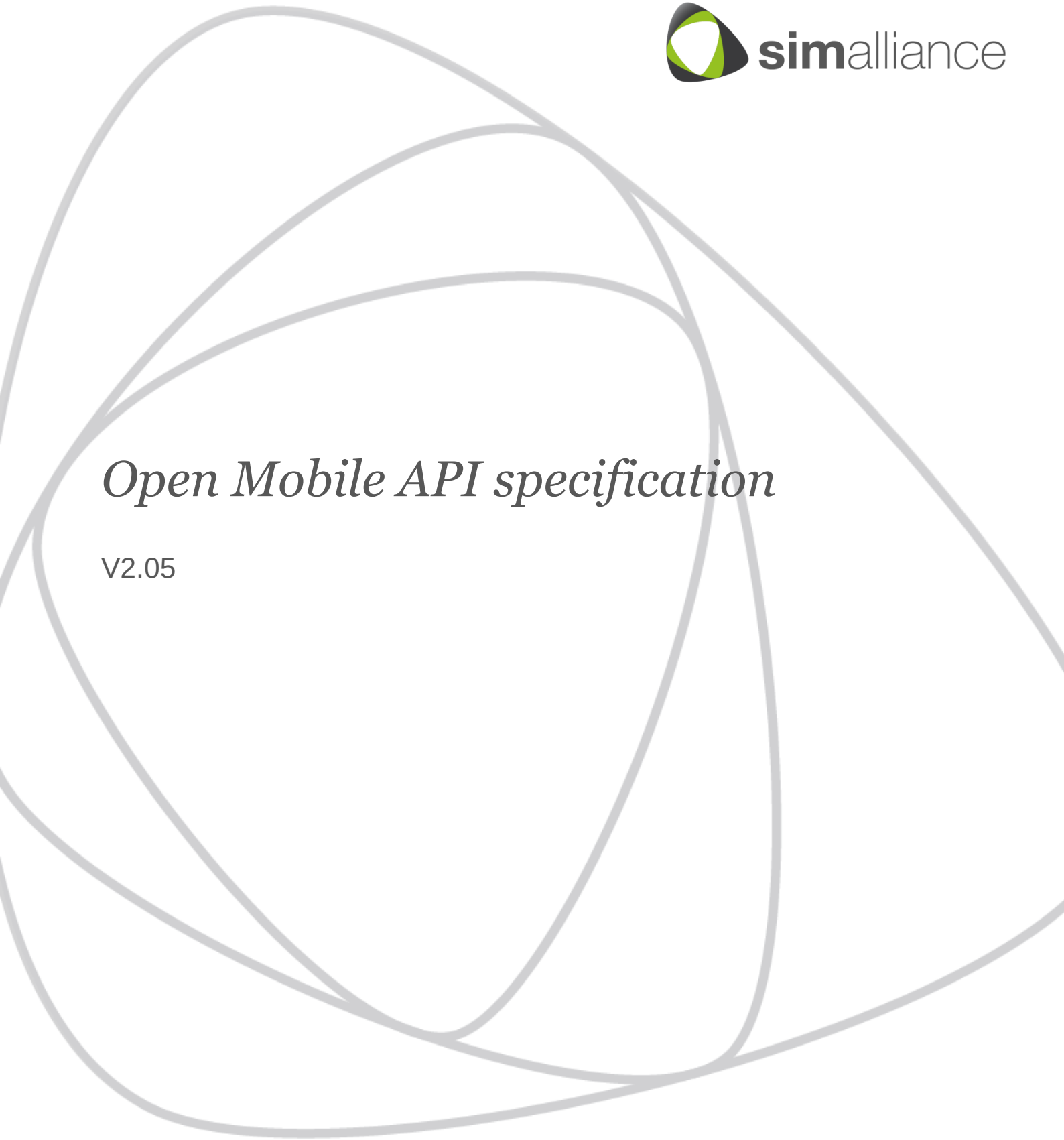

Open Mobile API specification

V2.05

Table of Contents

Table of Figures

Table of Tables

1. Terminology

The given terminology is used in this document.

