

E M S C B – Milestone No. I  
Secure Linux Hard-Disk Encryption

DESIGN SPECIFICATION

based on  
European Multilaterally Secure Computing Base (EMSCB)



**Abstract:** The aim of this EMSCB-based security service is to provide a secure hard-disk encryption for Linux, where the secret key information and all related security-critical operations are not under the control of Linux, but under control of an EMSCB application protected and isolated from Linux. This document describes the design specification of the EMSCB-based security service. It contains the architecture description, the design model, and the deployment model of the EMSCB HDD-Encrypter.

Version June 19, 2006

# Contents

<b>1</b>	<b>Architecture Description</b>	<b>2</b>
<b>2</b>	<b>Deployment Model</b>	<b>4</b>
<b>3</b>	<b>Design Model</b>	<b>5</b>
3.1	Design Subsystem LinuxStub . . . . .	5
3.1.1	Informational Part . . . . .	5
3.1.2	Technical Part . . . . .	6
3.2	Design Subsystem HddEnc Server . . . . .	9
3.2.1	Design Class HddEncServer . . . . .	9
3.2.2	Design Class UserModeStrategy . . . . .	11
3.2.3	Design Class HddEncCiphers . . . . .	11
3.2.4	Use-Case Realization of Register Encrypted Hard-Disk	11
3.2.5	Use-Case Realization of Deregister Encrypted Hard-Disk	12
3.2.6	Design Class ConfigurationManagement . . . . .	12
3.2.7	Design Class SystemConfig . . . . .	13
3.2.8	Design Class UserConfig . . . . .	13
3.2.9	Design Class ResourceKey . . . . .	13
3.2.10	Design Class AccessRight . . . . .	14
3.2.11	Use-Case Realization of Initialization . . . . .	14
3.2.12	Use-Case Realization of Add User . . . . .	15
3.2.13	Use-Case Realization of Delete User . . . . .	15
3.2.14	Use-Case Realization of Add Resource . . . . .	16
3.2.15	Use-Case Realization of Delete Resource . . . . .	17
3.2.16	Use-Case Realization of User Rights Administration .	17
3.2.17	Use-Case Realization of Password Change . . . . .	17
3.2.18	Design of GUI Classes . . . . .	18
3.3	Design Subsystem HddEnc Launcher . . . . .	20
3.4	Design Subsystem libCrypto . . . . .	23
3.4.1	Encryption Operation . . . . .	23
3.4.2	Decryption Operation . . . . .	23
3.5	Design Subsystem libUtils . . . . .	24
3.6	Design Subsystem Persistent Storage . . . . .	24
3.6.1	Design Class PersistentStorageElement . . . . .	24
3.6.2	Design Class PersistentStorageServer . . . . .	26
3.6.3	Design Class PersistentStorageDaemon . . . . .	27
3.6.4	TPM Support . . . . .	27
<b>4</b>	<b>References</b>	<b>28</b>

# 1 Architecture Description

The Secure Linux Hard-Disk Encryption is based on the microkernel-based EMSCB security kernel [1]. The Linux operating system is executed as a separate EMSCB application. This allows an architecture where the key critical information of a hard-disk encryption system is stored and handled in a special EMSCB service outside of Linux. This special EMSCB service is the main part of the EMSCB HDD-Encrypter as shown in Figure 1.

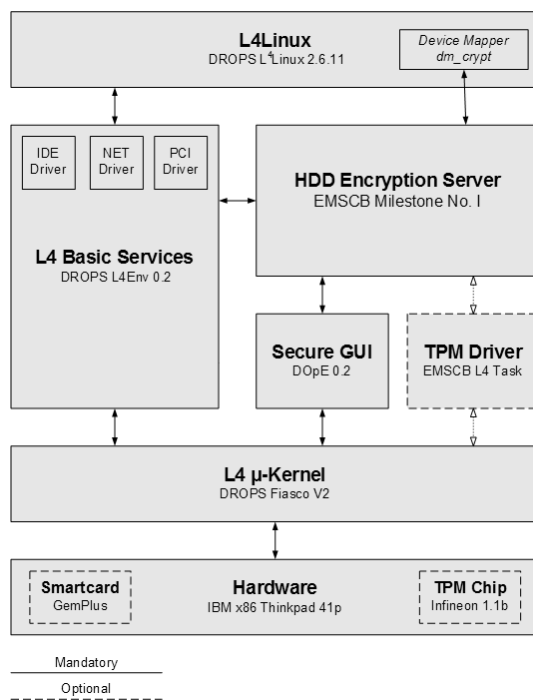


Figure 1: Architecture of secure Linux hard-disk encryption

All key critical information is handled by an EMSCB service, that itself is fully independent from Linux. After a successful authentication process against the encryption server, Linux function that handles the hard-disk encryption just sends the plain text to the encryption server and receives the cipher text afterwards and vice versa without having access to the secret key information.

The authentication process simply authenticates a qualified user, i.e. the data owner, and then provides access to the data to all applications of the respective user. The authentication is performed by providing a password, which is then used to derive an encryption key. Without the correct password the correct encryption key will not be accessible and hence confidentiality is preserved.

We use AES as encryption algorithm with a key size of 128 bits. We derive the key from a given password using the SHA-1 hash function. SHA-1 returns a 160-bit value. Thus we can use the first 128 out of these 160 bits to make up the AES key. Another option would be to use SHA256 and then use 256-bit AES keys.

We use CBC as operation mode of AES because of enhanced security properties. Every block received by the HDD-Encrypter is divided into 16 Bytes chunks, which are processed by the AES algorithm and chained by CBC mode. If we use 512 Bytes as block size, we can use the disk sector number as the Initialization Vector (IV) for the CBC mode regarding this block.

The EMSCB HDD-Encrypter can be run in three operational modes:

- Single-user mode (without Trusted GUI)
- Single-user mode (with Trusted GUI)
- Multi-user mode (with Trusted GUI)

In single-user mode all encrypted partitions are encrypted with one single key, which is derived from the single user's password. In multi-user mode every encrypted partition has its own individual encryption key. The user's password is used to derive another encryption key, which is used to encrypt/decrypt the partition's encryption key. This allows multiple users to share a common encrypted partition but having not to share a common password.

Since the EMSCB encryption server can be used to not only encrypt partitions but also all virtual block devices, we call those "partitions" resources. In multi-user mode it is necessary to define resources and user accounts. If users want to have access to certain resources their access rights to these resources must be specified. Thus, there is a need for user-management, which is handled by the HDD-Encrypter as well.

The EMSCB HDD-Encrypter is composed of several subsystems. Figure 2 shows the composition of the main subsystems. The LinuxStub contains the Linux kernel module, which provides an interface to `dm_crypt` of Linux and uses CORBA inter-process communication (IPC) to communicate with the HDD-Encrypter server. The server subsystem contains all classes that comprise the actual hard-disk encrypter and the user management for the multi-user mode. The third package that contains an individual process is the launcher subsystem. The launcher subsystem is only responsible for displaying a button to start the configuration management in the multi-user mode.

Besides these three subsystems there are some supportive subsystems. First, there is the subsystem common, which is contains commonly used functionality for CORBA IPC and a Message widget. Second, there are application-independent subsystems that can be used by other applications

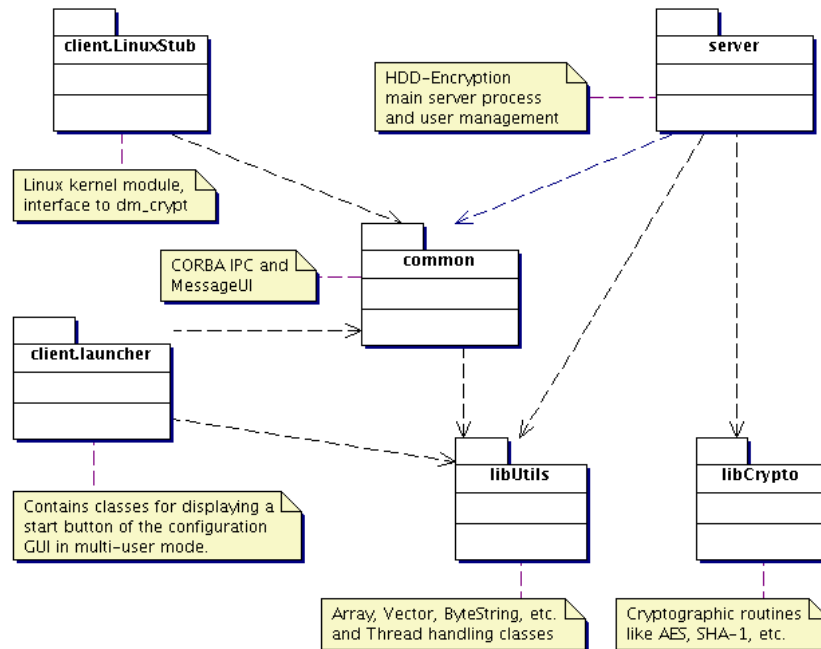


Figure 2: Design architecture of the HDD-Encrypter.

as well: `libUtils` and `libCrypto`. The former contains data structure classes like `Array`, `Vector`, `String`, `ByteString` and Thread handling classes. `libCrypto` contains cryptographic routines like symmetric ciphers, e.g. AES and hash functions, e.g. SHA-1.

