Parties0.properties");
List<Party> listOfParties = loadParties.getPartiesList();

//Create the communication setup.
CommunicationSetup commSetup = new CommunicationSetup();
```
// Choose the naive connectivity success algorithm.  
ConnectivitySuccessVerifier naive = new NaiveSuccess();  

long timeoutInMs = 60000; // The maximum amount of time we are willing to wait to set a connection.

Map<InetSocketAddress, Channel> map = commSetup.prepareForCommunication(listOfParties, naive, timeoutInMs);

// prepareForCommunication() returns a map with all the established channels,
// we return only the first one since this code assumes the two-party case.
return map.values().iterator().next();

Verifying that the connections were established

Different Multi-parties computations may require different types of success when checking the connections between all the parties that were supposed to participate. Some protocols may need to make sure that absolutely all parties participating in it have established connections one with another; other protocols may need only a certain percentage of connections to have succeeded. There are many possibilities and each one of them is represented by a class implementing the ConnectivitySuccessVerifier interface. The different classes that implement this interface will run different algorithms to verify the level of success of the connections. It is up to the user of the CommunicationSetup class to choose the relevant level and pass it on to the CommunicationSetup upon calling the prepareForCommunication function.

public interface ConnectivitySuccessVerifier

public boolean hasSucceeded(EstablishedConnections estCon, List<Party> originalListOfParties)

This function gets the information about the established connections as input and the original list of parties, then it runs a certain algorithm (determined by the implementing class), and it returns true or false according to the level of connectivity checked by the implementing algorithm.

Parameters

• estCon – the actual established connections
• originalListOfParties – the original list of parties to connect to

Returns true if the level of connectivity was reached (depends on implementing algorithm) and false otherwise.

Naive

public class NaiveSuccess implements ConnectivitySuccessVerifier

NaiveSuccess does not actually check the connections but rather always returns true. It can be used when there is no need to verify any level of success in establishing the connections.

Clique

public class CliqueSuccess implements ConnectivitySuccessVerifier

For future implementation.

• Check if connected to all parties in original list.
• Ask every party if they are connected to all parties in their list.
• If all answers are true, return true.
• Else, return false.
SecureClique
public class SecureCliqueSuccess implements ConnectivitySuccessVerifier
For future implementation.

• Check if connected to all parties in original list.
• Ask every party if they are connected to all parties in their list. USE SECURE BROADCAST. DO NOT TRUST THE OTHER PARTIES.
• If all answers are true, return true,
• Else, return false.

Layer 1: Basic Primitives

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 \text{DDH}.

The library therefore includes a hierarchy of security level interfaces; specifically in this layer for DlogGroup and CryptographicHash families. These interfaces have no methods and are only marker interfaces. Each concrete class that is based on any security level should implement the relevant interface, to declare itself as secure as the marker interface.
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.
The **CryptographicHash** interface

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.

```java
void update(byte[] in, int inOffset, int inLen)
```

Adds the byte array to the existing msg to hash.

**Parameters**

- `in` – input byte array
- `inOffset` – the offset within the byte array
- `inLen` – the length. The number of bytes to take after the offset

```java
void hashFinal(byte[] out, int outOffset)
```

Completes the hash computation.

**Parameters**

- `out` – the output in byte array
- `outOffset` – the offset which to put the result bytes from

**Usage**

The best way to use CryptographicHash is via the `CryptographicHashFactory` factory class.

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

//call the CryptographicHashFactory.
CryptographicHash hash = CryptographicHashFactory.getInstance().getObject("SHA-1");

//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 possible keys to the `CryptographicHashFactory`.

Default keys: (point to the Crypto++ implementation)
### Universal Hash

A **universal hash function** is a family of hash functions with the property that a randomly chosen hash function (from the family) yields very few collisions, with good probability. More importantly in a cryptographic context, universal hash functions have important properties, like good randomness extraction and pairwise independence. Many universal families are known (for hashing integers, vectors, strings), and their evaluation is often very efficient.

The notions of universal hashing and cryptographic hash are distinct, and should not be confused (it is unfortunate that they have a similar name). We therefore completely separate the two implementations so that cryptographic hash functions cannot be confused with universal hash functions.

The output length of universal hash function is fixed for any given instantiation. The input is fixed (maybe for a certain instantiation) for some implementations and may be varying for other implementations. Since the input can be either fixed or varying we supply a compute function with input length as an argument for the varying version. The function `getInputLength()` plays a slightly different role for each version.
The **UniversalHash interface**

public void **setKey**(SecretKey secretKey)
Sets the secret key for this UH. The key can be changed at any time.

**Parameters**
- secretKey – secret key

**Throws** InvalidKeyException

public boolean **isKeySet**()
An object trying to use an instance of UH needs to check if it has already been initialized.

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

public int **getInputSize**()
This function has multiple roles depending on the concrete hash function.

If the concrete class can get a varying input lengths then there are 2 possible answers:

1. The maximum size of the input if there is some kind of an upper bound on the input size (for example in the EvaluationHashFunction there is a limit on the input size due to security reasons). Thus, this function returns this bound even though the actual size can be any number between zero and that limit.

2. If there is no limit on the input size, this function returns 0.

Otherwise, if the concrete class can get a fixed length, this function returns a constant size that may be determined either in the init for some implementations or hardcoded for other implementations.

**Returns** the input size of this hash function

public int **getOutputSize**()
**Returns** the output size of this hash function

public SecretKey **generateKey**(AlgorithmParameterSpec keyParams)
Generates a secret key to initialize this UH object.

**Parameters**
- keyParams – contains the required parameters for the key generation

**Throws** InvalidParameterSpecException

**Returns** the generated secret key

public SecretKey **generateKey**(int keySize)
Generates a secret key to initialize this UH object.

**Parameters**
- keySize – is the required secret key size in bits

**Returns** the generated secret key

public void **compute**(byte[] in, int inOffset, int inLen, byte[] out, int outOffset)
Computes the hash function on the in byte array and put the result in the output byte array.

**Parameters**
- in – input byte array
- inOffset – the offset within the input byte array
- inLen – the number of bytes to take after the offset
- out – output byte array
• **outOffset** – the offset within the output byte array

**Throws** IllegalBlockSizeException if the input length is greater than the upper limit

**Example of Usage**

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

// initiates an EvaluationHashFunction object using the UniversalHashFactory
UniversalHash uh = UniversalHashFactory.getInstance().getObject("ScapiEvaluationHash");

// calls the compute() function in the UniversalHash interface
uh.compute(in, 0, in.length, out, 0);
```

**Supported Hash Types**

In this section we present possible keys to the UniversalHashFactory. Currently, there is only one supported implementation of UniversalHash.

<table>
<thead>
<tr>
<th>Key</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScapiEvaluationHash</td>
<td>edu.biu.scapi.primitives.universalHash.EvaluationHashFunction</td>
</tr>
</tbody>
</table>

**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**

- Pseudorandom Function (PRF)
  - The PseudorandomFunction Interface
    - Block Manipulation
    - Setting the Secret Key
  - Basic Usage
  - Pseudorandom Function with Varying Input-Output Lengths
    - How to use the Varying Input-Output Length PRF

**The PseudorandomFunction Interface**

The main function of this interface 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

public void computeBlock (byte[] inBytes, int inOff, byte[] outBytes, int outOff)

Computes the function using the secret key. The user supplies the input byte array and the offset from which to take the data from. The user also supplies the output byte array as well as the offset. The computeBlock function will put the output in the output array 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

Throws

- IllegalBlockSizeException –

public void computeBlock (byte[] inBytes, int inOff, byte[] outBytes, int outOff, int outLen)

Computes the function using the secret key. This function is provided in the interface 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 array
- inLen – the length of the input array
- outBytes – output bytes. The resulted bytes of compute
- outOff – output offset in the outBytes array to put the result from
- outLen – the length of the output array

Throws

- IllegalBlockSizeException –

public void computeBlock (byte[] inBytes, int inOffset, int inLen, byte[] outBytes, int outOffset)

Computes the function using the secret key.

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

Parameters

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

Throws
• IllegalBlockSizeException –

public int getBlockSize()

Returns the input block size in bytes

Setting the Secret Key

public SecretKey generateKey (AlgorithmParameterSpec keyParams)
Generates a secret key to initialize this prf object.

Parameters
• keyParams – algorithmParameterSpec contains the required parameters for the key generation

Throws
• InvalidParameterSpecException –

Returns the generated secret key

public SecretKey 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

public boolean 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.

public void setKey (SecretKey secretKey)
Sets the secret key for this prf. The key can be changed at any time.

Parameters
• secretKey – secret key

Throws
• InvalidKeyException –

Basic Usage

//Create secretKey and in, in2, out byte arrays
...

// initiate a PRF of type TripleDES using the PrfFactory
PseudorandomFunction prf = PrfFactory.getInstance().getObject("TripleDES")

//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 interface. An example for such PRF is Hmac.

How to use the Varying Input-Output Length PRF

```java
//Create secret key and in, out byte arrays
...

//call the PrfFactory.
PseudorandomFunction prf = PrfFactory.getInstance().getObject("IteratedPrfVarying(Hmac(SHA-1))");

//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);
```

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 Interface

The PseudorandomPermutation interface extends the PseudorandomFunction interface, and adds the following functionality.

```java
public void invertBlock (byte[] inBytes, int inOff, byte[] outBytes, int outOff)
Inverts the permutation using the given key.
```

This function is a part of the PseudorandomPermutation interface 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 array
- **outBytes** – output bytes. The resulted bytes of invert
- **outOff** – output offset in the outBytes array to put the result from

Throws

- **IllegalBlockSizeException** –

```java
public void invertBlock (byte[] inBytes, int inOff, 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 array
- **outBytes** – output bytes. The resulted bytes of invert
- **outOff** – output offset in the outBytes array to put the result from
- **len** – the length of the input and the output

Throws

- **IllegalBlockSizeException** –

Basic Usage

```java
//Create secretKey and in, out, inv byte arrays
...