1.1 Abbreviations and Notations

Table 1-1: Abbreviations and Notations

Abbreviation	Description
SE	Secure Element
API	Application Programming Interface
ATR	Answer to Reset (as per ISO/IEC 7816-4)
APDU	Application Protocol Data Unit (as per ISO/IEC 7816-4)
ISO	International Organization for Standardization
ASSD	Advanced Security SD cards (SD memory cards with an embedded security system) as specified by the SD Association.
OS	Operating system
RIL	Radio Interface Layer
SFI	Short File ID
FID	File ID
FCP	File Control Parameters
MF	Master File
DF	Dedicated File
EF	Elementary File
OID	Object Identifier
DER	Distinguished Encoding Rules of ASN.1
ASN.1	Abstract Syntax Notation One

1.2 Terms

Table 1-2: Terms

Term	Description
Secure Element	A Secure Element (SE) is a tamper-resistant component which is used to provide the security, confidentiality, and multiple application environments required to support various business models. For example UICC/SIM, embedded Secure Element, Secure SD card, …
Applet	General term for Secure Element application: An application as described in [1] which is installed in the SE and runs within the SE. For example a JavaCard™ application or a native application
Application	Device/Terminal/Mobile application: An application which is installed in the mobile device and runs within the mobile device
Session	An open connection between an application on the device (e.g. mobile phone) and a SE.
Channel	An open connection between an application on the device (e.g. mobile phone) and an applet on the SE.

2. Informative References

Table 2-1: Informative References

Specification	Description
[1] GP 2.2	Global Platform Card Specification 2.2
[2] ISO/IEC 7816-4:2005	Identification cards - Integrated circuit cards - Part 4: Organisation, security and commands for interchange
[3] ISO/IEC 7816-5:2004	Identification cards - Integrated circuit cards - Part 5: Registration of application providers
[4] ISO/IEC 7816-15:2004	Identification cards - Integrated circuit cards with contacts - Part 15: Cryptographic information application
[5] PKCS #11 v2.20	Cryptographic Token Interface Standard Go to following website for PKCS#15 documentation: http://www.rsa.com/rsalabs/node.asp?id=2133
[6] PKCS #15 v1.1	Cryptographic Token Information Syntax Standard
[7] Java™ Cryptography Architecture API Specification & Reference	Go to the following website for JCA documentation: http://download.oracle.com/javase/1.4.2/docs/guide/security/CryptoSpec.html
[8] ISO/IEC 8825-1:2002 | ITU-T Recommendation X.690 (2002)	Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)
[9] GlobalPlatform Secure Element Access Control, V1.0	Specification for controlling access to Secure Elements based on access policies that are stored in the Secure Element

3. Overview

The API specified in this document enables mobile applications to have access to different Secure Elements in a Mobile such as SIMs or embedded SEs.
This specification provides interface definitions and UML diagrams to allow the implementation on the various mobile platforms and in different programming languages.
The namespace used shall be org.simalliance.openmobileapi,

4. Architecture

The following picture provides an overview of the Open Mobile API architecture.
The architecture is divided into three functional layers:

- The Transport Layer is the foundation for the Service Layer APIs. It provides general access to Secure Elements when an application is accessing it via the generic Transport API. The Transport Layer is using APDUs for accessing a Secure Element (see chapter 6 for details).
- The Service Layer provides a more abstract interface to various functions on the Secure Element. They will be much easier to use by application developers than the generic transport API. One example could be an E-Mail application that uses a sign() function of the Crypto API and let the Crypto API do all the APDU exchange with the Secure Element (rather then handle all the needed APDUs directly in the E-Mail application).
- The Application Layer represents the various applications that make use of the Open Mobile API

