­­­­SCAPI Interactive Mid Crypto Protocols Layer

R&D Group

User Manual

Written by Moriya Farbstein

Created 17 October 2013

Modified 17 October 2013

PUBLIC

Change Control Information

|  |  |  |  |
| --- | --- | --- | --- |
| Version No. | Date | Changed by | Scope of Change |
| 1.00 | 17-Oct-2013 |  | Initial version |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1. Scope 4](#_Toc379701297)

[1.1. Introduction 4](#_Toc379701298)

[1.2. Purpose 4](#_Toc379701299)

[2. Definitions Acronyms and Abbreviations 4](#_Toc379701300)

[3. References 4](#_Toc379701301)

[4. General description 5](#_Toc379701302)

[4.1. General Structure 5](#_Toc379701303)

[4.2. Communication between parties 5](#_Toc379701304)

[4.3. Input/Output 5](#_Toc379701305)

[5. Main Components 6](#_Toc379701306)

[5.1. Oblivious transfer – Static view 6](#_Toc379701307)

[5.2. Oblivious Transfer – Dynamic View 7](#_Toc379701308)

[5.2.1 Interfaces 7](#_Toc379701309)

[5.2.2 Abstract classes 8](#_Toc379701310)

[5.2.3 Concrete implementations 8](#_Toc379701311)

[5.3. Sigma Protocol – Static view 9](#_Toc379701312)

[5.4. Sigma Protocol – Dynamic View 10](#_Toc379701313)

[5.4.1 Interfaces 11](#_Toc379701314)

[5.4.2 Concrete classes 11](#_Toc379701315)

[5.4.3 Computation classes 11](#_Toc379701316)

[5.5. Zero Knowledge – Static view 13](#_Toc379701317)

[5.6. Zero Knowledge – Dynamic View 14](#_Toc379701318)

[5.6.1 Interfaces 14](#_Toc379701319)

[5.6.2 Concrete classes 15](#_Toc379701320)

[5.7. Commitment – Static view 16](#_Toc379701321)

[5.8. Commitment – Dynamic View 17](#_Toc379701322)

[5.8.1 Interfaces 17](#_Toc379701323)

[5.8.2 Concrete classes 19](#_Toc379701324)

[5.9. Coin Tossing – Static View 20](#_Toc379701325)

[5.10. Coin Tossing – Dynamic View 20](#_Toc379701326)

[5.10.1 Interfaces 20](#_Toc379701327)

[5.10.2 Concrete classes 20](#_Toc379701328)

[6. Examples 21](#_Toc379701329)

# Scope

The purpose of this manual is to explain the main components of what we call the interactive mid crypto protocols layer**.**

## Introduction

The Interactive Mid Crypto Protocol layer contains interactive protocols which can be used as a standalone protocols or as building blocks of higher cryptographic schemes. This layer contains the following components:

* Oblivious Transfer
* Sigma Protocol
* Zero Knowledge and Zero Knowledge Proof of Knowledge
* Commitment
* Coin Tossing

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.

## **Purpose**

The purpose of this layer is to provide low level interactive protocols, that can be used in higher cryptographic protocols implementations or a standalone protocol. It aims to be as simple and clear as possible, and to enable general usage if needed.

# Definitions Acronyms and Abbreviations

|  |  |
| --- | --- |
| OT | Oblivious Transfer |
| ZK | Zero Knowledge |
| ZKPOK | Zero Knowledge Proof of Knowledge |
| CT | Coin Tossing |

# References

[1] The pseudo codes for all the protocols of this layer can be found at <http://crypto.biu.ac.il/scapi/SDK_Pseudocode_SCAPI_V2.0.0.pdf>.

# General description

## General Structure

As mentioned above, the interactive Mid Crypto Protocols layer contains various two party protocols, when each party has a specific role and can have different set of actions.

In order to be as clear as possible and to make this separation easy to understand, we chose not to put them together in one class but to have different classes for each protocol's party.