//call the PrfFactory and cast to prp
PseudorandomPermutation prp = (PseudorandomPermutation) PrfFactory.getInstance().getObject("OpenSSL", "AES");

//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

```java
//Create secretKey and in, out byte arrays
...

//call the PrfFactory and cast to prp
PseudorandomPermutation prp = (PseudorandomPermutation) PrfFactory.getInstance().getObject("LubyRackoffPrpFromPrfVarying");

//set the key
prp.setKey(secretKey);

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

public void getPRGBytes (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

Basic Usage

//Create secret key and out byte array
...

//Create prg using the PrgFactory
PseudorandomGenerator prg = PrgFactory.getInstance().getObject("RC4");
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);

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

• Trapdoor Permutation
   – The TPElement Interface
   – The TrapdoorPermutation Interface
      • Core Functionality
      • Generating TPElements
      • Checking Element Validity
      • Encryption Keys Functionality
   – BasicUsage
   – Supported Trapdoor Permutations
The TPElement Interface

The TPElement interface represents a trapdoor permutation element.

public BigInteger getElement()  
Returns the trapdoor element value as BigInteger.

public TPElementSendableData generateSendableData()  
This function extracts the actual value of the TPElement and wraps it in a TPElementSendableData that as it name indicates can be send using the serialization mechanism.

The TrapdoorPermutation Interface

This interface is the general interface of trapdoor permutation.

Core Functionality

public TPElement 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

Throws IllegalArgumentException if the given element is invalid for this permutation

public TPElement 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

Throws KeyException if there is no private key

Throws IllegalArgumentException if the given element is invalid for this permutation

public byte 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.

public byte[] 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

public TPElement `generateRandomTPElement ()`
creates a random TPElement that is valid for this trapdoor permutation

**Returns** the created random element

public TPElement `generateTPElement (BigInteger x)`
Creates a TPElement from a specific value $x$. It checks that the $x$ value is valid for this trapdoor permutation.

**Returns** If the $x$ value is valid for this permutation return the created random element

**Throws** IllegalArgumentException if the given value $x$ is invalid for this permutation

public TPElement `generateUncheckedTPElement (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

public TPElement `reconstructTPElement (TPElementSendableData data)`
Creates a TPElement from data that was probably obtained via the serialization mechanism.

**Parameters**
- data – serialized data necessary to reconstruct a given TPElement

**Returns** the reconstructed TPElement

### Checking Element Validity

public TPElValidity `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

public enum TPElValidity
Enum that represent the possible validity values of trapdoor element. There are three possible validity values:

**Parameters**
- VALID – it is an element
- NOT_VALID – it is not an element
- DONT_KNOW – there is not enough information to check if it is an element or not
Encryption Keys Functionality

public void setKey (PublicKey publicKey, 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

public void setKey (PublicKey publicKey)
Sets this trapdoor permutation with a public key. After this initialization, this object can do compute() but not invert(). This initialization is for user that wants to encrypt messages using the public key but cannot decrypt messages.

Parameters
• publicKey – the public key

Throws  InvalidKeyException if the key is not a valid key of this permutation

public boolean 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.

public 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 trapdoor permutation:
TrapdoorPermutation trapdoorPermutation = TrapdoorPermutationFactory.getInstance().getObject("RSA", "SCAPI");
//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
channel.send(publicKey.getEncoded());
channel.send(maskedSecret.generateSendableData());

Here is the code of the receiver:
Serializable pkey = channel.receive();
TPElementSendableData secretMsg = (TPElementSendableData) channel.receive();

// reconstruct publicKey from pkey
...

// instantiate the trapdoor permutation:
TrapdoorPermutation trapdoorPermutation = TrapdoorPermutationFactory.getInstance().getInstance("RSA", "SCAPI");
// set the keys for this trapdoor permutation
trapdoorPermutation.setKey(publicKey);

// reconstruct a TPElement from a TPElementSendableData
TPElement maskedSecret = trapdoorPermutation.reconstructTPElement(secretMsg);

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

**Supported Trapdoor Permutations**

In this section we present possible keys to the TrapdoorPermutationFactory.

Scapi’s own implementation of RSA trapdoor permutation:

<table>
<thead>
<tr>
<th>Key</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScapiRSA</td>
<td>edu.biu.scapi.primitives.trapdoorPermutation.ScRSAPermutation</td>
</tr>
</tbody>
</table>

Crypto++ implementation of RSA trapdoor permutation and Rabin trapdoor permutation:

<table>
<thead>
<tr>
<th>Key</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>CryptoPPRSA</td>
<td>edu.biu.scapi.primitives.trapdoorPermutation.cryptopp.CryptoPpRSAPermutation</td>
</tr>
<tr>
<td>CryptoPPRabin</td>
<td>edu.biu.scapi.primitives.trapdoorPermutation.cryptopp.CryptoPpRabinPermutation</td>
</tr>
</tbody>
</table>

OpenSSL implementation of RSA trapdoor permutation:

<table>
<thead>
<tr>
<th>Key</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSLRSA</td>
<td>edu.biu.scapi.primitives.trapdoorPermutation.openSSL.OpenSSLRSAPermutation</td>
</tr>
</tbody>
</table>

**Discrete Log Group**

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 \( \mathbb{Z}_p^* \) for a large \( p \), and some Elliptic curve groups.

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

- \( \mathbb{Z}_p^* \)
- Elliptic curve over the field \( GF(2^m) \)
- Elliptic curve over the field \( \mathbb{Z}_p \)

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.
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 \( \mathbb{Z}_p^* \) field.

At the third level we have:

1. DlogZpSafePrime: The order \( q \) is not only a prime but also is such that prime \( p = 2 \times q + 1 \).
2. DlogEcFp: Any elliptic curve over \( \mathbb{F}_p \).
3. DlogEcF2m: Any elliptic curve over \( \mathbb{F}_2^m \).

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 Interface

Group Parameters

```java
public GroupElement 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
public GroupElement **createRandomGenerator** ()
  Creates a random generator of this Dlog group

  **Returns** the random generator

public BigInteger **getOrder** ()
  **Returns** the order of this Dlog group

public GroupParams **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 \( \mathbb{Z}_p^* \) what defines the group is \( p \).

  **Returns** the GroupParams of that Dlog group

public String **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 \( \mathbb{F}_{2^m} \)” or “\( \mathbb{Z}_p^* \)”

  **Returns** the name of the group type

public GroupElement **getIdentity** ()
  **Returns** the identity of this Dlog group

---

**Exponentiation**

public GroupElement **exponentiate** (GroupElement **base**, BigInteger **exponent**)
  Raises the base GroupElement to the exponent. The result is another GroupElement.

  **Parameters**
  - base –
  - exponent –

  **Throws**
  - IllegalArgumentException –

  **Returns** the result of the exponentiation

public GroupElement **exponentiateWithPreComputedValues** (GroupElement **base**, 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 \( h_1, h_2, h_4, h_8, \ldots \) 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.

  **Parameters**
  - base –
  - exponent –

  **Returns** the exponentiation result

public void **endExponentiateWithPreComputedValues** (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.

  **Parameters**
public GroupElement **simultaneousMultipleExponentiations** (GroupElement[] groupElements, 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** –
- **exponentiations** –

**Returns** the exponentiation result

### Multiplication and Inverse

public GroupElement **getInverse** (GroupElement groupElement)

Calculates the inverse of the given GroupElement.

**Parameters**
- **groupElement** – to invert

**Throws**
- **IllegalArgumentException** –

**Returns** the inverse element of the given GroupElement

public GroupElement **multiplyGroupElements** (GroupElement groupElement1, GroupElement groupElement2)

Multiplies two GroupElements

**Parameters**
- **groupElement1** –
- **groupElement2** –

**Throws**
- **IllegalArgumentException** –

**Returns** the multiplication result

### Group Element Generation

public GroupElement **createRandomElement**()

Creates a random member of this Dlog group

**Returns** the random element

public GroupElement **generateElement** (boolean bCheckMembership, 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 –

Throws

• IllegalArgumentException –

Returns the generated GroupElement

Validation

public boolean 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.

public boolean 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

Throws

• IllegalArgumentException –

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

public boolean 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

public boolean isOrderGreaterThan (int numBits)
Checks if the order of this group is greater than 2^n numBits

Parameters

• numBits –

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

public boolean isPrimeOrder()
Checks if the order is a prime number

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

Group Element Serialization

public GroupElement reconstructElement (boolean 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

```java
public GroupElement encodeByteArrayToGroupElement (byte[] 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

```java
public byte[] 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

```java
public int 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

```java
public byte[] 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 Interface**

```java
public GroupElementSendableData generateSendableData()
```

This function is used when a group element needs to be sent via a `edu.biu.scapi.comm.Channel` or any other means of sending data (including serialization). 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

```java
public boolean 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 Interface**

```java
public BigInteger getQ()
```

**Returns** the group order q

**Basic Usage**

```java
// initiate a discrete log group (in this case the OpenSSL implementation of the elliptic curve group K-233)
DlogGroup dlog = new OpenSSLDlogECF2m("K-233");
SecureRandom random = new SecureRandom();

// get the group generator and order
GroupElement g = dlog.getGenerator();
BigInteger q = dlog.getOrder();
BigInteger qMinusOne = q.subtract(BigInteger.ONE);

// create a random exponent r
BigInteger r = BigIntegers.createRandomInRange(BigInteger.ZERO, qMinusOne, random);
// exponentiate g in r to receive a new group element
GroupElement g1 = dlog.exponentiate(g, r);
// create a random group element
GroupElement h = dlog.createRandomElement();
// multiply elements
GroupElement gMult = dlog.multiplyGroupElements(g1, h);
```

**key Derivation Function**

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).
The Key Derivation Function Interface:

public SecretKey deriveKey(byte[] entropySource, int inOff, int inLen, int outLen)
Generates a new secret key from the given seed.

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

Returns SecretKey the derivated key.

There is another variation of this function, that also takes into account an initial vector (iv):

public SecretKey deriveKey(byte[] entropySource, int inOff, int inLen, int outLen, byte[] iv)
Generates a new secret key from the given seed and iv.

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

```java
KeyDerivationFunction kdf = new HKDF(new BcHMAC());
byte[] source = "...";
int targetLen = 128;
byte[] kdfed = kdf.deriveKey(source, 0, source.length, targetLen).getEncoded();
```

Layer 2: Non Interactive Protocols

The second layer of Scapi includes different symmetric and asymmetric encryption schemes, 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; CBC-MAC uses any of the PRPs defined in the first level.

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. Scapi provides two implementations of message authentication codes: CBC-MAC and HMAC.
The Mac Interface

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

public interface Mac

Basic Mac and Verify Functionality

public byte[] mac(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 array to take the bytes from.
• msgLen – the length of the message in bytes.

Returns byte[] the return tag from the mac operation.

public boolean verify(byte[] msg, int offset, int msgLength, 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.

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

public void update(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.

