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 | 08-Jan-2006 |  | Initial version |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

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 9](#_Toc294537833)

[5.2.1 General Static View 9](#_Toc294537834)

[5.3 Detailed Description 10](#_Toc294537835)

[5.3.1 CBC-MAC 10](#_Toc294537836)

[5.3.1.1 Static View 11](#_Toc294537837)

[5.3.1.2 Dynamic View 11](#_Toc294537838)

[5.3.1.2.1 In CbcMacPrepending::CbcMacPrepending (PseudrandomPermutation p) do: 11](#_Toc294537839)

[5.3.1.2.2 In CbcMacPrepending::startMac do: 12](#_Toc294537840)

[5.3.1.2.3 In CbcMacPrepending::update do: 12](#_Toc294537841)

[5.3.1.2.4 In CbcMacPrepending:: doFinal do: 12](#_Toc294537842)

[5.3.1.2.5 In CbcMacPrepending::mac do: 12](#_Toc294537843)

[5.3.1.2.6 In CbcMacImp::verify do: 12](#_Toc294537844)

[5.3.1.2.7 In CbcMacPrepending::compute do: 13](#_Toc294537845)

[5.3.1.2.8 In CbcMacPrepending::getAlgorithmName 13](#_Toc294537846)

[5.3.1.3 Usage 13](#_Toc294537847)

[5.3.2 HMAC 14](#_Toc294537848)

[5.3.2.1 Static View 14](#_Toc294537849)

[5.3.2.2 Dynamic View 14](#_Toc294537850)

[5.3.2.2.1 In BcHmac::computeBlock do 14](#_Toc294537851)

[5.3.2.2.2 In BcHmac::mac do 15](#_Toc294537852)

[5.3.2.2.3 In BcHmac::verify do 15](#_Toc294537853)

[5.3.2.2.4 In BcHmac::update do 15](#_Toc294537854)

[5.3.2.2.5 In BcHmac::doFinal do 15](#_Toc294537855)

[5.4 External interface 19](#_Toc294537856)

[5.4.1 Class 1 API 20](#_Toc294537857)

[5.4.2 Class 2 API 20](#_Toc294537858)

[5.5 Components - Static view 21](#_Toc294537859)

[5.5.1 Functionality 1 21](#_Toc294537860)

[5.5.2 Functionality 2 21](#_Toc294537861)

[5.6 Data Structure Design 21](#_Toc294537862)

[6. User interface design 21](#_Toc294537863)

[7. Performance 21](#_Toc294537864)

[8. Multi-Platform Issues 21](#_Toc294537865)

[9. Backward-Compatibility 21](#_Toc294537866)

[10. Benchmarking 21](#_Toc294537867)

[Open Issues 21](#_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

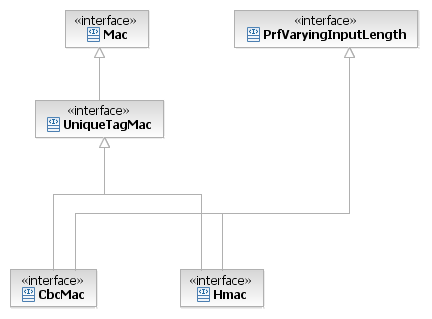
The same principles that guided us in [1] guide us here to achieve maximum flexibility and extensibility as at the same time being as efficient as possible.