Generally, each protocol family (i.e. OT, ZK, etc) will have an interface for each party. These interfaces declare the main functions of each role. Each concrete implementation should have classes that implement these interfaces.

For example, an OT protocol has OTSender and OTReceiver interfaces, each one contains the functionalities regarding its role. OTSemiHonest, which is a concrete protocol implementation of OT, has OTSemiHonestSender that implements OTSender, and OTSemiHonestReceiver that implements OTReceiver.

In most of the cases, the concrete implementations require some underlying objects to use during the protocol execution. The underlying objects can vary between different protocols, even in the same family. (For example, OTSemiHonest requires different underlying objects from OTOneSidedSimulation).

These objects should be given in the constructor of each concrete class.

## Communication between parties

The communication between the parties is done via SCAPI's communication layer. Each protocol party gets in the constructor a **connected** channel to use in its execution.

When a party wishes to send the other party a message, he should use a serializable object that wraps the message. Then, he sends this object using the connected channel.

## Input/Output

As mentioned above, there are general interfaces for each protocol party. These interfaces declare the main functions of the party, including functions that get an input and an output. These functions use high level marker interfaces of input and output, because input and output objects can vary among concrete implementations (that is, one implementation of OT protocol uses ByteArray input and a second one uses DlogGroup element input).

Each concrete protocol implements the input and output classes that are related to it.

For example, SigmaProverComputation interface contains the function:

**public** SigmaProtocolMsg computeFirstMsg(SigmaProverInput input)

SigmaProverInput is a general interface for prover's input, while the concrete implementation SigmaDlogProverComputation should accept an instance of SigmaDlogProverInput.

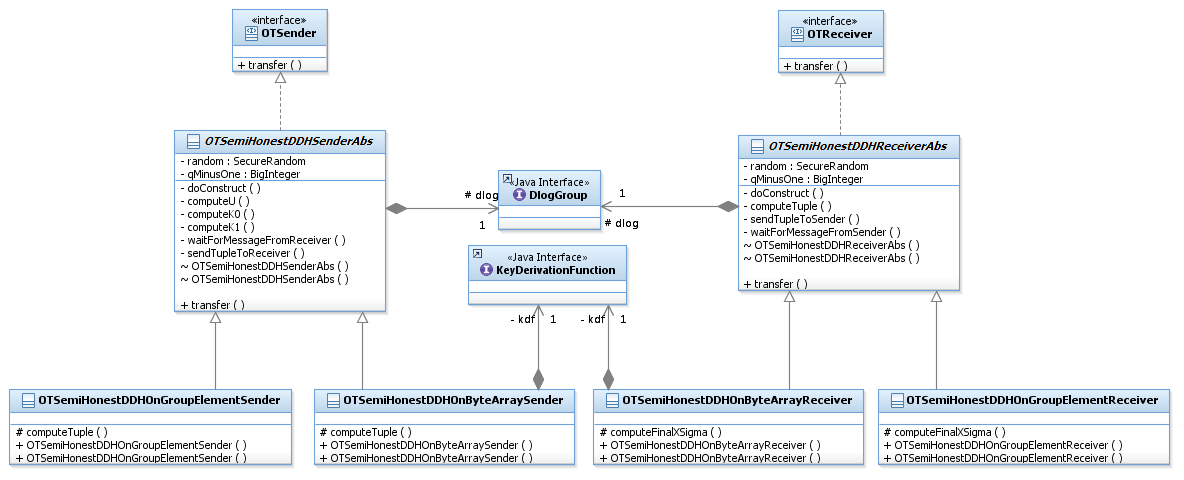
Thus, the constructor of SigmaDlogProverComputation checks that the given input is indeed an instance of SigmaDlogProverInput.

