SCAPI Non-Interactive Crypto Mid-Layer

R&D Group

Software Design Description

Written by Yael Ejgenberg and Moriya Farbstein

Approved by

Created 10 April 2011

Modified 1 March 2014

CONFIDENTIAL

Change Control Information

|  |  |  |  |
| --- | --- | --- | --- |
| Version No. | Date | Changed by | Scope of Change |
| 1.00 | 01-Jul-2011 | Yael Ejgenberg | Initial version |
|  |  |  |  |
| 2.00 | 29-Dec-2011 | Yael Ejgenberg | 1) Deleted the encryption with existing IV feature from symmetric encryption.  2) Added wide design decisions regarding  a) key generation in first layer and this (mid) layer.  b) source of randomness.  3) Added ElGamal encryption |
| 2.1 | 14-Feb-12 | Yael Ejgenberg | 1. Changed SymmetricCiphertext structure to decorator pattern. 2. Added Cramer-Shoup encryption 3. Added Damgard-Jurik encryption |
| 3.00 | 26-July-12 | Moriya Farbstein | 1. Updated diagrams 2. Updated dynamic views. 3. Added wide design decisions: generating Plaintext from a byte[] and vice versa. |
| 4.00 | 1-March-14 | Moriya Farbstein | Final version |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1. Scope 6](#_Toc379979065)

[1.1 Introduction 6](#_Toc379979066)

[2. Definitions Acronyms and Abbreviations 6](#_Toc379979067)

[3. References 6](#_Toc379979068)

[4. Wide Design Decisions 6](#_Toc379979069)

[4.1 Key generation 6](#_Toc379979070)

[4.2 Source of randomness. 7](#_Toc379979071)

[4.3 Packages 7](#_Toc379979072)

[4.4 Initialization parameters 8](#_Toc379979073)

[4.5 Generating corresponding Plaintext from a byte array 8](#_Toc379979074)

[4.6 Generating corresponding byte array from a Plaintext 10](#_Toc379979075)

[5. Architectural design 11](#_Toc379979076)

[5.1 General 11](#_Toc379979077)

[5.1.1 Main interfaces 11](#_Toc379979078)

[5.1.1.1 Message authentication codes 11](#_Toc379979079)

[5.1.1.2 Encryption Types 12](#_Toc379979080)

[5.1.1.3 Digital Signatures 12](#_Toc379979081)

[5.1.1.4 Security Levels 13](#_Toc379979082)

[5.2 High Level Description 14](#_Toc379979083)

[5.2.1 General Static View 14](#_Toc379979084)

[5.2.1.1 MAC – Message Authentication Codes 14](#_Toc379979085)

[5.2.1.2 Symmetric Encryption 14](#_Toc379979086)

[5.2.1.3 Asymmetric Encryption 15](#_Toc379979087)

[5.2.1.4 Digital Signatures 15](#_Toc379979088)

[5.2.1.5 Factories 15](#_Toc379979089)

[5.2.1.6 Padding schemes 16](#_Toc379979090)

[5.3 Detailed Description 17](#_Toc379979091)

[5.3.1 Key Generation 17](#_Toc379979092)

[5.3.2 Padding schemes 18](#_Toc379979093)

[5.3.2.1 Static View 18](#_Toc379979094)

[5.3.2.2 Dynamic View – NoPadding 18](#_Toc379979095)

[5.3.2.3 Dynamic View – BitPadding 18](#_Toc379979096)

[5.3.2.4 Dynamic View – PKCS7Padding 19](#_Toc379979097)

[5.3.3 Message Authentication Codes 20](#_Toc379979098)

[5.3.3.1 CBC-MAC 20](#_Toc379979099)

[5.3.3.1.1 Static View 20](#_Toc379979100)

[5.3.3.1.2 Dynamic View - ScCbcMacPrepending 21](#_Toc379979101)

[5.3.3.1.3 Usage 24](#_Toc379979102)

[5.3.3.2 HMAC 25](#_Toc379979103)

[5.3.3.2.1 Static View 25](#_Toc379979104)

[5.3.3.2.2 Dynamic View - BcHmac 26](#_Toc379979105)

[5.3.3.2.3 Usage 27](#_Toc379979106)

[5.3.4 Symmetric Encryption 28](#_Toc379979107)

[5.3.4.1 Static view – General 28](#_Toc379979108)

[5.3.4.2 Encryption with IV 30](#_Toc379979109)

[5.3.4.2.1 Static View – Encryption with IV 31](#_Toc379979110)

[5.3.4.2.2 Dynamic View – EncWithIVAbs 31](#_Toc379979111)

[5.3.4.2.3 Static View – CBCEnc 33](#_Toc379979112)

[5.3.4.2.4 Dynamic View – ScCBCEncRandomIV 34](#_Toc379979113)

[5.3.4.2.5 Static View – CTREnc 36](#_Toc379979114)

[5.3.4.2.6 Dynamic View - ScCTREncRandomIV 37](#_Toc379979115)

[5.3.4.2.7 Usage 39](#_Toc379979116)

[5.3.4.3 Authenticated Encryption 40](#_Toc379979117)

[5.3.4.3.1 Static View 40](#_Toc379979118)

[5.3.4.3.2 Dynamic View - ScEncryptThenMac 40](#_Toc379979119)

[5.3.4.3.3 Usage 42](#_Toc379979120)

[5.3.5 Asymmetric encryption 43](#_Toc379979121)

[5.3.5.1 Static View – General 43](#_Toc379979122)

[5.3.5.2 Static View – El Gamal 44](#_Toc379979123)

[5.3.5.3 Dynamic View – ElGamalAbs 45](#_Toc379979124)

[5.3.5.4 Dynamic View – ScElGamalOnGroupElement 46](#_Toc379979125)

[5.3.5.5 Dynamic View – ScElGamalOnByteArray 48](#_Toc379979126)

[5.3.5.6 Usage – ElGamalEnc 50](#_Toc379979127)

[5.3.5.7 Static View – CramerShoupDDH 51](#_Toc379979128)

[5.3.5.8 Dynamic View – CramerShoupAbs 52](#_Toc379979129)

[5.3.5.9 Dynamic View – ScCramerShoupDDHOnGroupElement 54](#_Toc379979130)

[5.3.5.10 Dynamic View – ScCramerShoupDDHOnByteArray 56](#_Toc379979131)

[5.3.5.11 Usage - CramerShoupDDHEnc 58](#_Toc379979132)

[5.3.5.12 Static View – DamgardJurikEnc 59](#_Toc379979133)

[5.3.5.13 Dynamic View – DamgardJurikEnc 60](#_Toc379979134)

[5.3.5.14 Usage - DamgardJurikEnc 64](#_Toc379979135)

[5.3.5.15 Static View – RSAOaepEnc 65](#_Toc379979136)

[5.3.5.16 Dynamic View – RSAOaepAbs 66](#_Toc379979137)

[5.3.5.17 Dynamic View – BcRSAOaep 67](#_Toc379979138)

[5.3.5.18 Dynamic View – CryptoPpRSAOaep 68](#_Toc379979139)

[5.3.5.19 Usage - RSAOaepEnc 69](#_Toc379979140)

[5.3.5.20 Usage - Asymmetric encryption 71](#_Toc379979141)

[5.3.6 Digital Signatures 73](#_Toc379979142)

[5.3.6.1 Static View – General 73](#_Toc379979143)

[5.3.6.2 Static View – RSAPss 74](#_Toc379979144)

[5.3.6.3 Dynamic View – RSAPssAbs 74](#_Toc379979145)

[5.3.6.4 Dynamic View – BcRSAPss 75](#_Toc379979146)

[5.3.6.5 Dynamic View – CryptoPpRSAPss 76](#_Toc379979147)

[5.3.6.6 Usage – RSAPss 77](#_Toc379979148)

[5.3.6.7 Static View – DSA 79](#_Toc379979149)

[5.3.6.8 Dynamic View – ScDSA 79](#_Toc379979150)

[5.3.6.9 Usage – DSA 82](#_Toc379979151)

[6. Performance 83](#_Toc379979152)

[7. Multi-Platform Issues 83](#_Toc379979153)

[8. Benchmarking 83](#_Toc379979154)

[9. Open Issues 83](#_Toc379979155)

# Scope

The purpose of this SDD is to analyze and define the elements of what we call the mid-level non-interactive primitives. It includes different encryption schemes, message authentication codes and digital signatures.

## Introduction

Functions in this level heavily use the primitives described in [[1](#_References)] to perform internal computations. For example, the ElGamal encryption scheme uses DlogGroup; CBC-MAC uses any of the PRPs defined in the first level.

# Definitions Acronyms and Abbreviations

|  |  |
| --- | --- |
| Gen | generate key |
| Enc | encrypt |
| Dec | decrypt |
| Mac | message authenticated algorithm |

# References

[1] FirstLevelSDK\_SDD.docx

[2] Introduction to modern cryptography- Jonathan Katz & Yehuda Lindell.

# Wide Design Decisions

The same principles that guided us in [[1](#_References)] guide us here to achieve maximum flexibility and extensibility as at the same time being as efficient as possible.

## Key generation

This layer mainly includes encryption schemes, macs and digital signatures. By definition all of them are tuples of three probabilistic polynomial-time algorithms:

1. Encryption schemes🡪 (Gen, Enc, Dec)
2. Mac 🡪 (Gen, Mac, Verify)
3. Digital signatures 🡪 (Gen, Sign, Verify)

It follows, that the generation of a relevant key is an inherent part of the scheme, therefore in all the interfaces that we will present below there will be a “generateKey” functionality as well as encrypt/decrypt or mac/verify or sign/verify functionalities. This is different from the way keys are generated in the JCA/JCE platform. There, the generation of the key is not an inherent part of the algorithm being implemented but something done externally by some service class, for example the KeyGenerator or KeyPairGenerator classes. Our approach here is to model the world in a “closer to the cryptographer’s view” way. Yet, when implementing the “generateKey” functionality in this layer we may fully or partially use the JCA/JCE approach as part of the implementing code. This will be further explained in relevant sections below.

## Source of randomness.

1. We allow the user to choose between providing his own source of randomness and using SCAPI’s default. However, for technical reasons, whenever a concrete implementation from a native library is used, it is not possible to use the user’s source. In that case, the source of randomness is the one provided in the native library. This will be properly documented in the relevant javadocs.
2. Whenever we need to choose a random element from a certain range of numbers we will use Bouncy Castle’s static function in org.bouncycastle.util.BigIntegers

**public** **static** BigInteger createRandomInRange(BigInteger min,

BigInteger max,

SecureRandom random)

## Packages

* edu.biu.scapi.midLayer
* edu.biu.scapi.midLayer.ciphertext
* edu.biu.scapi.midLayer.plaintext
* edu.biu.scapi.midLayer.signature
* edu.biu.scapi.midLayer.symmetricCrypto
* edu.biu.scapi.midLayer.symmetricCrypto .mac
* edu.biu.scapi.midLayer.symmetricCrypto .encryption
* edu.biu.scapi.midLayer.symmetricCrypto .keys
* edu.biu.scapi.midLayer.asymmetricCrypto
* edu.biu.scapi.midLayer.asymmetricCrypto.encryption
* edu.biu.scapi.midLayer.asymmetricCrypto.keys
* edu.biu.scapi.midLayer.asymmetricCrypto .digitalSignature
* edu.biu.scapi.securityLevel
* edu.biu.scapi.paddings

## Initialization parameters

The main elements in this layer are algorithms of type Mac, Symmetric Encryption, Asymmetric Encryption and Digital Signatures. In order to use the functionality of any of these algorithms we need to create an object or instance of the relevant type. Not only we need to have such an instance but also it has to be fully initialized. This includes having any underlying objects fully initialized. For example, El Gamal encryption uses an underlying Dlog Group which has to be fully initialized in order for El Gamal to encrypt and decrypt.

Classes implementing such said algorithms will only have regular constructors, with different freedom degrees. That is, for member variables that need initialization upon construction they may allow the user to enter all their values, some or none (default constructor). Those members that the user does not enter values for, will be initialized with default values. SCAPI will always try to choose the most efficient or secure (depending on the case) default values.

By having these constructors that do not require all the parameters, we allow a user to start working straight away with the algorithm he needs without having to worry about unnecessary parameters.

In most cases, there is one constructor that actually performs the object creation, usually the constructor that gets all the required parameters. The other constructors call this implementation with the parameters they got from the user and additional default parameters, if needed.

All the constructors get fully initialized underlying objects as parameters. As a result, the created object needs only the key(s) to start working.

For Symmetric Crypto there will be a setKey(SecretKey) function and for Asymmetric Crypto there will be a setKey(PublicKey, PrivateKey) and setKey(PublicKey) functions. These functions can be called at any time during the life of the object.

There will also be an isKeySet() : Boolean function. This function is used to check whether the SecretKey or PublicKey has been set.

## Generating corresponding Plaintext from a byte array

With the current design, a protocol that wishes to use an asymmetric encryption scheme, needs to be able to construct a proper plaintext, which is different for each different algorithm.

At the level of AsymmetricEncryption interface we have a function with signature:

AsymmetricCiphertext encrypt(Plaintext plaintext)

This should enable a general protocol to encrypt on an abstract level. But in reality since each algorithm needs a specific plaintext this is not the case.

In order to allow full generality we introduce another function at the interface level:

Plaintext generatePlaintext(byte[] text).

This function is implemented by each algorithm, and each implementation will return the specific plaintext, for example, Damagard-Jurik scheme returns an object of type BigIntegerPlaintext.

There are cases that there is a maximum byte array length that can be used to create a Plaintext, and other cases where there are no such limitations. To allow the user know if the current encryption has a maximum length and what it is, we provide the functions:

* boolean hasMaxByteArrayLengthForPlaintext() that returns true if the current encryption has a maximum byte array length and false otherwise.
* int getMaxLengthOfByteArrayForPlaintext() that return the maximum length.

In cases that there is no limitation on the byte array length that can be used to create a Plaintext, the function getMaxLengthOfByteArrayForPlaintext can be implemented in several ways:

* Return MAX\_BI which is a constant BigInteger with a huge value (For example, 24096). The disadvantage of this solution is that it contradicts the usage of BigInteger that can be of any length. This will cause the user do two checks to know if the array he have is in legal length:

1. Check if the current byteArray length is smaller than the value returned from getMaxLengthOfByteArrayForPlaintext.

2. If the current byteArray length is bigger than the value returned from getMaxLengthOfByteArrayForPlaintext, check if the value returned from the function is the same as MAX\_BI. If it is, the size of the byte array is legal.

* Return 0 or -1. The disadvantage of this solution is that some user may test his input length against this value and always reject the input, which is exactly the opposite of what we want. In other case the user will create an array of getMaxLengthOfByteArrayForPlaintext size, which will cause an ArrayOutOfBoundException.
* Throw a checked exception. The disadvantage of this solution is that we force the API users to use an exception for ordinary control flow. They will have to write the call for the getMaxLengthOfByteArrayForPlaintext in the try block and deal with the case that there is no maximum in the catch block. This will make the code annoying and less readable.
* Throw an unchecked exception. The disadvantage of this solution is that the exception can be thrown without any warning because there is no compiler alert of catching this exception.

We decided to use the last option. The recommended way to use these functions and avoid the problem shown above is a follows:

* Get the byte[] data that needs encryption.
* If encryptionObject.hasMaxByteArrayLengthForPlaintext ()
* Get the maximum length maxLen by calling encryptionObject.getMaxLengthOfByteArrayForPlaintext()
* For i=0 to data.length/maxLen
* Encrypt **each data block** of size maxLen by calling encryptionObject.encrypt(dataPart)
* Else, encrypt data at once by calling encryptionObject.encrypt(data)

## Generating corresponding byte array from a Plaintext

In addition to generating a Plaintext from a byte array, a protocol that wants to work with Asymmetric encryption on a high level, needs to be able to return back from an abstract Plaintext to a byte array. The decrypt function in the AsymmetricEncryption interface returns a “Plaintext”, and a protocol that does not know the specific Asymmetric encryption it has, cannot know the type of the plaintext and cannot do anything with it.

In order to allow full generality we introduce another function at the interface level:

byte [] generateBytesFromPlaintext(Plaintext plaintext).

This function is implemented by each algorithm. Each implementation will check that the given plaintext is an instance of the expected Plaintext type, generate a byte array from that plaintext and return it. For example, Damagard-Jurik scheme checks that the given plaintext is an instance of BigIntegerPlaintext and returns the byte array that represents the BigInteger value of the plaintext.

As mentioned above, the function encrypt gets a Plaintext as a parameter. Each Asymmetric encryption algorithm needs different type of Plaintext, and a protocol that wishes to use an asymmetric encryption scheme on an abstract level needs to be able to construct a plaintext that suits the encryption algorithm it has.

Although it is more intuitive to have these conversions from a Plaintext to a byte array and vice versa as functions of the Plaintext itself, or moreover, to have just a ByteArrayPlaintext, this cannot be done. Not all the plaintext types can be generated from a byte array. For example, there is just a small subset of binary strings that can be converted into a GroupElement, and that depends on the DlogGroup that is used. DlogGroup family has a function that converts a byte array into a GroupElement, if that can be done.

Because the protocol does not necessarily know which encryption algorithm it has, it cannot convert the byte array into a plaintext, because it does not know how. The only way it can construct a suitable Plaintext for the encryption algorithm is by the encryption object itself. Therefore the conversion from a byte array into a Plaintext must be in the AsymmetricEnc interface.

The second conversion from a Plaintext back to a byte array is a function of the AsymmetricEnc interface by a symmetric reason.

Note that those functions are used only when the protocol works in an abstract level, without knowing the specific asymmetric encryption algorithm it has. Otherwise, the protocol knows exactly which Plaintext to construct and how to do that. For example, see the [ElGamal usage section](#_Usage_–_ElGamalEnc).