# Architectural design

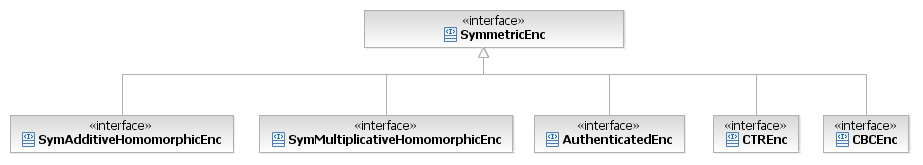
## General

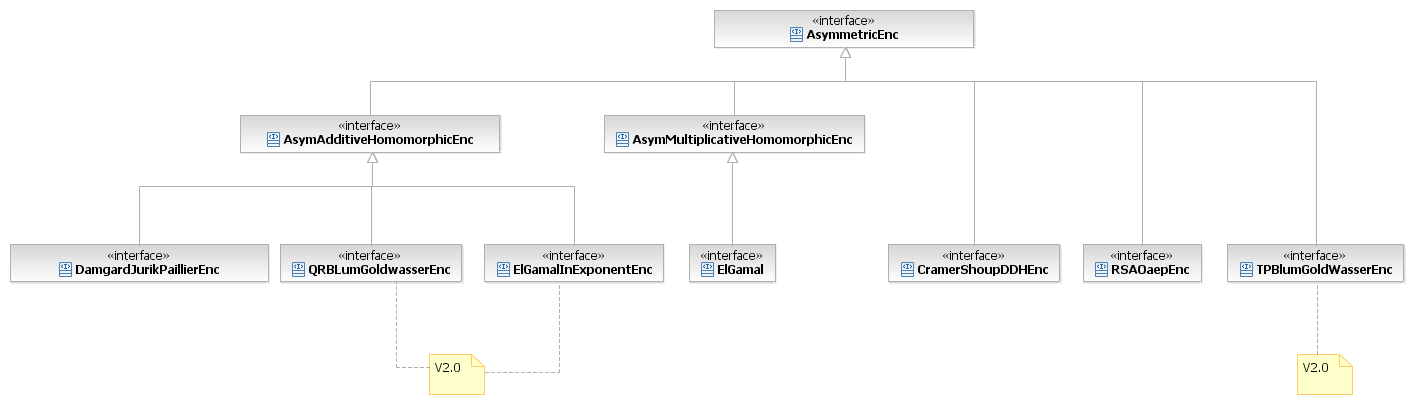
### Main interfaces

#### Message authentication codes

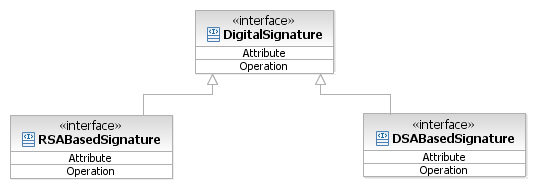


#### Encryption Types

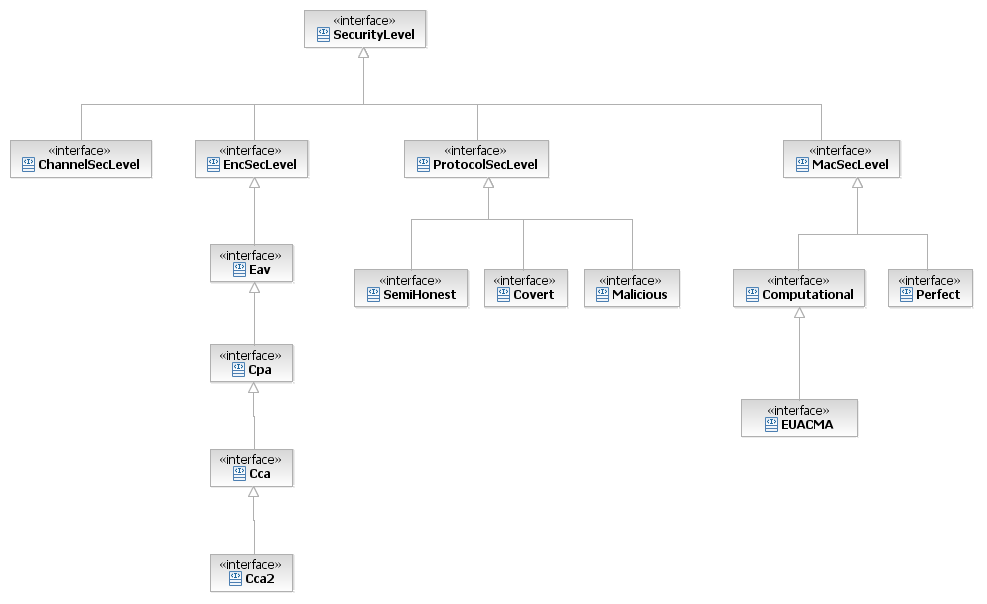




#### Digital Signatures

****

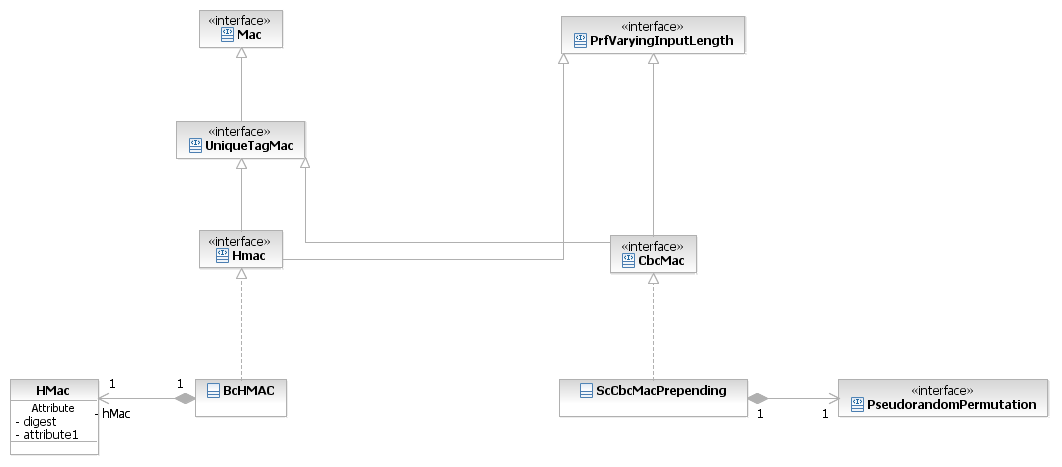
#### Security Levels



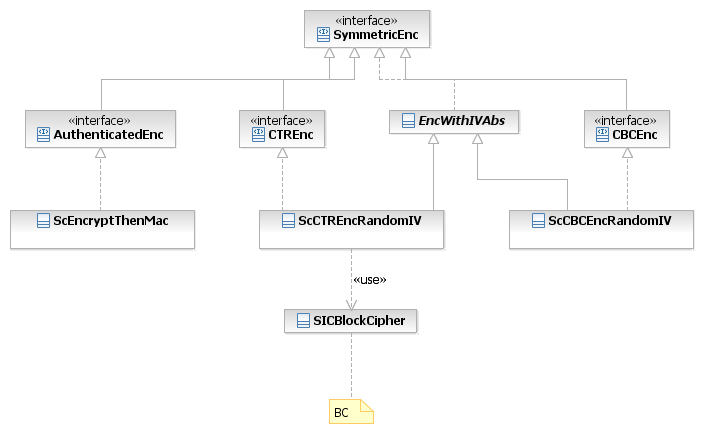