# Main Components

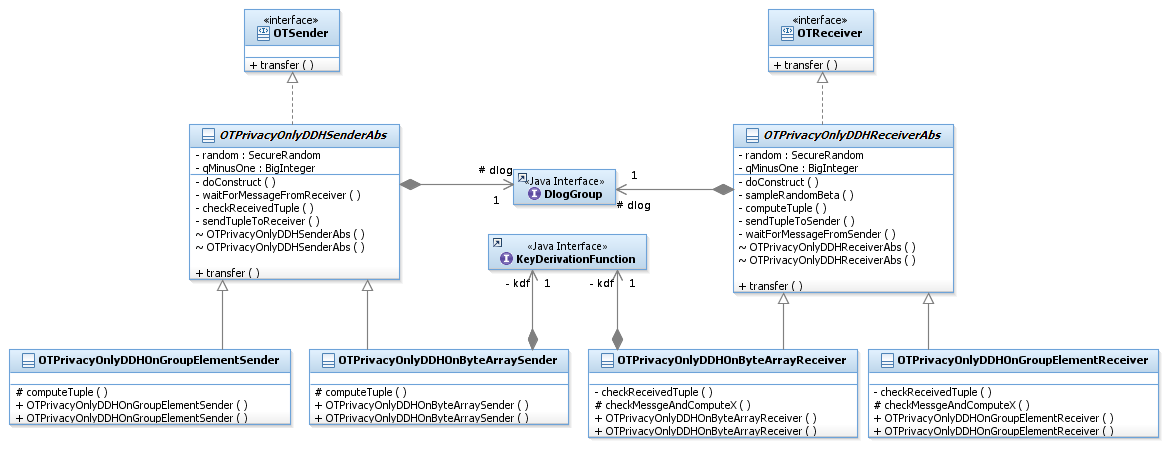
## Oblivious transfer – Static view

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.

The diagrams below show the classes hierarchy of some of OT protocols. Note that all concrete OT protocols have the same structure.



OT Semi Honest



OT Privacy Only

## Oblivious Transfer – Dynamic View

The general structure of OT protocols contains three components:

* Sender and Receiver interfaces
* Sender and receiver abstract classes
* Sender and receiver concrete classes

### 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.

In some cases, the OT protocol should execute a pre-process phase before the transfer. Usually, pre-process is done once at the beginning of the protocol and will not be executed later. The transfer function can be called many times.

In our implementation, the pre-process phase is done during the **construction time**. A protocol that needs to call the pre-process phase after the construction must create a new instance.

### 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.

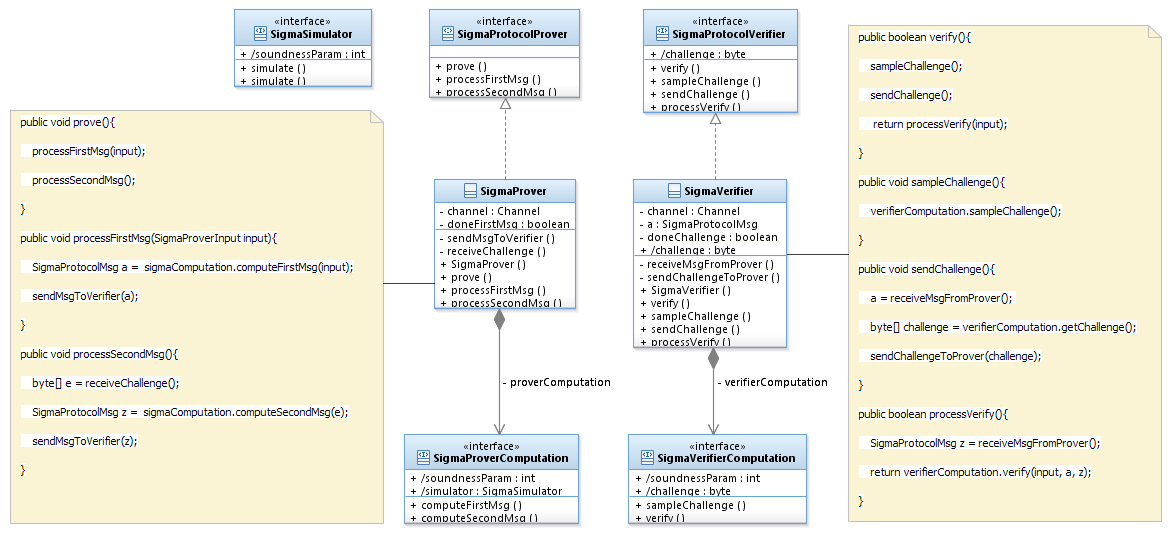
Concrete OT implemented so far are:

