SCAPI Non-Interactive Crypto Mid-Layer

R&D Group

Software Design Description

Written by Yael Ejgenberg

Approved by

Created 10 April 2011

Modified 10 April 2011

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 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1. Scope 5](#_Toc294537824)

[1.1 Introduction 5](#_Toc294537825)

[1.2 Goals 5](#_Toc294537826)

[2. Definitions Acronyms and Abbreviations 5](#_Toc294537827)

[3. References 5](#_Toc294537828)

[4. Wide Design Decisions 5](#_Toc294537829)

[5. Architectural design 6](#_Toc294537830)

[5.1 General 6](#_Toc294537831)

[5.1.1 Main interfaces 6](#_Toc294537832)

[5.2 High Level Description 10](#_Toc294537833)

[5.2.1 General Static View 10](#_Toc294537834)

[5.3 Detailed Description 11](#_Toc294537835)

[5.3.1 CBC-MAC 12](#_Toc294537836)

[5.3.1.1 Static View 12](#_Toc294537837)

[5.3.1.2 Dynamic View 14](#_Toc294537838)

[5.3.1.2.1 In CbcMacPrepending::CbcMacPrepending (PseudrandomPermutation p) do: 14](#_Toc294537839)

[5.3.1.2.2 In CbcMacPrepending::startMac do: 14](#_Toc294537840)

[5.3.1.2.3 In CbcMacPrepending::update do: 14](#_Toc294537841)

[5.3.1.2.4 In CbcMacPrepending:: doFinal do: 14](#_Toc294537842)

[5.3.1.2.5 In CbcMacPrepending::mac do: 15](#_Toc294537843)

[5.3.1.2.6 In CbcMacImp::verify do: 15](#_Toc294537844)

[5.3.1.2.7 In CbcMacPrepending::compute do: 15](#_Toc294537845)

[5.3.1.2.8 In CbcMacPrepending::getAlgorithmName 15](#_Toc294537846)

[5.3.1.3 Usage 15](#_Toc294537847)

[5.3.2 HMAC 16](#_Toc294537848)

[5.3.2.1 Static View 16](#_Toc294537849)

[5.3.2.2 Dynamic View 17](#_Toc294537850)

[5.3.2.2.1 In BcHmac::computeBlock do 17](#_Toc294537851)

[5.3.2.2.2 In BcHmac::mac do 17](#_Toc294537852)

[5.3.2.2.3 In BcHmac::verify do 17](#_Toc294537853)

[5.3.2.2.4 In BcHmac::update do 17](#_Toc294537854)

[5.3.2.2.5 In BcHmac::doFinal do 17](#_Toc294537855)

[5.4 External interface 23](#_Toc294537856)

[5.4.1 Class 1 API 33](#_Toc294537857)

[5.4.2 Class 2 API 33](#_Toc294537858)

[5.5 Components - Static view 34](#_Toc294537859)

[5.5.1 Functionality 1 34](#_Toc294537860)

[5.5.2 Functionality 2 34](#_Toc294537861)

[5.6 Data Structure Design 34](#_Toc294537862)

[6. User interface design 34](#_Toc294537863)

[7. Performance 34](#_Toc294537864)

[8. Multi-Platform Issues 34](#_Toc294537865)

[9. Backward-Compatibility 34](#_Toc294537866)

[10. Benchmarking 34](#_Toc294537867)

[Open Issues 34](#_Toc294537868)

# 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 RSAEncryption uses RSAPermutation and CBC-MAC uses any of the PRPs defined in the first level.

## Goals

*Explain the goals of this product. If many, consider a sub-numbering layout.*

# Definitions Acronyms and Abbreviations

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

# 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] 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, therefor 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 than what was presented in the first layer and also different than 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 KeyPairGeneratos 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.

### We allow the user to choose between providing her 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.

### 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

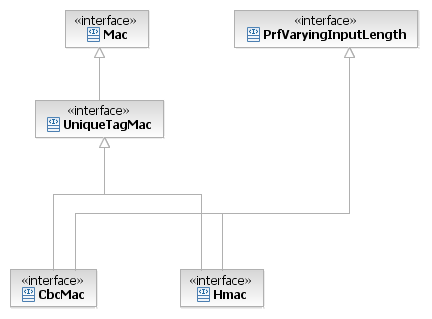
* edu.biu.scapi.midLayer.mac
* edu.biu.scapi.midLayer.symmetricEnc
* edu.biu.scapi.midLayer.asymmetricEnc
* edu.biu.scapi.midLayer.digitalSignature
* edu.biu.scapi.securityLevel