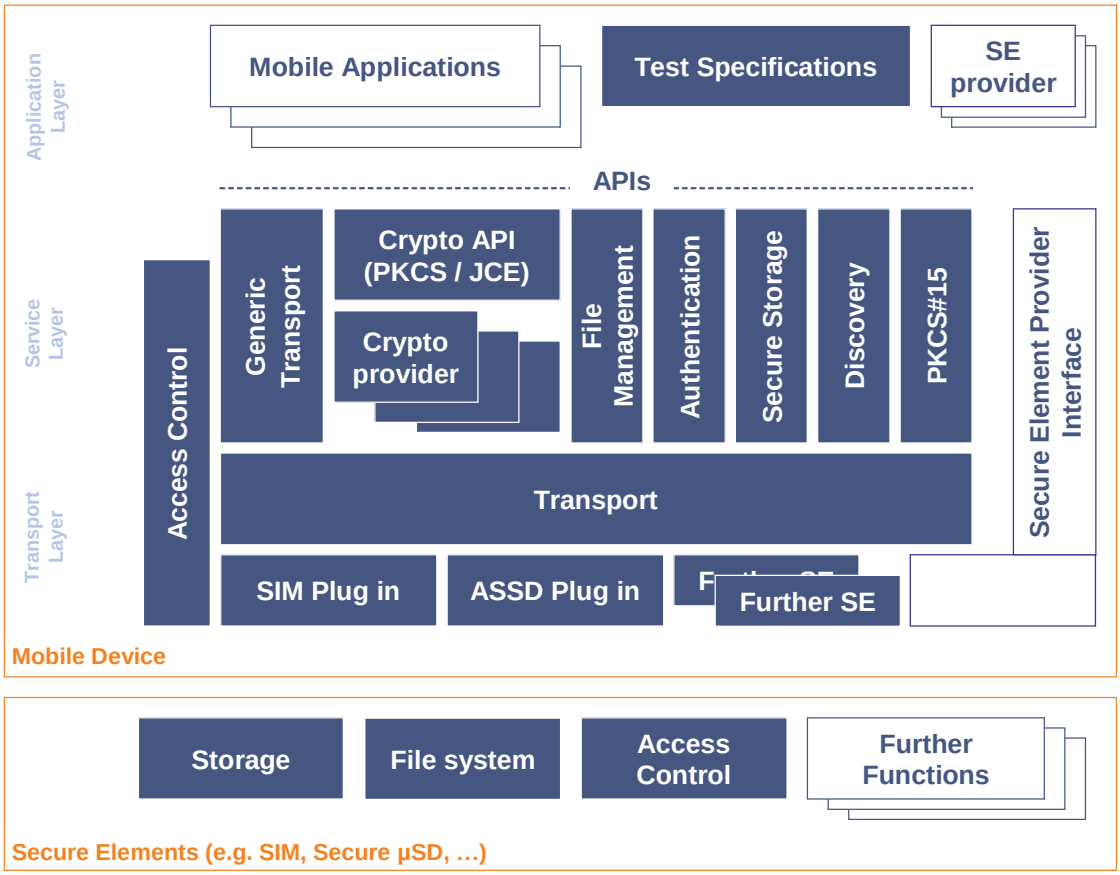

Figure 4-1: Architecture overview

The architecture can be mapped to different operating systems and depending on the OS it might look different.
The description of the APIs is using an abstraction level that allows an easy implementation in different programming languages (e.g. Java or C++).

5. API Description

This document defines the Open Mobile API using a class-by-class description.
The API is providing an interface definition that can be implemented on various programming languages.

The following types are used to describe return values, parameters and errors. If supported by the platform errors may be mapped to exceptions.

Value types;

Boolean:	a primitive type, can be true or false
Int:	a primitive type, contains a 4 byte integer value
byte[]:	an array of single byte (8 bits) values
String:	a string of characters
Context:	an object representing the execution context of an application
Void:	not a type, indicates that the method has no return value

Error types:

IOError:	an error related to communication (I/O)
SecurityError:	an :error related to security conditions not being satisfied
NoSuchElementError	raised when an AID is not available
IllegalStateError:	raised :when an object is used in the wrong context (e.g. being closed)
IllegalParameterError:	raised:when a method is given an incorrect parameter (e.g. bad format for an APDU or NULL when data is required)
IllegalReferenceError:	an error occurs if the reference cannot be found
OperationNotSupportedError:	an error occurs if the operation is not supported

The methods are described as followed:
<return value type> <method name> (<parameter1 type> <parameter1 name> …)

6. Transport API

The Transport API as part of the Open Mobile API provides a communication framework to Secure Elements available in the Open Mobile device.