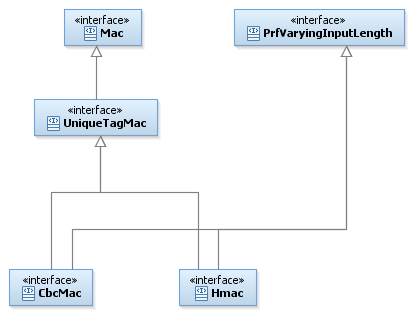
# Architectural design

## General

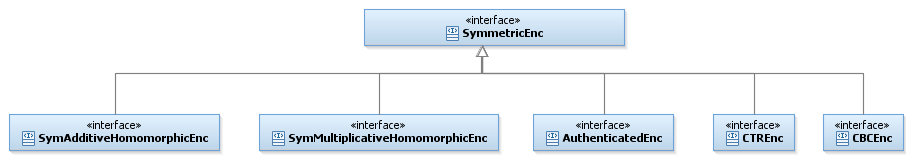
### Main interfaces

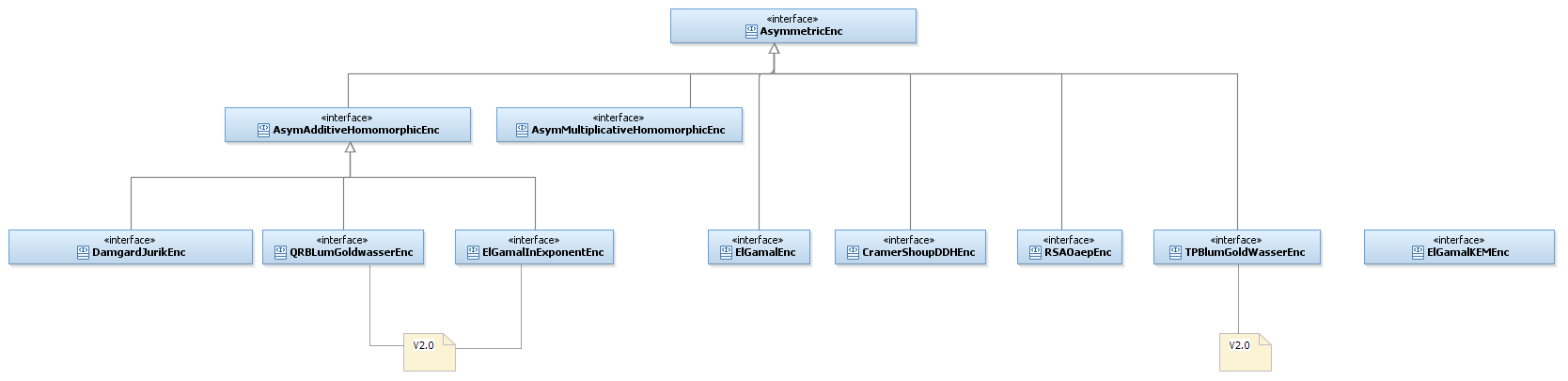
In this section we show the main elements that compose this layer: MAC, Encryption and Digital Signatures. We also present a diagram of the different Security Levels each element in this layer and in the first layer complies to.

#### Message authentication codes

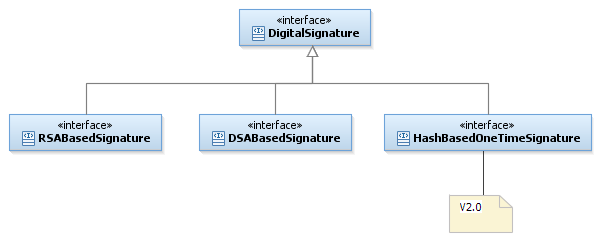


#### Encryption Types

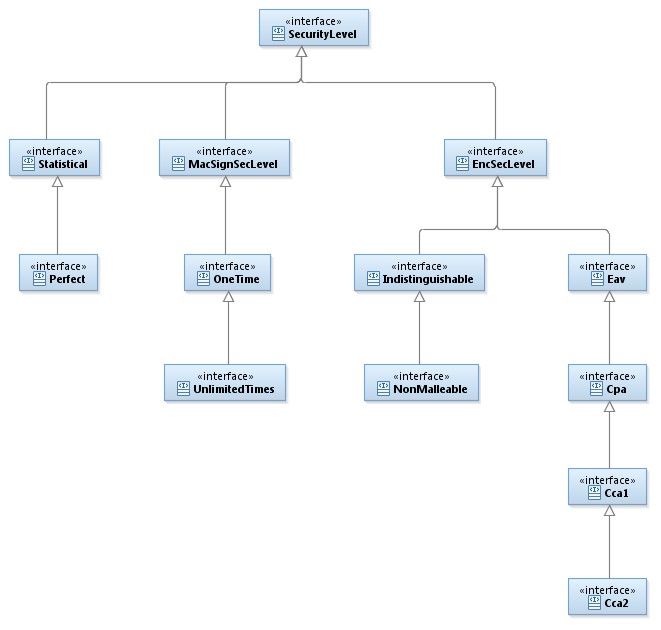




#### Digital Signatures



#### Security Levels

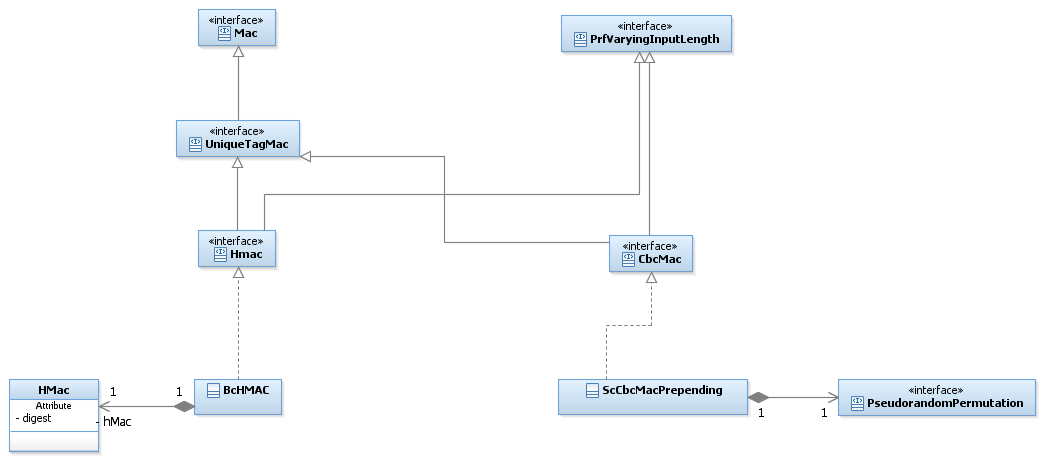
****

## High Level Description

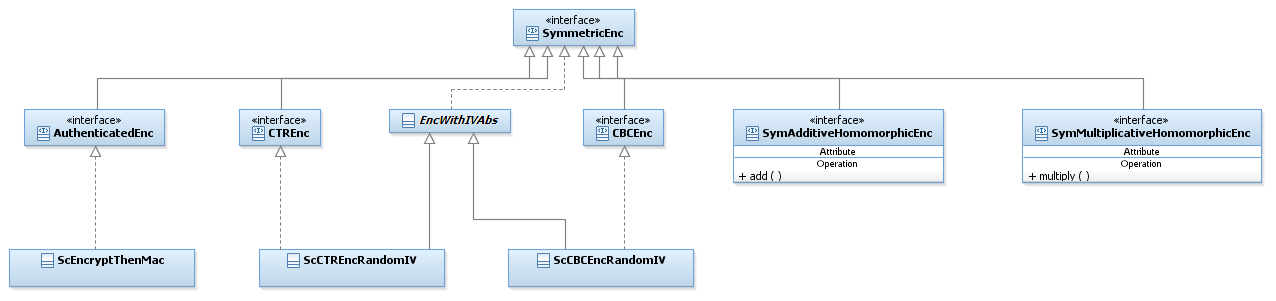
### General Static View

In this section we present all the elements of this layer that include, apart from the main elements, also the auxiliary classes and interfaces, like padding schemes.

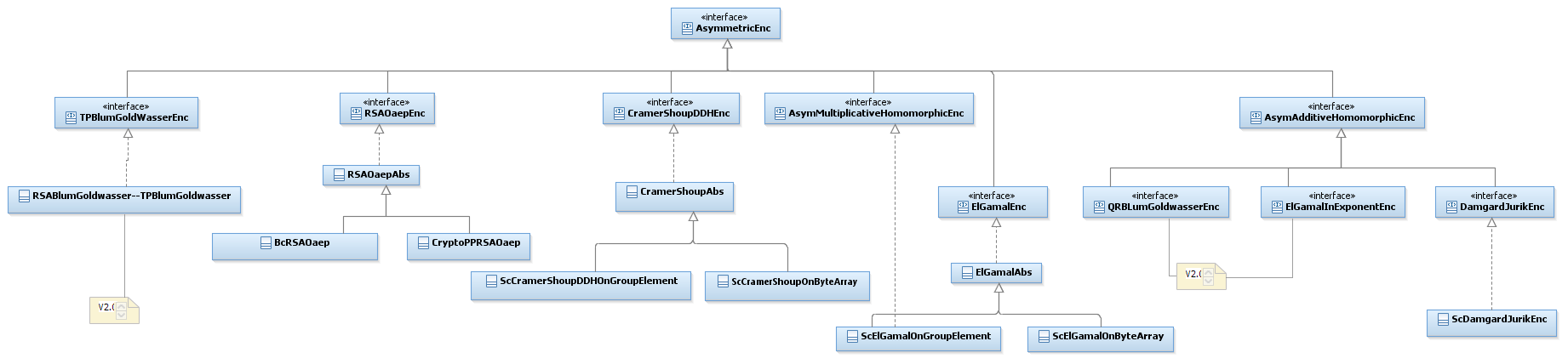
#### MAC – Message Authentication Codes

****

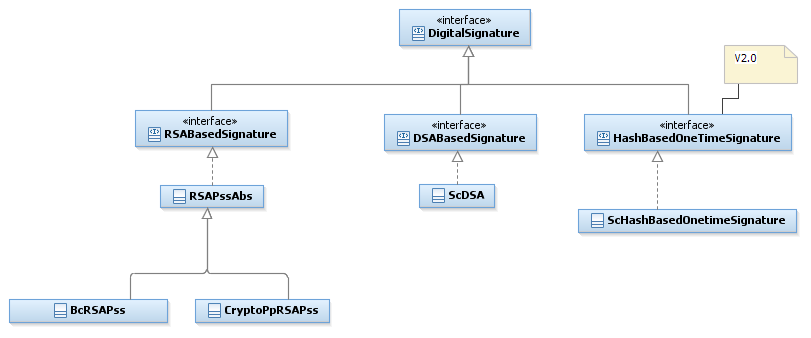
#### Symmetric Encryption

****

#### Asymmetric Encryption



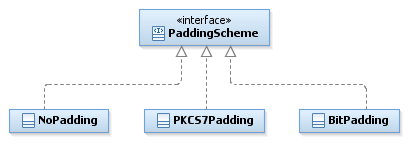
#### Digital Signatures



#### Factories

The current design of the factories (as mentioned in [[1](#_References)]) allows us to choose the provider of the main object only. The underlying objects are built with the default providers. This is a big drawback because almost every object holds some inner objects. Thus, we decided to remove our support in factories from now on. Factories for the first layer are still supported.

#### Padding schemes



## Detailed Description

In this detailed description we will present the implementation of Message Authentication Codes, Symmetric and Asymmetric Encryption, and Digital Signatures.

Notice that at this level we added to the Static Views of each family, the Security Level that each element belongs to.

### Key Generation

Every algorithm that needs a key has at least one member function *generateKey* that generates the corresponding key. Therefore, in order to generate a key for a certain algorithm we need an instance of the relevant algorithm and to call the generateKey(…) function. This is true for Symmetric Cryptography as well as for Asymmetric Cryptography and also true for the first layer of primitives and for this layer.

In Symmetric Crypto the key generated is a Secret Key and in Asymmetric Crypto it is actually a Key Pair which is a holder of a Public Key and a Private Key.

Key Generation in Symmetric Crypto

* SecretKey generateKey(int keySize)
* SecretKey generateKey(AlgorithmParameterSpec keyParams)

In most cases, a secret key is simply an array of random bits of a certain size. All the algorithms that need this type of a Secret Key will implement the generateKey function that accepts an int as an argument. Since there might be other algorithms that use a Secret Key that is created differently than by simply choosing some amount of random bits, we provide a more general generateKey function that accepts an argument of type AlgorithmParameterSpec. This means that any parameters can be passed to that function. **If an algorithm that needs such “special” keys is implemented, then a corresponding parameters’ class has to be implemented as well**. For example, ScEncryptThenMac needs to generate two secret keys, one for encryption and one for authentication. To allow this, we implemented AuthEncKeyGenParameterSpec that holds the sizes of the required keys.

In general, if one algorithm implements one of the generateKey(…) functions, then it does not implement the other. In this case, it throws UnsupportedOperationException for the unimplemented one.

Algorithms that use the key to initialize an underlying object will delegate the generation of the key in both *generateKey* functions to the corresponding underlying object’s *generateKey* function.

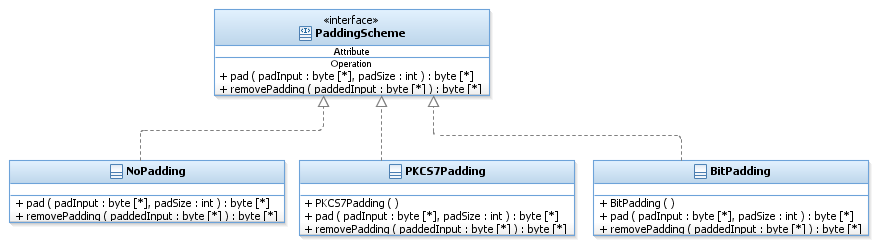
Key Generation in Asymmetric Crypto

* KeyPair generateKey()
* KeyPair generateKey(AlgorithmParameterSpec keyParams)

Again here in some cases the generateKey() with no arguments will be preferably implemented. It assumes that all that is needed for the function to generate the key is held by the algorithm object. An algorithm that implements this function will throw UnsupportedOperationExcepton if the second generateKey is called. Only algorithms that need special parameters for the generation of the key will implement the second function; and similarly to the Symmetric Crypto case the corresponding parameters class has to be created.

### Padding schemes

#### Static View



#### Dynamic View – NoPadding

Function: pad(byte[] padInput, int padSize) : Byte[] do

This class does not pad the given input.

* Return padInput.

Function: removePad(byte[] paddedInput) : byte[] do

This class does not pad the input, so the given paddedInput is not actually padded.

* Return paddedInput.

#### Dynamic View – BitPadding

Function: pad(byte[] padInput, int padSize) : Byte[] do

Pads the given byte array with padSize bytes according to BitPadding padding

scheme, Which add 10,...,0 bits to the given padInput.

The value of the first added byte is 10000000 and the values in the next added bytes

are 00000000.

* Allocate a byte array of size padInput’s length + padSize, named paddedArray.
* Copy padInput to the MSB of padded Array.
* Put in paddedArray[padInput.length] the byte 10000000.
* For i=1 to padSize-1
* Put in paddedArray[padInput.length + i] the byte 00000000.

Function: removePad(byte[] paddedInput) : byte[] do

* go over paddedArray from the LSB to the MSB until there is a byte with value 10000000 to find the first padding byte. The found index is denoted by i.
* Allocate a new byte array of size i.
* Copy the first i MSB bytes to the new array and return it.

#### Dynamic View – PKCS7Padding

Function: pad(byte[] padInput, int padSize) : Byte[] do

The input will be padded at the trailing end with k-(lth mod k) octets all having value

k- (lth mod k), where lth is the length of the input. In other words, the input is padded

at the trailing end with one of the following strings:

* 01 -- if lth mod k = k-1
* 02 02 -- if lth mod k = k-2
* .
* .
* .
* k k ... k k -- if lth mod k = 0

This padding method is well defined if and only if k is less than 256.

* Allocate a byte array of size padInput’s length + padSize, named paddedArray.
* Copy padInput to the MSB of padded Array.
* Compute padNum = the byte value of padSize.
* For i=0 to padSize-1
* Put padNum in paddedArray[padInput.length + i].

Function: removePad(byte[] paddedInput) : byte[] do

* Get the number of padded bytes from the LSB byte.
* Calculate originalSize = paddedInput.length – number of padded bytes
* Allocate a new byte array of size originalSize.
* Copy the first originalSize MSB bytes to the new array and return it.