# Architectural design

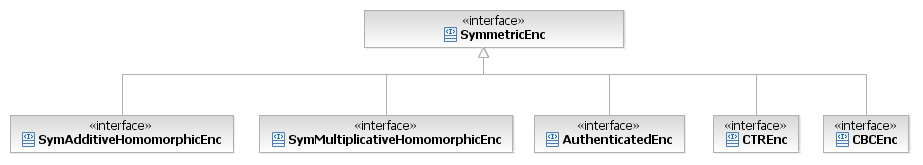
## General

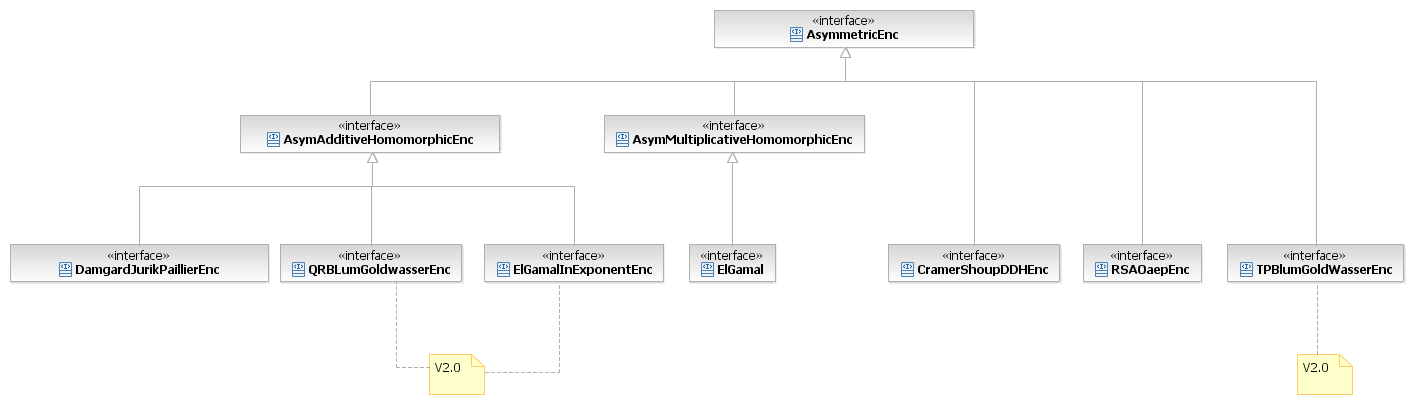
### Main interfaces

#### Message authentication codes

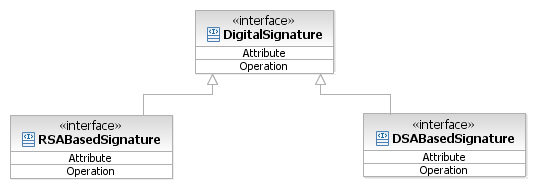


#### Encryption Types

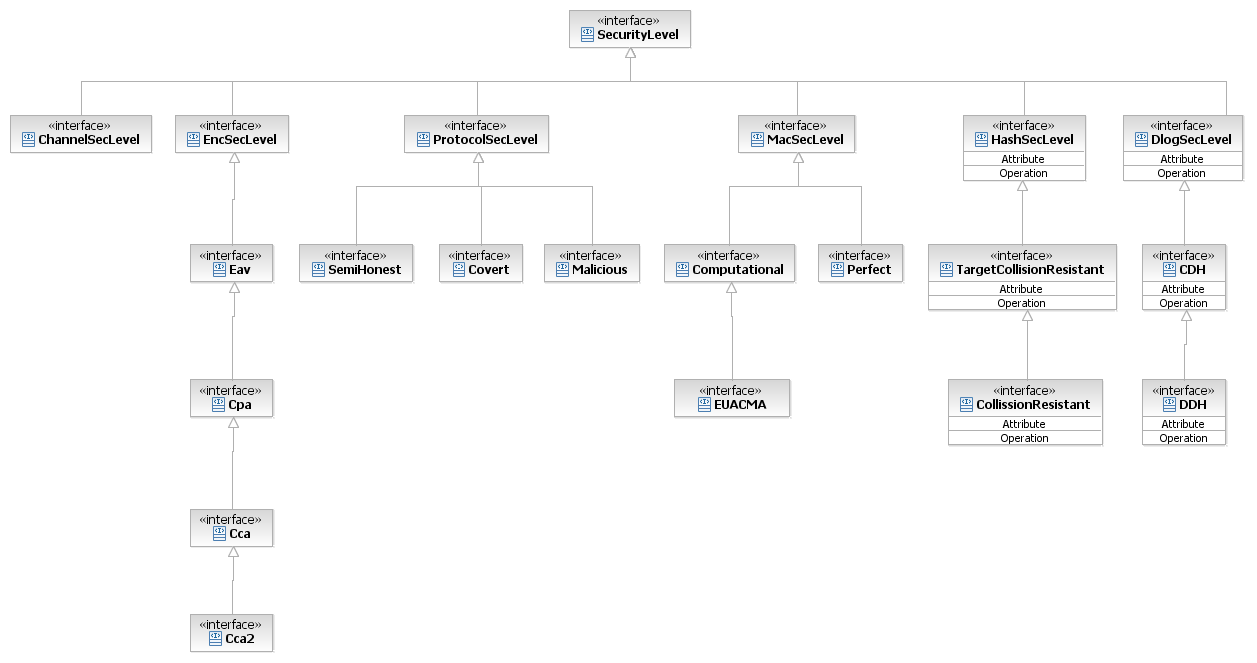




#### Digital Signatures

****

#### Security Levels

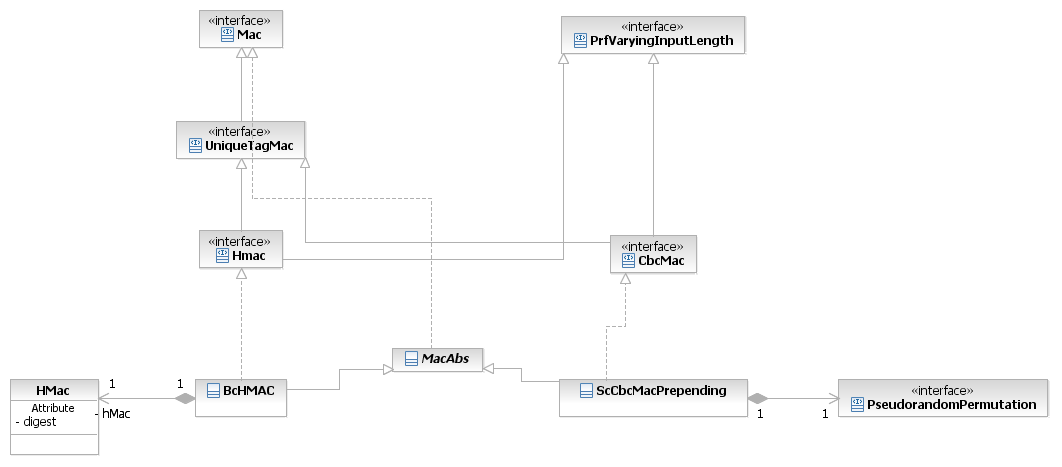


#### Factories

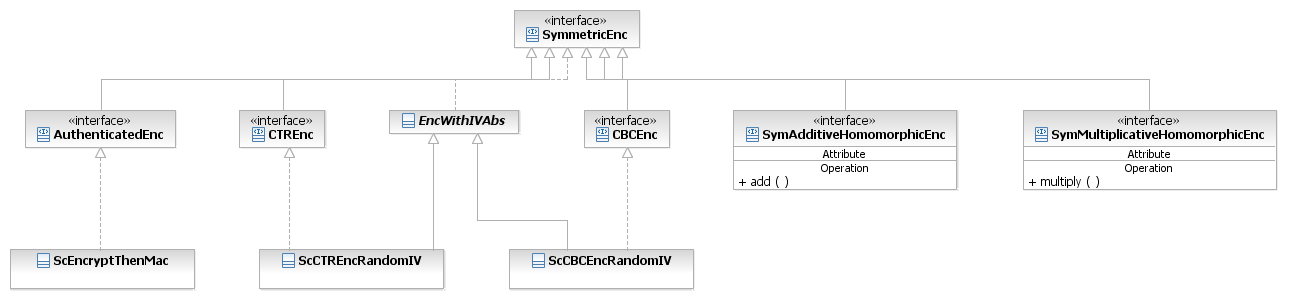
## High Level Description

### General Static View

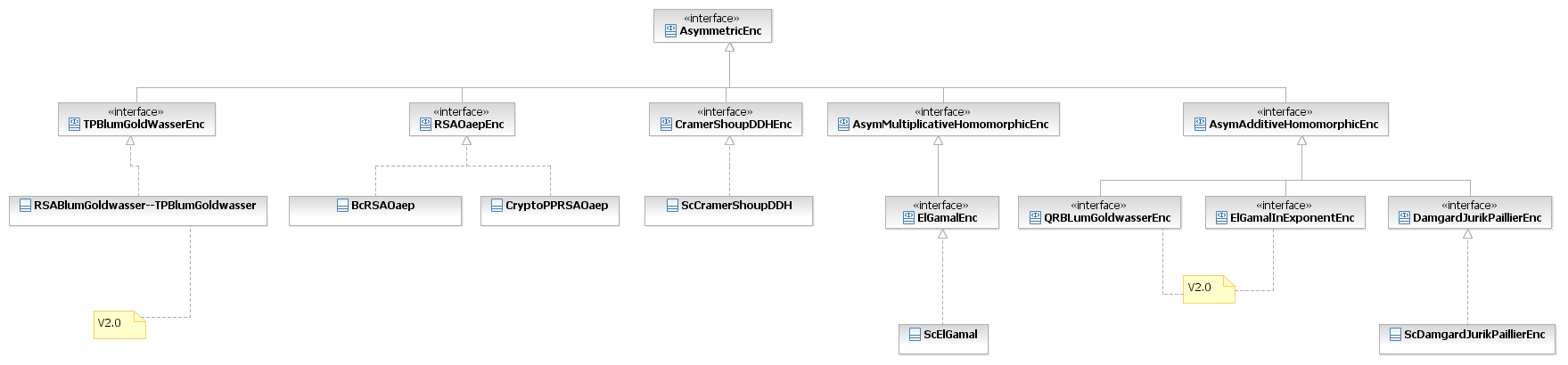
#### MAC – Message Authentication Codes

****

#### Symmetric Encryption

****

#### Asymmetric Encryption



#### Key Generation

Add diagram of provider class here.

## Detailed Description

### Message Authentication Codes

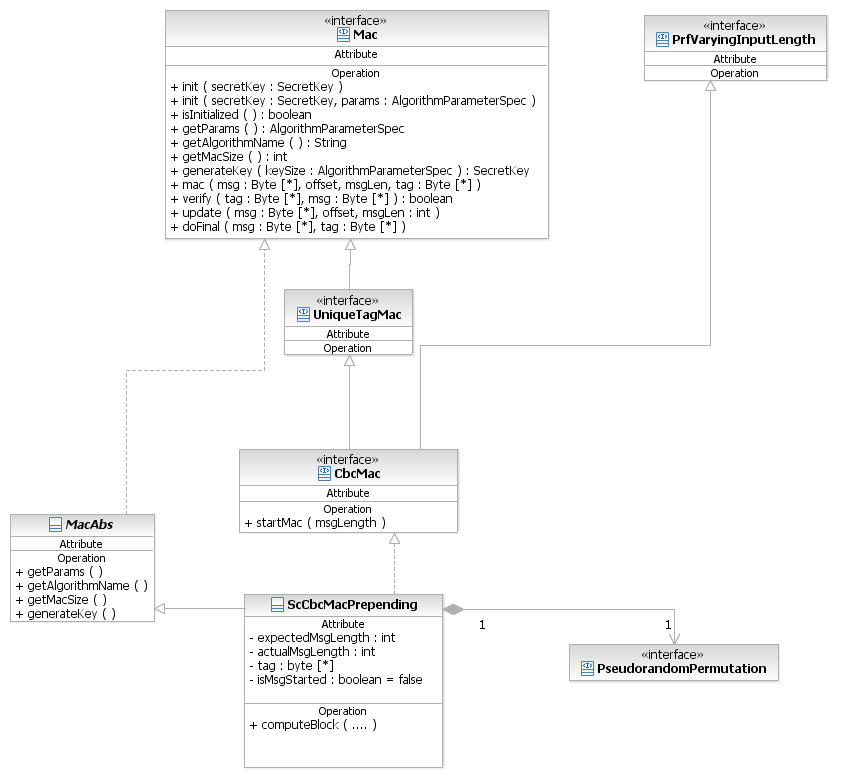
#### CBC-MAC

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

###### In CbcMacPrepending::CbcMacPrepending (String prp) do:

* Create a new PRP object from the string argument.
* Set the PRP member variable to it.

###### In CbcMacPrepending::CbcMacPrepending (PseudorandomPermutation p) do:

* Check that the PRP received as argument has been initialized. If not, throw exception UnIntializedException.
* Set the PRP member variable to the one in the argument.

###### In CbcMacPrepending::init functions do

* Call underlying PRP respective init function.

###### In CbcMacPrepending::isInitialized() do

* Return prp.isInitialized().

###### In CbcMacPrepending::startMac(int msgLength) do:

* If this object is not initialized, throw exception
* 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 msgStarted to true

###### In CbcMacPrepending::update do:

* If msg not marked as started, throw exception.
* If msg not aligned to underlying PRP’s block size, **throw exception**.
* 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 allocating and copying of arrays so put result into tag.
  + actualMsgLength += this block’s size

###### In CbcMacPrepending:: doFinal do:

* If msg not marked as started, throw exception.
* If msg not aligned to 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 allocating and copying of arrays.
* actualMsgLength += this msg’s size
* If actualMsgLength != expectedMsgLength, throw exception
* Return the calculated tag.

###### In CbcMacPrepending::mac do:

* Call startMac with received msgLength
* Call doFinal with whole msg
* Return the calculated tag

###### In CbcMacPrepending::verify do:

* Compute mac on the message
* If 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.

###### In CbcMacPrepending::compute do:

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

* Call mac ( msg : Byte [\*], offset : int, msgLen : int, tag : Byte [\*] )
* Set the out array argument to the resulting tag of the mac.