## 2 Deployment Model

The EMSCB HDD-Encrypter is designed to run on one machine solely without interaction or communication with other hardware nodes for the purpose of the hard-disk encryption functionality. In our scenario the user is equipped with a personal computer, e.g. a notebook, and uses the HDD-Encrypter to encrypt one, some or all hard-disk partitions to preserve the confidentiality of his data.

The working operating system, i.e. Linux [3], is executed as an isolated application on an underlying security kernel which uses the Fiasco microkernel [4, 5] as basis. The Linux task is used to configure encrypted partitions and to run user applications that use these transparently encrypted resources. Within this process, the `HddEnc LinuxStub` is executed as a Linux kernel module called `emscb-hddenc`. It provides an interface to the Linux `dm_crypt` subsystem and communicates with the `HddEnc Server` subsystem.

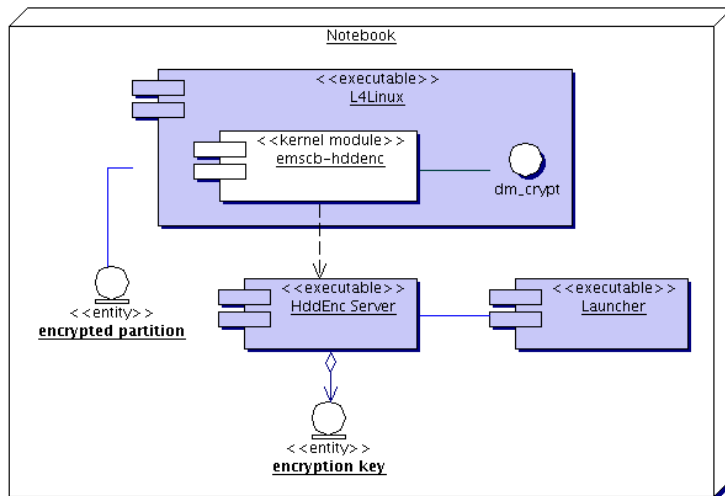


Figure 3: Deployment diagram of the HDD-Encrypter.

The HddEnc Server is executed as another isolated EMSCB service that performs the actual encryption and decryption of (virtual) device blocks. The HddEnc Server process has exclusive control over the used encryption keys and they are completely isolated from the Linux process. To start the user configuration in multi-user mode, there exists another EMSCB application called Launcher, whose only task is to display a button and to start the configuration management when the button is pressed by the user.

### 3 Design Model

#### 3.1 Design Subsystem LinuxStub

##### 3.1.1 Informational Part

Since Linux comes with a very good cryptographic interface inside their kernel with many different ciphers, cryptographic functions can easily be used under Linux. The software implementations of those algorithms are fast and flexible. Every cipher included into Linux has a struct with their algorithm information, e.g., the desired blocksize, key sizes, etc. Current encryption methods under Linux use these cryptographic functions of the Linux kernel to allow encryption of block devices. Therefore a device mapping has to be created, which assigns a cryptographic algorithm incl. the cryptographic key to a regular block device (e.g., a harddisc, USB stick, ...) in order to encrypt it. In the past there were several approaches, but most of them had design flaws. Thus, it was possible to gather information or even recover the cryptographic key during runtime (e.g. cryptographic loopback with

losetup).

Modern approaches use the methods of “device mapping”, a kernel internal function which allows the remapping of one block device to another by performing some action on the data. For example with the device mapping function it is possible to create an image file and map it to a block device, where it can easily be accessed and mounted with regular tools. These features can be used to setup a device mapping including cryptographic functions, called `dm_crypt`. It is now possible to assign a cryptographic function, like AES, with a key, e.g., derived from a user’s password, to a block device and use the new `dm_crypt-block` device instead of the original one.

*Example:* You have a harddisc with three partitions and you want to encrypt the third partition `/dev/hda3`. Then you can create a device mapping with your required algorithm:

```
cryptsetup -h sha1 -c aes-cbc create my_encrypted_hda3 /dev/hda3
```

Then, `cryptsetup` will prompt you for a password and will derive a cryptographic key from it by applying the SHA-1 hash function. You can access the device via the regarding device mapper

```
/dev/mapper/my_encrypted_hda3
```

and not via `/dev/hda3`, since the kernel internal function will route all access to the device mapping through the kernel cryptographic routine and pipe it then in an encrypted form to the real hard-disc.

### 3.1.2 Technical Part

Since we wanted to reuse as much as possible of the original Linux functions, the simplest way to achieve our goal was to build a new internal cryptographic module, which actually performs no encryption/decryption itself, but pipes the data to our HDD-Encrypter outside of Linux. Since regular kernel ciphers are operating on blocksizes between 8 and 16 Bytes, the device mapper uses the blocksize of the cipher in order to read blocks from the filesystem with the filesystem’s blocksize, splits them into the according number of blocks with the blocksize of the cipher and passes them one-by-one to the cipher. Afterwards, the en-/decrypted blocks are put together and stored on the filesystem again.

*Example:* The default block size for reiserfs under Linux is 4096 bytes, which means that encrypting one block of reiserfs-data needs 256 encryption calls with a cipher block length of 16 Byte.

Function calls under Linux are pretty fast, so it is no problem to perform the operation in this mode. But when Linux is running in a separated

environment on top of a microkernel, every encryption call to the HDD-Encrypter needs to perform a task switch, which is slower than direct function calls. For this reason we have set our blocksize to a much larger value, by default to 512 Bytes. This means, that the device mapper believes, that it can pass a block for encryption with 512 Bytes to our module, which would encrypt the 4096 Bytes in only 8 function calls. Surely, the HDD-Encryption server will split the received 512 Bytes block into encryptable blocks with the appropriate cipher block length. But this is much more faster than performing 256 IPC calls.

The operation mode of the HDD-Encrypter's AES is CBC and we could use the currently used disk sector number as Initialization Vector (IV). However, the `dm_crypt` interface for crypto algorithms, where our HDD-Encrypter plugs in, is not capable of passing any information about the disk sector number. This requires the implementation of another interface of `dm_crypt`: the operation mode. If you call `cryptsetup` you usually pass the crypto algorithm and its operation mode as argument:

```
cryptsetup -c aes-cbc ...
```

Using `emscb-cbc` as parameter would not work since the Linux CBC implementation would divide the block into chunks and perform the chaining operation. But our HDD-Encrypter server process already does the chaining operation. Therefore, we would have to implement a separate Linux operation mode module, which passes the currently used disk sector number to the EMSCB HDD-Encrypter but does not perform any chaining operation.

Currently, the HDD-Encrypter always uses 0 as IV. But this is subject to change.

For users' convenience and in order to allow fine-tuning of the blocksize depending on the actual used filesystem, the user is able to modify the `emscb-module` block length during loading.