### Message Authentication Codes

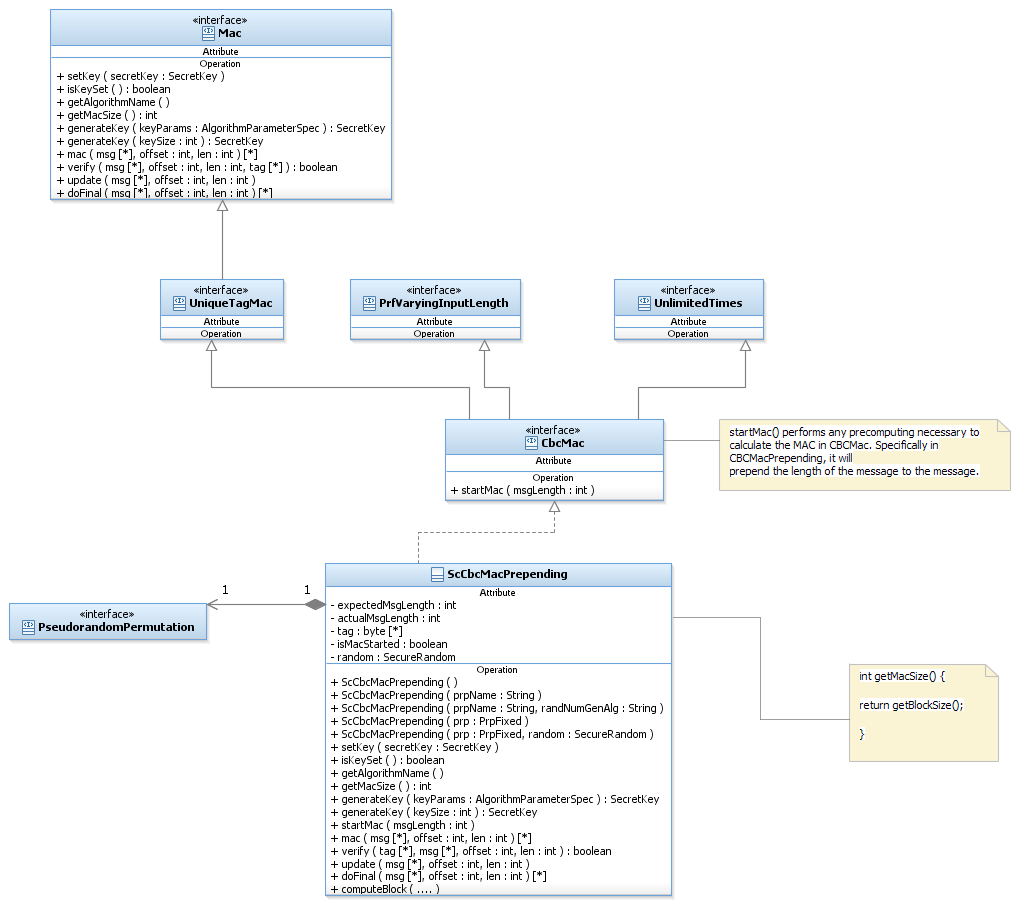
#### CBC-MAC

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.

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

##### Static View

In the following figure we present the interface and class structure for CBC-MAC. CBC-MAC is a type of MAC, but it is also, more specifically a type of Unique-Tag-MAC. The “type” CBC-MAC is represented by the interface CbcMac. This means that SCAPI can contain more than one concrete implementation of CBC-MAC. This complies with the generality and flexibility that we have tried to maintain for all the primitives in the first layer and in this layer (the middle layer). We actually implement the CBC-MAC in the class called ScCbcMacPrepending which implements CBC-MAC in the way explained above. Other implementations by us or by other providers can be added as needed.



##### Dynamic View - ScCbcMacPrepending

Function: ScCbcMacPrepending (String prpName, String randNumGenAlg) do:

* Create a new PRF object from the prpName string argument.
* If the created PRF is not an instance of PrpFixed, throw IllegalArgumentException.
* Set the PRP member variable to the created object.
* Create a new SecureRandom object from the randNumGenAlg string argument.
* Set the SecureRandom member variable to the created object.

Function: ScCbcMacPrepending (PrpFixed prp, SecureRandom random) do:

* Set the PRP member variable to the one in the argument.
* Set the SecureRandom member variable to instance of SecureRandom provided by the user.

Function: setKey(SecreteKey secretKey) do

* Call underlying PRP respective setKey function.

Function: isKeySet() do

* Return prp.isKeySet().

Function: startMac(int msgLength) do:

This function prepends the length of the message to the message.

* If this object has no secret key, throw IllegalStateException.
* Set actualMsgLength to zero.
* Set expectedMsgLength to msgLength.
* Calculate tag = prp.computeBlock(msgLength) //By doing this we are "pre-pending" the length of the msg to the message, and the mac will be calculated on [msgLength || msg].
* Set isMacStarted to true.

Function: update(byte[] msg, int offset, int len) do:

* If this object has no secret key, throw IllegalStateException.
* If msg is not marked as started, throw an exception.
* If msg is not aligned to the underlying PRP’s block size, **throw IllegalArgumentException.** (\*)
* While there are blocks to process in msg do
  + tag = prp.computeBlock( tag XOR current block in msg) //It’s important to save space here and to avoid unnecessary allocation and copy of arrays, so we put the result into tag.
  + actualMsgLength += this block’s size.

Function: doFinal(byte[] msg, int offset, int len) : byte[] do:

* If this object has no secret key, throw IllegalStateException.
* If msg is not marked as started, throw IllegalStateException.
* If msg is not aligned to the underlying PRP’s block size, then pad with zeroes. (\*)
* While there are blocks to process in msg do
* tag = prp.computeBlock( tag XOR current block in msg) //It’s important to save space here and avoid unnecessary allocation and copy of arrays.
* actualMsgLength += this msg’s size
* If actualMsgLength != expectedMsgLength, throw IllegalArgumentException.
* Return the calculated tag.

Note (\*): The alignment of the message can be checked in the following way:

If len % blockSize != 0 then throw Exception.

Function: mac(byte[] msg, int offset, int len) : byte[] do:

* If this object has no secret key, throw IllegalStateException.
* Call startMac with received msgLength.
* Call doFinal with whole msg.
* Return the calculated tag.

Function: verify(byte[] msg, int offset, int len, byte[] tag) : boolean do:

* If this object has no secret key, throw IllegalStateException.
* If the length of tag does not equal macSize then return false.
* Compute mac on the message.
* If the received tag equals the mac, return true, else return false. For code-security reasons, the comparison has to be fully performed. That is, even if we know already after the first few bits that the tag is not equal to the mac, we should not be inclined to think that we should cut short the check for performance.

Function: computeBlock do:

Since CBC-MAC is also a PRF we need to implement the three compute functions indicated in the PRF interface.

* If this object has no secret key, throw IllegalStateException.
* Call mac (msg : Byte [\*], offset : int, msgLen : int, tag : Byte [\*])
* Copy the resulting tag of the mac to the given out array.

Function: getAlgorithmName() : String

* Return the string: CBC-MAC/[current PRP].

Function: generateKey(AlgorithmParameterSpec keyParams) : SecretKey

Delegates the key generation to the underlying prp.

* Call and return prp.generateKey(keyParams).

Function: generateKey(int keySize) : SecretKey

Delegates the key generation to the underlying prp.

* Call and return prp.generateKey(keySize).

Function: getMacSize():int do:

* Return getBlockSize().

Function: getBlockSize():int do:

* Call and return prp.getBlockSize()

##### Usage

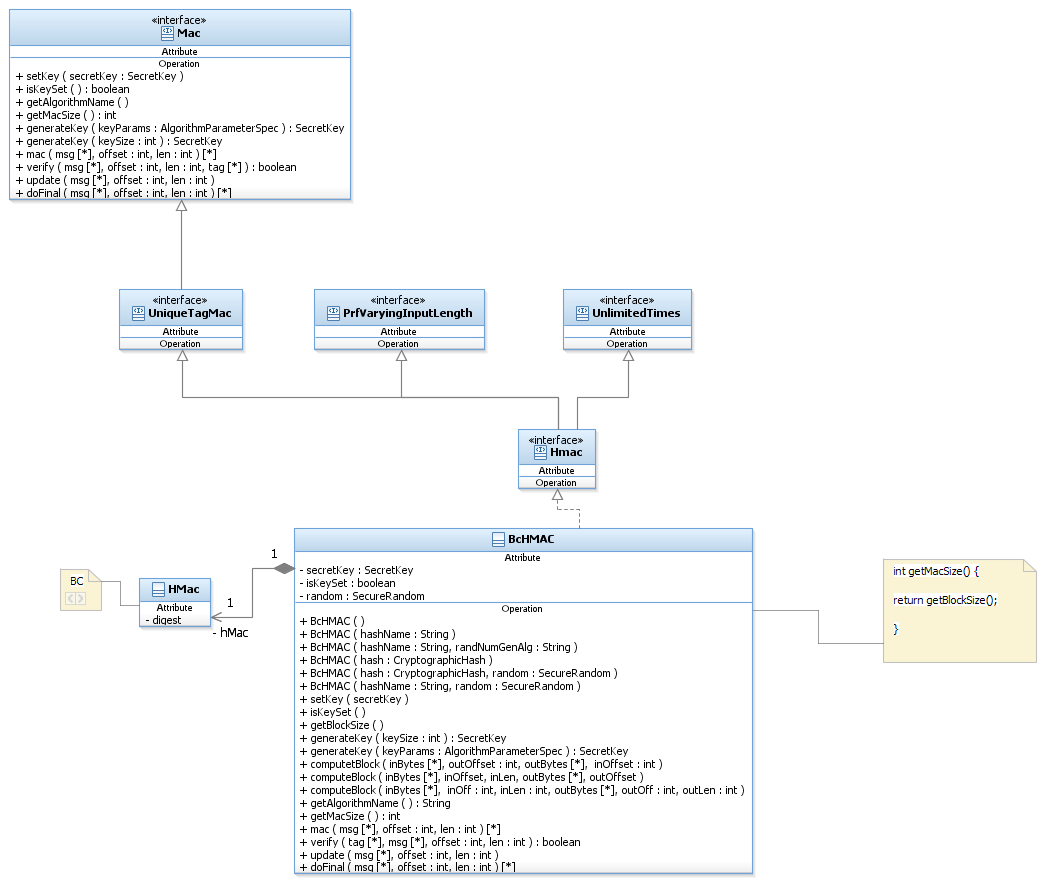
CbcMac is secure only if the length of the full message is known in advance and its computation with the underlying block cipher is prepended to the message. We allow two different ways to calculate the MAC with CBC-MAC:

1. Call the mac function on the full message. This method is useful when we hold the whole message and then it is obvious that we know its full size.
2. If we know the size of the full message but we do not hold the whole message at once but rather get pieces of it from some source, then we can divide the computation of the mac into a few steps. To do so, call the following series of functions in the order shown here:
   1. startMac with the length of the message
   2. call update as needed
   3. call doFinal

#### 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.

##### Static View



##### Dynamic View - BcHmac

Function: computeBlock do

* This function stays the same as specified for the first layer.

Function: mac do

* Call computeBlock and put into tag the result in outBytes.

Function: verify do

* Call mac on the message
* If the resulting mac equals the tag then return true, else return false.

Function: update do

* Call the update function of the underlying HMac from BC.

Function: doFinal do

* Call the update function of the underlying HMac from BC with the last part of the message.c
* Call the doFinal function of the underlying HMac from BC.

Function: generateKey(AlgorithmParameterSpec keyParams) do

* Throw UnSupportedOperationException.

Function: generateKey(int keySize) do

* Generate the required bytes using the random class member.

##### 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.

…

// 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

In the [general description section](#_Encryption_Types) we presented three main categories of symmetric encryption.

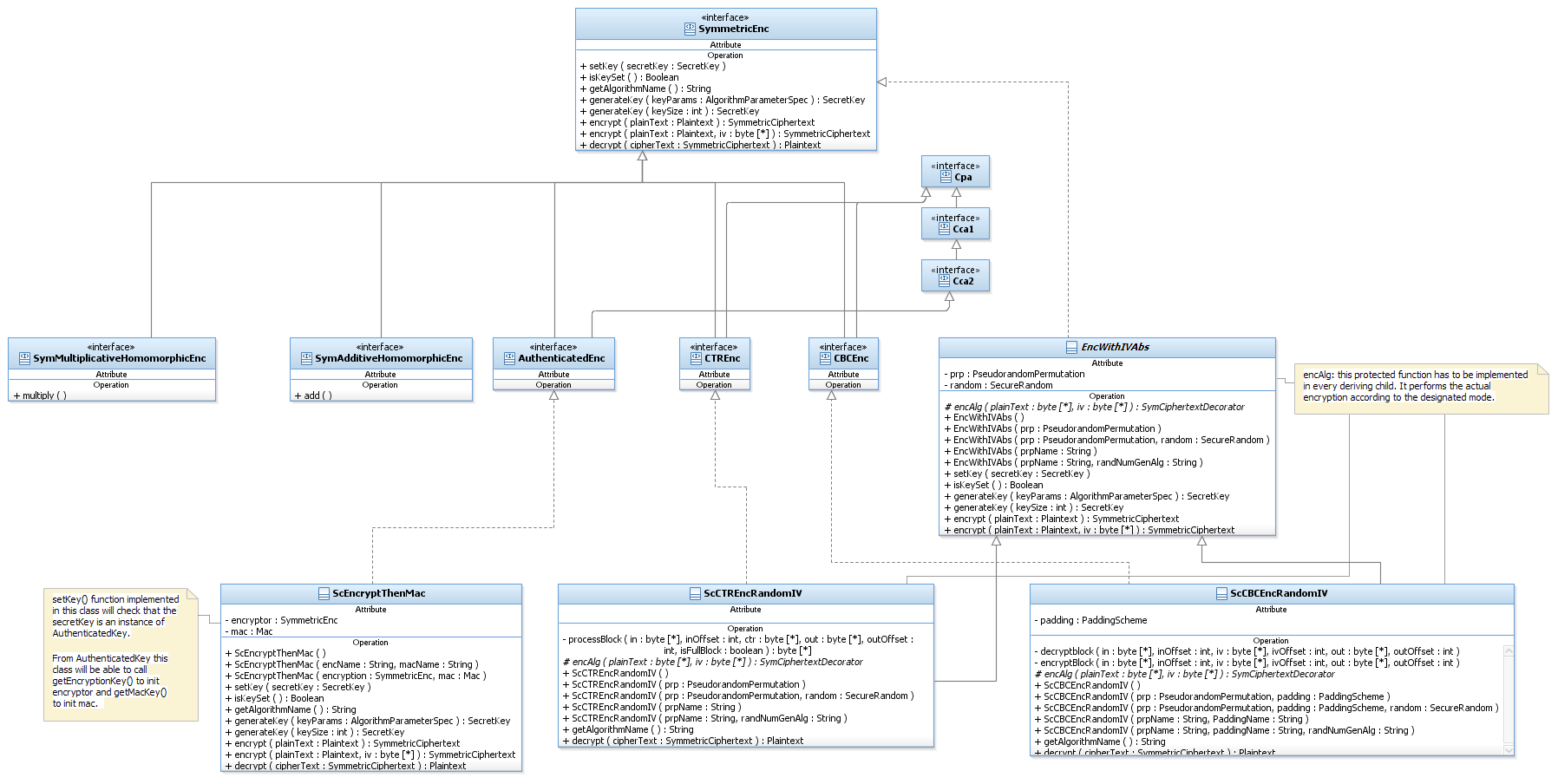
1. An encryption based on modes of operation using a pseudo-random permutation and a randomized IV. The randomized IV is crucial for security. CBCEnc and CTREnc belong to this category.
2. An authenticated encryption where the message gets first encrypted and then mac-ed. EncryptThenMac belongs to this category.
3. Homomorphic encryption. Even though we do not currently implement any concrete homomorphic encryption, we provide the interfaces for future possible implementations.

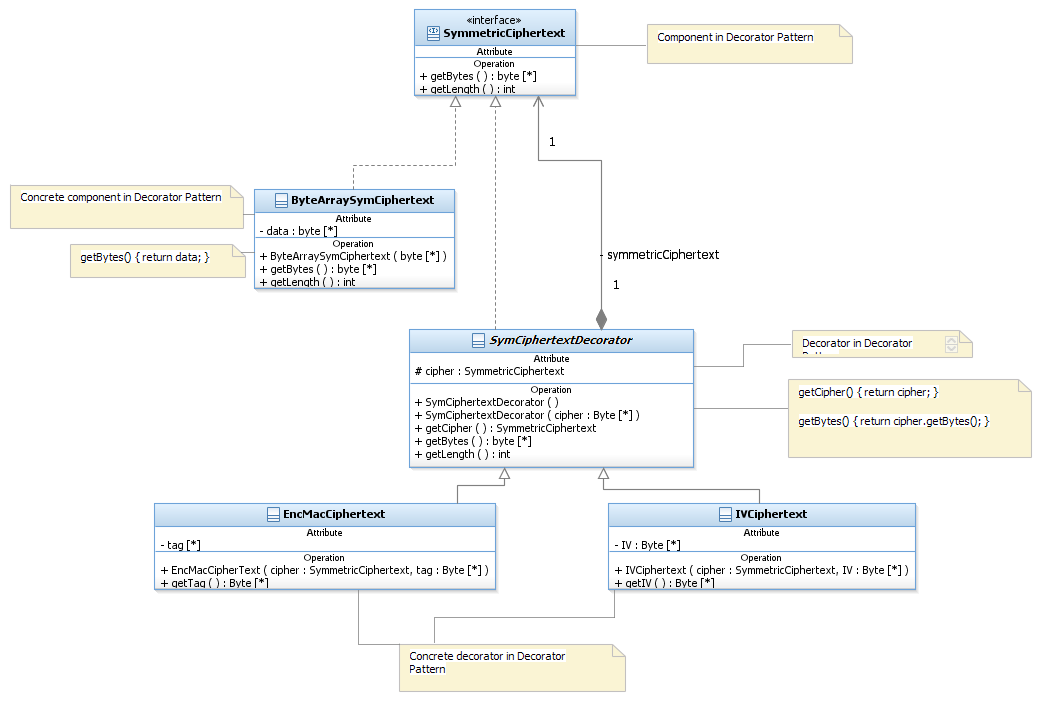
We note that for authenticated encryption, two secret keys will be used, one for the encryption and one for the authentication. Yet, the setKey() functions will require a single secret key object, so that a protocol working on a higher level will be able to perform initialization of the encryption object at a general level. In the case of authenticated encryption, the concrete class implementing the setKey functions will check that the secret key passed is actually an instance of AuthenticatedKey from which it will be able to initialize the encrypting object with the encryption key and the MAC object with the mac key.

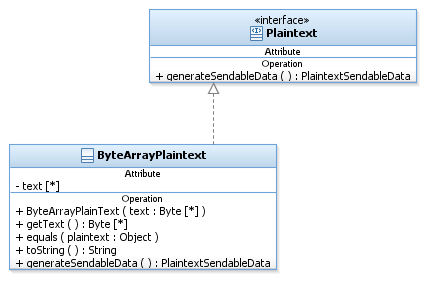
The symmetric encryption family of classes implements three main functionalities that correspond to the cryptographer’s language in which an encryption scheme is composed of three algorithms:

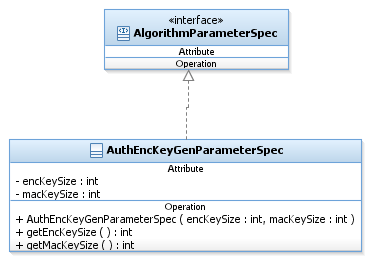
1. Generation of the key.
2. Encryption of the plaintext.
3. Decryption of the ciphertext.