## High Level Description

### General Static View

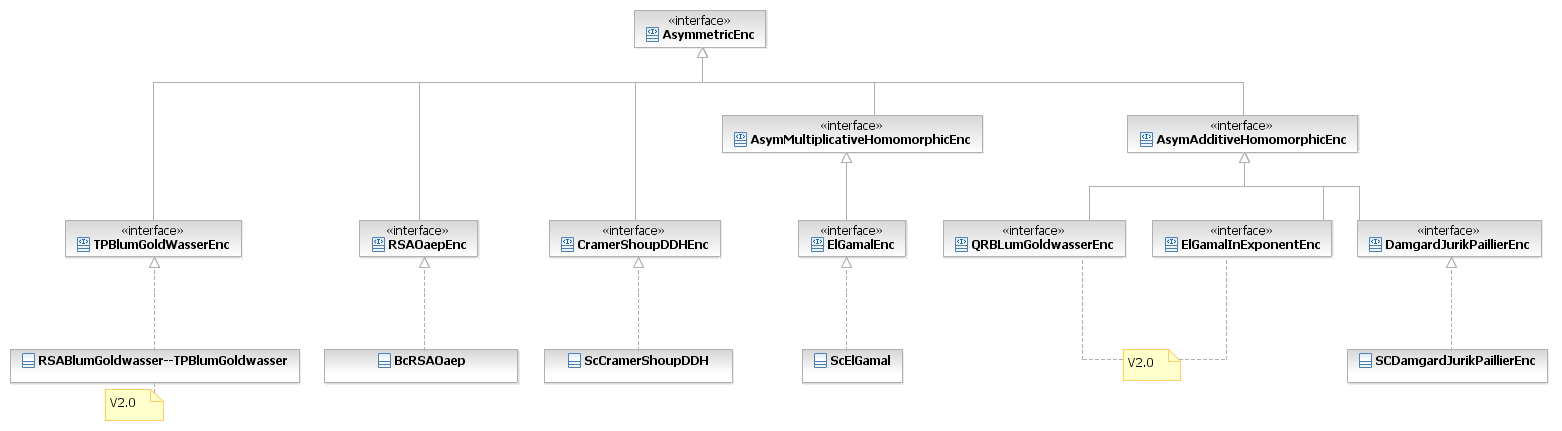
#### MAC – Message Authentication Codes

****

#### Symmetric Encryption

****

#### Asymmetric Encryption



## Detailed Description

### Message Authentication Codes

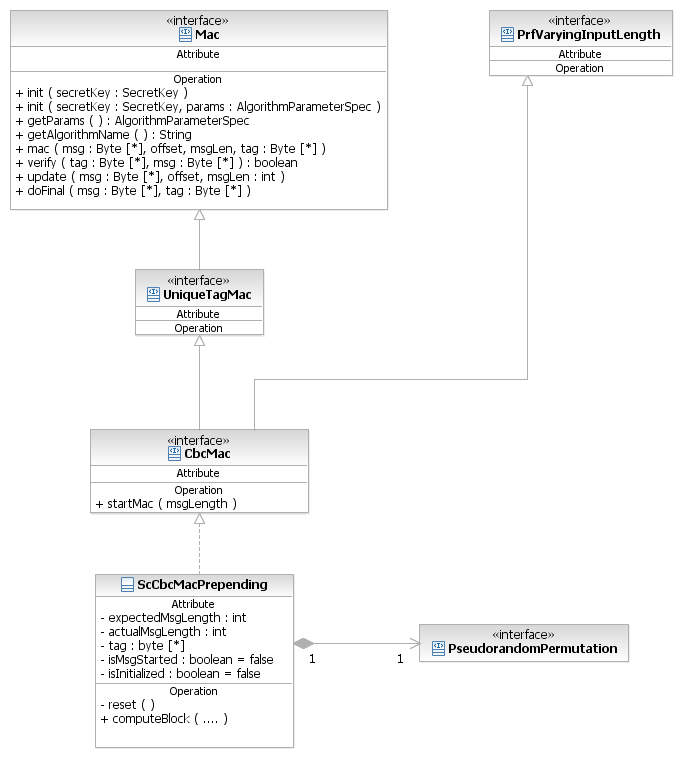
#### CBC-MAC

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: “NotIntializedException”.
* Set the PRP member variable to the one in the argument.

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