###### In CbcMacPrepending::getAlgorithmName

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

##### Usage

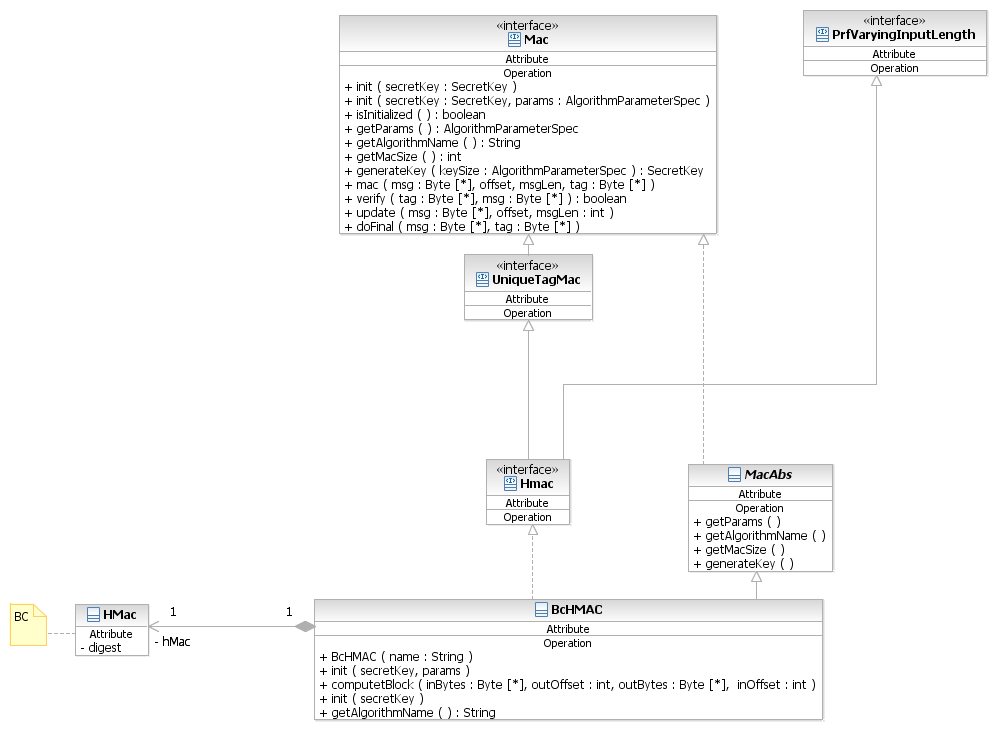
CbcMac is secure only if the length of the full message is known in advance and its computation with the underlying block cipher is prepend to the message. We allow two different ways to calculate the MAC with CBC-MAC:

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 this 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 to know the length of the message in advance.

##### Static View



##### Dynamic View

###### In BcHmac::computeBlock do

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

###### In BcHmac::mac do

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

###### In BcHmac::verify do

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

###### In BcHmac::update do

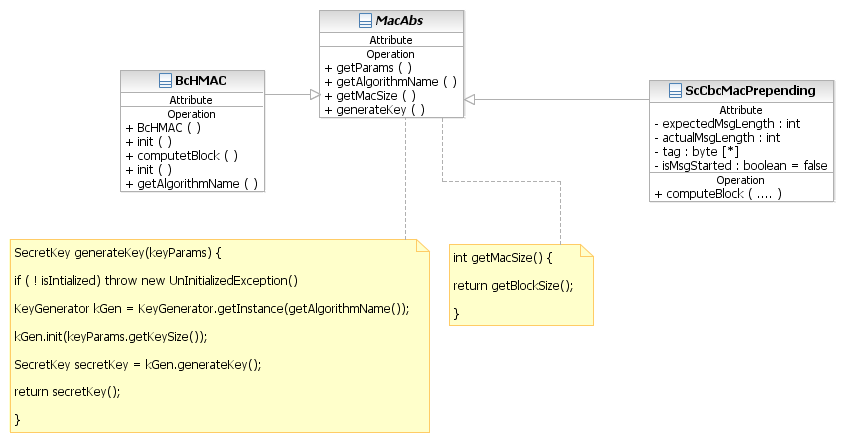
* Call the update function of the underlying HMac from BC.

###### In BcHmac::doFinal do

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

#### MacAbs

Functionality that is common to concrete classes implementing the Mac interface is placed in this abstract class.Static View



##### Dynamic view

###### In MacAbs::getMacSize do

* Return getBlockSize().

###### In MacAbs::generateKey

* Throw UnIntializedException if object is not initialized.
* Get an instance of KeyGenerator for current algorithm name.
* Init the generator with the size of the key.
* Create a secret key with the key generator.
* Return the secret key.

### Symmetric Encryption

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

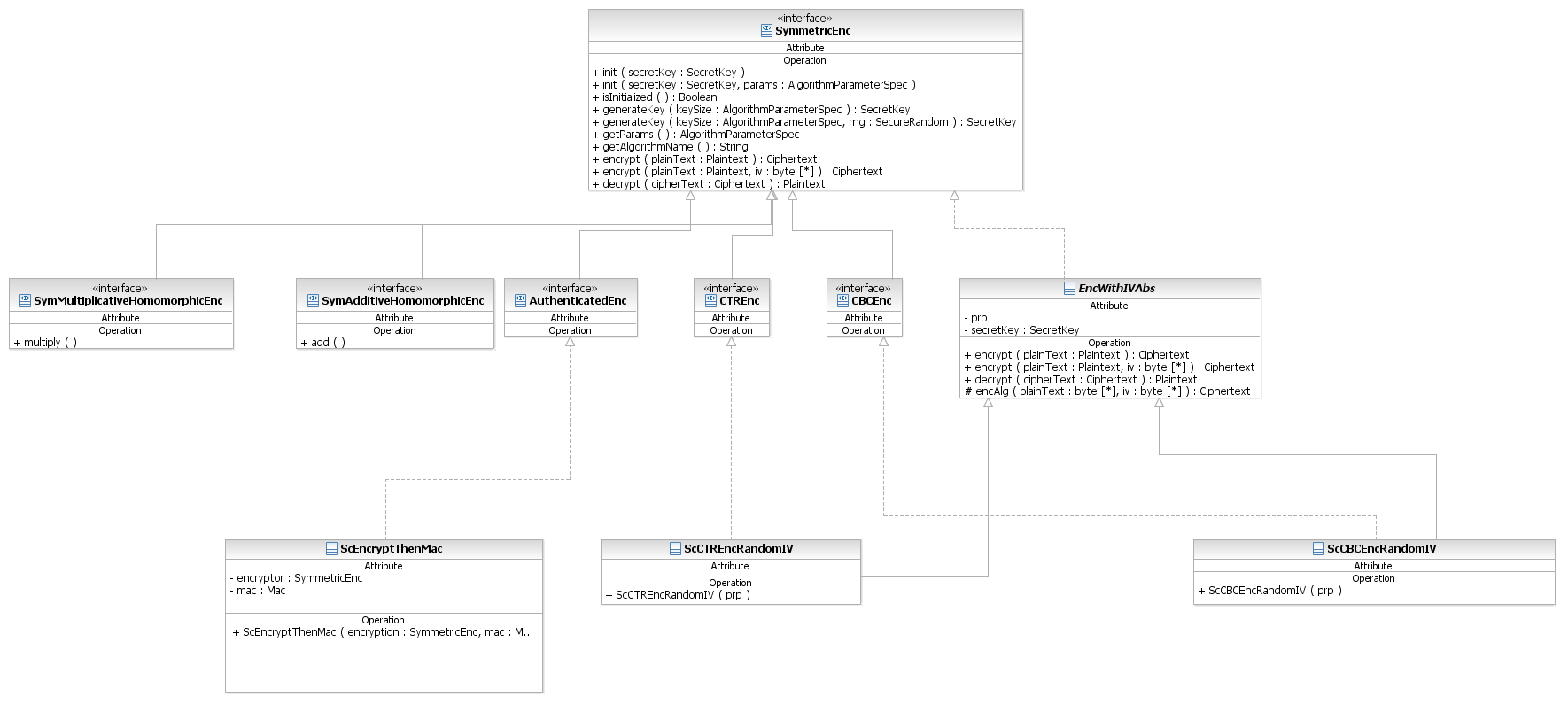
1. An encryption based on modes of operation using a pseudo-random permutation and a randomized IV. The randomized IV is crucial for security.
2. An authenticated encryption where the message gets first encrypted and then mac-ed. CBCEnc and CTREnc belong to the first category and EncryptThenMac belongs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