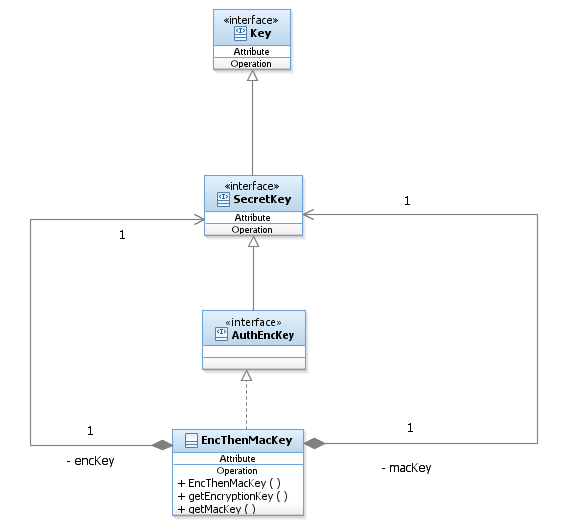
#### Static view – General











#### Encryption with IV

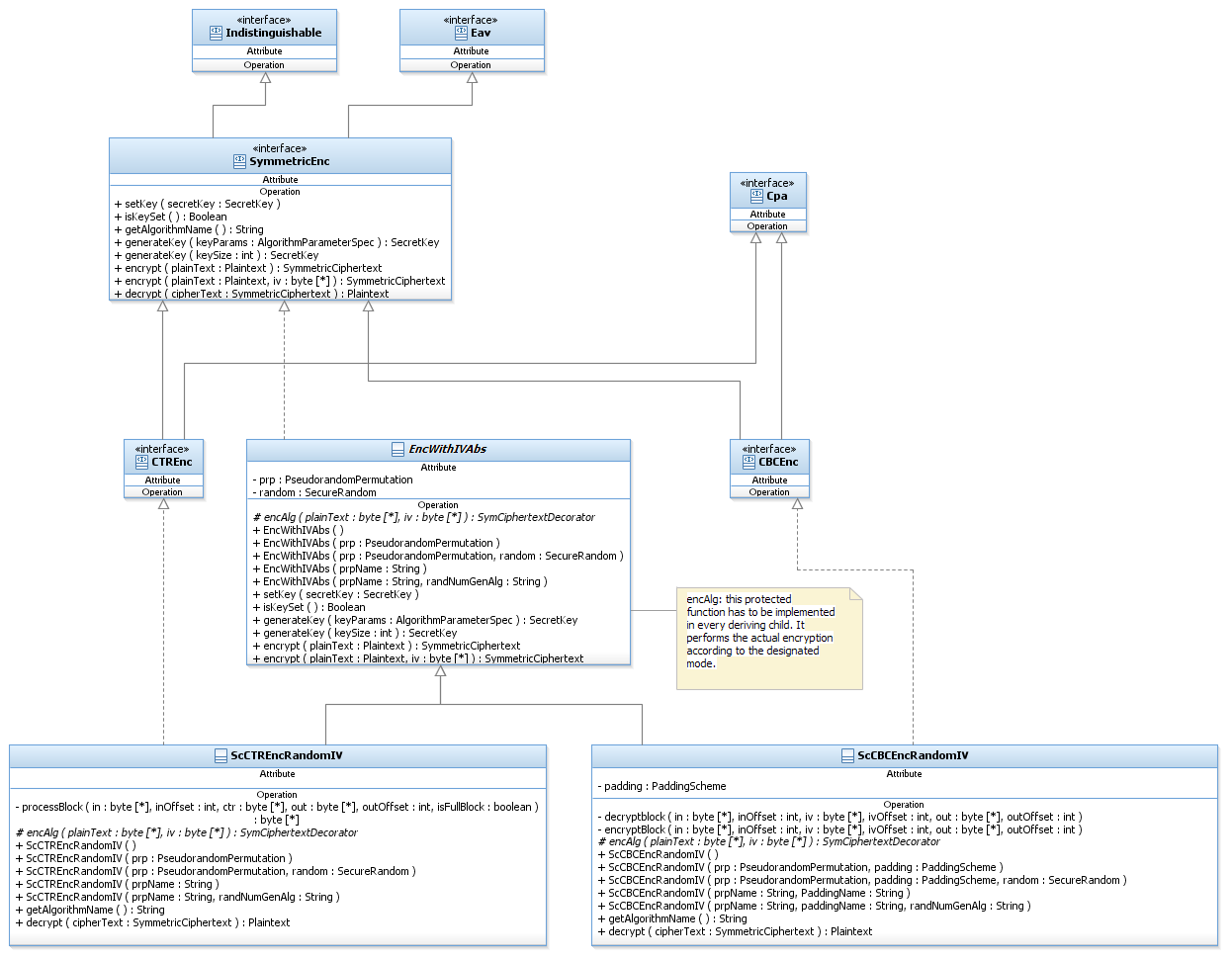
Since most of the IV work for encryption is the same for CBCEnc as for CTREnc we enclose all the common functionality in a package private abstract class called EncWithIVAbs. The actual mode of operation will be implemented in a protected function in each of the concrete classes.

The concrete class implementing the encryption using the CBC mode of operation is called ScCBCEncRandomIV; it extends the abstract class EncWithIVAbs and implements the CBCEnc interface.

The concrete class implementing the encryption using the CTR mode of operation is called ScCTREncRandomIV; it extends the abstract class EncWithIVAbs and implements the CTREnc interface.

In the Static and Dynamic Views, we present all the above mentioned elements since most of the functionality is common and is implemented in the parent abstract class. At the end, we will present the specific functionality of each concrete class.

##### Static View – Encryption with IV



##### Dynamic View – EncWithIVAbs

Constructor: EncWithIVAbs(PseudorandomPermutation prp, SecureRandom random) do:

* Set the prp member to the given prp.
* Set the random member to the given random.

Funcion: setKey ( SecretKey secretKey) do

setKey function can be called an indefinite amount of times. This may be very useful if many encryptions with the same permutation and different keys have to be performed. Then we save all the memory allocations of the encryption scheme and just change what is necessary.

* Call the underlying’s PRP setKey function.

Funcion: isKeySet ()

* Call and Return the underlying PRP isKeySet function.

Funcion: generateKey(AlgorithmParameterSpec keyParams): SecretKey and

Funcion: generateKey(int keySize) : SecretKey do

Delegates the key generation to the underlying PRP.

* Call and return the underlying PRP relevant generateKey function.

Encryption functions:

CBC and CTR encryption schemes must use a random IV parameter to be secure. Many times the best option is to let the encryption scheme generate its own random IV, which will be returned as part of the ciphertext. However, in some cases (for ex. SSL) there is a need for the user to generate the IV and then pass it to the encryption scheme. We allow these options by providing two "encrypt" functions, each with different arguments. All of them encrypt the plaintext using the mode of operation algorithm and the prp member variable. They differ in the way they obtain the IV. They always return the used IV as part of the ciphertext.

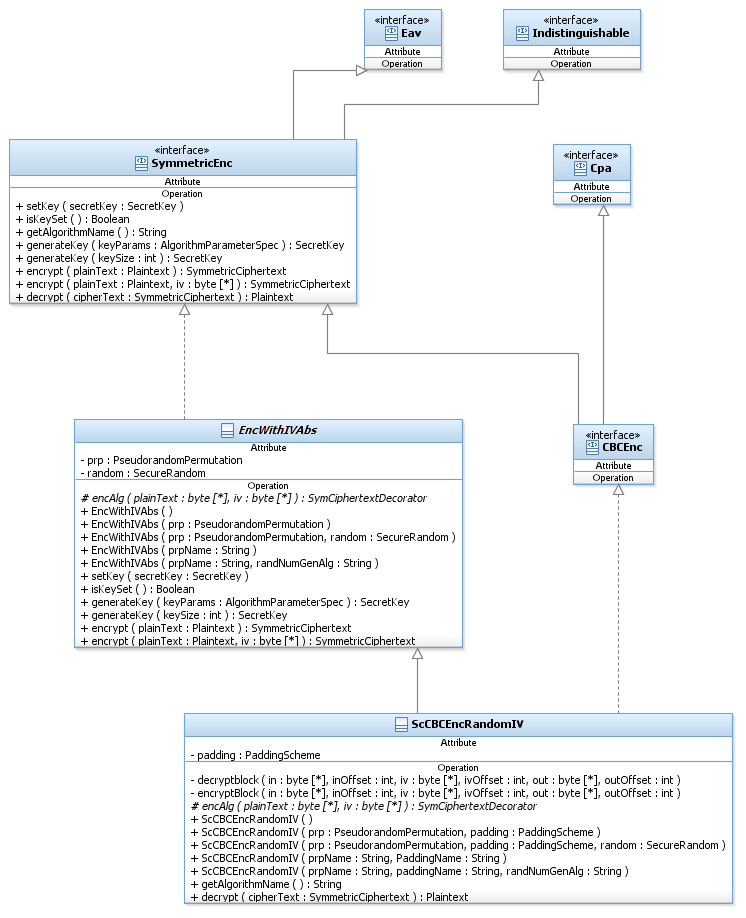
Funcion: encrypt (Plaintext plaintext): SymmetricCiphertext

* If SecretKey has not been set yet, throw IllegalStateException.
* Generate a new random IV of size equal to prp.getBlockSize. Use this.random.
* Return encrypt (plaintext, IV).

Funcion: encrypt (Plaintext plaintext, byte[ ] iv) : SymmetricCiphertext

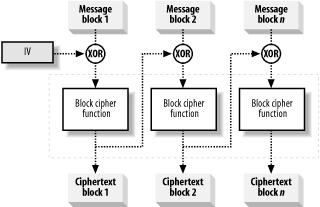
* If SecretKey has not been set yet, throw IllegalStateException.
* If the given IV is not of prp.getBlockSize length, throw IllegalBlockSizeException.
* If the given plaintext is not an instance of ByteArrayPlaintext throw IllegalArgumentException.
* Call encAlg (plaintext.getText(), iv) to obtain ciphertext.
* Return the ciphertext.

##### Static View – CBCEnc



##### Dynamic View – ScCBCEncRandomIV

The CBC encryption algorithm is depicted below (image taken from <http://programming4.us/security/1604.aspx>):



Constructor: ScCBCEncRandomIV (PseudoRandomPermutation prp, SecureRandom random, PaddingScheme padding) do

Sets the prp member variable to the one passed as argument. This means that from now on we have a specific CBC encryption scheme. For example, if the PRP is DES, then our object is a CBC-DES encryption scheme. We can encrypt different messages with different secret keys and parameters. To do so we use the “setKey” function presented in the abstract class.

* Call the relevant constructor of the super class.
* Set the padding member with the given padding scheme.

Function: decrypt (Ciphertext cipherText) : Plaintext

* If secretKey has not been set, throw IllegalStateException.
* Check if ciphertext is an instance of IVCiphertext. If not, throw IllegalArgumentException.
* iv= ciphertext.getIV()
* cipher = ciphertext.getBytes()
* Allocate a paddedPlaintext buffer of length ciphertext’s length
* paddedPlaintext[0] = prp.invert(cipher[0]) XOR iv
* For i from 1 to length of plaintext do:
* paddedPlaintext[i] : = prp.invert(cipher[i]) XOR ciphertext[i-1]
* Remove pad from the paddedPlaintext
* Return the plaintext.

Note: Each element in the loop is of size blockSize.

Function: encAlg (byte[] plainText, byte [] iv): IVCiphertext

This protected function must be implemented in this concrete class. In CBCEnc this function performs the CBC mode of operation:

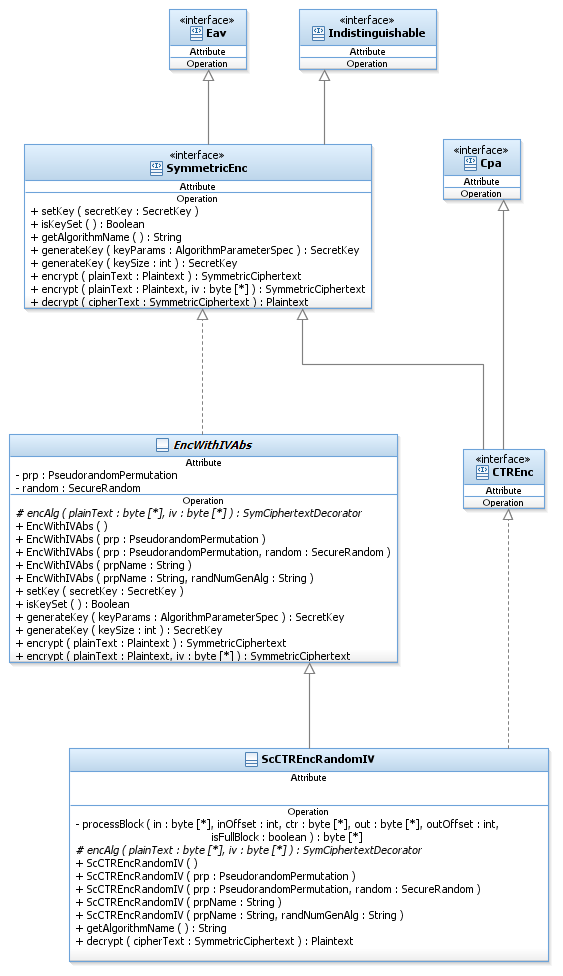
* pad the plaintext using the padding member.
* Allocate a byte[] ciphertext with the length of the padded plaintext.
* ciphertext[0] = prp.computeBlock(iv XOR paddedPlaintext[0])
* for next blocks in plaintext do: //i = 1
* ciphertext [i] = prp.computeBlock(ciphertext [i-1] XOR paddedPlaintext [i])
* Create an IVCiphertext with ciphertext and iv
* Return the IVCiphertext.

**Note:** The loop goes over blocks and not bytes therefore, plaintext[i] and cipher[i] refer to a block unit and not to a byte. For each PRP the blocks will be calculated according to the respective block size.

Function: getAlgorithmName (): String

* Return "CBCwith" + prp.getAlgorithmName();

##### Static View – CTREnc

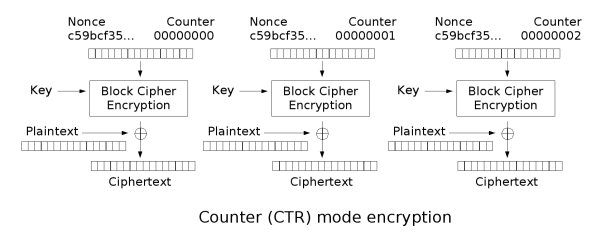


##### Dynamic View - ScCTREncRandomIV

Counter mode can be viewed as a way of generating a pseudorandom stream from a block cipher. First, a random IV ← {0, 1} n is chosen; the IV is denoted ***ctr****.* Then, a stream is generated by computing

ri :=Fk(ctr + *i*), (where ctr and *i* are viewed as integers and addition is performed modulo 2n). Finally, the i*th* ciphertext is computed as ci: = ri XOR mi, and the IV is sent as part of the ciphertext. (See [[2],](#_References) page 98) .

See figure below (taken from wiki) for a graphical description:



Constructor: ScCTREncRandomIV (PseudoRandomPermutation prp, SecureRandom random)

Sets the prp member variable to the one passed as argument. This means that from now on we have a specific CTR encryption scheme. For example, if the PRP is DES, then our object is a CTR-DES encryption scheme. We can encrypt different messages with different secret keys and parameter. To do so we use the “setKey” function presented in the abstract class.

* Call the relevant constructor of the super class.

Function: decrypt (Ciphertext ciphertext) : Plaintext

* If secretKey has not been set, throw IllegalStateException.
* Check if ciphertext is an instance of IVCiphertext. If not, throw IllegalArgumentException.
* Allocate a plaintext buffer of length ciphertext’s length.
* ctr = ciphertext.getIV
* For every block in ciphertext (i = 0 to n-1) do:
* Plaintext[i] : = ciphertext[i] XOR prp.computeBlock(ctr)
* ctr = ctr + 1 mod 2n
* Return the plaintext.

Function: encAlg (plainText : byte [], iv : byte []) : Ciphertext

This protected function must be implemented in this concrete class. In CTREnc this function performs the CTR mode of operation:

* Allocate a byte buffer cipher of length equal to the plaintext length.
* ctr = iv
* For each block in plaintext do: //i = 0
* cipher[i] = prp.computeBlock(ctr) XOR plaintext[i]
* ctr = ctr +1 mod 2n
* Create and return a ciphertext instance with cipher and iv.

**Note:** The loop goes over blocks and not bytes. Therefore, plaintext[i] and cipher[i] refer to a block unit and not to a byte. For each PRP the blocks will be calculated according to the respective block size.

Function: getAlgorithmName (): String

* Return "CTRwith" + prp.getAlgorithmName();

##### Usage

Sender usage:

//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);

//Sends cipher and secretKey to the decryptor.

Receiver usage:

//Create the same SymmetricEnc object as the sender’s encryption object, and set the key.

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

…

Plaintext plaintext = decryptor.decrypt(cipher);

//Get the plaintext bytes.