6.1 Overview

The role of the Transport API is to provide a mean for applications to access the Secure Element(s) available on the device. The access provided is based on the concepts defined by ISO/IEC 7816-4:

- APDUs: the format of the messages which are exchanged with the Secure Element, basically a byte array is sent to the SE (or more precisely to an Applet in the SE) and the SE responds with another byte array. For the details of the exact formatting of such byte arrays, refer to ISO/IEC 7816-4.
- Basic and Logical Channels: the communication abstraction to the SE: Channels are the way to transmit APDUs, and can be opened simultaneously (although one can send an APDU at a time, by waiting for the response before sending the next APDU).

This API relies on a "connection" pattern: the client application (running on the device connected to the SE, e.g. the phone) opens a connection to the SE (a.k.a session in the rest of the document) and then opens a logical or basic channel to an Applet running in the SE.

On top of this pattern, there are a number of constraints that are enforced by the system:
An application cannot send "channel management" APDUs on its own, as this would break the isolation feature given by the logical channel. Once a channel is opened, it is allocated to communicate with one and only one Applet in the SE. In the same manner, the SELECT by DF name APDU cannot be sent by the terminal application.
The restrictions for the system should be implemented in the modules that are directly handling the communication with the Secure Element and not in the API itself to ensure that attackers cannot overcome the APDU filters. Thus if possible, the baseband should take care of the filtering or at least the RIL that communicates with the baseband.

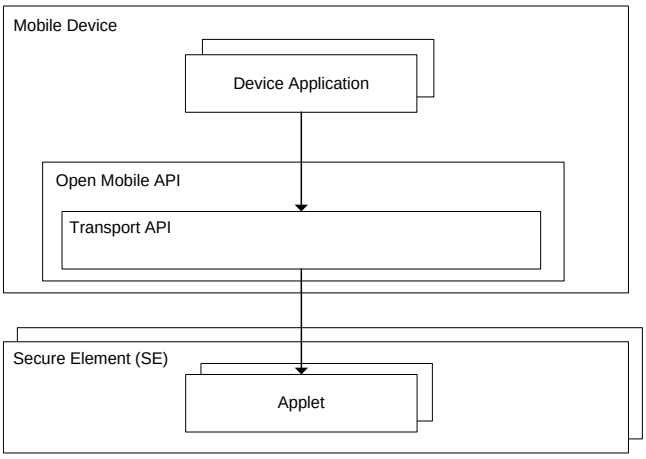

Figure 6-1: Transport API overview

6.2 Class diagram

This class diagram contains all classes of the Transport API. The SEService class realises a connector to the SE framework system and can be used to retrieve all SE readers available in the system. The Reader class can be used to access the SE connected with the selected reader. The Session class represents a session to an SE established by the reader and allows opening different communication channels represented by the Channel class.