We register our module as `emscb-hddenc` inside the kernel with the following struct:

```
#define EMSCB_MIN_KEY_SIZE      1
#define EMSCB_MAX_KEY_SIZE     32
#define EMSCB_BLOCK_SIZE       16

static struct crypto_alg emscb_alg = {
    .cra_name = "emscb",
    .cra_flags = CRYPTO_ALG_TYPE_CIPHER,
    .cra_blocksize = EMSCB_BLOCK_SIZE,
    .cra_ctxsize = sizeof(HddEnc_Resource),
    .cra_module = THIS_MODULE,
    .cra_list = LIST_HEAD_INIT(emscb_alg.cra_list),
    .cra_u = {
```

```

.cipher = {
    .cia_min_keysize = EMSCB_MIN_KEY_SIZE,
    .cia_max_keysize = EMSCB_MAX_KEY_SIZE,
    .cia_setkey = emscb_choose_part,
    .cia_encrypt = emscb_encrypt,
    .cia_decrypt = emscb_decrypt}
}

```

This struct is used by the `emscb` kernel module. Figure 4 shows the relevant “classes” of the kernel module. The kernel module is separated into two parts: a plain C Linux implementation (`emscb.c`) and C++ like classes for communication with the HDD Encrypter.

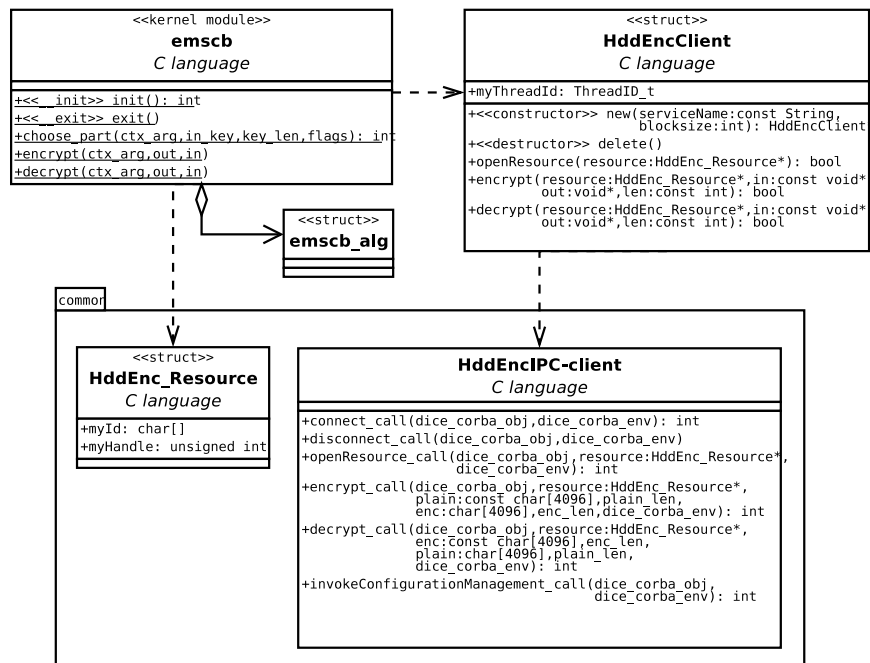


Figure 4: Design of the linux kernel module.

The kernel module uses the `HddEncClient` struct and associated functions to establish a connection between the Linux kernel, especially the `dm_crypt` subsystem and the `HddEnc` server task. Since C++ cannot be used within the Linux kernel, these classes are implemented in C in fact. The `HddEncClient` class hides the IDL interface, as provided by the `HddEncIPC-client` class.

The `HddEncClient` class has the responsibility to provide the following five functions:

1. Connect to the HDD Encrypter upon loading of the module.

2. Disconnect from the HDD Encrypter upon unloading of the module.
3. Open a resource (i.e. set the current encryption key).  
When `cryptsetup` is called, the `dm_crypt` creates a new cipher instance, called `emscb-hddenc`. During the creation, the set-key function is called. Instead of the key, the requested resource is used as a parameter. The kernel module invokes the `openResource` call at the HDD Encrypter. If the call succeeds, the encryption of the resource can be used.
4. Encryption. A complete block of data is being received from the Linux kernel and forwarded to the HDD Encrypter. The block is processed by the resource ID for which the encryption shall take place. The HDD Encrypter encrypts the data and returns the cipher text. The cipher text is send to the Linux kernel.
5. Decryption. The operation is similar to encryption, but with decryption instead.

## 3.2 Design Subsystem HddEnc Server

Within the EMSCB HDD-Encrypter, the encryption and decryption of virtual block devices is actually performed in an isolated service. The HddEnc Server subsystem mainly comprises this server process. Figure 5 shows the class structure of this subsystem.

### 3.2.1 Design Class HddEncServer

This is the main class of the HDD Encrypter. It is responsible for connecting client processes with user configuration and encryption operations. It provides the following operations:

- *Connect*: Only a connected client can call the other functions. Currently, at most one client can be connected at one time.
- *Disconnect*: Disconnects a connected client from the server. Logs off a currently logged-in user.
- *Open resource*: Requests the allocation of a cipher object for the given resource. Ensures that a user is logged in to the system (by methods from `UserModeStrategy`), then fetches the resource key and finally creates the cipher (by methods from `HddEncCiphers`). Returns the handle to the calling client received from `HddEncCiphers` assigned to the opened resource.
- *Encrypt*: Encrypts a data block (arbitrary size, multiple of the cipher block length) for a previously opened resource specified by a handle. Only connected clients may call this function.
- *Decrypt*: Same as encryption, but with decryption.

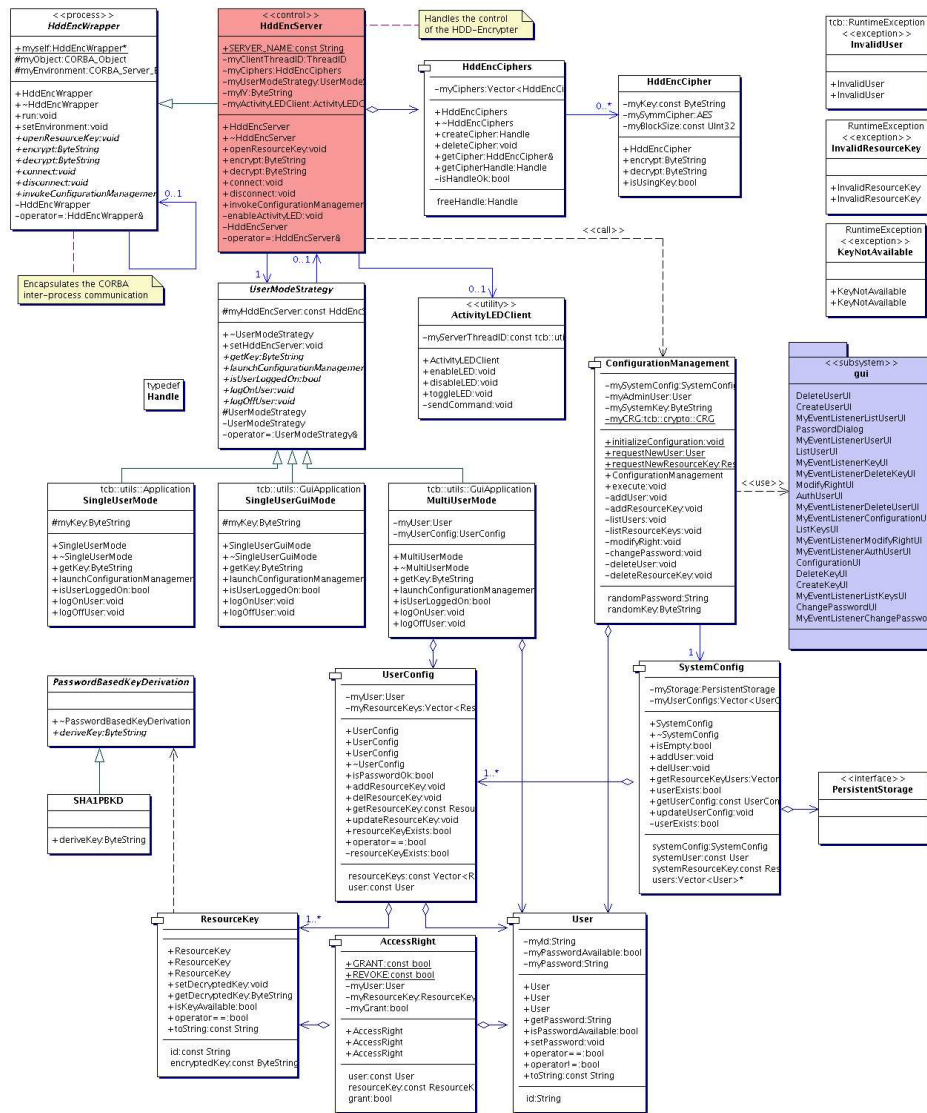


Figure 5: Class diagram of the HddEnc Server subsystem.