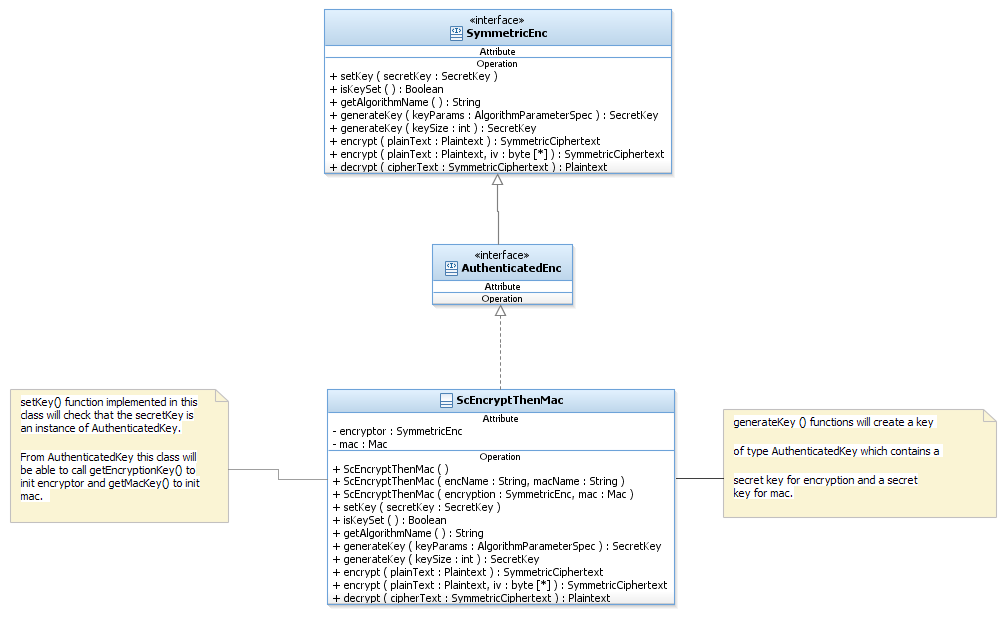
#### Authenticated Encryption

Authenticated encryption should be used when privacy and message integrity need to be achieved. Unfortunately, not all combinations of secure encryption schemes and secure message authentication codes provide these properties. There are three common approaches to combining encryption and message authentication.

1. Encrypt and authenticate
2. Authenticate then encrypt
3. Encrypt then authenticate.

Here we present the design for the third approach and call the class implementing it EncryptThenMac.

##### Static View



##### Dynamic View – ScEncryptThenMac

Constructor: ScEncryptThenMac(SymmetricEnc encryption, Mac mac) do

* If encryption is an instance of AuthenticatedEnc then throw exception IllegalArgumentException(“A symmetric encryption that is not an authenticated encryption is needed”).
* Set the encryption member to the given encryption.
* Set the mac member to the given mac.

Function: setKey(SecretKey secretKey) do

* Checks that the given secretKey is of type AuthenticatedKey. If not, throws InvalidKeyException(“This encryption requires a key of type AuthenticatedKey”)
* Calls encryptor’s setKey with a corresponding key and macs setKey with a corresponding key.

Function: IsKeySet():boolean do

* If the underlying encryption and mac objects are initialized with keys, return true.
* Else, return false.

Function: getAlgorithmName():String do

* Return "EncryptThenMacWith" + encryptor.getAlgorithmName() + "And" + mac.getAlgorithmName()

Function: generateKey(AlgorithmParameterSpec keyParams)

This function generates an authenticated key.

* If keyParams is not of type AuthEncKeyGenParameterSpec, throw InvalidParameterSpecException(“Key size has to be of type AuthEncKeyGenParameterSpec”)
* Generate encKey by calling encryptor.generateKey(keySize.getEncKeySize())
* Generate macKey by calling mac.generateKey(keySize.getMacKeySize())
* Create and return an EncThenMacKey object with encKey and macKey.

Function: generateKey (int keySize)

This function is not supported in authenticated encryption, since this encryption needs two key sizes, one for encryption key and one for mac key.

* Throw UnSupportedOperationException.

Function: encrypt (Plaintext plaintext):Ciphertext do

* If secretKey has not been set, throw IllegalStateException.
* cipher = encryptor.encrypt(plaintext)
* tag = mac (cipher.getBytes (), 0, cipher.getBytes().length)
* Create and return an EncMacCiphertext object with cipher and tag.

Function: encrypt (Plaintext plaintext, byte[] iv):Ciphertext do

* If secretKey has not been set, throw IllegalStateException.
* cipher = encryptor.encrypt(plaintext, iv)
* tag = mac (cipher.getBytes (), 0, cipher.getBytes().length)
* Create and return an EncMacCiphertext object with cipher and tag

Function: Decrypt(Ciphertext ciphertext) : Plaintext do

* If secretKey has not been set, throw IllegalStateException.
* If ciphertext is not an instance of EncMacCiphertext, throw IllegaStateException.
* Verify the tag using the mac object, if correct:
* Decrypt the message using the encryptor object
* Return the plaintext
* Else, return null.

##### Usage

Authenticated Encryption contains multiple underlying objects and therefore, the creation is as following:

//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 the created encryption and mac objects.

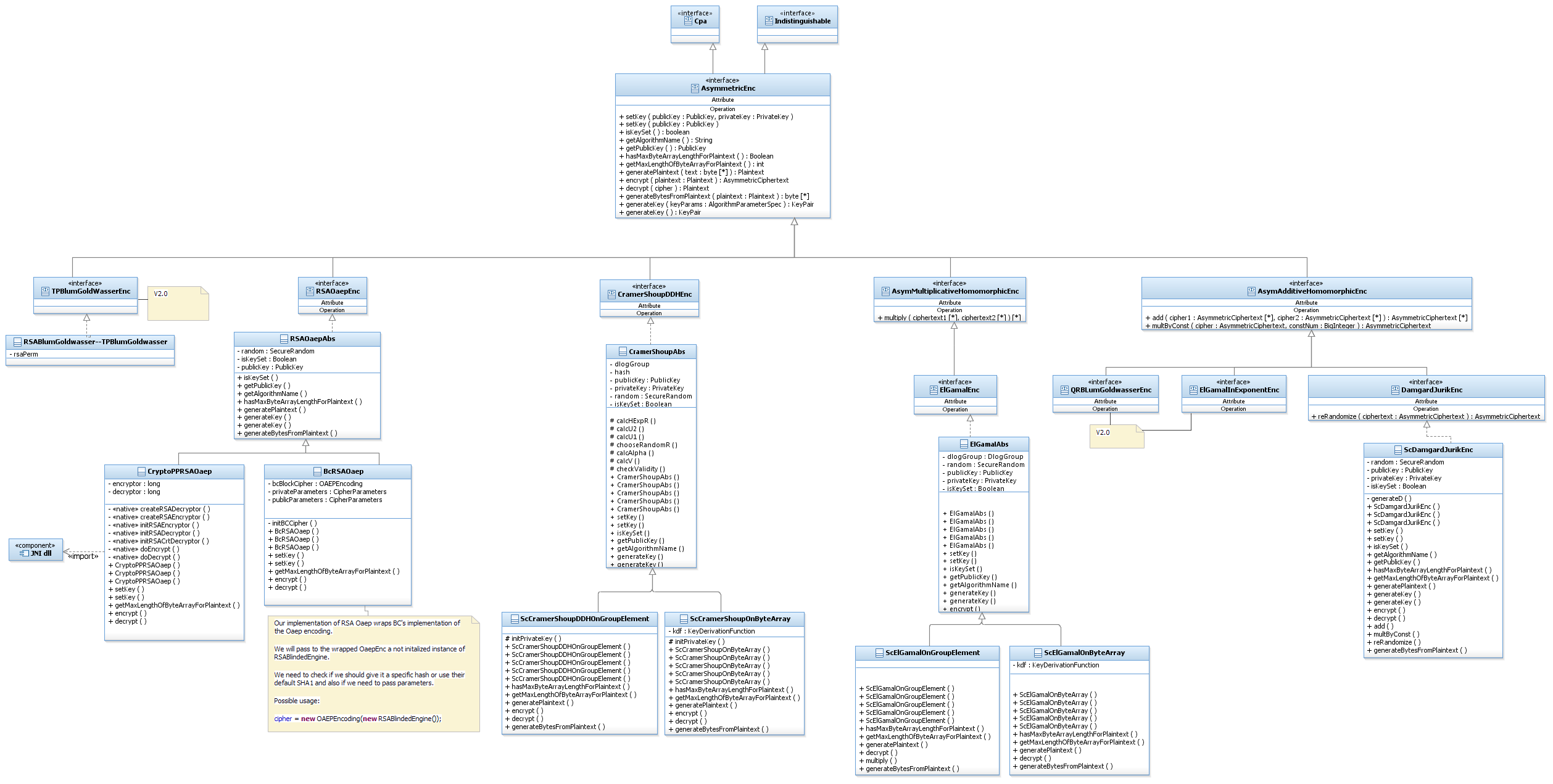
SymmetricEnc encThenMac = **new** ScEncryptThenMac(enc, cbcMac);

### Asymmetric encryption

The asymmetric 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.

#### Static View – General



#### Static View – El Gamal

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.

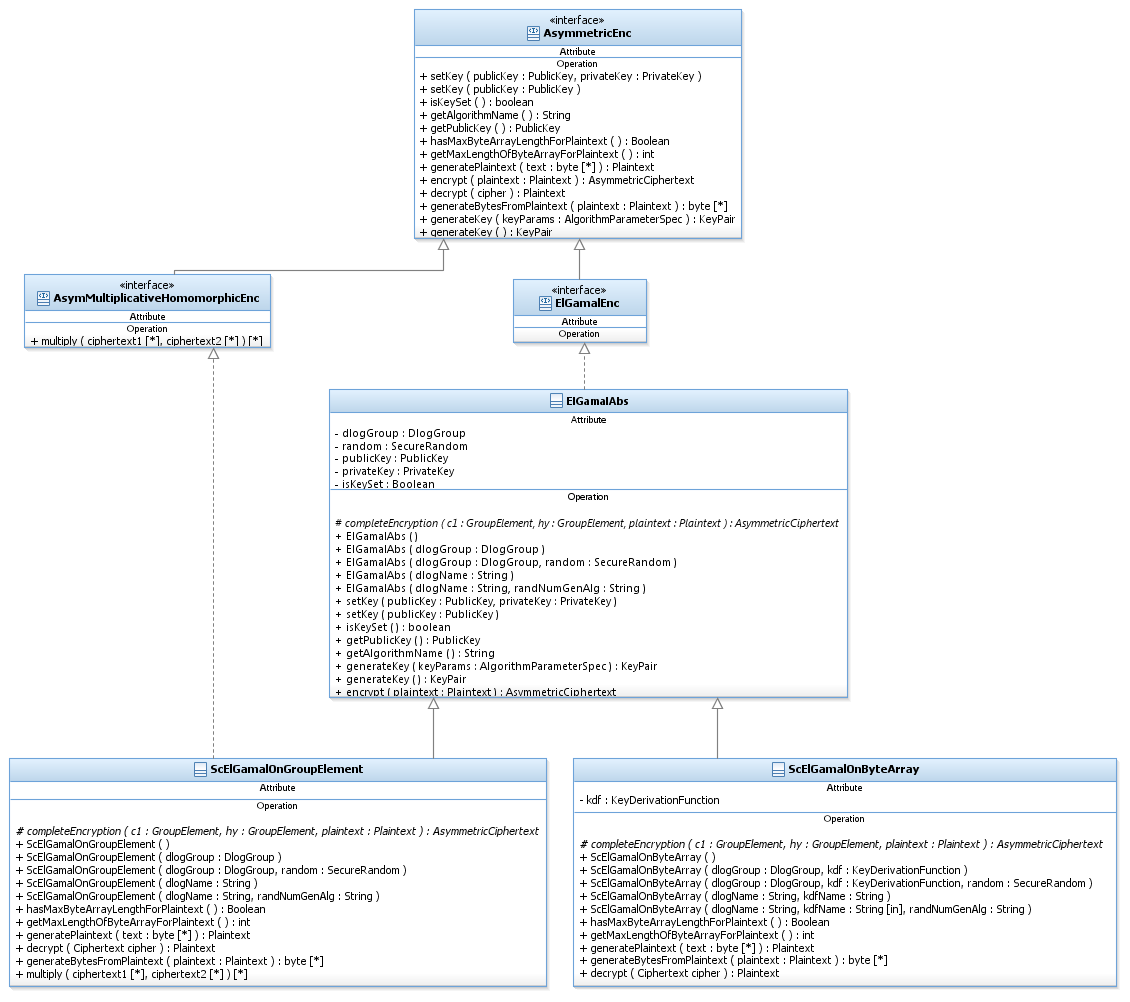
ElGamal encryption scheme can encrypt a group element and a byte array.

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.

There is some common functionality to both ElGamal types, for example, key generation. This functionality is implemented in an abstract class, ElGamalAbs, which the two concrete classes extend.

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



#### Dynamic View – ElGamalAbs

Constructor: ElGamalAbs(DlogGroup dlogGroup, SecureRandom random)

* If the given DlogGroup is not an instance of DDH security level, throw IllegalArgumentException.
* Set the dlog member variable to the given dlogGroup.
* Set the random member variable to the given random.

Function: setKey(PublicKey publicKey, PrivateKey privateKey) do

Initializes this ElGamal encryption scheme with (public, private) keys.

If private key is null, this encryption object can not decrypt and used for encryption only.

* If publicKey is not an instance of ElGamalPublicKey, throw InvalidKeyException.
* If privateKey is not null and is not an instance of ElGamalPrivateKey, throw InvalidKeyException.
* Set the given public key.
* If private key is not null, call initPrivateKey function.
* Set isKeySet member to true.

Function: setKey (PublicKey publicKey) do

Initializes this ElGamal encryption scheme with public key.

After this initialization, this encryption object can not decrypt and can be used for encryption only.

* Call setKey(publicKey, null)

Function: getAlgorithmName() : String do

* return “ElGamal/” + dlog.getGroupType()

Function: generateKey () : KeyPair do

* Given a Dlog Group (G, q, g) do:
* Choose a random x 🡨Zq
* Compute h = gx
* Set the public key part of the key pair to be h.
* Set the private key part of the key pair to be x.
* Return the key pair.

Function: generateKey (AlgorithmParameterSpec keyParams) : KeyPair do

In the case of El Gamal there are not any actual params that need to be passed since all it needs to generate the key is the Dlog Group, which was set upon construction.

* Throw UnSupportedOperationException.

Function: encrypt(Plaintext plaintext) : AsymmetricCiphertext do

* If no public key has been set, throw IllegalStateException.
* Choose a random y 🡨 Zq
* Calculate c1 = gy mod p //mod p operation are performed automatically by the group.
* Calculate hy = hy
* call and return completeEncryption abstract function with c1, hy and plaintext.

#### Dynamic View – ScElGamalOnGroupElement

Constructor: ScElGamalOnGroupElement(DlogGroup dlogGroup, SecureRandom random) do

* Call relevant super constructor.

Function: initPrivateKey(PrivateKey privateKey) do

ElGamalOGroupElement’s decrypt function can be optimized if instead of using the x value in the private key as is, we change it to be q-x, while q is the dlog group order. This function computes this change and saves the new private value as the private key member.

* Get x from privateKey
* Calculate xInv = dlog.getOrder().substract(x).
* Set the privateKey member a new instance of ScElGamalPrivateKey with xInv.

Function: hasMaxByteArrayLengthForPlaintext() : boolean do

ElGamal that encrypts GroupElement has a limit of the byte array length to generate a plaintext from.

* Return true.

Function: getMaxLengthOfByteArrayForPlaintext():int do

* Return dlog.getMaxLengthOfByteArrayForEncoding() .

Function: generatePlaintext(byte[] text):Plaintext do

Generates a plaintext to the encrypt function of this ElGamal type.

* If length of text exceeds maximum length allowed for this ElGamal, throw IllegalArgumentException.
* Create a groupElement from text using the dlog member.
* Create and return a GroupElementPlaintext with the created element.

Function: completeEncryption(GroupElement c1, GroupElement hy, Plaintext plaintext) : AsymmetricCiphertext do

* If plaintext is not an instance of GroupElementPlaintext throw IllegalArgumentException.
* Calculate c2 = hy \* plaintext.getMessage() mod p
* Create and return an ElGamalOnGroupElementCiphertext object with c1 and c2.

Function: decrypt(AsymmetricCiphertext ciphertext) : Plaintext

* If private key is null, cannot decrypt. Throw Keyexception.
* If ciphertext is not an instance of ElGamalOnGroupElementCiphertext throw IllegalArgumentException.
* Calculate s = ciphertext.getC1() ^ privateKey.getX()
* Calculate the inverse of s: invS = s ^ -1
* NOTE: The two steps above can be optimized because we keep as a secretKey the value of –x instead of x. Therefore, we calculates invS directly by calculating ciphertext.getC1() ^ privateKey.getX().
* Calculate m = ciphertext.getC2() \* invS
* m is a groupElement. Use it to create and return an instance of GroupElementPlaintext.

Function: generateBytesFromPlaintext(Plaintext plaintext):byte[] do

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 a byte array.

* If the given plaintext is not an instance of GroupElementPlaintext, throw IllegalArgumentException.
* Get the GroupElement from the plaintext.
* Call and return dlog.decodeGroupElementToByteArray with the group element.

Function: Multiply(AsymmetricCiphertext cipher1, AsymmetricCiphertext cipher2) : AsymmetricCiphertext do

Calculates the ciphertext resulting of multiplying two given ciphertexts, (cipher1 = enc(p1), cipher2 = enc(p2),) which is actually the result of encrypting the multiplication of p1 and p2. Denote c1 = (u1, v1); c2 = (u2, v2).

* If no public key has been set, throw IllegalStateException.
* If cipher1 or cipher2 is not an instance of ElGamalOnGroupElementCiphertext throw IllegalArgumentException.
* If one or more of u1, u2, v1, v2 is not a member of the dlog member, throw IllegalArgumentException.
* Choose a random w 🡨 Zq
* Compute u = gw\*u1\*u2

v = hw\*v1\*v2

* Create and return a new ElGamalOnGroupElementCiphertext with u and v.