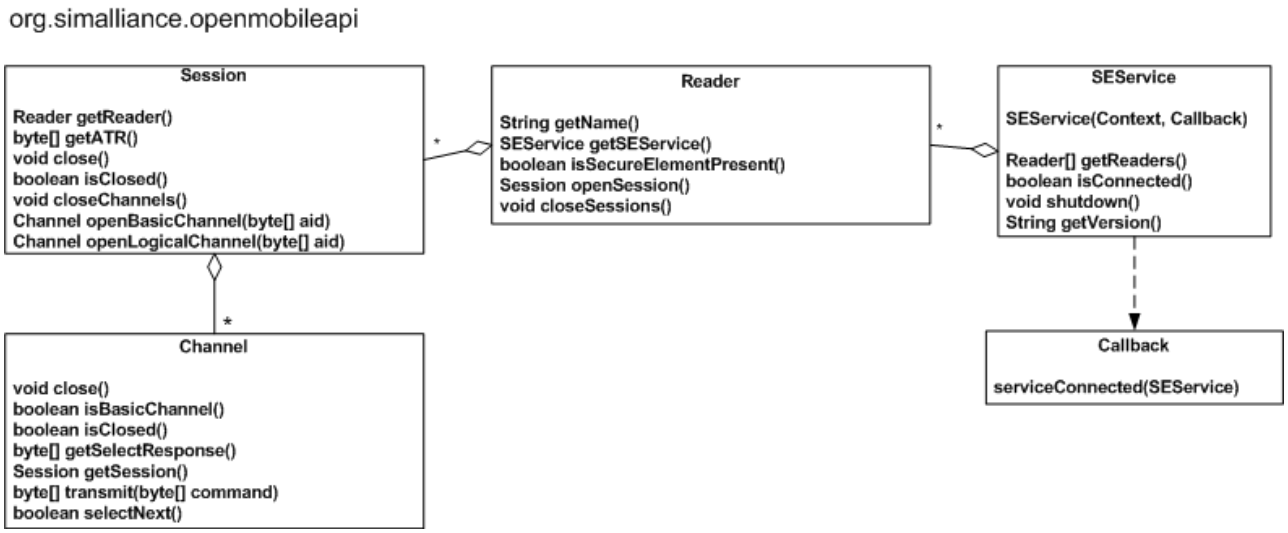

Figure 6-2: Transport API class diagram

6.3 Usage pattern

The usage pattern of the Transport API is as follows:

1. The application gets access to the Secure Element service(s):
   It creates an SEService class, passing an object implementing the SEService.Callback interface, whose serviceConnected method is called asynchronously when the connection is established.
   This doesn't represent a connection with the SE itself, but with the subsystem implementing the Secure Element access functionality.
2. The application enumerates the available readers.
   Readers are the slots where SEs are connected (in a removable or non removable manner). Once the user or an application-specific algorithm has selected a Reader, then the application opens a Session on this reader.
3. With this session, the application can retrieve the ATR of the SE, and if it matches with one of the known ATRs, it can start opening channels with Applets in the SE.
4. To open a channel, the application will use the AID of the Applet or use the default Applet on the newly opened channel.
5. Then the terminal application can start transmitting APDUs to the Applet.
6. Once done, the application can close any existing channels or even sessions, and its connection to the SEService.

6.4 Class: SEService

The SEService realizes the communication to available Secure Elements on the device.
This is the entry point of this API. It is used to connect to the infrastructure and get access to a list of Secure Element Readers.

6.4.1 Constructor: SEService(Context context, SEService.CallBack listener)

Establishes a new connection that can be used to connect to all the Secure Elements available in the system. The connection process can be quite long, so it happens in an asynchronous way. It is usable only if the specified listener is called or if isConnected() returns true.
The call-back object passed as a parameter will have its serviceConnected() method called when the connection actually happen.

Parameters:
context - the context of the calling application. Cannot be null.
listener - a SEService.CallBack object. Can be null.

6.4.2 Method: Reader[] getReaders()

Returns the list of available Secure Element readers. There must be no duplicated objects in the returned list. All available readers shall be listed even if no card is inserted.

Return value:
The readers list, as an array of Readers. If there are no readers the returned array is of length 0.

Errors:
IllegalStateError - if the SEService object is not connected

6.4.3 Method: boolean isConnected()

Tells whether or not the service is connected.

Return value:
True if the service is connected.

6.4.4 Method: void shutdown()

Releases all Secure Elements resources allocated by this SEService (including any binding to an underlying service). As a result isConnected() will return false after shutdown() was called. After this method call, the SEService object is not connected.
It is recommended to call this method in the termination method of the calling application (or part of this application) which is bound to this SEService.

6.4.5 Method: String getVersion()

Returns the version of the OpenMobile API specification this implementation is based on.

Return value:
String containing the OpenMobile API version (e.g. "2.05").

6.5 Class (or interface): SEService:CallBack

Interface to receive call-backs when the service is connected.
If the target language and environment allows it, then this shall be an inner interface of the SEService class.

6.5.1 Method: void serviceConnected(SEService service)

Called by the framework when the service is connected.

Parameters:

service - the connected service.

6.6 Class: Reader

Instances of this class represent Secure Element Readers supported by this device. These Readers can be physical devices or virtual devices. They can be removable or not. They can contain one Secure Element that can or cannot be removed.

6.6.1 Method: String getName()

Return the name of this reader.