```java
public byte[] doFinal (byte[] msg, int offset, int msgLength)
```
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

```java
public SecretKey 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.

```java
public SecretKey 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.

**Throws**

- **InvalidParameterSpecException** – if the given keyParams does not match this mac algorithm.

**Returns** the generated secret key.

```java
public boolean 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.

```java
public void setKey (SecretKey secretKey)
```
Sets the secret key for this mac. The key can be changed at any time.

**Parameters**

- **secretKey** – secret key

**Throws**

- **InvalidKeyException** – if the given key does not match this MAC algorithm.
Mac Properties

public int getMacSize()
    Returns the input block size in bytes.
    Returns the input block size.

CBC-MAC

A Cipher Block Chaining Message Authentication Code, abbreviated CBC-MAC, is a technique for constructing a message authentication code from a block cipher. The message is processed with some block cipher algorithm in CBC mode to create a chain of blocks such that each block depends on the previous blocks. This interdependence ensures that a change to any of the plaintext bits will cause the final encrypted block to change in a way that cannot be predicted or counteracted without knowing the key to the block cipher. The initialization vector (IV) usually present in CBC encryption is set to zero when a CBC MAC is computed (i.e., there is no IV). In addition, in order for CBC-MAC to be secure for variable-length messages, the length of the message has to be pre-pended to the message in the first block before beginning CBC-MAC. When computed in this way, CBC-MAC is a PRF and thus a secure MAC.

Note: We remark that if the length of the message is not known in advance then a different MAC algorithm should be used (for example: HMAC).

The CbcMac Interface

The CbcMac interface is the general interface for CBC-Mac. Every class that implement the CBC-Mac algorithm should implement this interface.

public interface CbcMac extends UniqueTagMac, PrfVaryingInputLength, UnlimitedTimes
public void startMac (int msgLength)
    Pre-pends the length if the message to the message. As a result, the mac will be calculated on [msgLength||msg].

Parameters
    • msgLength – the length of the message in bytes.

Basic Usage

// assume we have a message msg
byte[] msg;

// initialize a secure random object
SecureRandom random = new SecureRandom();

// initialize a prp to be used by the CbcMac algorithm
PseudorandomPermutation prp = new OpenSSLAES(random);
SecretKey secretKey = prp.generateKey(128);
prp.setKey(secretKey);

// initialize the CbcMac algorithm
CbcMac mac = new ScCbcMacPrepending(prp, random);

// calculate the tag on a complete message
byte[] tag = mac.mac(msg, 0, msg.length);
// compute the mac in stages (in case not all the message is known up front)
mac.startMac(100);
mac.update(msg, 0, 20);
mac.update(msg, 20, 20);
mac.update(msg, 40, 20);
mac.update(msg, 60, 20);
mac.doFinal(msg, 80, 20);

**HMAC**

We presented the same HMAC algorithm in the first layer of Scapi. 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 implement the Mac interface. 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 Interface**

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. Currently the BcHMAC class implements the Hmac interface.

```
public interface Hmac extends PrfVaryingInputLength, UniqueTagMac, UnlimitedTimes
```

**Basic Usage**

**Sender usage:**

```
//Create an hmac object.
Mac hmac = new BcHMAC("SHA-1");

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

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

//Get the message to mac and calculate the mac tag.
byte[] 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 throw new IllegalStateException() //Tag is not valid.
```
Symmetric Encryption

Scapi implements 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. Even though we do not currently implement any concrete homomorphic encryption, we provide the interfaces for future possible implementations.

The symmetric encryption family of classes 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.

**Note:** We note that for authenticated encryption, two secret keys will be used, one for the encryption and one for the authentication.

---

### Contents

- **Symmetric Encryption**
  - The SymmetricEnc Interface
    * Encryption and Decryption
    * Key Generation
    * Key Handling
  - The CBCEnc Interface
  - The CTREnc Interface
  - Basic Usage
  - The AuthenticatedEnc Interface
    * Basic Usage

---

### The SymmetricEnc Interface

**public interface** **SymmetricEnc** extends Eav, Indistinguishable

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

### Encryption and Decryption

**public SymmetricCiphertext encrypt (Plaintext plaintext)**

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

**Parameters**

- plaintext

**Throws**

- **IllegalArgumentException** – if the given plaintext does not match this encryption scheme.
• `IllegalStateException` – if no secret key was set.

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

```java
public SymmetricCiphertext encrypt (Plaintext plaintext, byte[] iv)
```

This function encrypts a plaintext. It lets the user choose the random IV.

**Parameters**

• `plaintext` –
• `iv` – random bytes to use in the encryption of the message.

**Throws**

• `IllegalArgumentException` – if the given plaintext does not match this encryption scheme.
• `IllegalBlockSizeException` – if the given IV length is not as the block size.
• `IllegalStateException` – if no secret key was set.

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

```java
public Plaintext decrypt (SymmetricCiphertext ciphertext)
```

This function performs the decryption of a ciphertext returning the corresponding decrypted plaintext.

**Parameters**

• `ciphertext` – The Ciphertext to decrypt.

**Throws**

• `IllegalArgumentException` – if the given ciphertext does not match this encryption scheme.
• `IllegalStateException` – if no secret key was set.

**Returns** the decrypted plaintext.

### Key Generation

```java
public SecretKey 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.

**Throws**

• `InvalidParameterSpecException` – if the given keyParams does not match this symmetric encryption.

**Returns** the generated secret key.

```java
public SecretKey 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

public boolean 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.

public void setKey (SecretKey secretKey)
    Sets the secret key for this symmetric encryption. The key can be changed at any time.

    Parameters
        • secretKey – secret key.

    Throws
        • InvalidKeyException – if the given key does not match this encryption scheme.

The CBCEnc Interface

These is a marker interface, for the CBC method:

![CBC Method Diagram]

public interface CBCEnc extends SymmetricEnc, Cpa

The CTREnc Interface

These is a marker interface, for the CTR method:
public interface **CTREnc** extends SymmetricEnc, Cpa

### Basic Usage

**Sender usage:**

```java
//Create an encryption object. The created object is a CTR-AES encryption scheme object.
SymmetricEnc encryptor = new ScCTREncRandomIV("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:**

```java
//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);
```

### The AuthenticatedEnc Interface

public interface **AuthenticatedEnc** extends SymmetricEnc, Cca2

public class **ScEncryptThenMac** implements AuthenticatedEnc

This class implements a type of authenticated encryption: encrypt then mac.

The encryption algorithm first encrypts the message and then calculates a mac on the encrypted message.

The decrypt algorithm receives an encrypted message and a tag. It first verifies the encrypted message with the tag. If verifies, then it proceeds to decrypt using the underlying decrypt algorithm, if not returns a null response.

This encryption scheme achieves Cca2 and NonMalleable security level.
Basic Usage

```java
//Create the PRP that is used by the encryption object.
AES aes = new BcAES();

//Create encryption object.
SymmetricEnc enc = new ScCTREncRandomIV(aes);

//Create the PRP that is used by the Mac object.
TripleDES tripleDes = new BcTripleDES();

//Create Mac object.
Mac cbcMac = new ScCbcMacPrepending(tripleDes);

//Create the encrypt-then-mac object using encryption and authentication objects.
ScEncryptThenMac encThenMac = new ScEncryptThenMac(enc, cbcMac);

// prepare the plaintext
Plaintext plaintext;
...

// encrypt
SymmetricCiphertext cipher = encThenMac.encrypt(plaintext);

// decrypt
Plaintext decrypted = encThenMac.decrypt(cypher);
```

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.
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 Interface

public interface AsymmetricEnc extends Cpa, Indistinguishable

General interface for asymmetric encryption. Each class of this family must implement this interface.
**Encryption and Decryption**

```java
public AsymmetricCiphertext encrypt (Plaintext plainText)
    Encrypts the given plaintext using this asymmetric encryption scheme.

    Parameters
    • plainText – message to encrypt

    Throws
    • IllegalArgumentException – if the given Plaintext doesn’t match this encryption type.
    • IllegalStateException – if no public key was set.

    Returns  Ciphertext the encrypted plaintext
```

```java
public Plaintext decrypt (AsymmetricCiphertext cipher)
    Decrypts the given ciphertext using this asymmetric encryption scheme.

    Parameters
    • cipher – ciphertext to decrypt

    Throws
    • IllegalArgumentException – if the given Ciphertext doesn’t match this encryption type.
    • KeyException – if there is no private key

    Returns  Plaintext the decrypted cipher
```

**Plaintext Manipulation**

```java
public Plaintext generatePlaintext (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.

    Throws
    • IllegalArgumentException – if the given message’s length is greater than the maximum.
```

```java
public byte[] 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.
```

```java
public int getMaxLengthOfByteArrayForPlaintext ()
    Returns the maximum size of the byte array that can be passed to generatePlaintext function. This is the maxi-
    mum size of a byte array that can be converted to a Plaintext object suitable to this encryption scheme.