- *Launch configuration management:* Invokes the configuration management GUI by means of the `UserModeStrategy` class. Can only be called as long as no user is logged in.

`HddEncServer` derives from `HddEncWrapper`, which provides an interface for IDL function calls. `HddEncWrapper` runs the server main loop and calls requested functions invoked by CORBA IPC calls.

### 3.2.2 Design Class `UserModeStrategy`

Provides functions for logging in and off a user, receiving a resource encryption key and calling the configuration management. This abstract class is an instance of the Strategy Design Pattern. Regarding the different user modes, i.e., single-user with or without GUI or multi-user mode, there exist concrete Strategy classes that implement the respective behavior of the `HddEncServer`.

For example, in single-user mode all resource keys are the same because they are all derived from the password provided by the single user. In multi-user mode each resource has its individual encryption key instead.

### 3.2.3 Design Class `HddEncCiphers`

The `HddEncCiphers` class is a set of cipher objects used by the `HddEncServer`. Each cipher object represents a separate encryption key. The cipher objects are instances of the `HddEncCipher` class.

### 3.2.4 Use-Case Realization of Register Encrypted Hard-Disk

**Resource Setup** The Linux command `cryptsetup` creates a new instance of the `emscb-hddenc` “cipher” kernel module. The kernel module calls “openResource” at the server process. `HddEncWrapper` receives the requests and calls the appropriate `HddEncServer` method.

The `HddEncServer` requests a user to be logged in (`isUserLoggedIn()` and `loginUser()` methods in `UserModeStrategy`), requests the resource key from `UserModeStrategy` and requests a handle for a `HddEncCipher` object from `HddEncCiphers`. Therefore, `HddEncCiphers` looks for a previously allocated `HddEncCipher` object for the given key. If it can find one it returns a handle to the `HddEncCipher` object.

The handle is passed to the `emscb-hddenc` kernel module. The module returns with success and stores the received handle in private data for the requested resource.

**Encryption/Decryption** The `emscb-hddenc` kernel module is called by the Linux kernel. The module retrieves the handle from private data and calls the encrypt or decrypt operation on the L4 server task, respectively.

The HddEncWrapper receives the request and calls the appropriate HddEncServer method. The HddEncServer receives the HddEncCipher object from HddEncCiphers by the received handle and calls the encrypt or decrypt operation on the HddEncCipher, respectively. The HddEncCipher object creates a new CBC object and hands in the complete data packet which is going to be processed for encryption or decryption. The encrypted (or decrypted) data is then returned back to the HddEncServer, which in turn returns the processed data to the `emscb-hddenc` kernel module by IPC calls.

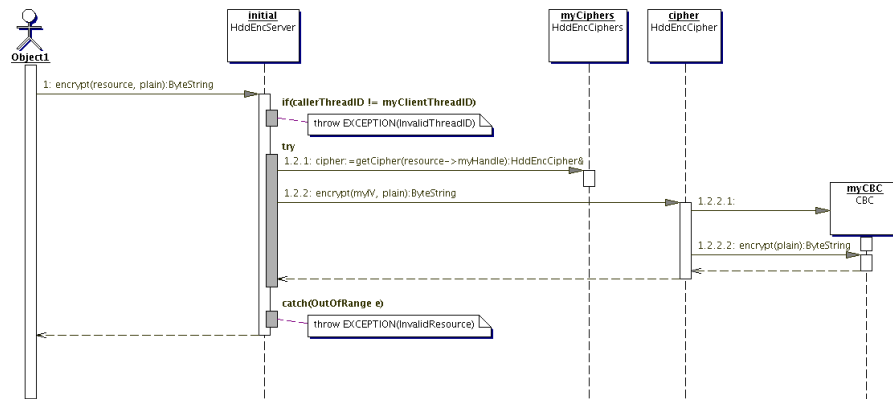


Figure 6: Sequence diagram of the encryption within HddEncServer.

### 3.2.5 Use-Case Realization of Deregister Encrypted Hard-Disk

The de-registering process is performed by using the `umount` and `cryptsetup remove` Linux commands within Linux.

If a different user wants to register encrypted hard-disks afterwards, the current user must log off. At the moment, this can only be done by unloading the `emscb-hddenc` kernel module in Linux. This results in disconnecting from the HDD Encrypter, which implicitly logs off a currently logged in user.

### 3.2.6 Design Class ConfigurationManagement

The ConfigurationManagement class is responsible for management control. It controls all tasks on SystemConfig and related classes, i.e. creation and deletion of users and resources, modification of access rights to resources. It depends on the GUI subsystem to display graphical user interface dialogs for human interaction.

The configuration data consist of the classes as depicted in Figure 7.

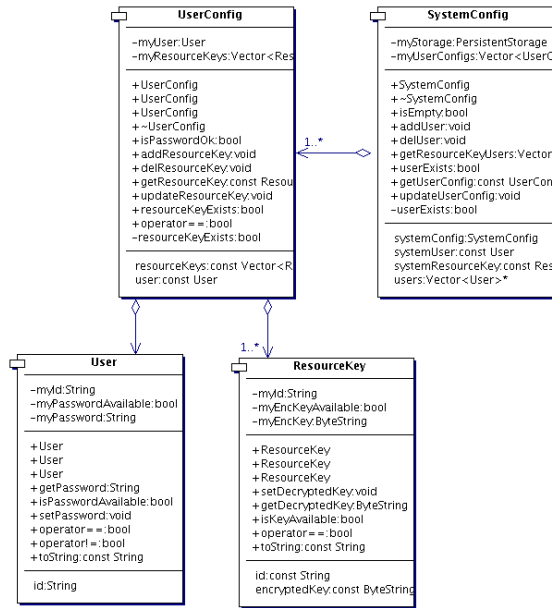


Figure 7: Classes used for configuration management.

A User object contains the characteristic data of a user, like the username and the password. The UserConfig is linked to a User object and contains a list of resources that can be accessed by the user. Each ResourceKey object consists of an ID (its name) and a key  $k_{\text{RES}}$ . For setting the  $k_{\text{RES}}$ , it is encrypted with a user key  $k_{\text{USER}}$ , which is derived from the user's password.

### 3.2.7 Design Class SystemConfig

The SystemConfig is a set of all known UserConfig objects. It provides methods for adding, retrieving and deleting User objects.

User objects contain a username, a password  $p_{\text{USER}}$ . The password is only held in RAM but never written to the persistent storage.

### 3.2.8 Design Class UserConfig

Objects of the UserConfig class possess one User object and a set of ResourceKey objects. The UserConfig class provides operations for adding, retrieving and deleting ResourceKey objects.

### 3.2.9 Design Class ResourceKey

ResourceKey objects consist of a key  $k_{\text{RES}}$  for an encrypted resource. The key  $k_{\text{RES}}$  itself is encrypted by a key  $k_{\text{USER}}$  that is derived from the user's

password  $p_{\text{user}}$  using an implementation of PasswordBasedKeyDerivation (currently we use SHA-1, but more sophisticated methods could be possible):

$$k_{\text{user}} = \text{PBKD}(p_{\text{user}})$$

The ResourceKey object stores  $\text{enc}_{k_{\text{user}}}(k_{\text{res}})$ . The key  $k_{\text{res}}$  can only be retrieved or stored by supplying the user’s password  $p_{\text{user}}$ . Each user having access to a resource has an own corresponding ResourceKey object stored in his UserConfig object. ResourceKey objects can be removed from a UserConfig without providing a password. However, this task can only be performed by an administrator user.

### 3.2.10 Design Class AccessRight

The class AccessRight is responsible for correlating ResourceKey objects with User objects. Instances of AccessRight are not stored within the SystemConfig. They are only needed for changing the available ResourceKeys in a UserConfig object.