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

## Sigma Protocol – Static view

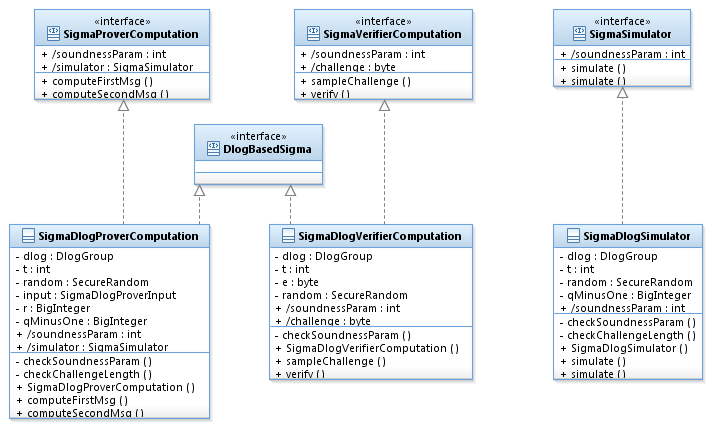
Sigma protocols are a basic building block for zero-knowledge proofs, zero-knowledge proofs of knowledge and more. A sigma protocol is a 3-round proof, comprised of a first message from the prover to the verifier, a random challenge from the verifier and a second message from the prover.

The diagram below shows the general relationships between Sigma Protocol components.



Each concrete Sigma protocol has dedicated computation classes, as will be explained later.

An example of those classes is shown below:



Sigma Dlog Protocol

## Sigma Protocol – Dynamic View

As mentioned above, 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 interfaces.
* Prover and verifier concrete classes.
* Prover and verifier computation classes.

### Interfaces

The **prover interface** has three functions: processFirstMessage, processSecondMessage and prove. This provides 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.

The **verifier interface** has four functions: sampleChallenge(), sendChallenge(), processVerify() and verify(). Similar to the sender interface, this provides 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.

The **simulator interface** 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.

### Concrete classes

There are two concrete classes that implement the above prover and verifier interfaces,

As shown in the above diagram. They both use SigmaComputation classes that actually implement the protocol phases and add the communication between the parties.

This way the sigma protocol can run as a standalone protocol.

For example, Sigma Prover has a function processFirstMessage. This function calls sigmaProverComputation.computaFirstMessage() and then sends the returned message to the verifier. See diagram above.

In addition, each concrete protocol should have a concrete simulator class that implements the simulator interface.

### 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 SigmaProver/VerifierComputation and will call the required function.

Each concrete sigma protocol should implement the computation interfaces.

Sigma Prover Computation has two main functions:

* ComputeFirstMessage that accepts the input for this prover, computes the first prover's message and returns it.
* ComputeSecondMessage that accepts the verifier's challenge, computes the second prover's message and returns it.

Sigma Verifier Computation has three main functions:

* SampleChallenge that samples the challenge.
* SendChallenge that accepts a challenge and sets it.

This function is called in cases the challenge is given from outside, like in Sigma AND protocol (where there is one challenge that needs to be set to all sub Sigma protocols).

* Verify function that accepts the input and two prover messages and computes the protocol's verification.