* If this object is not initialized, throw exception
* Call reset.
* 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

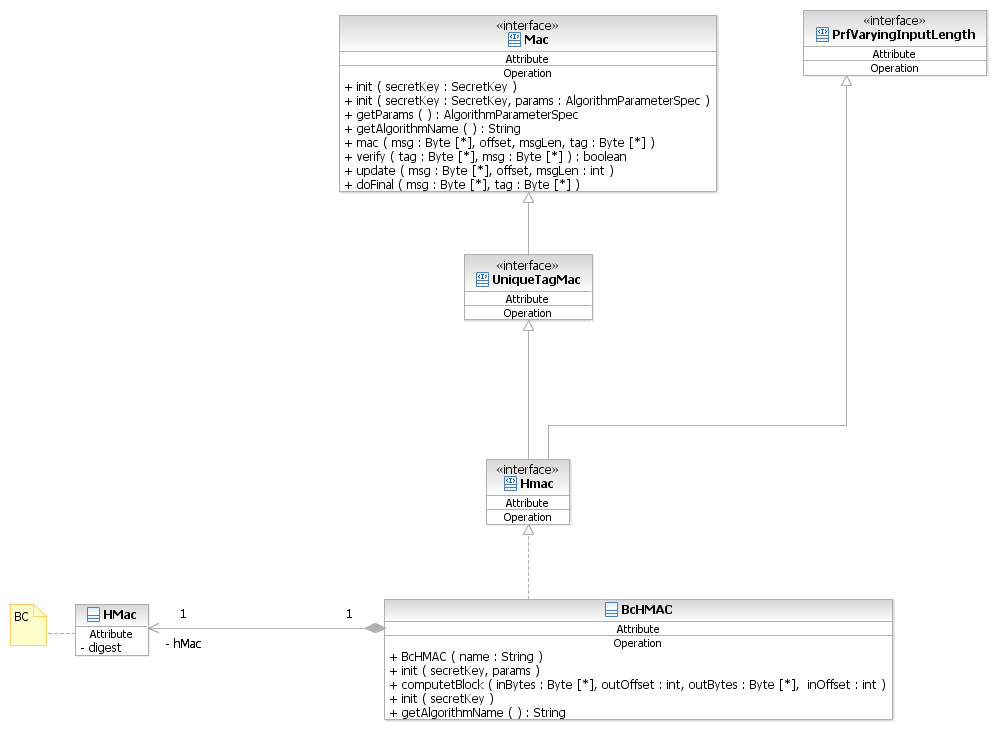
CbcMac is secure only if the length of the full message is known in advance and its computation with the underlying block cipher is prepend to the message. We allow two different ways to calculate the MAC with CBC-MAC:

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.