### 3.2.11 Use-Case Realization of Initialization

If the HddEnc server is started for the very first time in multi-user mode, initial configuration data must be created.

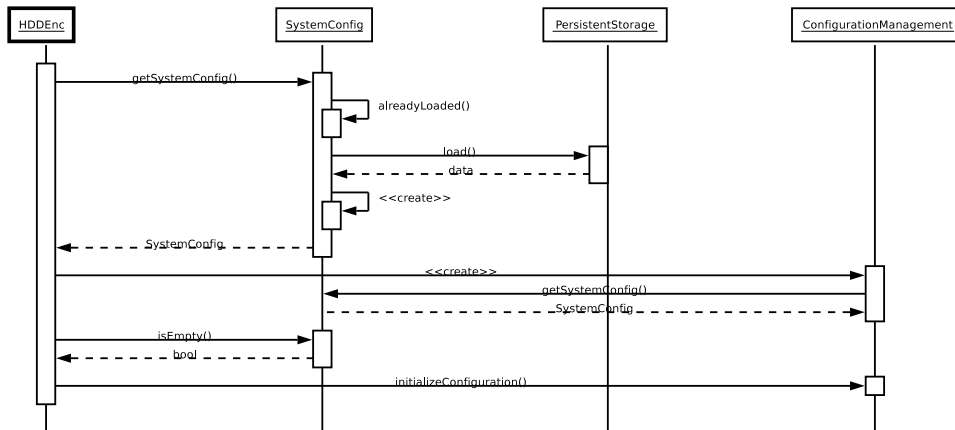


Figure 8: Sequence diagram Configuration Management: HDDEncStartup

**System User and System ResourceKey** A valid SystemConfig always contains a special UserConfig for a special system user called “\_SYSTEM\_”. The system user has access to all available resources: all ResourceKey objects are stored within the system user’s UserConfig and the corresponding

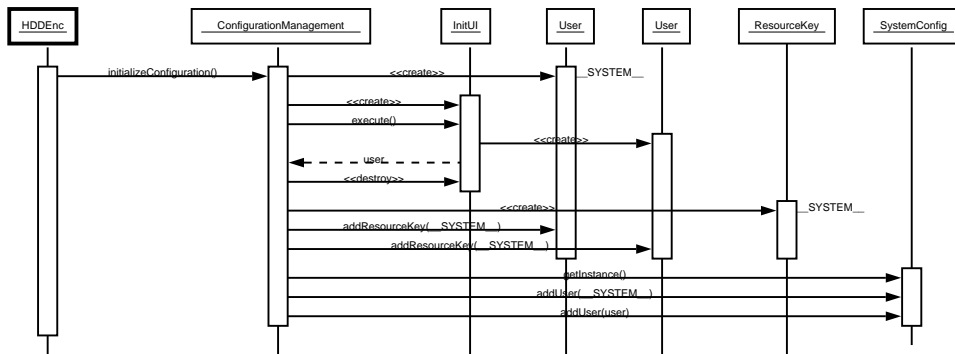


Figure 9: Sequence diagram Configuration Management: InitializeConfiguration

resource encryption keys are encrypted with the password of the system user.

The password of the system user is randomly chosen during the one-time initialization of the SystemConfig object. Creating new resources is only possible if the password of the system user is known. For a given access to the resources of the system user, an administrator user is granted access to the special system resource “`__SYSTEM__`”.

The stored resource key of the system resource, granted to a certain user, simply is the password of the system user. It is encrypted with the user’s password as usual:

$$k_{SysRes} := p_{SysUser}$$

A user with administration rights simply has a valid system ResourceKey object within his UserConfig object. Thus an administrator is able to access all resources in plain text, since all resources can be assigned to him. This results in the fact that the corresponding resource encryption keys would be encrypted with the administrator’s password, too.

### 3.2.12 Use-Case Realization of Add User

Addition of users can be done without restrictions. The new user object is simply added to the SystemConfiguration object. The SystemConfiguration then automatically creates an empty UserConfig object.

### 3.2.13 Use-Case Realization of Delete User

Deletion of users can also be done without any restrictions. However, the user currently logged in cannot be removed. This implies that there must exist one administrator user at least.

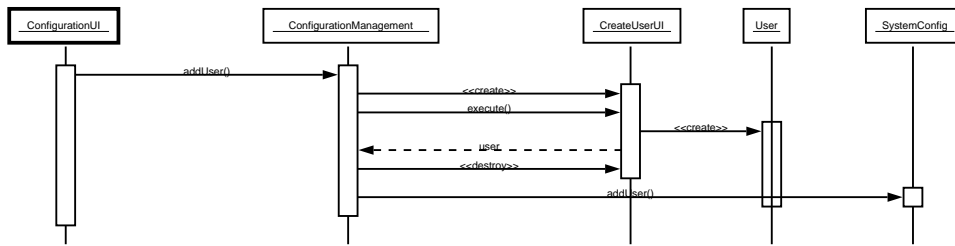


Figure 10: Sequence diagram Configuration Management: AddUser

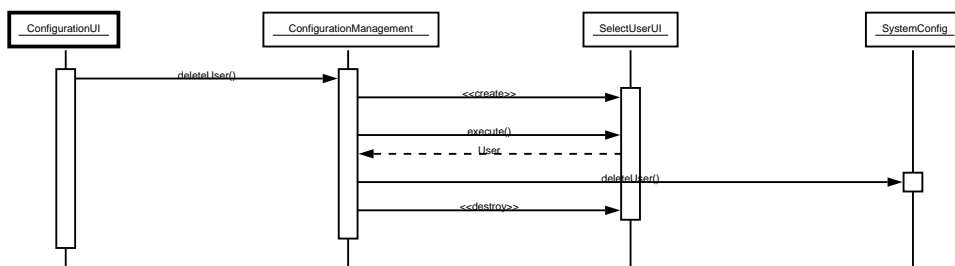


Figure 11: Sequence diagram Configuration Management: DeleteUser

By supplying the desired User object to the SystemConfiguration, it automatically removes the corresponding UserConfig object and all included ResourceKeys from the SystemConfiguration.

### 3.2.14 Use-Case Realization of Add Resource

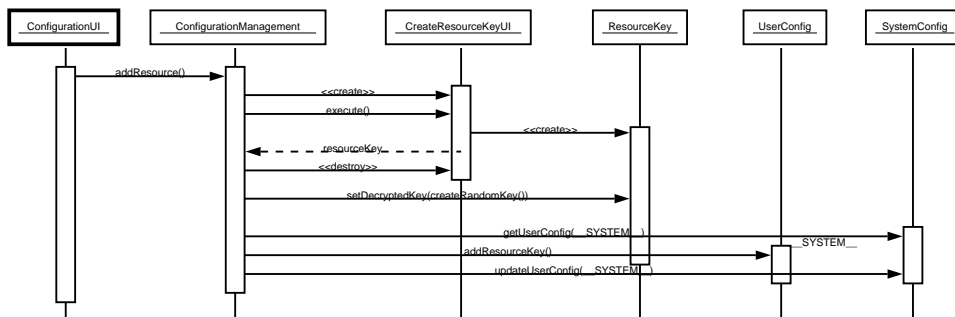


Figure 12: Sequence diagram Configuration Management: AddResource

To perform this task we need to have access to the system user. This implies that the administrator user must have the system resource in his UserConfig.

When a new ResourceKey object is created, a randomly chosen key  $k_{res}$  is created. The new ResourceKey object is stored in the UserConfig of the system user. The key  $k_{res}$  is encrypted with the password of the system user and it is stored in the ResourceKey object.