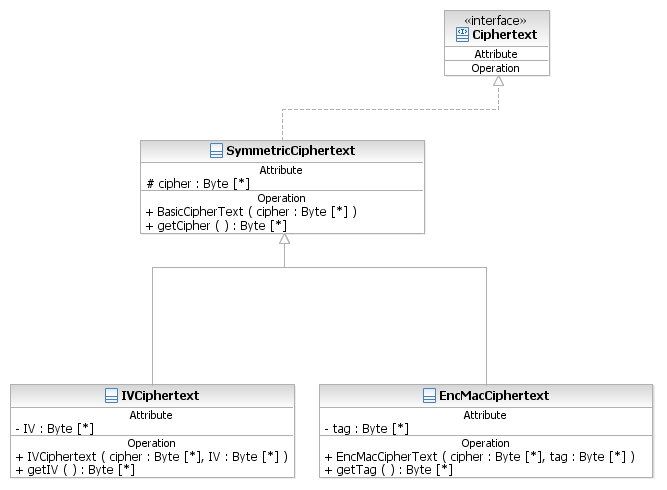
We note that for authenticated encryption two secret keys will be used, one for the encryption and one for the authentication. Yet, the init() functions will require a single secret key object, so that a protocol working on a higher level will be able to perform initialization of the encryption object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

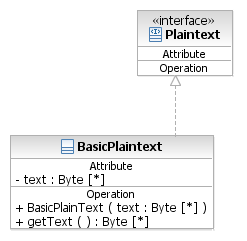
The symmetric encryption family of classes implements three main functionalities that correspond to the cryptographer’s language in which an encryption scheme is composed of three algorithms:

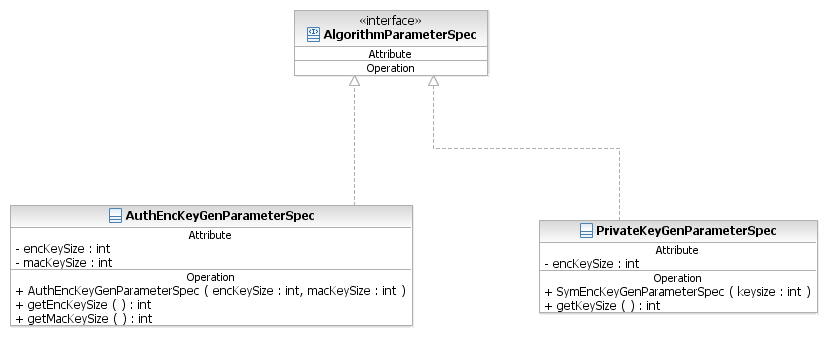
1. Generation of the key.
2. Encryption of the plaintext.
3. Decryption of the ciphertext.

#### Static view – General









#### Encryption with IV

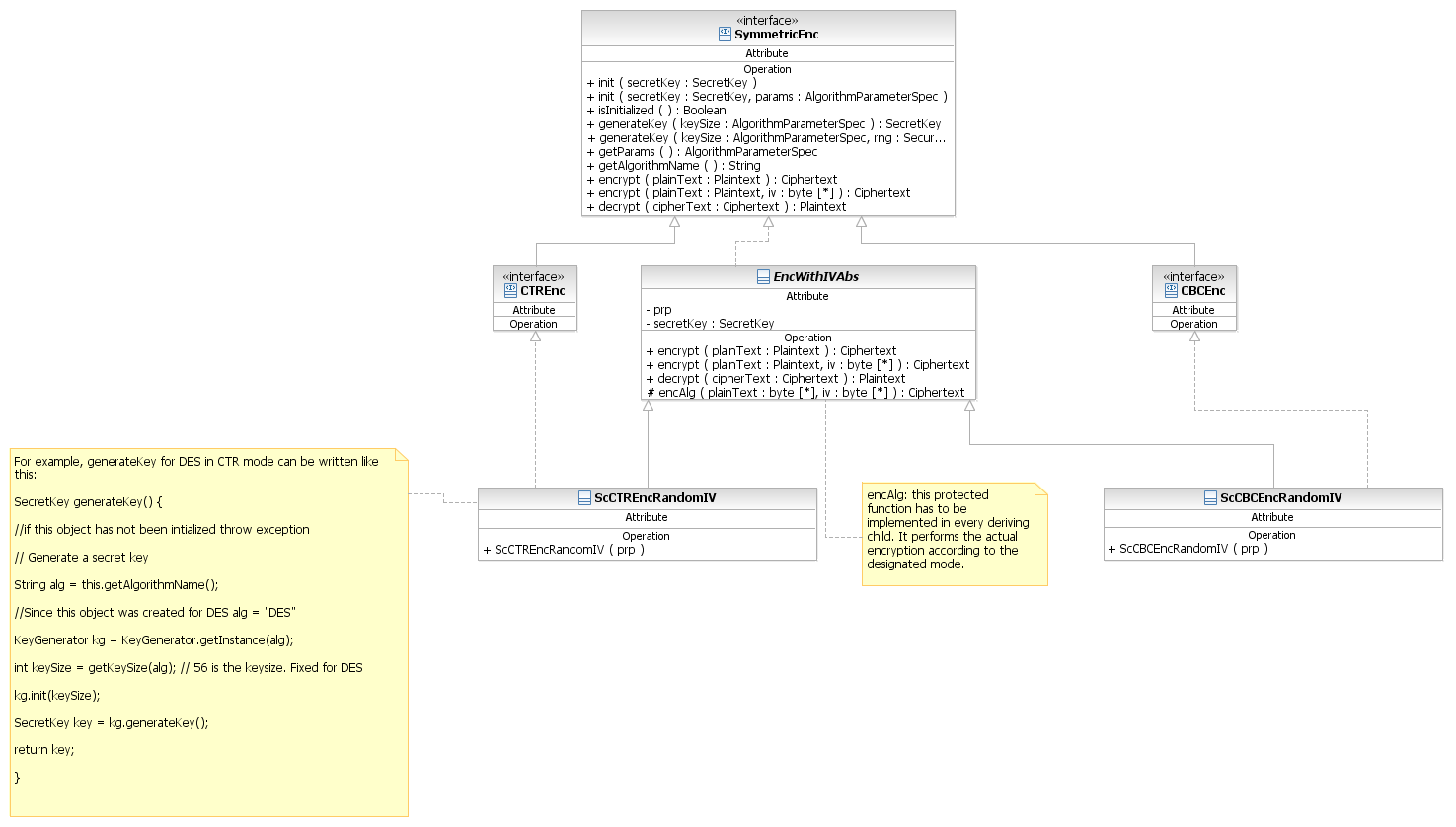
Since most of the IV work for encryption is the same for CBCEnc as for CTREnc we enclose all the common functionality in a package private abstract class called EncWithIVAbs. The actual mode of operation will be implemented in a protected function in each of the concrete classes.