Concrete Sigma protocols implemented so far are:

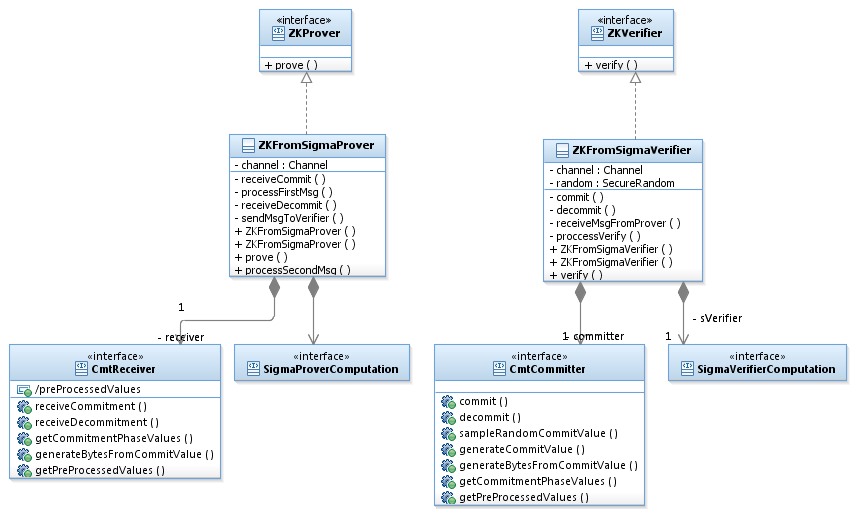
* Dlog
* DH
* Extended DH
* Pedersen commitment knowledge
* Pedersen committed value
* El Gamal commitment knowledge
* El Gamal committed value
* El Gamal private key
* El Gamal encrypted value
* Cramer-Shoup encrypted value
* Damgard-Jurik encrypted zero
* Damgard-Jurik encrypted value
* Damgard-Jurik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

## Zero Knowledge – Static view

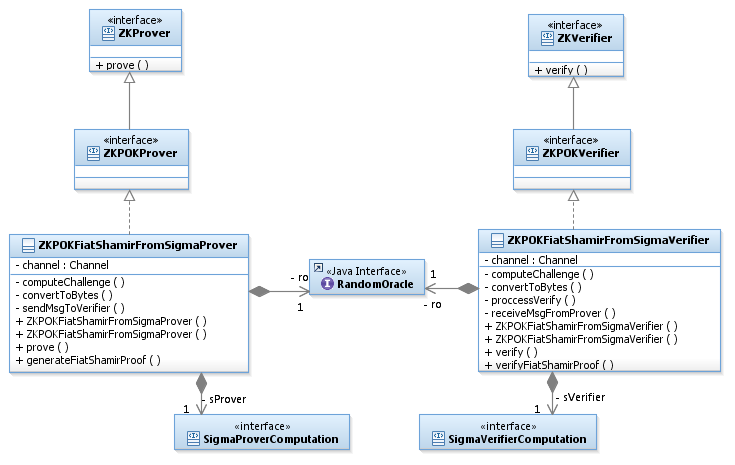
A zero-knowledge proof or a zero-knowledge protocol is a method by which one party (the prover) can prove to another party (the verifier) that a given statement is true, without conveying any additional information apart from the fact that the statement is indeed true.

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.

The diagrams below show the classes hierarchy of Zero Knowledge protocols.



Zero Knowledge from any Sigma Protocol



Zero Knowledge Proof of Knowledge Fiat Shamir

## Zero Knowledge – Dynamic View

The general structure of Zero Knowledge contains the following components:

* Prover and verifier interfaces.
* Prover and verifier concrete classes.

### Interfaces

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.

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.

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.

### Concrete classes

Each concrete ZK or ZKPOK protocol should implement the above interfaces.

A ZK proof is usually built on a Sigma Protocol with additional computations in order to obtain the zero knowledge property (such as commitments).

For this reason, ZKProver can hold instances of SigmaProverComputation, CommitmentScheme, RandomOracle, and more. Similarly, ZKVerifier can hold instances of SigmaVerifierComputation, CommitmentScheme, RandomOracle, and more.