    Throws
```
• **NoMaxException** – if this encryption scheme has no limit on the plaintext input.

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

    public boolean `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

public KeyPair `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

    **Throws**

    • `InvalidParameterSpecException` – if the given parameters don’t match this encryption scheme.

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

    public KeyPair `generateKey` ()

    Generates public and private keys for this asymmetric encryption.

    **Returns** KeyPair holding the public and private keys

### Key Handling

public PublicKey `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.

    ** Throws**

    • `IllegalStateException` – if no public key was set.

    **Returns** the PublicKey

    public boolean `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.

    public void `setKey` (PublicKey `publicKey`, PrivateKey `privateKey`)

    Sets this asymmetric encryption with public key and private key.

    **Parameters**

    • `publicKey` –

    • `privateKey` –

    **Throws**
- **InvalidKeyException** – if the given keys don’t match this encryption scheme.

```java
public void setKey (PublicKey publicKey)
```

Sets this asymmetric encryption with a public key.

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

**Parameters**

- **publicKey**

**Throws**

- **InvalidKeyException** – if the given key doesn’t match this encryption scheme.

### Reconstruction (from communication channel)

```java
public AsymmetricCiphertext 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

```java
public PrivateKey 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

```java
public PublicKey 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 keyPair using the encryptor. 
KeyPair pair = encryptor.generateKey();

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

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

//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 encryptor. 
KeyPair pair = encryptor.generateKey();

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

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

//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. 
byte[] 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 ScElGamalOnGroupElement, and the other one implements the encrypt function on a byte array and is called ScElGamalOnByteArray.

Note: Note that ElGamal on a groupElement is an asymmetric multiplicative homomorphic encryption, while ElGa-
ElGamalEnc Interface

public interface ElGamalEnc extends AsymmetricEnc

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

public AsymmetricCiphertext encryptWithGivenRandomValue (Plaintext plaintext, BigInteger y)

Encrypts the given message using ElGamal encryption scheme.

Parameters

• plaintext – contains message to encrypt. The given plaintext must match this ElGamal type.

Throws

• IllegalArgumentException – if the given Plaintext does not match this ElGamal type.

• IllegalStateException – if no public key was set.

Returns Ciphertext containing the encrypted message.

ScElGamalOnByteArray Interface

public class ScElGamalOnByteArray extends ElGamalAbs

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

public ScElGamalOnByteArray ()

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

public ScElGamalOnByteArray (DlogGroup dlogGroup, KeyDerivationFunction kdf)

Constructor that gets a DlogGroup and sets it to the underlying group. It lets SCAPI choose and source of randomness.

Parameters

• dlogGroup – must be DDH secure.

• kdf – a key derivation function.

Throws

• SecurityLevelException – if the given dlog group does not have DDH security level.

public ScElGamalOnByteArray (DlogGroup dlogGroup, KeyDerivationFunction kdf, SecureRandom 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.

Throws
• SecurityLevelException – if the given dlog group does not have DDH security level.

Complete Encryption
protected AsymmetricCiphertext completeEncryption (GroupElement c1, GroupElement hy, Plaintext plaintext)

Completes the encryption operation.

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

Throws
• IllegalArgumentException – if the given Plaintext is not an instance of ByteArrayPlaintext.

Returns Ciphertext of type ElGamalOnByteArrayCiphertext containing the encrypted message.

ScElGamalOnGroupElement Interface

public class ScElGamalOnGroupElement extends ElGamalAbs implements 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
public ScElGamalOnGroupElement ()

Default constructor. Uses the default implementations of DlogGroup, CryptographicHash and SecureRandom.

public ScElGamalOnGroupElement (DlogGroup dlogGroup)

Constructor that gets a DlogGroup and sets it to the underlying group. It lets SCAPI choose and source of randomness.

Parameters
• dlogGroup – must be DDH secure.

Throws
• SecurityLevelException –

public ScElGamalOnGroupElement (DlogGroup dlogGroup, SecureRandom random)

Constructor that gets a DlogGroup and source of randomness.

Parameters
• dlogGroup – must be DDH secure.
• random – source of randomness.

Throws
• SecurityLevelException – if the given dlog group does not have DDH security level.

Complete Encryption
protected AsymmetricCiphertext completeEncryption (GroupElement c1, GroupElement hy, Plaintext plaintext)

Completes the encryption operation.

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

Throws
• IllegalArgumentException – if the given Plaintext is not an instance of GroupElementPlaintext.

Returns Ciphertext of type ElGamalOnGroupElementCiphertext containing the encrypted message.

Multiply Ciphertexts (Homomorphic Encryption operation)
public AsymmetricCiphertext 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.

Throws
• IllegalArgumentException – in the following cases: 1. If one or more of the given ciphertexts is not instance of ElGamalOnGroupElementCiphertext. 2. If one or more of the GroupElements in the given ciphertexts is not a member of the underlying DlogGroup of this ElGamal encryption scheme.
• IllegalStateException – if no public key was set.

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:

//Create an underlying DlogGroup.
DlogGroup dlog = new MiraclDlogECFp();

//Create an ElGamalOnGroupElement encryption object.
ElGamalEnc elGamal = new ScElGamalOnGroupElement(dlog);

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

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

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

//Create a GroupElementPlaintext to encrypt and encrypt the plaintext.
Plaintext plaintext = new GroupElementPlaintext(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.
KeyPair pair = elGamal.generateKey();

//Publish your public key.
Publish(pair.getPublic());
// Set private key and party1’s public key:
elGamal.setKey(party1PublicKey, pair.getPrivate());

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

// Get the plaintext element and use it as needed.
GroupElement element = plaintext.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.

Similarly to ElGamal, Cramer Shoup encryption scheme can encrypt a group element and a byte array. In order to allow these two encryption types, we provide two Cramer Shoup concrete classes. One implements the encrypt function on a group element and is called \texttt{ScCramerShoupDDHOnGroupElement}, and the other one implements the encrypt function on a byte array and is called \texttt{ScCramerShoupDDHOnByteArray}.

The CramerShoupDDHEnc Interface

public interface \texttt{CramerShoupDDHEnc} extends \texttt{AsymmetricEnc}, \texttt{Cca2}

General interface for CramerShoup encryption scheme. Every concrete implementation of CramerShoup encryption should implement this interface. By definition, this encryption scheme is CCA-secure and NonMalleable.

The ScCramerShoupDDHOnByteArray Interface

public class \texttt{ScCramerShoupDDHOnByteArray} extends CramerShoupAbs

public \texttt{ScCramerShoupDDHOnByteArray}()

Default constructor. It uses a default Dlog group and CryptographicHash.

public \texttt{ScCramerShoupDDHOnByteArray}(DlogGroup \texttt{dlogGroup}, CryptographicHash \texttt{hash}, KeyDerivationFunction \texttt{kdf})

Constructor that lets the user choose the underlying dlog and hash. Uses default implementation of SecureRandom as source of randomness.

Parameters

- \texttt{dlogGroup} – underlying DlogGroup to use, it has to have DDH security level
- \texttt{hash} – underlying hash to use, has to have CollisionResistant security level

Throws

- \texttt{SecurityLevelException} – if the Dlog Group or the Hash function do not meet the required Security Level
public ScCramerShoupDDHOnByteArray (DlogGroup dlogGroup, CryptographicHash hash, KeyDerivationFunction kdf, SecureRandom random)

Constructor that lets the user choose the underlying dlog, hash and source of randomness.

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.

Throws

- SecurityLevelException – if the Dlog Group or the Hash function do not meet the required Security Level

The ScCramerShoupDDHOnGroupElement Interface

generics

public class ScCramerShoupDDHOnGroupElement

extends CramerShoupAbs

Concrete class that implement Cramer-Shoup encryption scheme. By definition, this encryption scheme is CCA-secure and NonMalleable.

public ScCramerShoupDDHOnGroupElement ()

Default constructor. It uses a default Dlog group and CryptographicHash.

public ScCramerShoupDDHOnGroupElement (DlogGroup dlogGroup, CryptographicHash hash)

Constructor that lets the user choose the underlying dlog and hash. Uses default implementation of SecureRandom as source of randomness.

Parameters

- dlogGroup – underlying DlogGroup to use, it has to have DDH security level
- hash – underlying hash to use, has to have CollisionResistant security level

Throws

- SecurityLevelException – if the Dlog Group or the Hash function do not meet the required Security Level

public ScCramerShoupDDHOnGroupElement (DlogGroup dlogGroup, CryptographicHash hash, SecureRandom random)

Constructor that lets the user choose the underlying dlog, hash and source of randomness.

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.

Throws

- SecurityLevelException – if the Dlog Group or the Hash function do not meet the required Security Level

Basic Usage

Sender usage:
//Create an underlying DlogGroup.
DlogGroup dlog = new MiraclDlogECF2m();

//Create a CramerShoupOnByteArray encryption object.
CramerShoupDDHEnc encryptor = new ScCramerShoupDDHOnByteArray(dlog);

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

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

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

//Get a byte[] message to encrypt. Check if the length of the given msg is valid.
if (encryptor.hasMaxByteArrayLengthForPlaintext())
    if (msg.length>encryptor.getMaxLengthOfByteArrayForPlaintext())
        throw new IllegalArgumentException("message too long");

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

//Encrypt the plaintext
AsymmetricCiphertext cipher = encryptor.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.
KeyPair pair = encryptor.generateKey();

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

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

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

//Get the plaintext bytes and use it as needed.
byte[] text = plaintext.getText();

**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.
Interface

public interface DamgardJurikEnc extends AsymAdditiveHomomorphicEnc

General interface for DamgardJurik encryption scheme. Every concrete implementation of DamgardJurik encryption should implement this interface. By definition, this encryption scheme is CPA-secure and Indistinguishable.

public AsymmetricCiphertext 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.

Parameters

• cipher –

Throws

• IllegalArgumentException – if the given ciphertext does not match this asymmetric encryption.
• IllegalStateException – if no public key was set.

Scapi Implementation

public class ScDamgardJurikEnc implements DamgardJurikEnc

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

public ScDamgardJurikEnc()

Default constructor. Uses the default implementations of SecureRandom.

public ScDamgardJurikEnc(SecureRandom rnd)

Constructor that lets the user choose the source of randomness.