### 3.2.15 Use-Case Realization of Delete Resource

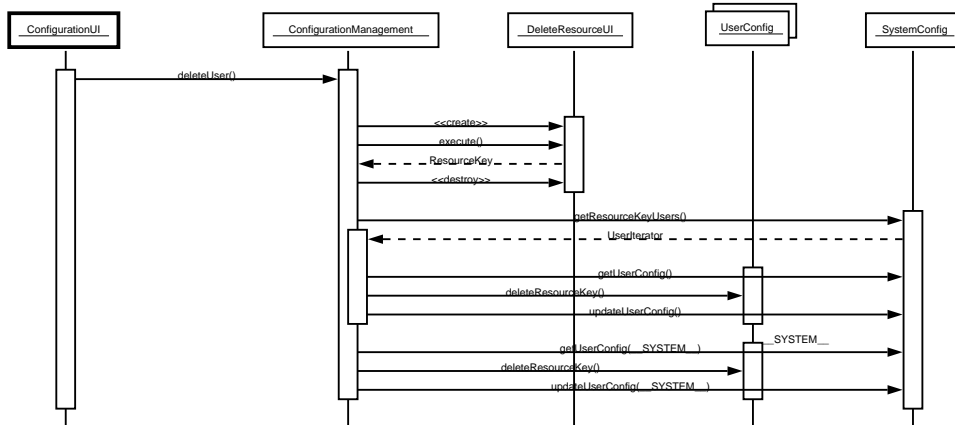


Figure 13: Sequence diagram Configuration Management: DeleteResource

Deletion of resources can be done without restrictions. Upon deletion of a resource from the system user, all instances of that resource at all UserConfig objects must be deleted as well.

### 3.2.16 Use-Case Realization of User Rights Administration

For granting a resource to a user, both the password of the system user and of the user are needed. The ResourceKey object is retrieved from the system user. The resource key  $k_{res}$  is retrieved from the ResourceKey object by supplying the password of the system user. The decrypted resource key can then be stored to the UserConfig object of the requested user within a new ResourceKey object by encrypting the key with the password of the user.

### 3.2.17 Use-Case Realization of Password Change

**Single-user mode** To change the password used for deriving the encryption key, the user has to create a new Linux (virtual) device and call `cryptsetup` with this device as argument together with the new password. Then, the user has to copy all data from the old encrypted device to the newly created one. Afterwards he must delete the old encrypted device.

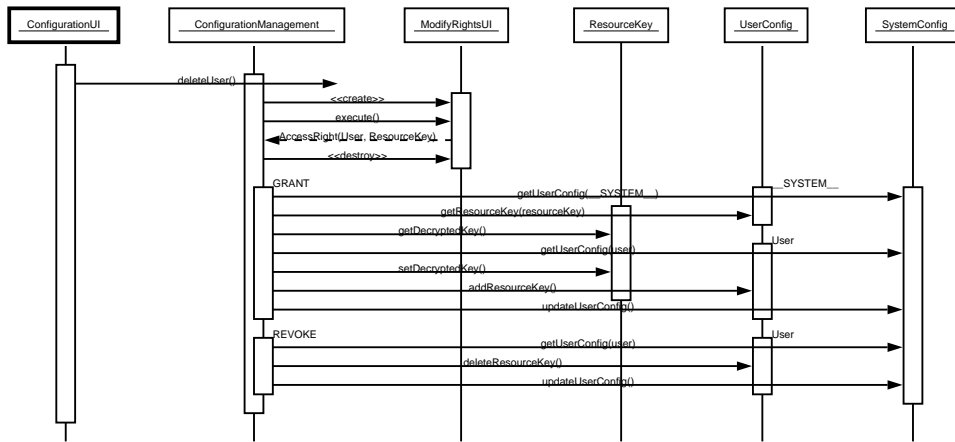


Figure 14: Sequence diagram Configuration Management: ModifyRights

**Multi-user mode** Changing a user’s password in multi-user mode means re-encrypting all resource keys associated with the user using the new password (i.e. a password-derived key).

The respective dialog could be implemented within the configuration menu. However, this would require the administrator user to log in first. Another solution would be to integrate the Password Change dialog call button within the launcher. This would require the HddEncServer to be able to have two users logged in simultaneously or restrict Password Change to the currently logged in user.

### 3.2.18 Design of GUI Classes

The GUI of the HddEncServer, which is needed in multi-user mode for configuration and message information, uses the DOpE [2] server subsystem. DOpE uses a Tk-script like programming language to define and display graphical user interface elements. The use of this script language and the associated event handling mechanism is encapsulated within several C++ classes.

Each GUI dialog of the HDD-Encrypter has its own class. And each class has an associated event listener class. The structure of the HddEnc GUI is shown in Figure 15.

Upon creation of a GUI object the respective constructor defines the script commands for DOpE to create the dialog. Furthermore the constructor creates corresponding event listener objects which are responsible for event handle callbacks of DOpE.

The event handling uses listener objects. An instance of the Reflector Pattern is used to transform a static callback to an instance method call:

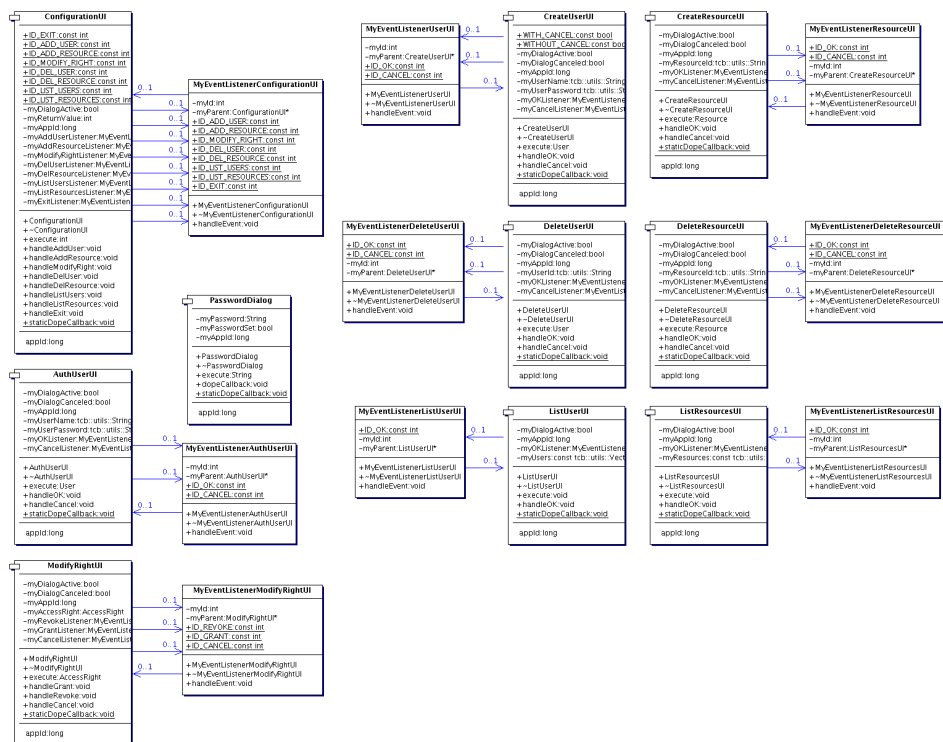


Figure 15: Structure of the GUI classes.

```

void ConfigurationUI::staticDopeCallback(dope_event *e, void *arg) {
    ((MyEventListenerConfigurationUI*)arg)->handleEvent(e);
}

```

The corresponding event listener object then calls back an instance method of the GUI class, to which the event listener has a parent link:

```

void MyEventListenerConfigurationUI::handleEvent(dope_event *e)
{
    if (myId == ID_ADD_USER) {
        myParent->handleAddUser();
    } else if (myId == ID_ADD_RESOURCE) {
        myParent->handleAddResource();
    }
    ...
}

```

As an example, Figure 16 shows the event handling mechanism for the configuration management in the case when the “Add User” button is pressed.

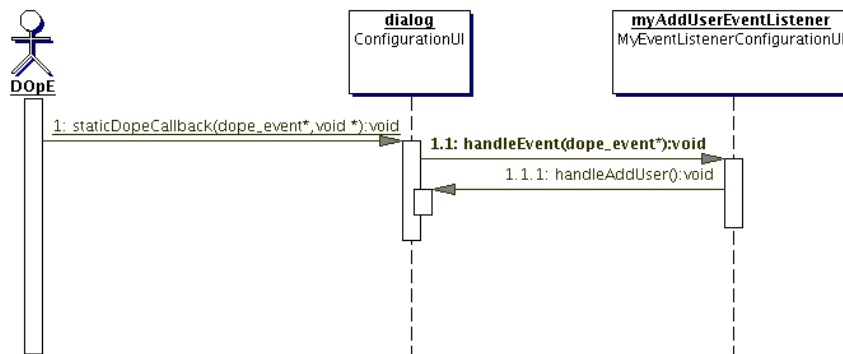


Figure 16: Sequence diagram of the GUI event handling.