- If this reader is a SIM reader, then its name must be "SIM[Slot]"
- If the reader is a SD or micro SD reader, then its name must be "SD[slot]"
- If the reader is a embedded SE reader, then its name must be "eSE[slot]"

Slot is a decimal number without leading zeros. The Numbering must start with 1 (e.g. SIM1, SIM2, … or SD1, SD2, … or eSE1, eSE2, …). The slot number "1" for a reader is optional (SIM and SIM1 are both valid for the first SIM-reader, but if there are two readers then the second reader must be named SIM2). This applies also for other SD or SE readers.

Return value:

The reader name, as a String.

6.6.2 Method: SEService getSEService()

Return the Secure Element service this reader is bound to.

Return value:

The SEService object.

6.6.3 Method: boolean isSecureElementPresent()

Check if a Secure Element is present in this reader.

Return value:

True if the SE is present, false otherwise.

6.6.4 Method: Session openSession()

Connects to a Secure Element in this reader.

This method prepares (initialises) the Secure Element for communication before the Session object is returned (i.e. powers the Secure Element by ICC ON if it is not already on).

There might be multiple sessions opened at the same time on the same reader. The system ensures the interleaving of APDUs between the respective sessions.

Return value:

A Session object to be used to create Channels.

Errors:

IOError - if something went wrong when communicating with the Secure Element or the reader.

6.6.5 Method: void closeSessions()

Close all the sessions opened on this reader. All the channels opened by all these sessions will be closed.

6.7 Class: Session

Instances of this class represent a connection session to one of the Secure Elements available on the device. These objects can be used to get a communication channel with an Applet in the Secure Element. This channel can be the basic channel or a logical channel.

6.7.1 Method: Reader getReader()

Get the reader that provides this session.

Return value:
The Reader object.

6.7.2 Method: byte[] getATR()

Get the Answer to Reset of this Secure Element.
The returned byte array can be null if the ATR for this Secure Element is not available.

Return value:
The ATR as a byte array or null.

6.7.3 Method: void close()

Close the connection with the Secure Element. This will close any channels opened by this application with this Secure Element.

6.7.4 Method: boolean isClosed()

Tells if this session is closed.

Return value:
True if the session is closed, false otherwise.

6.7.5 Method: void closeChannels()

Close any channel opened on this session.

6.7.6 Method: Channel openBasicChannel(byte[] aid)

Get an access to the basic channel, as defined in the ISO/IEC 7816-4 specification (the one that has number 0). The obtained object is an instance of the Channel class.
If the AID is null, it means no Applet is to be selected on this channel and the default Applet is used. If the AID is defined then the corresponding Applet is selected.
Once this channel has been opened by a device application, it is considered as "locked" by this device application, and other calls to this method will return null, until the channel is closed. Some Secure Elements (like the UICC) might always keep the basic channel locked (i.e. return null to applications), to prevent access to the basic channel, while some other might return a channel object implementing some kind of filtering on the commands, restricting the set of accepted command to a smaller set.
It is recommended for the UICC to reject the opening of the basic channel to a specific applet, by always answering null to such a request.
For other Secure Elements, the recommendation is to accept opening the basic channel on the default applet until another applet is selected on the basic channel. As there is no other way than a reset to select again the default applet, the implementation of the transport API should guarantee that the openBasicChannel(null) command will return null until a reset occurs.

The implementation of the underlying SELECT command within this method shall be based on ISO 7816-4 with following options:
CLA = '00'
INS = 'A4'
P1='04' (Select by DF name/application identifier)
P2='00' (First or only occurrence)

The select response data can be retrieved with byte[] getSelectResponse().

The API shall handle received status word as follow. If the status word indicates that the Secure Element was able to open a channel (e.g. status word '90 00' or status words referencing a warning in ISO-7816-4: '62 XX" or "63 XX') the API shall keep the channel opened and the next getSelectResponse() shall return the received status word.
Other received status codes indicating that the Secure Element was able not to open a channel shall be considered as an error and the corresponding channel shall not be opened.

Parameters:
aid - the AID of the Applet to be selected on this channel, as a byte array, or null if no Applet is to be selected.

Return value:
An instance of Channel if available or null.

Errors:
IOError - if there is a communication problem to the reader or the Secure Element
NoSuchElementError – If the AID on the Secure Element is not available or cannot be selected.
IllegalStateError - if the Secure Element session is used after being closed
IllegalParameterError - if the aid's length is not within 5 to 16 (inclusive).
SecurityError - if the calling application cannot be granted access to this AID or the default Applet on this session.