#### Dynamic View – ScElGamalOnByteArray

Constructor: ScElGamalOnByteArray (DlogGroup dlogGroup, KeyDerivationFunction kdf, SecureRandom random) do

* Call relevant super constructor.
* Set the kdf member with the given kdf.

Function: initPrivateKey(PrivateKey privateKey) do

* Set the privateKey member with the given privateKey.

Function: hasMaxByteArrayLengthForPlaintext() : boolean do

ElGamal that encrypts a ByteArray has no limit of the byte array length to generate a plaintext from, because it uses a kdf to generate bytes in the required length.

* Return false.

Function: getMaxLengthOfByteArrayForPlaintext():int do

* Throw new NoMaxException().

Function: generatePlaintext(byte[] text):Plaintext do

Generates a plaintext for the encrypt function of this ElGamal type.

* Create and return a ByteArrayPlaintext with the given text.

Function: completeEncryption(GroupElement c1, GroupElement hy, Plaintext plaintext) : AsymmetricCiphertext do

* If plaintext is not an instance of ByteArrayPlaintext throw IllegalArgumentException.
* Get the bytes of the given hy using the dlog member.
* Calculate c2 = KDF(hyBytes) ^ plaintext.getText().
* Create and return an ElGamalOnByteArrayCiphertext object with c1 and c2.

Function: decrypt(AsymmetricCiphertext ciphertext) : Plaintext

* If private key is null, cannot decrypt. Throw KeyException.
* If ciphertext is not an instance of ElGamalOnByteArrayCiphertext, throw IllegalArgumentException.
* Calculate s = ciphertext.getC1() ^ privateKey.getX()
* Get the bytes of s using the dlog member.
* Calculate m = KDF(sBytes) ^ ciphertext.getC2()
* m is a ByteArray. Use it to create and return an instance of ByteArrayPlaintext.

Function: generateBytesFromPlaintext(Plaintext plaintext):byte[] do

Generates a 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.

* If the given plaintext is not an instance of ByteArrayPlaintext, throw IllegalArgumentException.
* Return the bytes of the plaintext.

#### Usage – ElGamalEnc

There are two types of ElGamal encryptions, one is on a GroupElement and the other is on a byte array. The example below shows the ElGamal on a GroupElement. Example for the second encryption type can be found as [CramerShoup usage section](#_Usage), which is quite the same.

Party1 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();

…

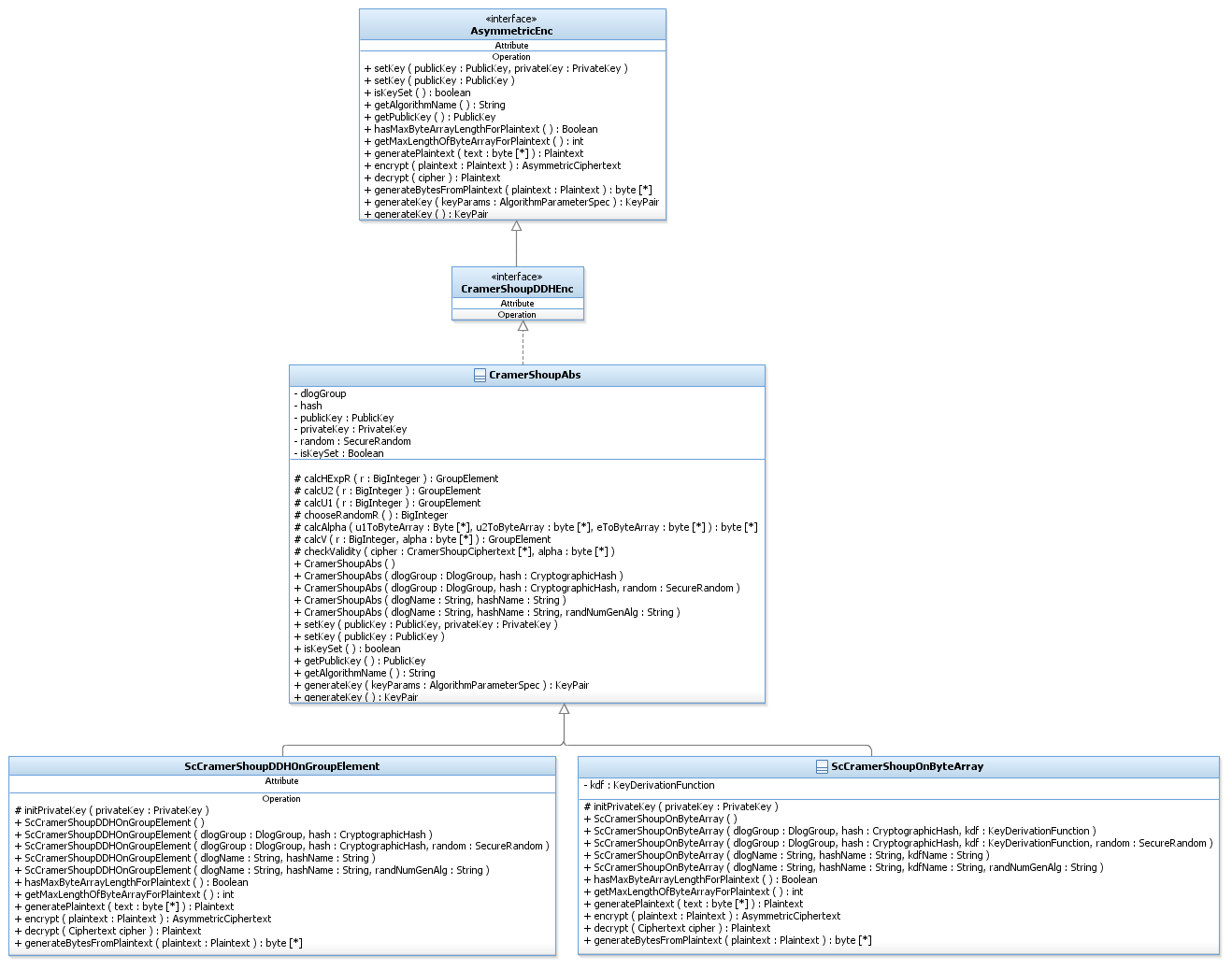
#### Static View – CramerShoupDDH

The Cramer Shoup encryption scheme’s security is based on the hardness of the decisional Diffie-Hellman (DDH) problem, like El Gamal encryption scheme. Cramer Shoup encryption can be defined over any cyclic group *G*. Its security depends upon the difficulty of a certain problem in *G* related to computing discrete logarithms. We implement Cramer Shoup over a Dlog Group (G, q, g) where q is the order of group G and g is the generator. In contrast to El Gamal, which is extremely malleable, Cramer–Shoup adds other elements to ensure non-malleability even against a resourceful attacker. This non-malleability is achieved through the use of a hash function and additional computations, resulting in a ciphertext which is twice as large as in El Gamal.

Similary to ElGamal, Cramer Shoup encryption scheme can encrypt a group element and a byte array.

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 ScCramerShoupDDHOnGroupElement, and the other one implements the encrypt function on a byte array and is called ScCramerShoupDDHOnByteArray.

There are some common functionalities to both Cramer Shoup types, which are implemented in an abstract class, CramerShoupAbs, which the two concrete classes extend.



#### Dynamic View – CramerShoupAbs

Constructor: CramerShoupAbs (DlogGroup dlogGroup, CryptographicHash hash, SecureRandom random) do

Our implementations of CramerShoupDDH encryption need a Dlog Group and CryptographicHash objects for the encryption.

* If the given dlogGroup is not an instance of DDH security level, throw IllegalArgumentException.
* If the given hash is not an instance of CollisionResistant security level, throw IllegalArgumentException.
* Set the given dlogGroup as dlogGroup class member.
* Set the given hash as hash class member.
* Set the given random as random class member.

Function: setKey(PublicKey publicKey, PrivateKey privateKey) do

Initializes this CramerShoup encryption scheme with (public, private) keys.

If private key is null, this encryption object can not decrypt and is used for encryption only.

* If publicKey is not an instance of CramerShoupPublicKey throw InvalidKeyException.
* Set the given public key.
* If privateKey is not null and is not an instance of ElGamalPrivateKey, throw InvalidKeyException.
* If privateKey is not null, call initPrivateKey function.
* Set isKeySet member to true.

Function: setKey (PublicKey publicKey) do

Initializes this CramerShoup encryption scheme with public key.

After this initialization, this encryption object can not decrypt and can be used for encryption only.

* Call setKey(publicKey, null)

Function: generateKey () : KeyPair do

* Given a Dlog Group (G, q, g) do:
* Choose two distinct, random generators g1, g2.
* Choose five random values (x1, x2, y1, y2, z) 🡨Zq.
* Compute , , .
* Set the public key part of the key pair to be c, d, h, g1, g2.
* Set the private key part of the key pair to be x1, x2, y1, y2, z.
* Return the key pair.

Function: generateKey (AlgorithmParameterSpec keyParams) : KeyPair do

In the case of CramerShoup there are not any actual params that need to be passed since all it needs to generate the key is the Dlog Group, which was set upon construction.

* Throw UnSupportedOperationException.

Function: getAlgorithmName():String do

* return “CramerShoup/” + dlog.getGroupType()

The next four functions are protected functions that are used in the encrypt or decrypt functions and are called from the implementation of the derived classes. The functionality the functions perform is equal to both types of CramerShoup encryption schemes.

Function: chooseRandomR() : BigInteger do

* Choose and return a random r 🡨 Zq.

Function: calcU1(BigInteger r) : BigInteger do

* Calculate and return u1 = g1r.

Function: calcU2(BigInteger r) :BigInteger

* Calculate and return u2 = g2r.

Function: calcHExpR(BigInteger r) : BigInteger do

* Calculate and return hr.

Function: calcAlpha(byte[] u1, byte[] u2, byte[] e) : byte[] do

Computes the hash function on the concatenation of the three given values.

* Allocate a byte[] msgToHash of length u1.length + u2.length + e.length.
* Concatenate u1, u2 and e into msgToHash.
* Call hash.update with msgToHash
* Allocate a byte[] alpha
* Call hash.hashFinal with alpha.
* Return alpha.

Function: calcV(BigInteger r, byte[] alpha) : GroupElement do

Calculate the v value of the encryption.

* Calculate and return v = crdrα

Function: checkValidity(CramerShoupCiphertext cipher, byte[] alpha) do

Validates that the given cipher is correct.

* Calculate
* If the result value is different from v, throw ScapiRuntimeException.

#### Dynamic View – ScCramerShoupDDHOnGroupElement

Constructor: ScCramerShoupDDHOnGroupElement (DlogGroup dlogGroup, CryptographicHash hash, SecureRandom random) do

* Call the relevant super constructor.

Function: initPrivateKey(PrivateKey privateKey) do

CramerShoupDDHOGroupElement’s decrypt function can be optimized if instead of using the z value in the private key as is, we change it to be q-z, while q is the dlog group order. This function computes this changing and saves the new private value as the private key member.

* Get z from privateKey
* Calculate zInv = dlog.getOrder().substract(z).
* Set the privateKey member to a new instance of ScCramerShoupPrivateKey with x1, x2, y1, y2 and zInv.

Function: hasMaxByteArrayLengthForPlaintext() : boolean do

CramerShoup that encrypts GroupElement has a limit of the byte array length to generate a plaintext from.

* Return true.

Function: getMaxLengthOfByteArrayForPlaintext():int do

* Return dlog.getMaxLengthOfByteArrayForEncoding().

Function: generatePlaintext(byte[] text):Plaintext do

Generates a plaintext for the encrypt function of this CramerShoup type.

* If the length of text exceeds maximum length allowed for this CramerShoup, throw IllegalArgumentException.
* Create a groupElement from text using the dlog member.
* Create and return a GroupElementPlaintext with the created element.

Function: Encrypt(Plaintext plaintext) : Ciphertext

* If no public key has been set, throw IllegalStateException.
* If plaintext is not an instance of GroupElementPlaintext throw IllegalArgumentException.
* Choose a random r 🡨 Zq by calling chooseRandomR function.
* Calculate u1 = g1r by calling calcU1 function

u2 = g2r by calling calcU1 function

hExpR = hr by calling calcHExpR function

e = hExpR \*msgEl

* Convert u1, u2, e to byte[] using the dlogGroup
* Compute α - the result of computing the hash function on the concatenation u1+ u2+ e. This is done by calling calcAlpha function.
* Calculate v = c^r\*d^(r\*α) by calling calcV function.
* Create and return a CramerShoupOnGroupElementCiphertext object with u1, u2, e and v.

Function: Decrypt(Ciphertext ciphertext) : Plaintext

* If private key is null, cannot decrypt. Throw KeyException.
* If cipher is not an instance of CramerShoupOnGroupElementCiphertext, throw IllegalArgumentException.
* Convert u1, u2, e to byte[] using the dlogGroup.
* Compute α - the result of computing the hash function on the concatenation of u1+ u2+ e. This is done by calling calcAlpha function.
* If != v throw exception. This is done by calling checkValidity function.
* Calculate m = e\*(u1z)-1 // equal to msg = e/u1z . We don’t have a divide

// operation in DlogGroup so we calculate it in

// equivalent way

**NOTE**: The step above can be optimized because we keep as a secretKey the value of –z instead of z. Therefore, we calculate (u1z)-1 directly by calculating ciphertext.getU1() ^ privateKey.getZ().

* m is a groupElement. Use it to create and return an instance of GroupElementPlaintext.

Function: generateBytesFromPlaintext(Plaintext plaintext):byte[] do

Generates a 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 a byte array.

* If the given plaintext is not an instance of GroupElementPlaintext, throw IllegalArgumentException.
* Get the GroupElement from the plaintext.
* Call and return dlogGroup.decodeGroupElementToByteArray with the group element.

#### Dynamic View – ScCramerShoupDDHOnByteArray

Constructor: ScCramerShoupDDHOnByteArray (DlogGroup dlogGroup, CryptographicHash hash, KeyDerivationFunction kdf, SecureRandom random) do

* Call relevant super constructor.
* Set the kdf member with the given kdf.

Function: initPrivateKey(PrivateKey privateKey) do

* Set the privateKey member with the given privateKey.

Function: hasMaxByteArrayLengthForPlaintext() : boolean do

CramerShoup that encrypts ByteArray has no limit of the byte array length to generate a plaintext from, because it uses a kdf to generate bytes in the required length.

* Return false.

Function: getMaxLengthOfByteArrayForPlaintext():int do

* Throw new NoMaxException().

Function: generatePlaintext(byte[] text):Plaintext do

Generates a plaintext to the encrypt function of this CramerShoup type.

* Create and return a ByteArrayPlaintext with the given text.

Function: Encrypt(Plaintext plaintext) : Ciphertext

* If no public key has been set, throw IllegalStateException.
* If plaintext is not an instance of ByteArrayPlaintext throw IllegalArgumentException.
* Choose a random r 🡨 Zq by calling chooseRandomR function.
* Calculate u1 = g1r by calling calcU1 function

u2 = g2r by calling calcU1 function

hExpR = hr by calling calcHExpR function

e = KDF(hExpR) XOR plaintext.getText().

* Convert u1, u2 to byte[] using the dlogGroup
* Compute α - the result of computing the hash function on the concatenation of u1+ u2+ e. This is done by calling calcAlpha function.
* Calculate v = = c^r\*d^(r\*α) by calling calcV function.
* Create and return a CramerShoupOnByteArrayCiphertext object with u1, u2, e and v.

Function: Decrypt(Ciphertext ciphertext) : Plaintext

* If private key is null, we cannot decrypt. Throw KeyException.
* If cipher is not an instance of CramerShoupOnByteArrayCiphertext, throw IllegalArgumentException.
* Convert u1, u2 to byte[] using the dlogGroup.
* Compute α - the result of computing the hash function on the concatenation of u1+ u2+ e. This is done by calling calcAlpha function.
* If != v throw exception. This is done by calling checkValidity function.
* Calculate m = KDF(u1z) XOR e.
* m is a ByteArray. Use it to create and return an instance of ByteArrayPlaintext.

Function: generateBytesFromPlaintext(Plaintext plaintext):byte[] do

Generates a 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 a byte array.

* If the given plaintext is not an instance of ByteArrayPlaintext, throw IllegalArgumentException.
* Return the bytes of the plaintext.

#### Usage - CramerShoupDDHEnc

There are two types of CramerShoup encryptions, one is on a GroupElement and the other is on a byte array. The example below shows the CramerShoup on a byte array. An example for the second encryption type can be found as [ElGamal usage section](#_Usage_1), which is quite the same.

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.getMaxByteArrayLengthForPlaintext()){

throw new IllegalArgumentException(“message too long”);

}

}

//Generate a plaintext suitable to this CramerShoup object.

Plaintext plaintext = encryptor.generatePlaintext(msg);

//Encrypt the plaintext

AsymmetricCiphertext cipher = encrypor.encrypt(plaintext);

//Send cipher and keys to the receiver.

Receiver usage:

//Create a CramerShoup object with the same DlogGroup definition as party1.

//Generate a keyPair using the CramerShoup object.

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.