Parameters

• rnd – source of randomness.

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 = new ScDamgardJurikEnc();

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

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

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

//Get the BigInteger value to encrypt, create a BigIntegerPlaintext with it and encrypt the plaintext...
BigIntegerPlainText plaintext = new BigIntegerPlainText(num);
AsymmetricCiphertext 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.
KeyPair pair = encryptor.generateKey();

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

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

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

//Get the plaintext element and use it as needed.
BigInteger element = plaintext.getX();

**RSA Oaep Encryption Scheme**

RSA-OAEP is a public-key encryption scheme combining the RSA algorithm with the Optimal Asymmetric Encryption Padding (OAEP) method.

**Interface**

public interface RSAOaepEnc extends AsymmetricEnc, Cca2

General interface for RSA OAEP encryption scheme. Every concrete implementation of RSA OAEP encryption should implement this interface. By definition, this encryption scheme is CCA-secure and NonMalleable.

**Scapi Implementation**

public class BcRSAOaep extends RSAOaepAbs

RSA-OAEP encryption scheme based on BC library’s implementation. By definition, this encryption scheme is CCA-secure and NonMalleable.

public BcRSAOaep ()

Default constructor. Uses default implementation of SecureRandom as source of randomness.

public BcRSAOaep (SecureRandom random)

Constructor that lets the user choose the source of randomness.

Parameters

- **random** – source of randomness.

**Crypto++ Implementation**

public class CryptoPPRSAOaep extends RSAOaepAbs

RSA-OAEP encryption scheme based on Crypto++ library’s implementation. By definition, this encryption scheme is CCA-secure and NonMalleable.
public CryptoPPRSAOaep()
    Default constructor. Uses default implementation of SecureRandom as source of randomness.

public CryptoPPRSAOaep(SecureRandom secureRandom)
    Constructor that lets the user choose the source of randomness.

    Parameters
    • secureRandom – source of randomness.

OpenSSL Implementation

public class OpenSSLRSAOaep extends RSAOaepAbs
    RSA-OAEP encryption scheme based on OpenSSL library’s implementation. By definition, this encryption
    scheme is CCA-secure and NonMalleable.

public OpenSSLRSAOaep()
    Default constructor. Uses default implementation of SecureRandom as source of randomness.

public OpenSSLRSAOaep(SecureRandom secureRandom)
    Constructor that lets the user choose the source of randomness.

    Parameters
    • secureRandom – source of randomness.

Basic Usage

Sender code:

    //Create an RSA encryption object.
    RSAOaepEnc encryptor = new CryptoPPRSAOaep();

    //Generate a keyPair using the RSAOaep object.
    KeyPair pair = encryptor.generateKey(new RSAKeyGenParameterSpec(1024, null));

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

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

    //Get a byte[] message to encrypt. Check if the length of the given msg is valid.
    if (encryptor.hasMoreBytesForPlaintext()){
        if (msg.length>encryptor.getMaxLengthOfByteArrayForPlaintext()) {
            throw new IllegalArgumentException("message too long");
        }
    }

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

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

    //Send cipher and keys to the receiver.

Receiver code:

1.6. Layer 2: Non Interactive Protocols
//Create the same RSAOaep object with the same definition as the sender’s object.
//Generate a keyPair using the RSAOaep object.
KeyPair pair = encryptor.generateKey();

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

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

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

//Get the plaintext bytes and use it as needed.
byte[] text = plaintext.getText();
...

Digital Signatures

A digital signature is a mathematical scheme for demonstrating the authenticity of a digital message or document. A valid digital signature provides the recipient with a reason to believe that the message was created by a known sender, and that it was not altered in transit.

The Digital Signatures family of classes 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 keys.
2. Signing a message.
3. Verifying a signature with a message.

Contents

- Digital Signatures
  - The DigitalSignature Interface
    - Sign and Verify
    - Key Generation and Handling
  - RSA Based Digital Signature
    - The RSABasedSignature Interface
    - BouncyCastle Implementation
    - Crypto++ Implementation
    - OpenSSL Implementation
    - Example of Usage
  - DSA Digital Signature
    - The DSABasedSignature Interface
    - Scapi Implementation
    - OpenSSL Implementation
    - Example of Usage
The DigitalSignature Interface

public interface DigitalSignature

General interface for digital signatures. Each class of this family must implement this interface. A digital signature is a mathematical scheme for demonstrating the authenticity of a digital message or document. A valid digital signature gives a recipient reason to believe that the message was created by a known sender, and that it was not altered in transit.

Sign and Verify

public Signature sign(byte[], int offset, int length)
Signs the given message

Parameters

• msg – the byte array to sign.
• offset – the place in the msg to take the bytes from.
• length – the length of the msg.

Throws

• ArrayIndexOutOfBoundsException – if the given offset and length are wrong for the given message.
• KeyException – if PrivateKey is not set.

Returns the signatures from the msg signing.

public boolean verify(Signature signature, byte[], int offset, int length)
Verifies the given signature

Parameters

• signature – to verify
• msg – the byte array to verify the signature with
• offset – the place in the msg to take the bytes from
• length – the length of the msg

Throws

• ArrayIndexOutOfBoundsException – if the given offset and length are wrong for the given message.
• IllegalArgumentException – if the given Signature does not match this signature scheme.
• IllegalStateException – if no public key was set.

Returns true if the signature is valid. false, otherwise.

Key Generation and Handling

public KeyPair generateKey(AlgorithmParameterSpec keyParams)
Generates public and private keys for this digital signature.

Parameters

• keyParams – hold the required key parameters
Throws
  • **InvalidParameterSpecException** – if the given keyParams does not match this signature scheme.

Returns KeyPair holding the public and private keys

```java
generateKey()
```
Generates public and private keys for this digital signature.

Returns KeyPair holding the public and private keys

```java
getPublicKey()
```
Returns the PublicKey of this signature scheme.

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

Throws
  • **IllegalStateException** – if no public key was set.

Returns the PublicKey

```java
isKeySet()
```
Checks if this digital signature object has been given a key already.

Returns true if the object has been given a key; false otherwise.

```java
setKey(PublicKey publicKey, PrivateKey privateKey)
```
Sets this digital signature with public key and private key.

Parameters
  • publicKey –
  • privateKey –

Throws
  • **InvalidKeyException** – if the given keys do not match this signature scheme.

```java
setKey(PublicKey publicKey)
```
Sets this digital signature with a public key.

In this case the signature object can be used only for verification.

Parameters
  • publicKey –

Throws
  • **InvalidKeyException** – if the given key does not match his signature scheme.

**RSA Based Digital Signature**

**The RSABasedSignature Interface**

```java
public interface RSABasedSignature extends DigitalSignature, UnlimitedTimes
```
General interface for RSA PSS signature scheme. Every concrete implementation of RSA PSS signature should implement this interface. The RSA PSS (Probabilistic Signature Scheme) is a provably secure way of creating signatures with RSA.
BouncyCastle Implementation

public class BcRSAPss extends RSAPssAbs
This class implements the RSA PSS signature scheme, using BC RSAPss implementation. The RSA PSS
(Probabilistic Signature Scheme) is a provably secure way of creating signatures with RSA.

public BcRSAPss ()
Default constructor. uses default implementations of CryptographicHash and SecureRandom.

public BcRSAPss (CryptographicHash hash, SecureRandom random)
Constructor that receives hash and secure random to use.

Parameters

• hash – underlying hash to use.
• random – secure random to use.

Throws

• FactoriesException – if there is no hash with the given name.

Crypto++ Implementation

public class CryptoPPRSAPss extends RSAPssAbs
This class implements the RSA PSS signature scheme, using Crypto++ RSAPss implementation. The RSA PSS
(Probabilistic Signature Scheme) is a provably secure way of creating signatures with RSA.

public CryptoPPRSAPss ()
Default constructor. uses default implementation of SecureRandom.

public CryptoPPRSAPss (SecureRandom random)
Constructor that receives the secure random object to use.

Parameters

• random – secure random to use

OpenSSL Implementation

public class OpenSSLRSAPss extends RSAPssAbs
This class implements the RSA PSS signature scheme, using OpenSSL RSAPss implementation. The RSA PSS
(Probabilistic Signature Scheme) is a provably secure way of creating signatures with RSA.

public OpenSSLRSAPss ()
Default constructor. uses default implementation of SecureRandom.

public OpenSSLRSAPss (SecureRandom random)
Constructor that receives the secure random object to use.

Parameters

• random – secure random to use

Example of Usage

Sender usage:
//Create an RSAPss signature object.
RSAPss signer = new BcRSAPss();

//Generate a keyPair using the RSAPss object.
KeyPair pair = signer.generateKey(new RSAKeyGenParameterSpec(1024, null));

//Generate a keyPair using the signer.
KeyPair pair = signer.generateKey();

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

//Set private key and party2’s public key:
signer.setKey(party2PublicKey, pair.getPrivate());

//Get a byte[] message to sign, and sign it.
Signature signature = signer.sign(msg, offset, length); //Send signature, msg and keys to the receiver

Receiver usage:
//Create the same RSAPss object as the sender’s object.
//Generate a keyPair using the signer object.
KeyPair pair = signer.generateKey();

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

//Set private key and party1’s public key:
signer.setKey(party1PublicKey, pair.getPrivate());

//Get the signature and message and verify it.
...
if (!signer.verify(signature, msg, offset, length)) {
    Throw new IllegalArgumentException("the message is not verified!");
}

//Message verified, continue working with it.
...

DSA Digital Signature

The DSABasedSignature Interface

public interface DSABasedSignature extends DigitalSignature, UnlimitedTimes
    General interface for DSA signature scheme. Every concrete implementation of DSA signature should implement this interface.

Scapi Implementation

public class ScDSA implements DSABasedSignature
    This class implements the DSA signature scheme.

public ScDSA ()
    Default constructor. uses default implementations of CryptographicHash, DlogGroup and SecureRandom.
public **ScDSA** (CryptographicHash *hash*, DlogGroup *dlog*, SecureRandom *random*)

Constructor that receives hash, dlog and secure random to use.