### 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.

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.

#### 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.

We introduce a new class IVParam that extends the AlgorithmParameterSpec class in order to check if the IV has been set or not. This will be very useful when a special way of encryption (a sort of loop encryption) is used and needs to use the previous IV. This will be further explained below.

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

##### Static View - General



****

##### Dynamic View - General

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 the first block in 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. In other cases, we need to allow reuse of pre-calculated IV as input for the next encryption. We allow these options by providing three "encrypt" functions, each with different arguments.

###### init ( secretKey: SecretKey) and 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 should be called using an IVParam object as the second argument. This type of initialization can be used in conjunction with the loop encryption that will be described below.

###### encryption functions:

We provide three options for the encryption function. All of them encrypt the plaintext using the mode of operation algorithm and the prp member variable. At the end of the encryption set the iv member variable to be the last block in the cipher-text. They differ in the way they obtain the IV and whether they return the used IV as part of the ciphertext.

###### encrypt ( plaintext: byte[ ]): byte[ ]

* Always generates a new random IV. Set the iv member variable with it.
* Call encryptAndReturnIV (plaintext) to obtain ciphertext
* Return the ciphertext.

###### encrypt ( plaintext: byte[ ], IV : byte[ ]) : byte[ ] throws IlegalArgumentException

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

###### encryptAndReturnIV(plaintext: byte[]) : byte[] //private function

* Allocate a ciphertext buffer with length equal to plaintext’s length plus one block more.
* Call encAlg(plaintext, this.iv, ciphertext, offset) to obtain ciphertext, where offset is set to 1 block. This will leave the first block in the resulting ciphertext empty.
* Fill first (empty) block of ciphertext with current iv.
* Set iv member variable to last block in cipher-text.
* Return the ciphertext.

###### encryptWithExistingIV( plaintext: byte[ ] ) : byte[ ] throws IllegalStateException

* If the iv member variable is null or has not been set
  + Throw IllegalStateException
* Allocate a ciphertext buffer with length equal to plaintext’s length exactly
* Call encAlg(plaintext, this.iv, ciphertext, offset) to obtain ciphertext, where offset is set to 0 block.
* Return the ciphertext.

This encrypt function allows chaining the encryption of many messages using the resulting IV of the previous encryption. Since what we consider the new IV differs for the different modes, the encAlg function implemented in each concrete class is responsible for setting it.

In the figures below we present examples for the CBC encryption scheme and for the CTR encryption scheme:



CBC chaining of message



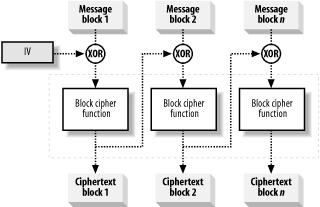
CTR chaining of messages

##### 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 : byte [] ) : byte []

This function assumes that the first block of the ciphertext argument is the IV. We will call it c0. All the other remaining blocks will be called ci with i = 1 to n.

* Allocate a plaintext buffer of length ciphertext’s length – 1 block.
* For i from 0 to length of plaintext do:
  + Plaintext[i] : = prp.computeBlock(ciphertext[i+1]) XOR ciphertext[i]
* Set iv member variable to last block in ciphertext.
* Return the plaintext.

###### decryptWithExistingIV(cipherText : byte[]) : byte[]

***NOTE for Yehuda, Meital, Moryiah and Yael*** : *because we may use an existing IV (researcher requirement) this can cause lots of problems. You should never use the same encrypting object both for encryption and decryption. For example: if part 1 sent some encrypted message to party 2 and set the corresponding IV and in the meantime it receives an encrypted message from party 3, it cannot call function decryptWithExisitngIv because ivs are not related. This brings me back to the design of BC in which a cipher object can be inited either for encryption or for decryption. We can add a boolean argument to the init functions to set for encryption or decryption and throw an InvalidStateException if the operation requested doesn’t suit the mode. This doesn’t comply with anything we have done before...*

This function assumes the first block is already an encrypted block.

* Allocate a plaintext buffer of length ciphertext’s length.
* Plaintext[0] := prp.computeBlock(ciphertext[0]) XOR iv member variable
* For i from 1 to length of plaintext do:
  + Plaintext[i] : = prp.computeBlock(ciphertext[i]) XOR ciphertext[i-1]
* Set iv member variable to last block in ciphertext.
* Return the plaintext.