…

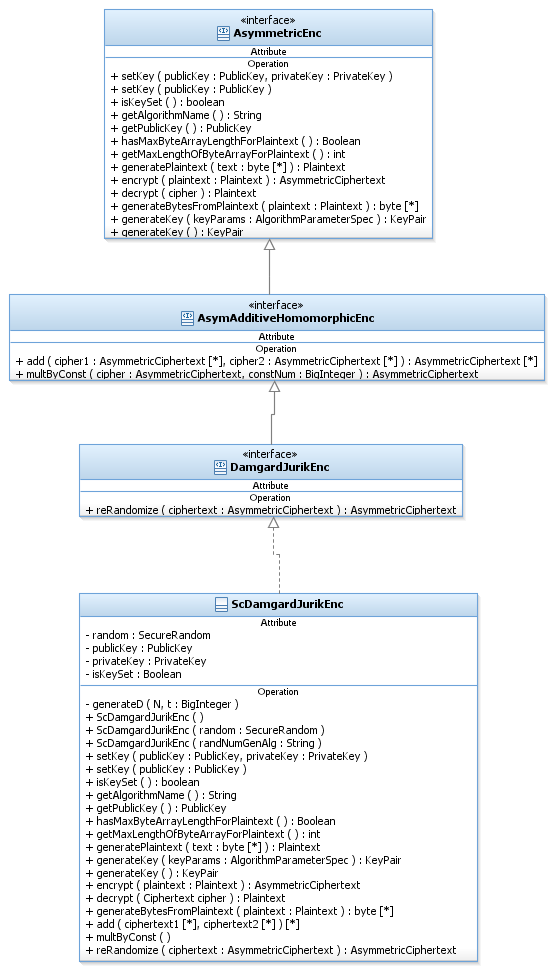
ByteArrayPlaintext plaintext = ((ByteArrayPlaintext)encryptor).decrypt(cipher);

//Get the plaintext bytes and use it as needed.

byte[] text = plaintext.getText();

…

#### Static View – DamgardJurikEnc

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

#### Dynamic View – DamgardJurikEnc

Constructor: ScDamgardJurikPaillierEnc (SecureRandom random) do

* Set the given random as the random member.

Function: setKey(PublicKey publicKey, PrivateKey privateKey) do

Initializes this encryption scheme with (public, private) keys.

* If publicKey is not an instance of DamgardJurikPublicKey throw InvalidKeyException.
* Set the given public key to the public key member.
* If privateKey is not null and is not an instance of DamgardJurikPrivateKey, throw InvalidKeyException.
* Set the given private key to the private key member.
* Set isKeySet member to true.

Function: setKey(PublicKey publicKey) do

Initializes this DamgardJurik encryption scheme with public key.

After this initialization, this encryption object can not decrypt andcan be used for encryption only.

* Call setKey(publicKey, null)

Function: getAlgorithmName():String do

* return “DamgardJurik”.

Function: hasMaxByteArrayLengthForPlaintext() : boolean do

DamagardJurik encryption has no limit on the BigInteger given to encrypt.

* Return false.

Function: getMaxLengthOfByteArrayForPlaintext():int do

* Throw new NoMaxException().

Function: generatePlaintext(byte[] text):Plaintext do

DamgardJurik encrypts BigIntegers.

* Create and return a BigIntegerPlaintext with the given text.

Function: generateKey (AlgorithmParameterSpec keyParams) : KeyPair do

DamgardJurik encryption needs the modulus length and certainty in order to generate a keyPair. These values are given in keyParams.

* If KeyParams is not an instance of DJKeyGenParameterSpec, throw InvalidParameterSpecException.
* Choose a random RSAModulus n = p\*q of the given length with the given certainty.
* Compute t=lcm(p-1, q-1) //lcm is the least common multiple and can be computed as .
* Call generateD function with n,t to precalculate for the case that s=1.
* Create ScDamgardJurikPublicKey with n
* Create ScDamgardJurikPrivateKey with t, d.
* Create and return a KeyPair with the created public and private keys.

Function: generateKey () : KeyPair do

DamgardJurik encryption scheme needs parameters to generate keys.

* Throw UnSupportedOperationException.

Function: generateD(BigInteger N, BigInteger t) : BigInteger do

Generates a value d such that d = 1modN and d = 0Modt, using the chinese remainder theorem.

* Create a vector of BigInteger congruences. Put in it 0 and 1.
* Create a congruences muduli. Put in it N and t.
* Call and return MathAlgorithms.cheineseRemainderTheorem with congruences and moduli.

Function: encrypt(Plaintext plaintext) : AsymmetricCiphertext do

* If no public key has been set, throw IllegalStateException.
* If plaintext is not an instance of BigIntegerPlaintext, throw IllegalArgumentException.
* Compute
* Compute N=n^s
* If plaintext.getX() is not in ZN throw IllegalArgumentException
* Compute N’=n^(s+1)
* Choose a random r 🡨 ZN’\*
* Compute c = ((1+n)^x\*r^N) mod N’
* Create and return BigIntegerCiphertext with c.

Function: decrypt(AsymmetricCiphertext ciphertext) : Plaintext do

* If privateKey is null, throw KeyException.
* If ciphertext is not an instance of BigIntegerCiphertext, throw IllegalArgumentException.
* Compute
* Compute N = ns
* N’ = ns+1
* If cipher.getCipher()is not in ZN’ throw IllegalArgumentException.
* Compute d:
* If s=1, get d from privateKey.
* Else, call generateD with N and t.
* Compute a= (cipher\*d)modN’
* Compute plaintext as the discrete logarithm of cd to the base (1+N)modN’ by the following computation:
* x=0
* for j=1 to s do:
* t1 =
* t2=x
* for k=2 to j do:
* x = x-1
* t2 = (t2\*x) mod nj
* t1 = (t1-) mod nj
* x=t1
* Create and return BigIntegerPlaintext with x.

Function: generateBytesFromPlaintext(Plaintext plaintext):byte[] do

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 a byte array.

* If the given plaintext is not an instance of BigIntegerPlaintext, throw IllegalArgumentException.
* Return the bytes of the BigInteger in the plaintext.

Function: reRandomized(AsymmetricCiphertext cipher): AsymmetricCiphertext do

This function takes an encryption of a plaintext (originalPlaintext) and returns a ciphertext that “looks” differnet but it is also an encryption of originalPlaintext. The given cipher should have been generated with the same public key as this encryption object’s public key.

* If no public key has been set, throw IllegalStateException.
* If ciphertext is not an instance of BigIntegerCiphertextm throw IllegalArgumentException.
* Compute
* Compute N = ns
* N’ = ns+1
* If cipher.getCipher()is not in ZN’ throw IllegalArgumentException.
* Choose a random r 🡨 ZN’\*
* Compute c = (cipher.getCipher()\*rN) mod N’
* Create and return BigIntegerCiphertext with c.

Function: add(AsymmetricCiphertext cipher1, AsymmetricCiphertext cipher2) : AsymmetricCiphertext do

Given two ciphertexts c1= enc(p1) and c2 = enc(p2), this function returns c1+c2 = enc(p1+p2). Both ciphers should have been generated with the same public key as this encryption object’s public key.

* If no public key has been set, throw IllegalStateException.
* If cipher1 or cipher2 is not an instance of BigIntegerCiphertext, throw IllegalArgumentException.
* Compute
* Compute
* If s1≠s2 throw IllegalArgumentException
* Compute N = ns1
* N’ = ns1+1
* c1 = cipher1.getCipher()
* c2 = cipher2.getCipher()
* If c1 or c2 is not in ZN’ throw IllegalArgumentException.
* Compute c = c1\*c2 mod N’
* Choose a random r 🡨 ZN’\*
* Compute c = (c\*rN) mod N’
* Create and return BigIntegerCiphertext with c.

Function: multByConst(AsymmetricCiphertext cipher, BigInteger constNumber) : AsymmetricCiphertext do

This function calculates the homomorphic multiplication of a ciphertext by a constant number. The given cipher should have been generated with the same public key as this encryption object’s public key.

* Compute
* Compute N = ns
* N’ = ns+1
* If cipher.getCipher()is not in ZN’ throw IllegalArgumentException.
* If constNumber is not in ZN throw IllegalArgumentException.
* Compute c = (cipher constNumber) mod N’
* Choose a random r 🡨 ZN’\*
* Compute c = (c\*rN) mod N’
* Create and return BigIntegerCiphertext with c.

#### Usage - DamgardJurikEnc

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

//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 usage:

//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.

…

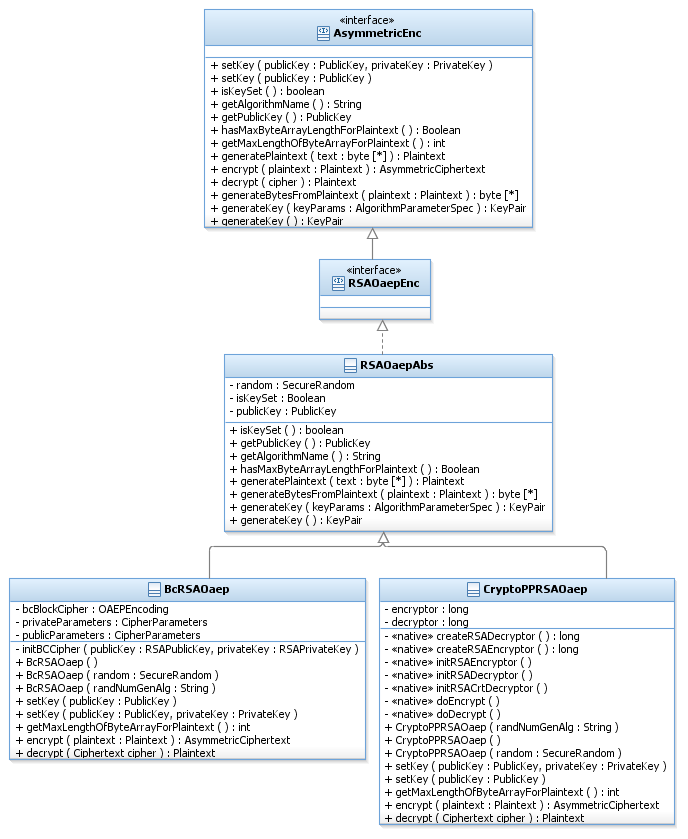
BigIntegerPlainText plaintext = (BigIntegerPlainText)elGamal.decrypt(cipher);

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

BigInteger element = plaintext.getX();

#### Static View – RSAOaepEnc

RSAES-OAEP is a public-key encryption scheme combining the RSA algorithm [39] with the Optimal Asymmetric Encryption Padding (OAEP) method.



#### Dynamic View – RSAOaepAbs

Function: getAlgorithmName() : String do

* return “RSA/OAEP”.

Function: hasMaxByteArrayLengthForPlaintext() : boolean do

RSA OAEP encryption has a limit on the byte array length to generate a plaintext

from.

* Return true.

Function: generatePlaintext(byte[] text):Plaintext do

Generates a ByteArrayPlaintext from the given text to the encrypt function of this RSAOaep.

* If the length of text exceeds maximum length allowed for this RSAOaep, throw IllegalArgumentException.
* Create and return a ByteArrayPlaintext with the given text.

Function: generateBytesFromPlaintext(Plaintext plaintext):byte[] do

Generates a 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 a byte array.

* If the given plaintext is not an instance of ByteArrayPlaintext, throw IllegalArgumentException.
* Return the bytes of the plaintext.

Function: generateKey(AlgorithmParameterSpec keyParams):KeyPair do

Generates RSA public and private keys using the KeyGenerator mechanism.

* If KeyParams is not an instance of RSAKeyGenParameterSpec, throw IllegalParameterSpecException.
* Get a KeyPairGenerator using KeyPairGenerator.getInstance(“RSA”).
* Call generator.init(keyParams, random).
* Return generator.generateKeyPair().

Function: generateKey():KeyPair do

RSA Oaep needs some parameters in order to generate keys.

* Throw UnSupportedOperationException.

#### Dynamic View – BcRSAOaep

Constructor: BcRSAOaep(SecureRandom random) do

* Sets the given random to the random member.
* Creates a new OAEPEncoding(new RSABlindedEngine()).
* Sets the created BC’s object to the bcBlockCipher member.

Function: setKey(PublicKey publicKey, PrivateKey privateKey) do

Initializes this RSAOaep encryption scheme with (public, private) keys.

If private key is null, this encryption object can not decrypt and can be used for encryption only.

* If publicKey is not an instance of RSAPublicKey, throw InvalidKeyException.
* If privateKey is not null and is not an instance of RSAPrivateKey, throw InvalidKeyException.
* Set the given public key.
* Call initBcCipher function with publicKey and PrivateKey.
* Set isKeySet member to true.

Function: setKey (PublicKey publicKey) do

Initializes this RSAOaep encryption scheme with public key.

After this initialization, this encryption object can not decrypt and can be used for encryption only.

* Call setKey(publicKey, null)

Function: initBCCipher(RSAPublicKey publicKey, RSAPrivateKey privateKey) do

Initializes BC OAEPEncoding object with the given key’s parameters and random. The initialization is to encryption mode.

* If privateKey is not null, create privateParameters using BcParameterTranslator.getInstance().translateParameters(privateKey, random).
* Create publicParameters using BcParameterTranslator.getInstance().translateParameters(publicKey, random).
* Call bcBlockCipher.init(forEncryption, publicParameters).

Function: getMaxLengthOfByteArrayForPlaintext() : int do

Returns the maximum byteArray length that can be passes as a plaintext.

* Return bcBlockCipher.getInputBlockSize()

Function: encrypt(Plaintext plaintext):AsymmetricCiphertext do

* If no public key has been set, throw IllegalStateException.
* If plaintext is not an instance of ByteArrayPlaintext, throw IllegalArgumentException.
* If bcBlockCipher is not initialized for encryption, initialize it to encryption.
* Compute ciphertext =bcBlockCipher.encodeBlock(plaintext).
* Create and return ByteArrayAsymCiphertext with ciphertext.

Function: decrypt(AstmmetricCiphertext ciphertext) : Plaintext do

* If privateParameters is null, cannot decrypt. Throw KeyException.
* If ciphertext is not an instance of ByteArrayAsymCiphertext, throw IllegalArgumentException.
* If bcBlockCipher is initialized for encryption, initialize it to decryption.
* Compute plaintext =bcBlockCipher.encodeBlock(ciphertext).
* Create and return ByteArrayPlaintext with plaintext.

#### Dynamic View – CryptoPpRSAOaep

Constructor: CryptoPpRSAOaep (SecureRandom random) do

* Set the given random to the random member.
* Call createRSAEncryptor native function and set the returned pointer to this.encryptor.
* Call createRSADecryptor native function and set the returned pointer to this.decryptor.

Function: setKey(PublicKey publicKey, PrivateKey privateKey) do

Initializes this RSAOaep encryption scheme with (public, private) keys.

If private key is null, this encryption object can not decrypt and can be used for encryption only.

* If publicKey is not an instance of RSAPublicKey, throw InvalidKeyException.
* If privateKey is not null and is not an instance of RSAPrivateKey, throw InvalidKeyException.
* Set the given public key.
* Call initRSAEncryptor native function with the pointer to the native encryptor, modulus and public exponent from the given public key.
* If privateKey is not null,
* If privateKey is an instance of RSAPrivateCrtKey, call initRSACrtDecryptor native function with the pointer to the native decryptor, modulus, public and private exponents, p, q, dp, dq and crt from the given keys.
* Else, call initRSADecryptor native function with the pointer to the native decryptor, modulus and public and private exponents from the given keys.
* Set isPrivateKeySet to true.
* Set isKeySet member to true.

Function: setKey (PublicKey publicKey) do

Initializes this RSAOaep encryption scheme with public key.

After this initialization, this encryption object can not decrypt and can be used for encryption only.

* Call setKey(publicKey, null)

Function: getMaxLengthOfByteArrayForPlaintext() : int do

Returns the maximum byteArray length that can be passes as a plaintext.

* Call and return getPlaintextLength native function with the pointer to the native encryption object.

Function: encrypt(Plaintext plaintext):AsymmetricCiphertext do

* If no public key has been set, throw IllegalStateException.
* If plaintext is not an instance of ByteArrayPlaintext, throw IllegalArgumentException.
* Call doEncrypt native function with the pointer to the native encryptor and plaintext.getText()
* Create and return ByteArrayAsymCiphertext with the returned ciphertext.

Function: decrypt(AstmmetricCiphertext ciphertext) : Plaintext do

* If private key is null, we cannot decrypt. Throw KeyException.
* If ciphertext is not an instance of ByteArrayAsymCiphertext, throw IllegalArgumentException.
* Call doDecrypt native function with the pointer to the native decryptor and ciphertext.getBytes().
* Create and return ByteArrayPlaintext with the returned plaintext.

#### Usage - RSAOaepEnc

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

Sender usage:

//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.hasMaxByteArrayLengthForPlaintext()){

If (msg.length>encryptor.getMaxByteArrayLengthForPlaintext()){

throw new IllegalArgumentException(“message too long”);

}

}

//Generate a plaintext suitable to this RSAOaep object.

Plaintext plaintext = encryptor.generatePlaintext(msg);

//Encrypt the plaintext

AsymmetricCiphertext cipher = encrypor.encrypt(plaintext);

//Send cipher and keys to the receiver.

Receiver usage:

//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();

…

#### Usage - Asymmetric encryption

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

* 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.

A code example for that usage can be found at the Asymmetric encryptions concrete implementations usage sections, for example, [DamgardJurik usage](#_Usage_-_DamgardJurikEnc).

* 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.

If the protocol can encrypt a message with any length, it is recommended that it will check in the constructor that the given encryption object has no length limitation. If the given encryption object has a length limitation, the protocol should throw an exception. This is better than computing the whole protocol steps and to find out in the end that the message to encrypt is too long.

The recommended usage in these cases is as followed:

//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 = encrypor.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 encryption 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.