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 (ROM)

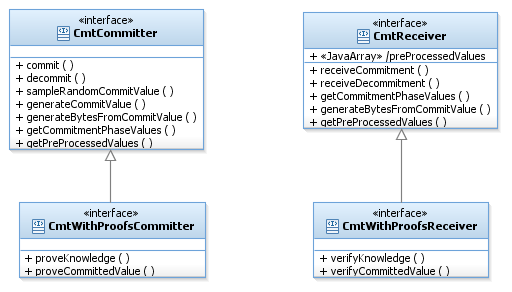
## Commitment – Static view

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.

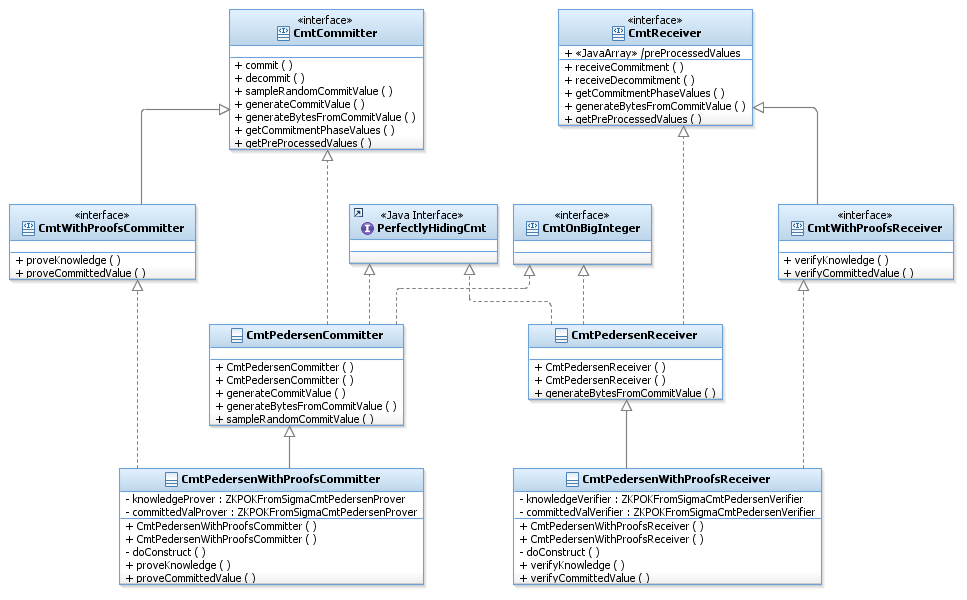
There exist some commitment schemes that can be proven by ZK protocols.

The diagram below shows the main interfaces of the commitment package.