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 – Encryption with IV

###### EncWithIVAbs::init ( secretKey: SecretKey) and

###### EncWithIVAbs::init (secretKey: SecretKey, params: AlgorithmParameterSpec)

Init functions 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.

The init function with the AlgorithmParameterSpec can be called using a PaddingParameterSpec object as the second argument if padding is required.

* Call the underlying’s PRP relevant init function.

###### EncWithIVAbs::generateKey(keySize : AlgorithmParameterSpec)

This function generates a key according to the underlying pseudorandom permutation and uses SCAPI’s default source of randomness. The given keySize is in bytes.

* If keySize is not of type PrivateKeyGenParameterSpec throw InvalidParameterSpecException (“Key size has to be of type PrivateKeyGenParameterSpec”)
* Return generateKey(keySize, defaultRandom)

###### EncWithIVAbs::generateKey(keySize : AlgorithmParameterSpec , rnd :SecureRandom)

This function generates a key according to the underlying pseudorandom permutation and gets the source of randomness from the caller of the function. The given keySize is in bytes. It uses the default provider to do so:

* If keySize is not of type PrivateKeyGenParameterSpec throw InvalidParameterSpecException (“Key size has to be of type PrivateKeyGenParameterSpec”)
* Get an instance of the default KeyGenerator for the underlying PRP’s name.
* Init the key generator with the key size.
* Generate the key with the generator and return it.

###### 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 cipher-text. 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.

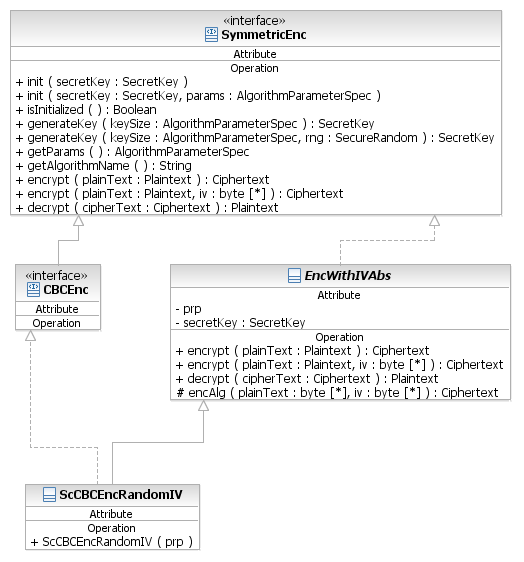
###### EncWithIVAbs::encrypt ( plaintext: Plaintext): Ciphertext

* Generate a new random IV.
* Return encrypt (plaintext, IV).

###### EncWithIVAbs::encrypt ( plaintext: Plaintext, IV : byte[ ]) : Ciphertext

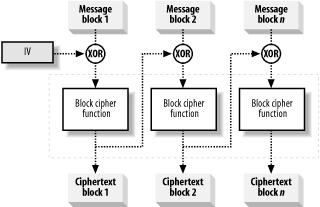
* Set the iv member variable to be the IV in the argument.
* Call encAlg (plaintext.getText(), IV) to obtain ciphertext.
* Return the ciphertext.

##### Static View – CBCEnc



##### Dynamic View –CBCEnc

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



###### CBCEnc( prp : PseudoRandomPermutation)

This is the constructor. It 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 parameter. To do so we use the “init” functions presented in the abstract class.

###### decrypt ( cipherText : Ciphertext ) : Plaintext

* Check if cipherText is instance of IVCiphertext. If not throw IllegalArgumentException.
* c­0 = cipherText.getIV()
* Allocate a plaintext buffer of length ciphertext’s length
* cipher = ciphertext.getCipher()
* plaintext[0] = prp.invert(cipher[0]) XOR c0
* For i from 1 to length of plaintext do:
  + plaintext[i] : = prp.invert(cipher[i]) XOR ciphertext[i-1]
* Return the plaintext.

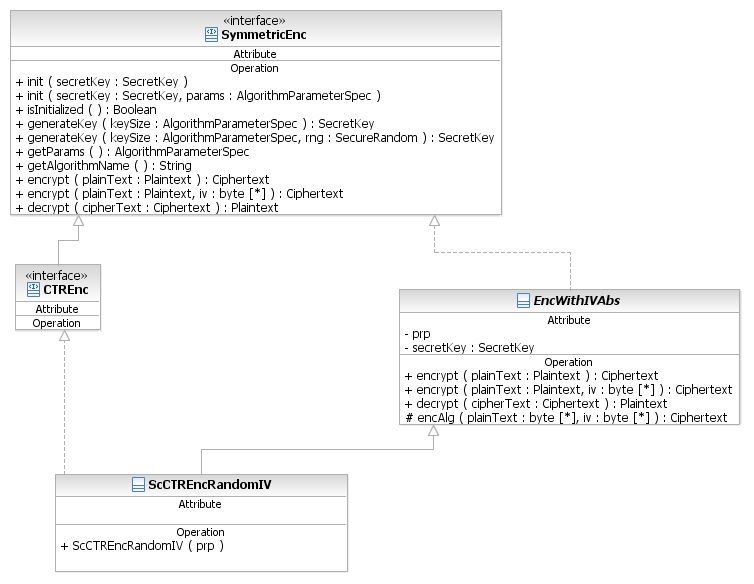
###### encAlg (plainText : byte [], iv : byte []) IVCiphertext

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

* Allocate a byte[] ciphertext with the length of plaintext.
* ciphertext[0] = prp.computeBlock(iv XOR plaintext[0])
* for next blocks in plaintext do: //i = 1
  + ciphertext [i+1] = prp.computeBlock(ciphertext [i] XOR plaintext[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.

##### Static View – CTREnc

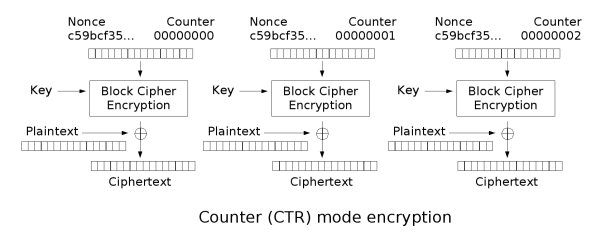


##### Dynamic View - CTREnc

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:



###### CTREnc( prp : PseudoRandomPermutation)

This is the constructor. It 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 “init” functions presented in the abstract class.