**Parameters**

- *hash* – underlying hash to use.
- *dlog* – underlying DlogGroup to use.
- *random* – secure random to use.

**OpenSSL Implementation**

public class **OpenSSLDSA** implements DSABasedSignature

This class implements the DSA signature scheme using OpenSSL library.

public **OpenSSLDSA** ()

Default constructor. uses default implementations of DlogGroup.

public **OpenSSLDSA** (DlogGroup *dlog*)

Constructor that receives a dlog to use.

**Parameters**

- *dlog* – underlying DlogGroup to use.

**Example of Usage**

Sender usage:

```java
//Create a DSA signature object.
DSA signer = new ScDSA(new MiraclDlogECFp());

//Generate a keyPair using the DSA object.
KeyPair pair = signer.generateKey();

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

//Set private key and party2’s public key:
signer.setKey(party2PublicKey, pair.getPrivate());

//Get a byte[] message to sign, and sign it.
Signature signature= signer.sign(msg, offset, length);

//Send signature, msg and keys to the receiver.
...
```

Receiver usage:

```java
//Create the same DSA object as the sender’s object.
//Generate a keyPair using the signer object.
KeyPair pair = signer.generateKey();

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

//Set private key and party1’s public key:
signer.setKey(party1PublicKey, pair.getPrivate());
```
//Get the signature and message and verify it.
...

if (!signer.verify(signature, msg, offset, length)) {
    throw new IllegalArgumentException("the message is not verified!");
}

//Message verified, continue working with it.
...

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.
Interfaces

Both Sender and Receiver interfaces 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.

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.

The input types are `OTSInput` and `OTRInput`. These are marker interfaces 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. While the sender’s transfer function returns void, the receiver’s transfer function returns `OTROutput`, which is a marker interface. Each concrete OT receiver should implement a dedicated output class that holds the necessary output objects.

The OTSender Interface

```java
public interface OTSender
public void transfer (Channel 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 batch OT by creating the OT sender once and call the transfer function for each input couple. In order to enable the 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.

Throws

- `ClassNotFoundException` – if there was a problem in the serialization mechanism.
- `IOException` – if there was a problem during the communication.
- `CheatAttemptException` – if the sender suspects that the receiver is trying to cheat.
- `InvalidDlogGroupException` – if the given DlogGRoup is not valid.

The OTReceiver Interface

```java
public interface OTReceiver
public OTROutput transfer (Channel 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 batch OT by creating the OT receiver once and call the transfer function for each input couple. In order to enable the 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.
Throws

- **CheatAttemptException** – if there was a cheat attempt during the execution of the protocol.
- **IOException** – if the send or receive functions failed
- **ClassNotFoundException** – if there was a problem during the serialization mechanism

Returns OTROutput, the output of the protocol.

The Input/Output Interfaces

**public interface OTSInput**

Every OT sender needs inputs during the protocol execution, but every concrete protocol needs different inputs. This interface is a marker interface for OT sender input, where there is an implementing class for each OT protocol.

**public interface OTRInput**

Every OT receiver needs inputs during the protocol execution, but every concrete protocol needs different inputs. This interface is a marker interface for OT receiver input, where there is an implementing class for each OT protocol.

**public interface OTROutput**

Every OT receiver outputs a result in the end of the protocol execution, but every concrete protocol output different data. This interface is a marker interface for OT receiver output, where there is an implementing class for each OT protocol.

Abstract classes

Each concrete OT protocol has abstract classes for both sender and receiver. Both classes implement common behavior of sender and receiver, accordingly. Each of the abstract classes implements the corresponding interface (sender/receiver).

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 OT implemented so far are:

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

```java
//Creates the OT sender object.
OTSemiHonestDDHOnGroupElementSender sender = new OTSemiHonestDDHOnGroupElementSender();

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

//call the transfer part of the OT protocol
try {
    sender.transfer(channel, input);
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (CheatAttemptException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (InvalidDlogGroupException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

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.

```java
//Creates the OT receiver object.
OTSemiHonestDDHOnGroupElementReceiver receiver = new OTSemiHonestDDHOnGroupElementReceiver();

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

OTROutput output = null;
try {
    output = receiver.transfer(channel, input);
} catch (CheatAttemptException e) {

```

1.7. Layer 3: Interactive Protocols
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 interfaces.
- Prover and Verifier abstract classes.
- ProverComputation and VerifierComputation classes (Specific to each protocol).

The Prover Interface

The SigmaProtocolProver interface 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.

public interface SigmaProtocolProver
General interface for Sigma Protocol prover. Every class that implements it is signed as Sigma Protocol prover. 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.

public void processFirstMsg (SigmaProverInput input)
Processes the first step of the sigma protocol. It computes the first message and sends it to the verifier.

Parameters
• input –

Throws
• IOException – if failed to send the message.

public void 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!

Throws
• CheatAttemptException – if the received challenge’s length is not equal to the soundness parameter.
• IOException – if there was a problem during the communication phase.
• ClassNotFoundException – if there was a problem during the serialization mechanism.

public void prove (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.

Parameters
• input –

Throws
• CheatAttemptException – if the received challenge’s length is not equal to the soundness parameter.
• IOException – if there was a problem during the communication phase.
• ClassNotFoundException – if there was a problem during the serialization mechanism.

The Verifier Interface

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.
public interface SigmaProtocolVerifier
    General interface for Sigma Protocol verifier. Every class that implements it is signed as Sigma Protocol verifier.

public byte[] getChallenge ()
    Returns the sampled challenge.

public boolean processVerify (SigmaCommonInput input)
    Waits to the prover’s second message and then verifies the proof. This is a blocking function!

    Parameters
    • input –

    Throws
    • ClassNotFoundException – if there was a problem during the serialization mechanism.
    • IOException – if there was a problem during the communication phase.

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

public void sampleChallenge ()
    Samples the challenge for this protocol.

public void sendChallenge ()
    Waits for the prover’s first message and then sends the chosen challenge to the prover. This is a blocking function!

    Throws
    • ClassNotFoundException – if there was a problem during the serialization mechanism.
    • IOException – if there was a problem during the communication phase.

public void setChallenge (byte[] challenge)
    Sets the given challenge.

    Parameters
    • challenge –

public boolean 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.

    Parameters
    • input –

    Throws
    • ClassNotFoundException – if there was a problem during the serialization mechanism.
    • IOException – if there was a problem during the communication phase.

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

The Simulator Interface

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.

public interface `SigmaSimulator`

General interface 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.

public int `getSoundnessParam`()

Returns the soundness parameter for this Sigma simulator.

**Returns** t soundness parameter

public `SigmaSimulatorOutput` `simulate` (`SigmaCommonInput` `input`, byte[] `challenge`)

Computes the simulator computation.

**Parameters**

- `input` –
- `challenge` –

**Throws**

- `CheatAttemptException` – if the received challenge’s length is not equal to the soundness parameter.

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

public `SigmaSimulatorOutput` `simulate` (`SigmaCommonInput` `input`)

Chooses random challenge and computes the simulator computation.

**Parameters**

- `input` –

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

**Computation classes**

The classes that operate the actual protocol phases implement the `SigmaProverComputation` and `SigmaVerifierComputation` interfaces. `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 interfaces.

**SigmaProverComputation**

public interface `SigmaProverComputation`

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

public `SigmaProtocolMsg` `computeFirstMsg` (`SigmaProverInput` `input`)

Computes the first message of the sigma protocol.

**Parameters**

- `input` –

public SigmaProtocolMsg computeSecondMsg (byte[] challenge)
  Computes the second message of the sigma protocol.

  **Throws**
  
  • **CheatAttemptException** – if the received challenge’s length is not equal to the soundness parameter.

**SigmaVerifierComputation**

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

public void sampleChallenge ()
  Samples the challenge for this protocol.

public void setChallenge (byte[] challenge)
  Sets the given challenge.

  **Parameters**
  
  • **challenge** –

public byte[] getChallenge ()
  Returns the sampled challenge.

  **Returns** the challenge.

public boolean verify (SigmaCommonInput input, SigmaProtocolMsg α, SigmaProtocolMsg τ)
  Verifies the proof.

  **Parameters**
  
  • **input** –

  **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
• Damgad-Jurik encrypted value
• Damgad-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:

```java
//Creates the dlog group.
DlogGroup dlog = null;
try {
    //use the koblitz curve.
    dlog = new MiraclDlogECF2m("K-233");
} catch (FactoriesException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}

//Creates sigma prover computation.
SigmaProverComputation proverComputation = new SigmaDlogProverComputation(dlog, t, new SecureRandom());

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

//Creates input for the prover.
SigmaProverInput input = new 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:

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

//Creates input for the prover.
SigmaProverInput input = new SigmaDlogProverInput(h, w);

//Calls the prove function of the prover.
prover.prove(input);
```
//Creates the dlog group
DlogGroup dlog = null;
try {
    //use the koblitz curve
    dlog = new MiraclDlogECF2m("K-233");
} catch (FactoriesException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}

//Creates sigma verifier computation.
SigmaVerifierComputation verifierComputation = new SigmaDlogVerifierComputation(dlog, t, new SecureRandom());

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

// Creates input for the verifier.
SigmaCommonInput input = new 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.

#### Zero Knowledge Interfaces

**ZKProver**

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

```java
public interface ZKProver {

    void prove(ZKProverInput input);
}
```

---

**Zero Knowledge Proofs and Zero Knowledge Proofs of Knowledge**

### 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.

#### Zero Knowledge Interfaces

**ZKProver**

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