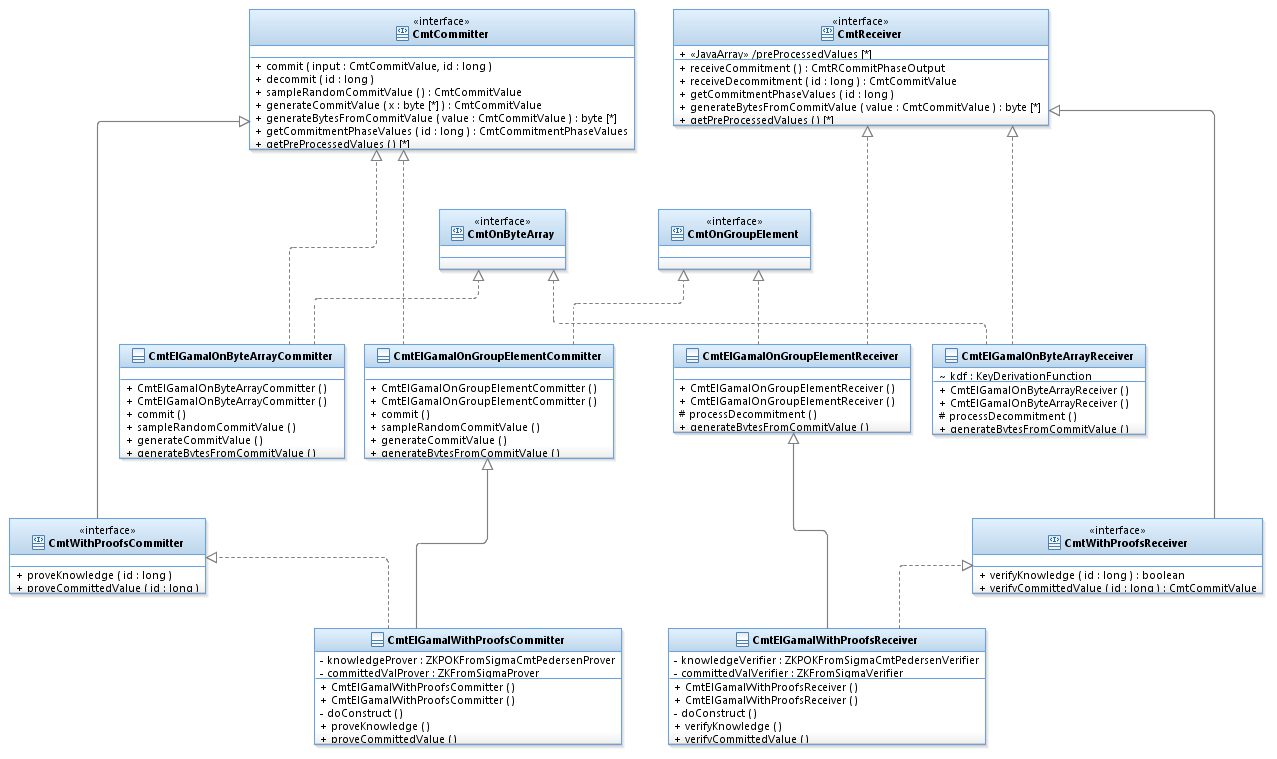
One can see that there are general “committer” and “receiver” interfaces along with the interfaces that add the proofs on the commitment schemes.



For example, the diagram below shows the case of the Pedersen commitment scheme, which can be proven by a ZK proof. So, in addition to the CmtPedersenCommitter/Receiver classes, we have the CmtPedersenWithProofsCommitter/Receiver classes that manage the proofs of the commitment scheme.



The structure of the ElGamal commitment scheme is similar, except that there is a separation between the inputs types. This can be shown in the following diagram:



## Commitment – Dynamic View

The general structure of a Commitment scheme contains the following components:

* “Committer” and “receiver” interfaces along with dedicated interfaces for a commitment scheme that can be proven.
* Committer and receiver concrete classes.

### Interfaces

The Committer interface declares the following functions:

1. commit(CmtCommitValue input, long id)

This function receives a value that should be committed to and runs the commitment scheme to obtain the commitment of the given value.

Attached to the input value to commit to, the function receives a unique value, called the id, that is used to keep track of the commitments in case that many commitments are performed one after the other without decommiting them yet.

Different commitment schemes can commit to different types. There are schemes that commit on a byte array, Dlog element, BigInteger, etc. To enable to commit to different types, the commit function receives an object that implements a marker interface, CmtCommitValue.

1. Decommit(long id)

This function accepts an id value used to identify which previously committed value needs to be decommitted and decommits it.

1. sampleRandomCommitValue()

This function samples a random commit value.

1. generateCommitValue(byte[] x)

This function accepts a byte array and converts it into a CmtCommitValue type.

When the user works in an abstract way, that is, by holding an instance of CmtCommitter without knowing the specific scheme, he does not know which concrete CmtCommitValue he should create. To allow this usage, we added this function that converts a byte array (which any object can be converted to) to CmtCommitValue. Each specific commitment scheme implements this function and returns the concrete CmtCommitValue type that is related to it.