Each GUI class has an `execute()` method, which displays the dialog window, starts the DOpE event loop, and closes the dialog window when the event loop has finished, that is when a special flag `myDialogActive` is set to false. This flag is set within a event handling method of the GUI class, respectively.

### 3.3 Design Subsystem HddEnc Launcher

The sole function of the HddEnc Launcher subsystem is to provide a separate L4 process which displays a start button for the configuration management.

In case this button is pressed the LauncherUI sends an IPC call to the HddEnc server task to start the configuration management user interface dialog.

If the HddEnc server is executed in multi-user mode the configuration management dialog will show up after a successful user authentication of the administrator user.

If the HddEnc server is executed in single-user mode or if any user is currently logged on, the configuration request will be declined and an error message will be displayed to the user instead.

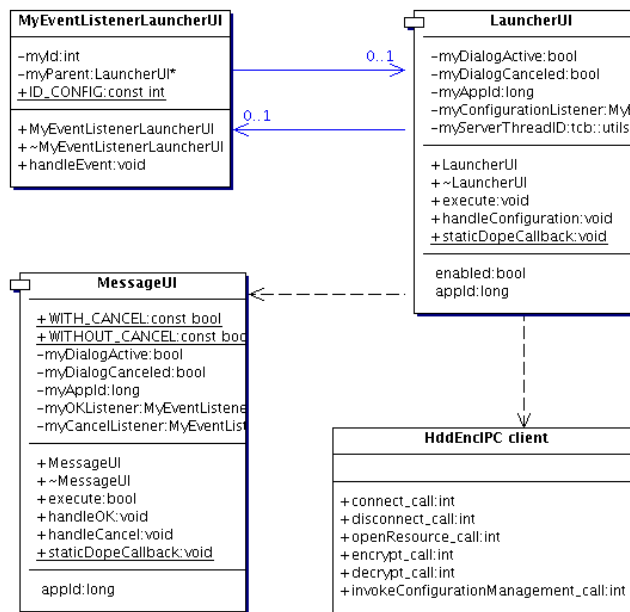


Figure 17: Class diagram of the HddEnc Launcher subsystem.

The class LauncherUI is the central class of the HddEnc Launcher process. It is responsible for handling the static callback of the DOpE subsystem, which is called when the button of LauncherUI is pressed. The static callback method in turn calls the event listener object passed as an argument of the static callback.

The event listener object calls the LauncherUI object again to handle the configuration call. The LauncherUI now calls the respective function of the HddEncIPC-client to invoke the configuration management control within the HddEnc Server. The sequence of calls can be seen in Figure 18.

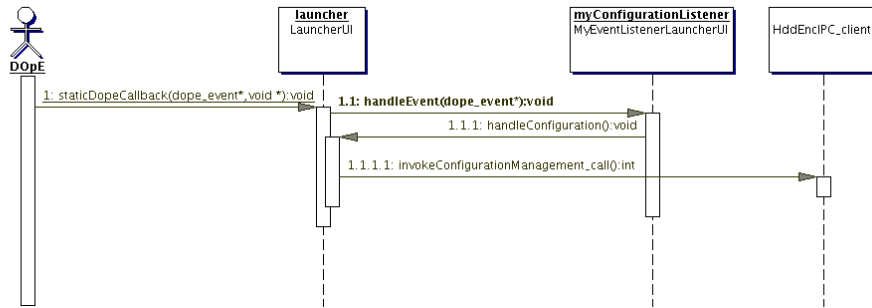


Figure 18: Sequence diagram of the Launcher event handling.

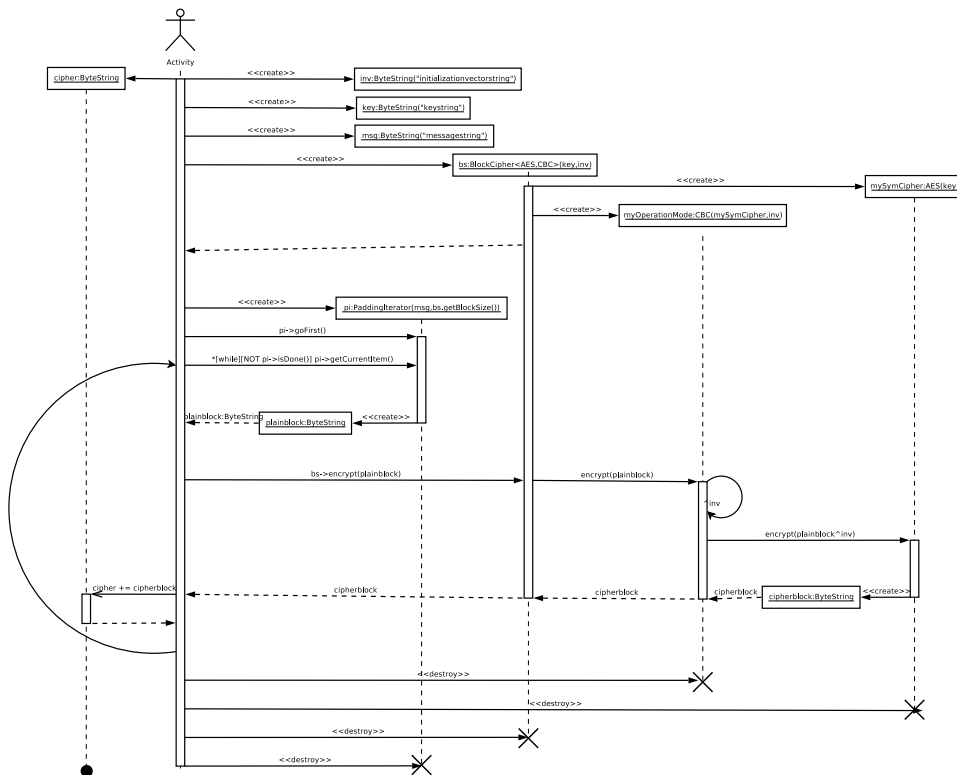


Figure 19: Sequence diagram encryption

### 3.4 Design Subsystem libCrypto

#### 3.4.1 Encryption Operation

First, the block cipher template *bc* creates an AES *SymmetricCipher* object named *mySymmetricCipher* using the *ByteString* *key* as well as the CBC *SymmetricCipher* operation mode object named *myOperationMode* using *mySymmetricCipher* and the initialization *ByteString* object *inv*. Then the *PaddingIterator* *pi* is used to divide the message *ByteString* object *msg* into *ByteString* objects of *mySymmetricCipher* blocksize including appropriate padding. Now the block cipher *encrypt()* method passes all *plainblock* objects to its *myOperationMode* CBC encryption object that in turn applies the CBC chaining followed by an AES encryption using *mySymmetricCipher*. Finally, the *cipher* *ByteString* combines all cipher blocks to a single cipher text object.