```java
public interface ZKProver {

    void prove(ZKProverInput input);
}
```
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 interface is a general interface that simulates the prover side of the Zero Knowledge proof. Every class that implements it is signed as Zero Knowledge prover.

public void prove (ZKProverInput input)
   Runs the prover side of the Zero Knowledge proof.

   Parameters
   • input – holds necessary values to the proof calculations.

   Throws
   • CheatAttemptException – if the prover suspects the verifier is trying to cheat.
   • IOException – if there was a problem during the communication.
   • ClassNotFoundException – if there was a problem during the serialization mechanism.
   • CommitValueException – can occur when using ElGamal commitment scheme.

ZKVerifier

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

public interface 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 interface is a general interface that simulates the verifier side of the Zero Knowledge proof. Every class that implements it is signed as Zero Knowledge verifier.

public boolean verify (ZKCommonInput input)
    Runs the verifier side of the Zero Knowledge proof.

    Parameters
    • input – holds necessary values to the.varification calculations.

    Throws
    • CheatAttemptException – if the prover suspects the verifier is trying to cheat.
    • IOException – if there was a problem during the communication.
    • ClassNotFoundException – if there was a problem during the serialization mechanism.
    • CommitValueException – can occur when using ElGamal commitment scheme.

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

ZKProverInput

public interface ZKProverInput
    Marker interface. Each concrete ZK prover’s input class should implement this interface.
ZKCommonInput

public interface ZKCommonInput

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

Zero Knowledge Proof of Knowledge Interfaces

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

public interface ZKPOKProver extends ZKProver

This interface is a general interface that simulates the prover side of the Zero Knowledge proof of knowledge. Every class that implements it is signed as ZKPOK prover.

public interface ZKPOKVerifier extends ZKVerifier

This interface is a general interface that simulates the verifier side of the Zero Knowledge proof of knowledge. Every class that implements 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:

```
try {
    //create the ZK prover
    DlogGroup dlog = new MiraclDlogECF2m("K-233");
    ZKProver prover = new ZKFromSigmaProver(channel, new SigmaDlogProverComputation(dlog, 40, new SecureRandom()));

    //create the input for the prover
    SigmaDlogProverInput input = new 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:

```java
try {
    //create the ZK verifier
    DlogGroup dlog = new MiraclDlogECF2m("K-233");
    ZKVerifier verifier = new ZKFromSigmaVerifier(channel, new SigmaDlogVerifierComputation(dlog, 40,
    new SecureRandom()),
    new SecureRandom());

    //create the input for the verifier
    SigmaDlogCommonInput input = new SigmaDlogCommonInput(h);

    //Call verify function
    System.out.println(verifier.verify(input));

    // TODO auto-generated catch block
    catch (IllegalArgumentException e) {
        e.printStackTrace();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    catch (CheatAttemptException e) {
        e.printStackTrace();
    }
    catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    catch (CommitValueException e) {
        e.printStackTrace();
    } // TODO auto-generated catch block

} catch (IllegalArgumentException e) {
    // TODO auto-generated catch block
    e.printStackTrace();
} catch (IOException e) {
    // TODO auto-generated catch block
    e.printStackTrace();
} catch (CheatAttemptException e) {
    // TODO auto-generated catch block
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    // TODO auto-generated catch block
    e.printStackTrace();
} catch (CommitValueException e) {
    // TODO auto-generated catch block
    e.printStackTrace();
} catch (InvalidDlogGroupException e) {
    // TODO auto-generated catch block
    e.printStackTrace();
}

1.7. Layer 3: Interactive Protocols
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.

The Committer Interface

public interface \texttt{CmtCommitter}

This the general interface 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

public void \texttt{commit} (CmtCommitValue \texttt{input}, long \texttt{id})

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

Parameters

- \texttt{input} – The value that the committer commits about.
- \texttt{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 decommitting them yet.

Throws

- \texttt{IOException} – if there is any problem at the communication level

public void \texttt{decommit} (long \texttt{id})

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

Parameters

- \texttt{id} – Unique value used to identify which previously committed value needs to be decommitted now.
Throws

- **CheatAttemptException** – if the Committer suspects that the receiver attempted cheating
- **IOException** – if there is any problem at the communication level
- **ClassNotFoundException** – if the commitment cannot be serialized
- **CommitValueException** – if the commit value does not match the implementing commitment.

Conversion to and from CmtCommitValue

```java
public byte[] generateBytesFromCommitValue(CmtCommitValue value)
```
This function converts the given commit value to a byte array.

**Parameters**

- **value** – to get its bytes.

**Returns** the generated bytes.

```java
public CmtCommitValue generateCommitValue(byte[] x)
```
This function wraps the raw data x with a suitable CommitValue instance according to the actual implementation.

**Parameters**

- **x** – array to convert into a CommitValue.

**Throws**

- **CommitValueException** – if the commit value does not match the implementing commitment

**Returns** the created CommitValue.

Inner state functions

```java
public CmtCommitmentPhaseValues 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

```java
public Object[] 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

```java
public CmtCommitValue sampleRandomCommitValue()
```
This function samples random commit value to commit on.

**Returns** the sampled commit value.
The Receiver Interface

public interface CmtReceiver

This the general interface 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

public CmtRCommitPhaseOutput receiveCommitment ()

This function is the heart of the commitment phase from the Receiver’s point of view.

Throws
- ClassNotFoundException – if the commitment received cannot be deserialized
- IOException – if there is any problem at the communication level

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

public CmtCommitValue 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

Throws
- CheatAttemptException – if there is an error that could have been caused by a cheating attempt
- ClassNotFoundException – if the decommitment received cannot be deserialized
- IOException – if there is any problem at the communication level.
- CommitValueException – if the commit value does not match the implementing commitment.

Returns the commitment

Conversion to and from CmtCommitValue

public byte[] 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

public Object 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.

```java
public Object[] 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

### Implementations in Scapi

Each concrete commitment protocol should have committer and receiver classes that implement the `CmtCommitter` and `CmtReceiver` interfaces 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:

```java
try {
    //create the committer
    DlogGroup dlog = new MiraclDlogECF2m("K-233");
    CmtCommitter committer = new CmtPedersenCommitter(ch, dlog, new SecureRandom());

    //generate CommitValue from string
    CmtCommitValue val = committer.generateCommitValue(new String("commit this string!").getBytes());

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

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

1.7. Layer 3: Interactive Protocols
catch (ClassNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (CheatAttemptException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (CommitValueException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

Steps in receiver creation:

• Given a Channel object ch do:
  – Create a CmtReceiver (for example, :java:ref:`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:

```
try {
    //create the receiver
dlog = new MiraclDlogECF2m("K-233");
    CmtReceiver receiver = new CmtPedersenReceiver(ch, dlog, new SecureRandom());

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

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

    //Convert the commitValue to bytes.
    String committedString = new String(receiver.generateBytesFromCommitValue(val));

    System.out.println(committedString);
} catch (IllegalArgumentException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (SecurityLevelException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (InvalidDlogGroupException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```
Coin Tossing

The basic case of coin tossing is the practice of throwing a coin in the air to choose between two alternatives, sometimes to resolve a dispute between two parties. The “coin” in our implementation can be various data types, like a bit or a byte array. Each implementation achieves different security levels.

The general structure of Coin Tossing protocols contains two levels:

- PartyOne and PartyTwo interfaces
- PartyOne and PartyTwo concrete classes for each coin tossing implementation.

Contents

- Coin Tossing
  - Coin Tossing Interfaces
    * CTPartyOne
    * CTPartyTwo
    * CTOutput
  - Implementations in Scapi

Coin Tossing Interfaces

The only function in the coin tossing interfaces (both party one, and party two) is the toss() function. This function executes the coin tossing protocol and returns a CTOutput object. CTOutput is a marker interface for the tossed “coin”. For each concrete coin type we should implement a dedicated class which will be returned. For example, Blum protocol tosses a single bit, therefore, CTBlumPartyOne and CTBlumPartyTwo return CTBitOutput object as the output of the toss function.

CTPartyOne

public interface CTPartyOne

Coin tossing is the practice of throwing a coin in the air to choose between two alternatives, sometimes to resolve a dispute between two parties. This is a general interface plays as party one of a coin tossing protocol. Each concrete party one class of a coin tossing protocol should implement this interface.

public CTOutput toss ()

Executes party one role of this coin tossing protocol.

Throws

- CheatAttemptException – if party one suspects that party two is trying to cheat.
- ClassNotFoundException – if there was a problem in the serialization mechanism
CTPartyTwo

public interface CTPartyTwo

Coin tossing is the practice of throwing a coin in the air to choose between two alternatives, sometimes to resolve a dispute between two parties. This is a general interface plays as party two of a coin tossing protocol. Each concrete party two class of a coin tossing protocol should implement this interface.

public CTOutput toss ()

Executes party two role of this coin tossing protocol.

Throws

• CheatAttemptException – if party one suspects that party two is trying to cheat.
• ClassNotfoundException – if there was a problem in the serialization mechanism
• IOException – can occur in the commit phase.
• CommitValueException – can occur in case the protocol uses an ElGamal receiver.

Returns CTOutput contains the tossed “coin”.

CTOutput

public interface CTOutput

Each coin tossing protocol outputs different “coin”. It can be a single bit, a string, etc. Each concrete output class should implement this interface.

public Object getOutput ()

Returns the output of the coin tossing protocol. The tossed value of the Coin Tossing protocol can vary. Returns Object instance to enable any return value.

Returns the tossed output.

Implementations in Scapi

Concrete Coin Tossing protocols implemented so far are:

• Coin Tossing of a single bit (Blum)
• Coin Tossing of a String
• Semi-Simulatable Coin-Tossing of a String

SCAPI in C++ (Beta)

Until recently SCAPI was a java-only library that in some cases wrapped native elements using JNI. SCAPI is now supported and maintained in C++ version as well. Roughly speaking, Java-SCAPI is great for fast development of application and for POCing cryptographic protocols. However, when performance is a main concern, a c++ implementation might be more suitable.
This is still “work in progress” and should be considered as beta. There are few modules that exist in the java version but were still not ported to the C++ version. The SCAPI team is working these days on expending the c++ implementation further.