###### encAlg (plainText : byte [], iv : byte [], ciphertext : byte [], offset : int )

This function must be implemented in this concrete class. We use a return buffer in the argument in order to save unnecessary copies of the array (which can be very large) in the case that we do not need to return the IV. One time encrypts will call the function with offset equal to 1, whereas loop encryptions will call it with offset seto to 0 because the IV is not needed.

In CBCEnc this function performs the CBC mode of operation:

* if offset equals 1 block
  + if lenght of ciphertext is less than length of plaintext +1
    - throw IlegalArgumentException (with explanation)
  + set first block of out buffer with the IV: ciphertext [0] = IV
  + for each block in plaintext do: //i = 0
    - ciphertext [i+1] = prp.computeBlock(ciphertext [i] XOR plaintext[i])
* else if it’s zero
  + ciphertext[0] = prp.computeBlock(iv XOR plaintext[0i])
  + for next blocks in plaintext do: //i = 1
    - ciphertext [i+1] = prp.computeBlock(ciphertext [i] XOR plaintext[i])
* Set this.iv to last block in cipher-text. //Crucial!

##### Static View – CTREnc



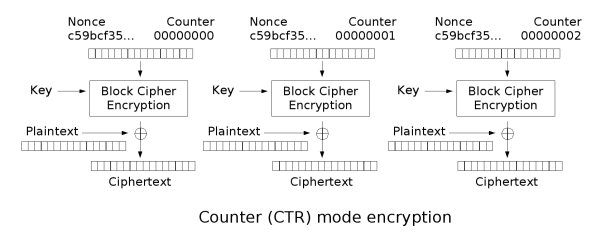
We implement CTREnc by wrapping Bouncy Castle’s SICBlockCipher class. Note that this is not a simple wrapper as presented in the First Layer of SCAPI. Here, the processBlock function in SICBlockCipher class represents one repetition of the loop depicted in the Dynamic View below. Our CTR encryption class is responsible for using a randomized IV and calling the one-step operation of SICBlockCipher in a loop in order to generate the stream.

##### 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.

###### init functions

* call super.init function
* call underlying SICBlockCipher object’s init function

###### decrypt ( cipherText : byte [] ) : byte []

This function assumes that the first block of the ciphertext argument is the CTR. We will call it c0. All the other remaining blocks will be called ci with i = 1 to n, the blocks of the decrypted message will be called mi with i =1 to n.

* Allocate a plaintext buffer of length ciphertext’s length – 1 block.
* Set this.iv:= c0
* For every block in ciphertex ( starting after the CTR) do:
  + mi : = ci XOR prp.computeBlock(this.iv + 1 mod N)
  + set this.iv += 1
* iv member variable is set every step of the previous loop.
* Return the plaintext.

###### decryptWithExistingIV(cipherText : byte[]) : byte[]

This function assumes the first block is already an encrypted block.

* Allocate a plaintext buffer of length ciphertext’s length.
* We use the CTR already set as this.iv in previous call to regular decrypt function.
* For every block in ciphertex do:
  + mi : = ci XOR prp.computeBlock(this.iv + 1 mod N)
  + set this.iv += 1
* iv member variable is set every step of the previous loop.
* Return the plaintext.

###### encAlg (plainText : byte [], iv : byte [], ciphertext : byte [], offset : int )

This function must be implemented in this concrete class. We use a return buffer in the argument in order to save unnecessary copies of the array (which can be very large) in the case that we do not need to return the IV. One time encrypts will call the function with offset equal to 1, whereas loop encryptions will call it with offset seto to 0 because the IV is not needed.

In CTREnc this function performs the CTR mode of operation:

* if offset equals 1 block
  + if lenght of ciphertext is less than length of plaintext +1
    - throw IlegalArgumentException (with explanation)
  + set first block of out buffer with the IV: ciphertext [0] = IV
  + for each block in plaintext do: //i = 0
    - keep a counter of blocks processed (blocksCounter)
    - call SICBlockCipher::processBlock(plaintext[i], 0, ciphertext[**i+1**], 0)
* else if it’s zero
  + for each block in plaintext do: //i = 0
    - keep a counter of blocks processed (blocksCounter)
    - call SICBlockCipher::processBlock(plaintext[i], 0, ciphertext[**i**], 0)
* Set this.iv to this.iv + blocksCounter mod 2n ( Unfortunately SICKBlockCipher does not return the incremented IV so we have to calculate it).

#### 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

## 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.