#### 3.4.2 Decryption Operation

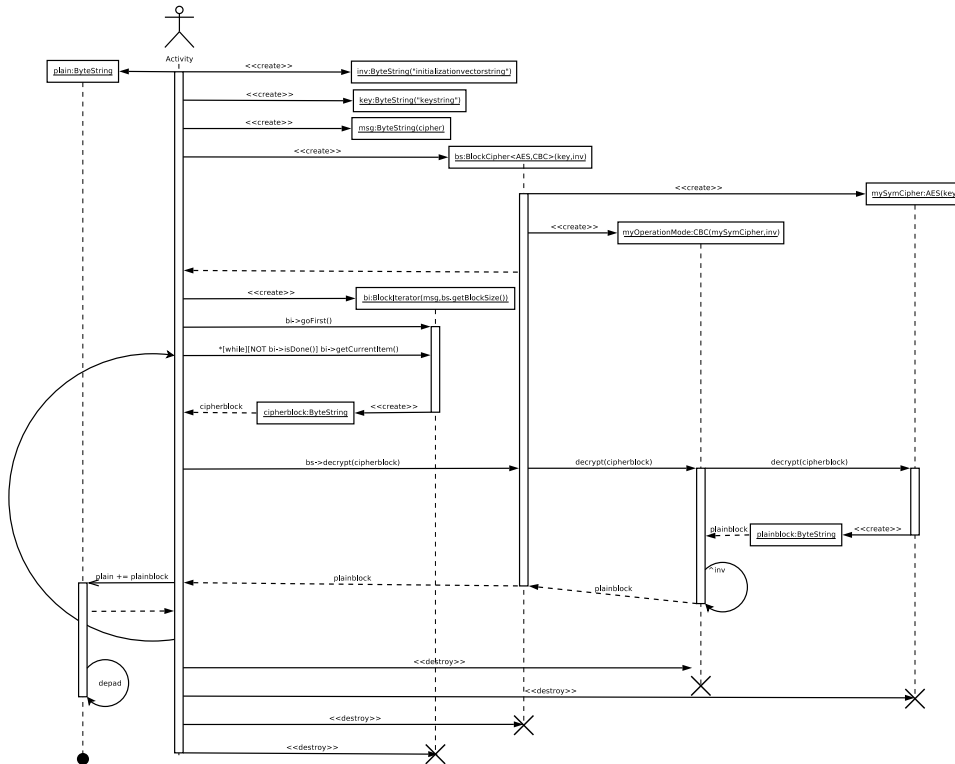


Figure 20: Sequence diagram decryption

First, the block cipher template *bc* creates an AES *SymmetricCipher*

object named *mySymmetricCipher* using the *ByteString* *key* as well as the CBC *SymmetricCipher* operation mode object named *myOperationMode* using *mySymmetricCipher* and the initialization *ByteString* object *inv*. Then the *BlockIterator* *bi* is used to divide the (cipher) message *ByteString* object *msg* into *ByteString* objects of *mySymmetricCipher* blocksize. Now the block cipher *decrypt()* method passes all *cipherblock* objects to its *myOperationMode* CBC encryption object that in turn applies the AES decryption using *mySymmetricCipher* followed by the CBC chaining. The *plain ByteString* combines all plain blocks to a single plain text object. Finally, the *depad()* function removes the padding bytes added in the encryption process before.

### 3.5 Design Subsystem libUtils

This subsystem contains helper classes like Array, Map, String, etc. One important class is *ByteString*, that is used within the encryption and decryption operations of the HDD-Encrypter. Another interesting class is *ThreadId*, which is used for thread handling purposes. Figure 21 shows the class structure of the *libUtils* subsystem.

### 3.6 Design Subsystem Persistent Storage

Persistent Storage saves objects of a client process which are momentarily in the transient memory on a persistent medium. To do that, the objects which shall be stored must be serialized. This means, that a complex object has to be converted into a byte array. The Persistent Storage saves the array and allocates it to a specific client process. Thus a client can reclaim back the object later.

Within the multi-user mode of the HDD-Encrypter the Persistent Storage subsystem is used to store user and key configuration data persistently. This subsystem can be divided into three different parts, see Figure 22 for an overview.

#### 3.6.1 Design Class PersistentStorageElement

*PersistentStorageElement* is used by every client process that wants to store objects persistently. This class is responsible for encapsulating the IDL interface. It is designed to assist the user and to provide a minimal interface. An improvement of the IPC interface to the *PersistentStorageServer* will not resolve in a change of the class interface.

This class is responsible to perform the following tasks:

- register the client process at the application manager, if not happened already
- save the *ByteBuffer* which contains the serialized object



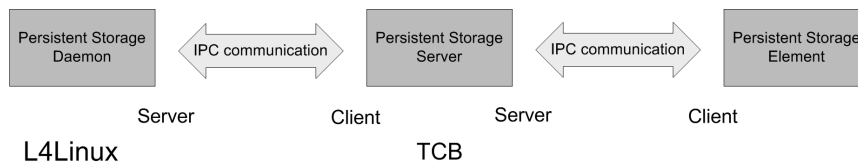


Figure 22: Design of the Persistent Storage subsystem.

- transmission of the ByteBuffer to the PersistentStorageServer
- loading of the ByteBuffer from the PersistentStorageServer
- access control on the ByteBuffer
- access control on the IPC functions

### 3.6.2 Design Class PersistentStorageServer

The PersistentStorageServer is a process that runs independently on the same system. It provides an IPC interface which is used by the PersistentStorageElement class to store data persistently. In the first iteration, the IPC interface is based on strings, and will later be converted to a transfer via shared memory.

The PersistentStorageServer must definitely be part of the TCB, because it deals with sensible and security-critical data, which is sent to the PersistentStorageServer. The PersistentStorageServer can also inquire the Hash ID of a demanding client in order to allocate the data to an according client process.

At a later time, the PersistentStorageServer shall overtake the encryption of the data it has received, so that there is no possibility for unencrypted data to escape the system.

Currently, the possibility to store data persistently in the TCB does not exist. For this reason, data which is transferred to the PersistentStorageServer has to be forwarded to an L4 Linux process. This task is managed by the PersistentStorageDaemon as long as data can not be stored in the TCB persistently. Sending confidential data from the TCB to the L4Linux process includes a security-risk, which can not be avoided at the present time.

To minimize the security-risk in a concrete application, all data from the PersistentStorageServer has to be encrypted before it can be forwarded to the PersistentStorageDaemon.

Tasks:

- Identification of the client process for definite assignment of the data
- Transfer of the data to the Persistent Storage Daemon for persistent storage

Tasks in a coming iteration:

- Creation and maintenance of a shared memory division for transmission of data between Persistent Storage Server and Persistent Storage Element
- Encryption of data and transmission into the Persistent Storage Daemon

### 3.6.3 Design Class PersistentStorageDaemon

The PersistentStorageDaemon is a discrete L4Linux process which receives data per string IPC from the PersistentStorageServer on the TCB. This data is stored persistently within L4Linux. In addition to the actual data, the PersistentStorageDaemon also receives the name of the file in which the data will be stored from the PersistentStorageServer. The PersistentStorageDaemon does not need any specifications about the client process, because these have already been evaluated by the PersistentStorageServer.

Tasks:

- Persistent storage of the data of the Persistent Storage Server
- Loading of data from the persistent storage and transmission to the PersistentStorageServer

### 3.6.4 TPM Support

The PersistentStorageServer binds encryption keys to a user authorization secret, hardware components or the trusted software modules. Binding to hardware and/or software components requires a trusted hardware component, e.g. a TPM.

The TPM uses on-chip registers (Platform Configuration Registers, PCRs) to store measurements (i.e. hash values) of hardware and software components securely. The TPM sealing command may subsequently bind data to these PCRs. The resulting binary data is then stored persistently.

For our application certain PCRs should reflect the integrity of the trusted components. This can be achieved as follows:

1. A TPM-aware (trusted) BIOS measures the MBR before execution.
2. The bootloader measures each boot stage before execution.
3. The bootloader is completely loaded. The PCRs now reflect the integrity of the boot process (“authenticated boot”).
4. The trusted software components are digitally signed. The bootloader checks their signatures before execution. The corresponding public key is hard-coded into the bootloader. If a signature check fails the PCR values are invalidated and the user is requested for interaction (“secure boot”)

The alternation of authenticated and secure boot allows secure updating of system components without “resealing” of secrets.

## 4 References

- [1] EMSCB PROJECT CONSORTIUM. The EMSCB project. <http://www.emscb.org>.
- [2] FESKE, N., AND HELMUTH, C. Overlay window management: User interaction with multiple security domains. Tech. Rep. TUD-FI04-02-März-2004, TU Dresden, Dresden, Germany, Mar. 2004.
- [3] HÄRTIG, H., HOHMUTH, M., AND WOLTER, J. Taming linux. In *Proceedings of PART'98* (1998), TU Dresden.
- [4] LIEDKE, J. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)* (Copper Mountain Resort, Colorado, Dec. 1995). Appeared as ACM Operating Systems Review 29.5.
- [5] LIEDKE, J. *L4 Reference Manual*. GMD/IBM Watson Technical Report, 1996.