…

Plaintext plaintext = encryptor.decrypt(cipher);

//Get the plaintext bytes **using the encryption object** and use it as needed.

byte[] text = encryptor.generatesBytesFromPlaintext(plaintext);

…

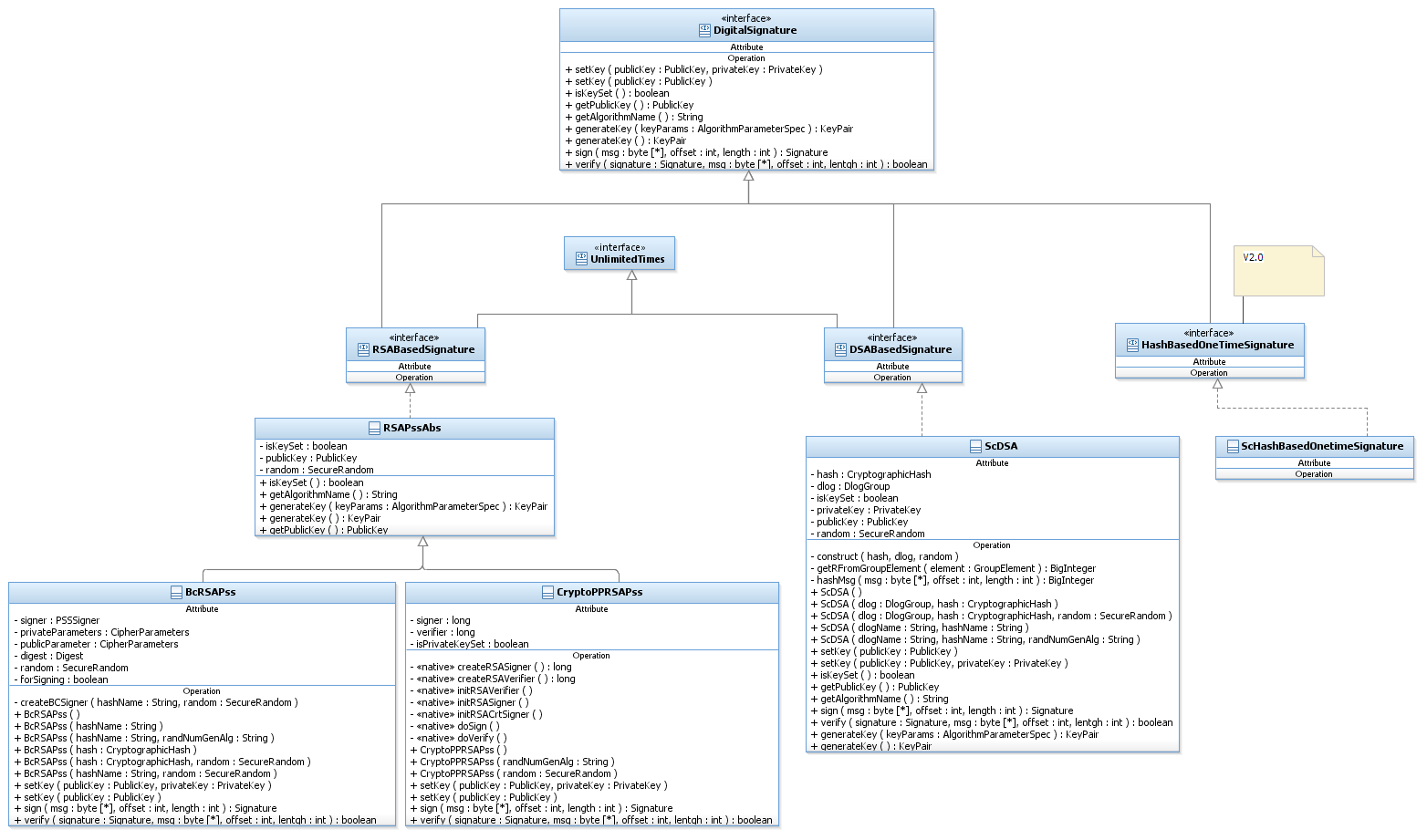
### Digital Signatures

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.

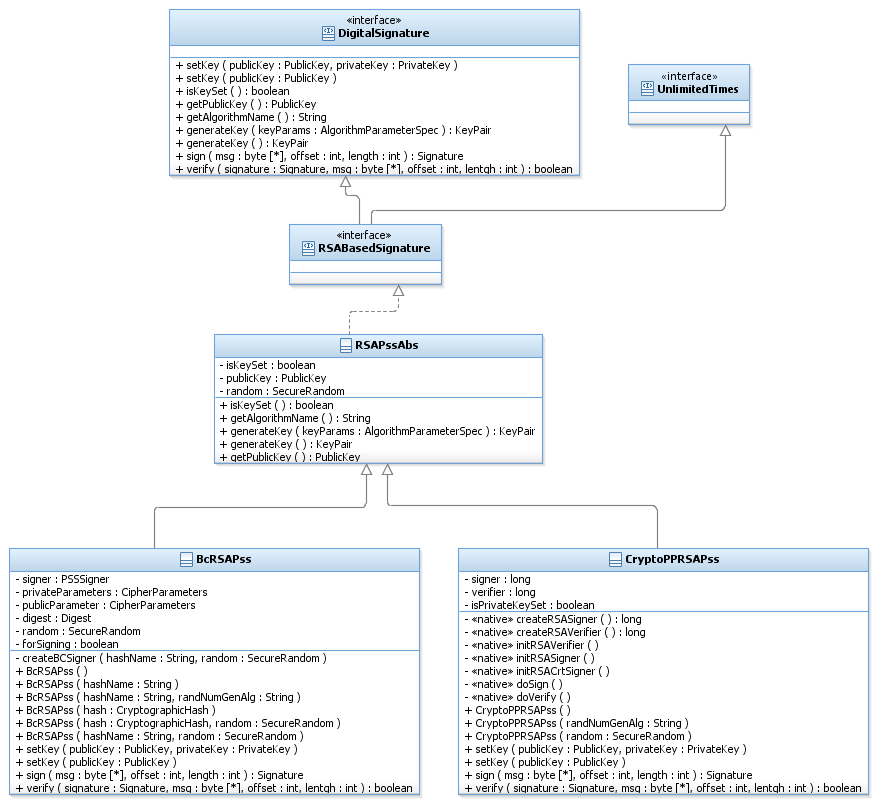
The Digital Signatures family of classes implements three main functionalities that correspond to the cryptographer’s language in which an encryption scheme is composed of three algorithms:

1. Generation of the keys.
2. Signing a message.
3. Verifying a signature with a message.

#### Static View – General



#### Static View – RSAPss



#### Dynamic View – RSAPssAbs

Function: getAlgorithmName() : String do

* return “RSA/PSS”.

Function: generateKey(AlgorithmParameterSpec keyParams):KeyPair do

Generates RSA public and private keys using the KeyGenerator mechanism.

* Get a KeyPairGenerator using KeyPairGenerator.getInstance(“RSA”)
* Call generator.init(keyParams, random)
* Return generator.generateKeyPair()
* If an InvalidAlgorithmParameterSpecException was thrown, throw new InvalidAlgorithmParameterSpec("KeyParams should be an instance of RSAKeyGenParameterSpec").

Function: generateKey():KeyPair do

RSA Pss needs some parameters in order to generate keys.

* Throw UnSupportedOperationException

#### Dynamic View – BcRSAPss

Constructor: BcRSAPss (String hashName, SecureRandom random) do

* Call createBCSigner(hashName, random)

Function: createBCSigner(String hashName, SecureRandom random) do

* Create a digest using BCFactory and the given hashName.
* Sets the given random to the random member.
* Create BC RSABlindedEngine.
* Create new PSSSigner with the created rsa and digest.
* Set the created BC’s signer object to the signer member.

Function: setKey(PublicKey publicKey, PrivateKey privateKey) do

Initializes this RSAPss scheme with (public, private) keys.

If private key is null, this object can not sign and can be used for verification only.

In any case, the initialization is for verification mode.

* If publicKey is not an instance of RSAPublicKey, throw InvalidKeyException.
* If privateKey is not null and is not an instance of RSAPrivateKey, throw InvalidKeyException.
* Set the given public key.
* Create publicParameters using BcParameterTranslator.getInstance().translateParameters(publicKey, random).
* If privateKey is not null, create privateParameters using BcParameterTranslator.getInstance().translateParameters(privateKey, random).
* Call signer.init(forSigning, publicParameters).
* Set isKeySet member to true.

Function: setKey (PublicKey publicKey) do

Initializes this RSAPss scheme with public key.

After this initialization, this object can not sign and can be used for verification only.

* Call setKey(publicKey, null)

Function: sign(byte[] msg, int offset, int length) : Signature do

* If privateParameters is null, cannot sign. Throw KeyException.
* If offset or length is not valid, throw ArrayIndexOutOfBoundException.
* If signer is not initialized for signing, initialize it to signing.
* Call signer.update(msg, offset, length)
* Compute byte[] signature = signer.generateSignature()
* Create and return RSASignature object with the signature array.

Function: Verify(Signature signature, byte[] msg, int offset, int length) : Boolean do

* If no public key has been set, throw IllegalStateException.
* If signature is not an instance of RSASignature, throw IllegalArgumentException.
* If offset or length is not valid, throw ArrayIndexOutOfBoundException.
* If signer is initialized for signing, initialize it to verifying.
* Call signer.update(msg, offset, length)
* Call and return signer.verifySignature with the given signature bytes

#### Dynamic View – CryptoPpRSAPss

Constructor: CryptoPpRSAPss (SecureRandom random) do

* Set the given random to the random member.
* Call createRSASigner native function to create native signer object and set the returned pointer to this.signer.
* Call createRSAVerifier native function to create native verifier object and set the returned pointer to this.verifier.

Function: setKey(PublicKey publicKey, PrivateKey privateKey) do

Initializes this RSAPss scheme with (public, private) keys.

If private key is null, this object can not sign and can be used for verification only.

* If publicKey is not an instance of RSAPublicKey, throw InvalidKeyException.
* If privateKey is not null and is not an instance of RSAPrivateKey, throw InvalidKeyException.
* Set the given public key.
* Call initRSAVerifier native function with the pointer to the native verifier, modulus and public exponent from the given public key.
* If privateKey is not null,
* If privateKey is an instance of RSAPrivateCrtKey, call initRSACrtSigner native function with the pointer to the native signer, modulus, public and private exponents, p, q, dp, dq and crt from the given keys.
* Else, call initRSASigner native function with the pointer to the native signer, modulus and public and private exponents from the given keys.
* Set isPrivateKeySet to true.
* Set isKeySet member to true.

Function: setKey (PublicKey publicKey) do

Initializes this RSAPss scheme with public key.

After this initialization, this object can not sign and cbe used for verification only.

* Call setKey(publicKey, null)

Function: sign(byte[] msg, int offset, int length) : Signature do

* If privateKey is null, we cannot sign. Throw KeyException.
* If offset or length is not valid, throw ArrayIndexOutOfBoundException.
* If offset is greater than 0, copy the relevant bytes to a new message array (the native function that performs sign needs the message to start at offset 0).
* Call doSign native function with the pointer to the native signer object, message and length.
* Create and return RSASignature object with the returned signature bytes.

Function: Verify(Signature signature, byte[] msg, int offset, int length) : Boolean do

* If no public key has been set, throw IllegalStateException.
* If signature is not an instance of RSASignature throw IllegalArgumentException.
* If offset or length is not valid throw ArrayIndexOutOfBoundException.
* If offset is greater than 0, copy the relevant bytes to a new message array (the native function that performs verification needs the message to start at offset 0).
* Call and return doVerify native function with the pointer to the native verifier object, signature bytes, message and length.

#### Usage – RSAPss

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)){

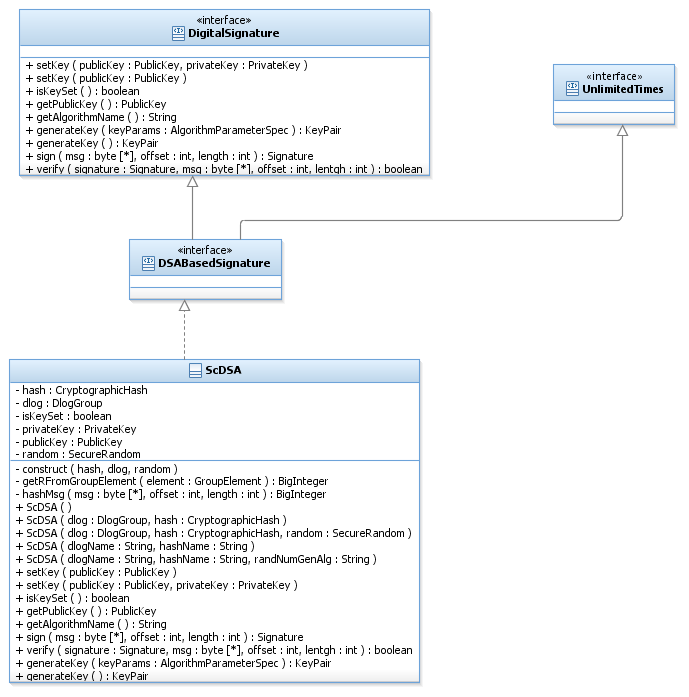
Throw new IllegalArgumentException(“the message is not verified!”);

}

//Message verified, continue working with it.

…

#### Static View – DSA



#### Dynamic View – ScDSA

Construrctor: ScDSA(CryptographicHash hash, DlogGroup dlog, SecureRandom random) do

* Call construct(hash, dlog, random).

Function: construct(CryptographicHash hash, DlogGroup dlog, SecureRandom random) do

* Set the given hash to the hash member.
* Set the given dlog to the dlog member.
* Set the given random to the random member.

Function: setKey(PublicKey publicKey, PrivateKey privateKey) do

Initializes this DSA scheme with (public, private) keys.

If private key is null, this object can not sign and can be used for verification only.

In any case, the initialization is for verification mode.

* If publicKey is not an instance of DSAPublicKey, throw InvalidKeyException.
* If privateKey is not null and is not an instance of DSAPrivateKey, throw InvalidKeyException.
* Set the given public key to the publicKey member.
* If privateKey is not null, set it to the privateKey member.
* Set isKeySet member to true.

Function: setKey (PublicKey publicKey) do

Initializes this DSA scheme with public key.

After this initialization, this object can not sign and can be used for verification only.

* Call setKey(publicKey, null).

Function: getAlgorithmName() : String do

* Return “DSA/” + dlog.getGroupType().

Function: generateKey ( ) : KeyPair do

Generates DSA public and private keys.

* Given a Dlog Group (G, q, g) do:
* Choose a random x 🡨Zq\*
* Compute y = gx
* Set the public key part of the key pair to be y.
* Set the private key part of the key pair to be x.
* Return the key pair.

Function: generateKey (AlgorithmParameterSpec keyParams) : KeyPair do

In the case of DSA there are not any actual params that need to be passed since all it needs for generating the key is the Dlog Group, which was set upon construction.

* Throw UnSupportedOperationException.

Function: sign(byte[] msg, int offset, int length) : Signature do

* If privateKey is null, cannot sign. Throw KeyException.
* If offset or length is not valid throw ArrayIndexOutOfBoundException.
* Compute s=0

r=0

* While s=0
* While r=0
  + Choose a random k 🡨Zq\*
  + Compute h = gk
  + Compute r, the BigInteger value of h, by calling getRFromGroupElement function with h.
* Compute z = H(msg) and return the left Lq bits of the result (Lq is the bit length of the group order q) as BigInteger, by calling hashMsg(msg, offset, length) function.
* Calculate s = k-1(z+xr)mod q.
* Create and return new DSASignature with r and s.

Function: HashMsg(byte[] msg, int offset, int length) : BigInteger do

Hash the given message using the CryptographicHash class member and return the left Lq bits of the result as BigInteger.

Lq is the bit length of the group order.

* Call hash.update(msg, offset, length)
* Call hash.hashFinal() and put the result in a byte[].
* If the result array length is greater than the byte length of the group order, copy the relevant bytes of the result and create a BigInteger z from them.
* Else, create a BigInteger z from the result bytes.
* Return z.

Function: getRFromGroupElement(GroupElement element) : BigInteger do

Computes a BigInteger value from the given GroupElement.

* If element is an instance of ZpElement, calculate r = the element value.
* If element is an instance of ECElement, calculate r = the x coordinate of the point.
* r = r mod q.
* Return r.

Function: verify(Signature signature, byte[] msg, int offset, int length) : Boolean do

* If no public key has been set, throw IllegalStateException.
* If signature is not an instance of DSASignature, throw IllegalArgumentException.
* If offset or length is not valid, throw ArrayIndexOutOfBoundException.
* Get r ans s from the given signature.
* If r or s is not in Zq\*, return false.
* Compute w = s-1 mod q.
* Compute e = H(msg) and return the left lq bits of e. This is done by calling hashMsg(msg, offset, length) function and set the return value to z.
* Compute u1 = z\*w mod q

u2 = r\*w mod q

v = gu1yu2.

* Get the BigInteger value of v, vBI, by calling getRFromGroupElement(v) function.
* If r equals vBI, return true.
* Else return false.

#### Usage – DSA

Sender usage:

//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:

//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.

…

# Performance

As in the primitives' layer, some of the algorithms were implemented both in Java and in C++. Mostly, the native implementation is faster than the Java implementation, although there are some JNI calls.

# Multi-Platform Issues

See multi-platform issues in [[1](#_References)].

# Benchmarking

Following are benchmarking for algorithms which we have multiple implementations for.

Times is measured in seconds.

RSA OAEP encryption

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Implementation/iterations | 10 | 100 | 1000 | 10000 |
| BcRSAOaep | 0.003 | 0.022 | 0.086 | 0.553 |
| CryptoPpRSAOaep | 0.001 | 0.004 | 0.032 | 0.297 |

RSA OAEP decryption

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Implementation/iterations | 10 | 100 | 1000 | 10000 |
| BcRSAOaep | 0.028 | 0.263 | 2.527 | 25.735 |
| CryptoPpRSAOaep | 0.004 | 0.038 | 0.358 | 3.537 |

RSA PSS signing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Implementation/iterations | 10 | 100 | 1000 | 10000 |
| BcRSAPss | 0.029 | 0.269 | 2.546 | 25.438 |
| CryptoPpRSAPss | 0.005 | 0.037 | 0.357 | 3.498 |

RSA PSS verification

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Implementation/iterations | 10 | 100 | 1000 | 10000 |
| BcRSAPss | 0.002 | 0.008 | 0.54 | 0.441 |
| CryptoPpRSAPss | 0 | 0.002 | 0.022 | 0.220 |

# Open Issues