###### decrypt ( ciphertext : Ciphertext ) : Plaintext

* Check if ciphertext is 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.

###### 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 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.

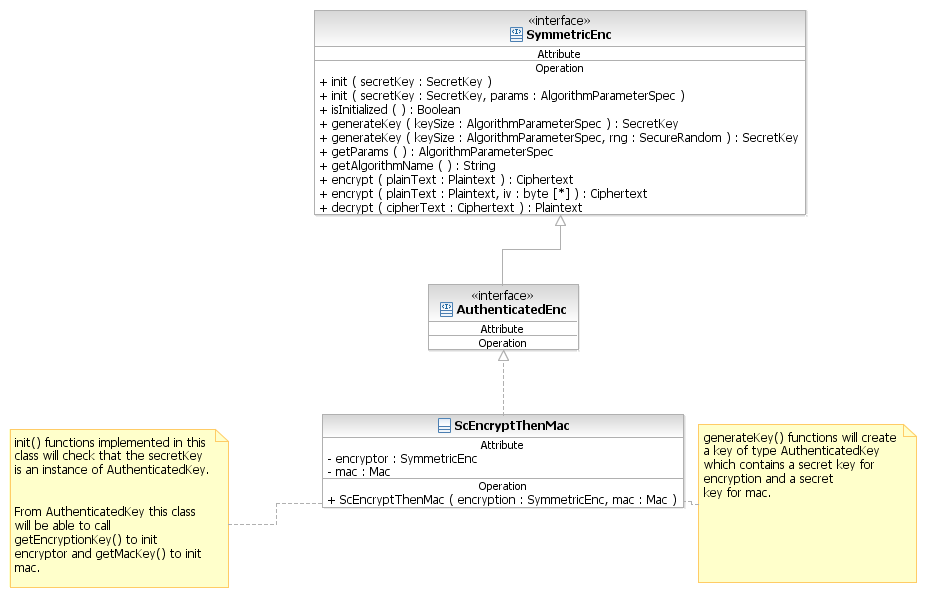
#### Authenticated Encryption

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

###### ScEncryptThenMac(encryption: SymmetricEnc, mac : Mac)

* If encryption is instance of AuthenticatedEnc then throw exception IllegalArgumentException(“A symmetric encryption that is not an authenticated encryption is needed”). //**NOTE: may be we want to allow it and let it be the user’s responsibility.**

###### init( secretKey) and init ( secretKey , params )

* ScEncryptThenMac checks that the given secretKey is of type AuthenticatedKey. If not throws InvalidKeyException(“This encryption requires a key of type AuthenticatedKey”)
* It calls encryptor’s relevant init with corresponding key and mac’s relevant init with corresponding key.

###### generateKey( keySize : AlgorithmParameterSpec )

This function generates an authenticated key and uses SCAPI’s default source of randomness. The given keySize is in bytes.

* If keySize is not of type AuthEncKeyGenParameterSpec throw InvalidParameterSpecException(“Key size has to be of type AuthEncKeyGenParameterSpec”)
* Return generateKey(keySize, defaultRandom)

###### generateKey (keySize, rnd)

This function generates an authenticated key according suitable for the underlying encrypting object and mac object. It gets the source of randomness from the caller of the function. The given keySize is in bytes. It uses the default provider to do so:

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

###### encrypt ( plaintext : Plaintext) : ciphertext : Ciphertext

* Allocate a cipher byte buffer of size equal to plaintext length.
* Allocate a tag byte buffer of size mac.getTagSize()
* cipher = encryptor.encrypt(plaintext)
* Calculate mac ( plaintext.getText(), 0, plaintext.length, tag )
* Create and return an EncMacCiphertext object with cipher and tag.

###### encrypt ( plaintext : Plaintext, iv : byte[]) : ciphertext : Ciphertext

* Allocate a cipher byte buffer of size equal to plaintext length.
* Allocate a tag byte buffer of size mac.getTagSize()
* cipher = encryptor.encrypt(plaintext, iv)
* Calculate mac ( cipher.getCipher(), cipher.getCpher().length, 0, .length, tag )
* Create and return an EncMacCiphertext object with cipher and tag.

###### Decrypt( ciphertext : Ciphertext ) : Plaintext

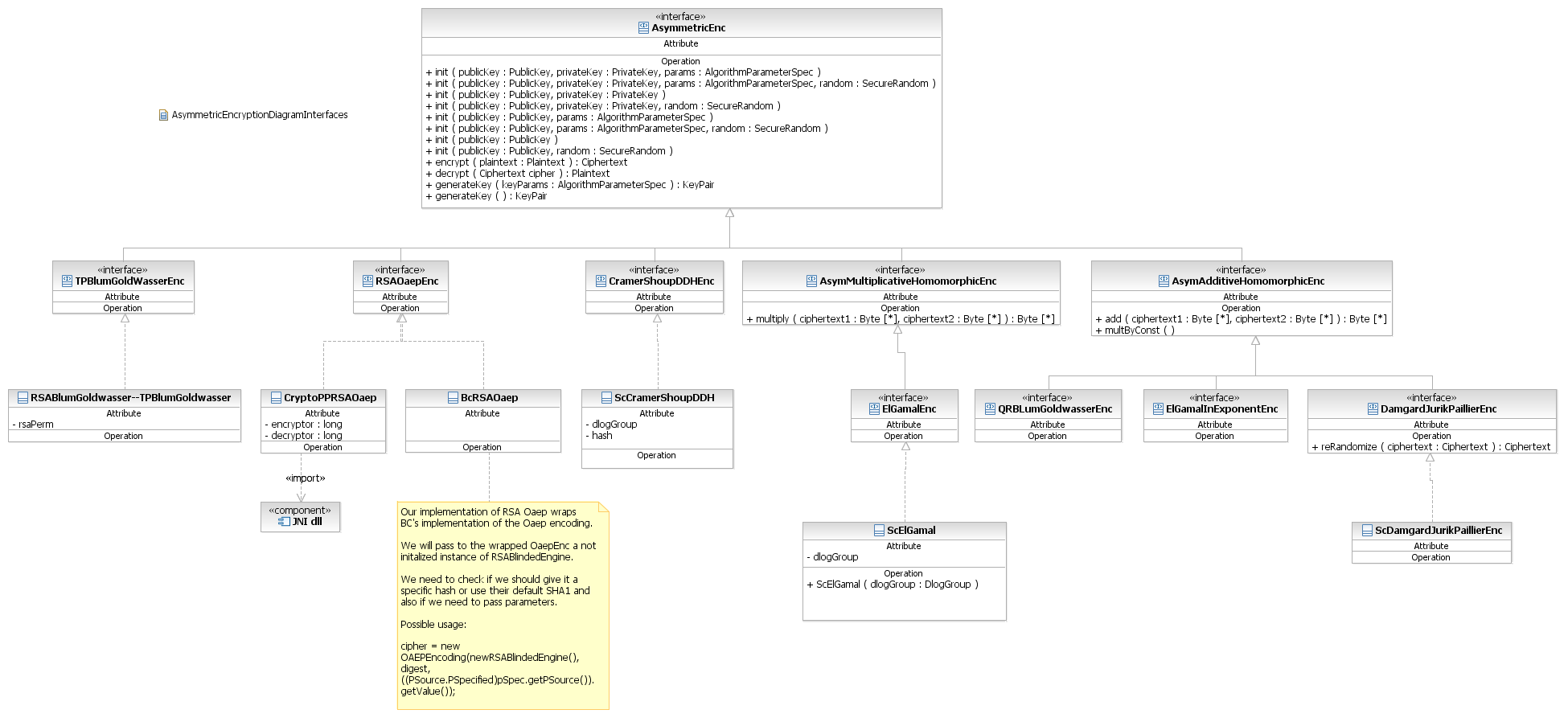
* Verify the tag using the mac object, if correct:
  + Decrypt the message using the encryptor object
  + Return the plaintext
* Else, return null.