## Installing SCAPI - Linux

The following explains how to install libscapi (SCAPI c++) 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_60_0.tar.bz2 http://sourceforge.net/projects/boost/files/boost/1.60.0/boost_1_60_0.tar.bz2
$ tar --bzip2 -xf boost_1_60_0.tar.bz2
$ cd boost_1_60_0
$ ./bootstrap.sh
$ ./b2
```

More details about boost here: [http://www.boost.org/doc/libs/1_60_0/more/getting_started/unix-variants.html](http://www.boost.org/doc/libs/1_60_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-honest 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 SigmaProtocols/SigmaConfig.txt

And in the second terminal:
$ ./libscapi_example.exe sigma 2 SigmaProtocols/SigmaConfig.txt

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

Installing SCAPI - 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
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\ScGarbledCircuitWin64
   (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 exaplined in the linux section
   (e) You can edit the different config file to play with the parameters

**Further Reading**

For further reading - refer to the extensive Java documentation. The c++ implementation usually follows the same concept except when language specific need required some change.

**License**

Copyright (c) 2012 - SCAPI.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

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 http://crypto.biu.ac.il/scapi.

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
Index

A
ActiveMQCommunicationSetup(String, PartyData, PartyData) (Java method), 18
ActiveMQMultipartyCommunicationSetup (Java class), 21
ActiveMQMultipartyCommunicationSetup(String, List) (Java method), 21
AuthenticatedChannel (Java class), 26
AuthenticatedChannel(Channel, Mac) (Java constructor), 26
AuthenticatedEnc (Java interface), 62

B
BeRSAOaep (Java class), 76
BeRSAOaep() (Java constructor), 76
BeRSAOaep(SecureRandom) (Java constructor), 76
BeRSAPss (Java class), 81
BeRSAPss() (Java constructor), 81
BeRSAPss(CryptographicHash, SecureRandom) (Java constructor), 81

C
CBCEnc (Java interface), 61
CbcMac (Java interface), 57
Channel (Java interface), 25
CliqueSuccess (Java class), 32
close() (Java method), 25
CmtCommitter (Java interface), 98
CmtReceiver (Java interface), 100
commit(CmtCommitValue, long) (Java method), 98
CommunicationSetup (Java class), 31
completeEncryption(GroupElement, GroupElement, Plaintext) (Java method), 70
compute(byte[], int, int, byte[], int) (Java method), 39
compute DOJEB [byte[], int, int, byte[], int, int] (Java method), 39
computeFirstMsg(SigmaProverInput) (Java method), 91
computeSecondMsg(byte[]) (Java method), 91
ConnectivitySuccessVerifier (Java interface), 32
CramerShoupDDHEnc (Java interface), 72
createRandomElement() (Java method), 50
decommit(long) (Java method), 98
decrypt(AsymmetricCiphertext) (Java method), 65
decrypt(SymmetricCiphertext) (Java method), 60
deriveKey(byte[], int, int, int) (Java method), 54
deriveKey(byte[], int, int, int, byte[]) (Java method), 54
DigitalSignature (Java interface), 79
doFinal(byte[], int, int) (Java method), 56
DSABasedSignature (Java interface), 82

D
DamgardJurikEnc (Java interface), 75
decodeGroupElementToByteArray(GroupElement) (Java method), 52
decommit(long) (Java method), 98
decrypt(AsymmetricCiphertext) (Java method), 65
decrypt(SymmetricCiphertext) (Java method), 60
deriveKey(byte[], int, int, int) (Java method), 54
deriveKey(byte[], int, int, int, byte[]) (Java method), 54
DigitalSignature (Java interface), 79
doFinal(byte[], int, int) (Java method), 56
DSABasedSignature (Java interface), 82

E
ElGamalEnc (Java interface), 69
encrypt(Plain, byte[]) (Java method), 52
e encrypt(Plaintext, byte[]) (Java method), 59, 65
e encrypt(Plaintext, byte[]) (Java method), 60
EncryptedChannel (Java class), 27
PlainTCPChannel (Java class), 26
PlainTCPSocketChannel (Java class), 26
prepareForCommunication(int, long) (Java method), 18
prepareForCommunication(List, ConnectivitySuccessVerifier, long, boolean) (Java method), 31
prepareForCommunication(List, long) (Java method), 8
prepareForCommunication(Map, long) (Java method), 22
prepareForCommunication(String[], long) (Java method), 18
processFirstMsg(SigmaProverInput) (Java method), 89
processSecondMsg() (Java method), 89
processVerify(SigmaCommonInput) (Java method), 90
prove(SigmaProverInput) (Java method), 89
prove(ZKProverInput) (Java method), 95
QueueChannel (Java class), 26
QueueCommunicationSetup (Java class), 16
QueueCommunicationSetup(ConnectionFactory, DestroyDestinationUtil, PartyData, PartyData) (Java method), 17
receive() (Java method), 9, 25
receiveCommitment() (Java method), 100
receiveDecommitment(long) (Java method), 100
reconstructCiphertext(AsymmetricCiphertextSendableData) (Java method), 67
reconstructElement( boolean, GroupElementSendableData) (Java method), 67
reconstructPrivateKey(KeySendableData) (Java method), 67
reconstructPublicKey(KeySendableData) (Java method), 67
reconstructTPElement(TPElementSendableData) (Java method), 45
reRandomize(AsymmetricCiphertext) (Java method), 75
RSABasedSignature (Java interface), 80
RSAOaepEnc (Java interface), 76
ScCramerShoupDDHOnGroupElement (Java class), 73
ScCramerShoupDDHOnGroupElement() (Java constructor), 73
ScCramerShoupDDHOnGroupElement(DlogGroup, CryptographicHash) (Java constructor), 73
ScCramerShoupDDHOnGroupElement(DlogGroup, CryptographicHash, SecureRandom) (Java constructor), 73
ScDamgardJurikEnc (Java class), 75
ScDamgardJurikEnc() (Java constructor), 75
ScDamgardJurikEnc(SecureRandom) (Java constructor), 75
ScDSAPrivateKey (Java interface), 82
ScDSAPublicKey (Java interface), 82
ScElGamalOnByteArray (Java class), 69
ScElGamalOnByteArray(DlogGroup) (Java constructor), 69
ScElGamalOnByteArray(DlogGroup, KeyDerivationFunction) (Java constructor), 69
ScElGamalOnByteArray(DlogGroup, KeyDerivationFunction, SecureRandom) (Java constructor), 69
ScElGamalOnGroupElement (Java class), 70
ScElGamalOnGroupElement() (Java constructor), 70
ScElGamalOnGroupElement(DlogGroup) (Java constructor), 70
ScElGamalOnGroupElement(DlogGroup, SecureRandom) (Java constructor), 70
ScEncryptThenMac (Java class), 62
SecureCliqueSuccess (Java class), 33
send(Serializable) (Java method), 9, 25
sendChallenge() (Java method), 90
setChallenge(byte[]) (Java method), 90, 92
setKey(PublicKey) (Java method), 46, 67, 80
setKey(PublicKey, PrivateKey) (Java method), 46, 66, 80
setKey(SecretKey) (Java method), 26, 28, 37, 40, 56, 61
SigmaProtocolProver (Java interface), 89
SigmaProtocolVerifier (Java interface), 90
SigmaProverComputation (Java interface), 91
SigmaSimulator (Java interface), 91
SigmaVerifierComputation (Java interface), 92
sign(byte[], int, int) (Java method), 79
simulate(SigmaCommonInput) (Java method), 91
simulate(SigmaCommonInput, byte[]) (Java method), 91
simultaneousMultipleExponentiations(GroupElement[], BigInteger[]) (Java method), 50
SocketCommunicationSetup (Java class), 16
SocketCommunicationSetup(PartyData, PartyData) (Java method), 17
SocketMultipartyCommunicationSetup (Java class), 21
SocketMultipartyCommunicationSetup(List) (Java method), 21
SampleChallenge() (Java method), 90, 92
SampleRandomCommitValue() (Java method), 99
ScCramerShoupDDHOnByteArray (Java class), 72
ScCramerShoupDDHOnByteArray() (Java constructor), 72
ScCramerShoupDDHOnByteArray(DlogGroup, CryptographicHash, KeyDerivationFunction) (Java constructor), 72
ScCramerShoupDDHOnByteArray(DlogGroup, CryptographicHash, SecureRandom) (Java constructor), 72
Index 111
SSLActiveMQCommunicationSetup(String, PartyData, PartyData, String) (Java method), 18
SSLActiveMQCommunicationSetup(String, PartyData, PartyData, String, String, String) (Java method), 18
SSLActiveMQMultipartyCommunicationSetup (Java class), 21
SSLActiveMQMultipartyCommunicationSetup(String, List, String) (Java method), 21
SSLActiveMQMultipartyCommunicationSetup(String, List, String, String, String) (Java method), 22
SSLSocketCommunicationSetup (Java class), 16
SSLSocketCommunicationSetup(PartyData, PartyData, String) (Java method), 17
SSLSocketCommunicationSetup(PartyData, PartyData, String, String, String) (Java method), 17
SSLSocketMultipartyCommunicationSetup (Java class), 21
SSLSocketMultipartyCommunicationSetup(List, String) (Java method), 21
SSLSocketMultipartyCommunicationSetup(List, String, String, String) (Java method), 21
startMac(int) (Java method), 57
SymmetricEnc (Java interface), 59
T
toss() (Java method), 103, 104
TPEIValidity (Java enum), 45
transfer(Channel, OTRInput) (Java method), 85
transfer(Channel, OTSInput) (Java method), 85
U
update(byte[], int, int) (Java method), 35, 55
V
validateGroup() (Java method), 51
verify(byte[], int, int, byte[]) (Java method), 55
verify(SigmaCommonInput) (Java method), 90
verify(SigmaCommonInput, SigmaProtocolMsg, SigmaProtocolMsg) (Java method), 92
verify(Signature, byte[], int, int) (Java method), 79
verify(ZKCommonInput) (Java method), 95
Z
ZKCommonInput (Java interface), 96
ZKPOKProver (Java interface), 96
ZKPOKVerifier (Java interface), 96
ZKProver (Java interface), 94
ZKProverInput (Java interface), 95
ZKVerifier (Java interface), 95