1. generateBytesFromCommitValue(CmtCommitValue val)

This is the inverse function of the previous function. It accepts a CmtCommitValue object and returns a byte array that represents this object.

The reason for not using the function toByteArray of CmtCommitValue is that not all the concrete CmtCommitValue classes can implement the conversion to a byte array. For example, CmtGroupElementCommitValue needs the DlogGroup, which it does not have, in order to convert to byte array.

The Receiver interface declares the following functions:

1. receiveCommitment()

This function receives the commitment from the committer through a shared channel and saves it. It returns an object of type CmtRCommitPhaseOutput that holds the id of the commitment and some other information if necessary according to the implementing class.

1. receiveDecommitment(long id)

This function receives the unique id number used to identify which previously committed value needs to be decommitted. Then, receives from the committer the decommitment message, computes the committed value and returns it.

1. generateBytesFromCommitValue(CmtCommitValue val)

As same as the sender's function, this function accepts a CmtCommitValue object and returns a byte array that represents this object.

In addition, there are some commitment schemes that can be proven. For those schemes, we have the CmtWithProofsCommitter and CmtWithProofsReceiver interfaces that extend the above interfaces.

CmtWithProofsCommitter interface declares the functions proveKnowledge to prove that the committer knows what has been committed to and proveCommittedValue to prove that a specific value has been committed to.

CmtWithProofsReceiver interface declares the functions verifyKnowledge to verify that the committer knows what has been committed to and verifyCommittedValue to verify that a specific value has been committed to.

### Concrete classes

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

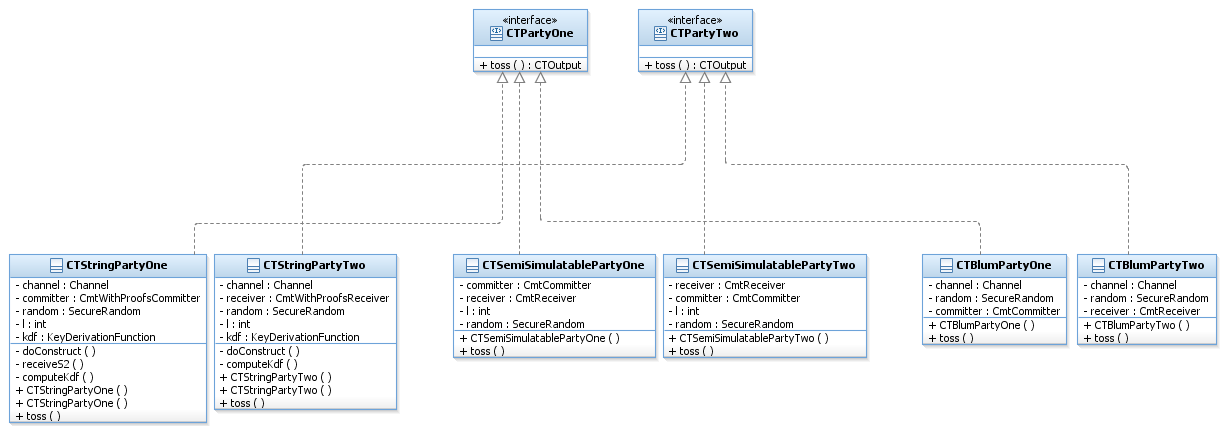
## Coin Tossing – Static View

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 main components of coin tossing are shown in the following diagram:



## Coin Tossing – Dynamic View

The general structure of Coin Tossing protocols contains two levels:

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

### 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.

### Concrete classes

As mentioned above, each concrete coin tossing protocol should have a class that implements PartyOne and another class that implement PartyTwo.

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

# Examples

Very simple examples of how to use OT, Sigma protocol, Zero Knowledge, commitment schemes and Coin Tossing protocols can be found in:

<http://crypto.biu.ac.il/examples-scapi.php>.