### Asymmetric encription

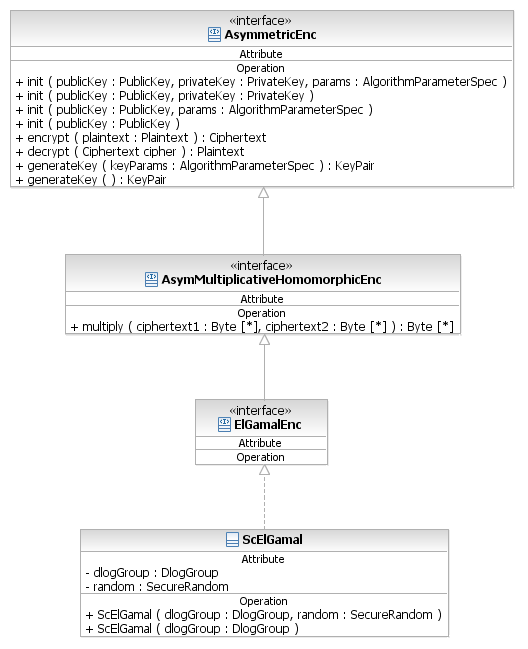
The asymmetric encryption family of classes implements three main functionalities that correspond to the cryptographer’s language in which an encryption scheme is composed of three algorithms:

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

#### Static View – General



#### Static View – El Gamal



#### Dynamic 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.

##### Constructors: ScElGamal(dlogGroup : DlogGroup) and ScElGamal(dlogGroup : DlogGroup, random : SecureRandom)

Our implementation of El Gamal encryption receives an initialized Dlog Group object in its constructor. It must be initialized since El Gamal can work with any Dlog group, i.e. elliptic curve or Zp, but their initializations are completely different. If so desired, the user can choose to pass a specific SecureRandom object in the constructor. We assume that once the user chooses to work with certain SecureRandom implementation there is no need to change it for this ElGamalEnc object.

* If the given dlog group object has not been initialized, throw UnInitializedException.
* If grp is not instance of DDH security level, throw IllegalArgumentException
* El Gamal algorithm needs a source of randomness. If an init function with SecureRandom is used,
  + then set the given argument as the source of randomness for all the work of this instance,
  + else use SCAPI’s default source of randomness.

##### Init( publicKey : PublicKey) and

##### Init (publicKey : PublicKey, params : AlgorithmParameterSpec) and

##### Init (publicKey : PublicKey, privateKey : PrivateKey) and

##### Init (publicKey : PublicKey, privateKey : PrivateKey, params : AlgorithmParameterSpec) and

* The init functions that require public key and private are used when the user knows both keys. In this case the user can encrypt and decrypt messages.
* The init functions that require only the public key are used when the user only knows the public key and/or only wants to encrypt a message.
* The init functions with the AlgorithmParameterSpec can be called using a PaddingParameterSpec object as the third argument if padding is required.

##### generateKey ( keyParams : AlgorithmParameterSpec ) : KeyPair and

##### generateKey ( ) : KeyPair and

All the generateKey functions generate a KeyPair which is a place holder for a Public Key and a Private Key objects. 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.

* 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. (Or (G, q, g, h) ?)
  + Set the private key part of the key pair to be x. (Or (G, q, g, x) ?)
  + Return the key pair.

##### Encrypt(plaintext : Plaintext ) : Ciphertext

* If !dlogGroup.convertByteArrayToGroupElement(plaintext.getBytes()) throw exception.
* Choose a random y 🡨 Zq
* Calculate c1 = gy mod p //mod p operation are performed automatically by the group.
* Calculate c2 = hy \* plaintext.getMessage() mod p
* Create and return an ElGamalCiphertext object with c1 and c2.

##### Decrypt(ciphertext : Ciphertext ) : Plaintext

* If private key is null, then cannot decrypt. Throw exception. [TODO decide which exception]
* Calculate s = ciphertext.getC1() ^ privateKey
* Calculate the inverse of s: invS = s ^ -1
* Calculate m = ciphertext.getC2() \* invS
* m is a groupElemenet. Use it to create and return an instance of ElGamalPlaintext.

## External interface

### Class 1 API

### Class 2 API

## Components - Static view

*Add detailed UML figure of each component. Put each functionality under a different numbering.*

### Functionality 1

### Functionality 2

…

## Data Structure Design

# User interface design

*Add if relevant.*

# Performance

# Multi-Platform Issues

# Backward-Compatibility

# Benchmarking

# Open Issues

Appendix A

For Elliptic curves:

Let p be the prime of the field over which the elliptic curve is defined. Then, allow a mapping of any string that is of length less than log p (i.e., less than the bit-length of p). Then, let x be UnsignedBigInteger conversion of that string (since it is of length less than the length of p, this should be a string smaller than p and so it is legal). Then compute y via the elliptic curve equation (y^2 = x^3 + ax + b). Regarding the mapping back from a point to a string: do the same in reverse. Ignore the y part and just convert the integer in the x part to a string.

For DLOG with safe primes:

We have the method written in the book so let's just do this. Specifically, given a string that is smaller than (p-1)/2 = q, map it into the group by squaring it modulo p. Then, mapping a group element back to a string is carried out by finding its square root mod p. (There are actually 2 square roots but only one between 1 and (p-1)/2, so this is the one that is taken.) In order to find a square root mod p, use the algorithm in the Handbook of Applied Cryptography, page 102, Algorithm 3.44.