6.7.7 Method: Channel openLogicalChannel(byte[] aid)

Open a logical channel with the Secure Element, selecting the Applet represented by the given AID. If the AID is null, which means no Applet is to be selected on this channel, the default Applet is used. It's up to the Secure Element to choose which logical channel will be used.

The implementation of the underlying SELECT command within this method shall be based on ISO 7816-4 with following options:
CLA = '01' to '03', '40 to 4F'
INS = 'A4'
P1='04' (Select by DF name/application identifier)
P2='00' (First or only occurrence)

The select response data can be retrieved with byte[] getSelectResponse().

The API shall handle received status word as follow. If the status word indicates that the Secure Element was able to open a channel (e.g. status word '90 00' or status words referencing a warning in ISO-7816-4: '62 XX" or "63 XX') the API shall keep

the channel opened and the next getSelectResponse() shall return the received status word.
Other received status codes indicating that the Secure Element was able not to open a channel shall be considered as an error and the corresponding channel shall not be opened.

Parameters:
aid - the AID of the Applet to be selected on this channel, as a byte array.

Return value:
An instance of Channel. Null if the Secure Element is unable to provide a new logical channel.

Errors:
IOError - if there is a communication problem to the reader or the Secure Element.
NoSuchElementError – If the AID on the Secure Element is not available or cannot be selected or a logical channel is already open to a non-multiselectable Applet.
IllegalStateError - if the Secure Element is used after being closed.
IllegalParameterError - if the aid's length is not within 5 to 16 (inclusive).
SecurityError - if the calling application cannot be granted access to this AID or the default Applet on this session.

6.8 Class: Channel

Instances of this class represent an ISO/IEC 7816-4 channel opened to a Secure Element. It can be either a logical channel or the basic channel.
They can be used to send APDUs to the Secure Element. Channels are opened by calling the Session.openBasicChannel(byte[]) or Session.openLogicalChannel(byte[]) methods.

6.8.1 Method: void close()

Closes this channel to the Secure Element. If the method is called when the channel is already closed, this method will be ignored.
The close() method shall wait for completion of any pending transmit(byte[] command) before closing the channel.

6.8.2 Method: boolean isBasicChannel()

Returns a boolean telling if this channel is the basic channel.

Return value:
True if this channel is a basic channel.
False if this channel is a logical channel.

6.8.3 Method: boolean isClosed()

Tells if this channel is closed.

Return value:
True if the channel is closed, false otherwise.

6.8.4 Method: byte[] getSelectResponse()

Returns the data as received from the application select command inclusively the status word received at applet selection.
The returned byte array contains the data bytes in the following order:
[<first data byte>, …, <last data byte>, <sw1>, <sw2>]

Return value:
- The data as returned by the application select command inclusively the status word.
- Only the status word if the application select command has no returned data.
- Null if an application select command has not been performed or the selection response can not be retrieved by the reader implementation.

6.8.5 Method: Session getSession()

Get the session that has opened this channel.

Return value:

The session object this channel is bound to.

6.8.6 Method: byte[] transmit(byte[] command)

Transmit an APDU command (as per ISO/IEC 7816-4) to the Secure Element. The underlying layers generate as many TPDUs as necessary to transport this APDU. The transport part is invisible from the application. The generated response is the response of the APDU which means that all protocols related responses are handled inside the API or the underlying implementation.

For status word '61 XX' the API or underlying implementation shall issue a GET RESPONSE command as specified by ISO 7816-4 standard with LE=XX; for the status word '6C XX', the API or underlying implementation shall reissue the input command with LE=XX. For other status words, the API (or underlying implementation) shall return the complete response including data and status word to the device application. The API (or underlying implementation) shall not handle internally the received status words. The channel shall not be closed even if the Secure Element answered with an error code.
The system ensures the synchronization between all the concurrent calls to this method, and that only one APDU will be sent at a time, irrespective of the number of TPDUs that might be required to transport it to the SE. The entire APDU communication to this SE is locked to the APDU.
The channel information in the class byte in the APDU will be ignored. The system will add any required information to ensure the APDU is transported on this channel.
There are restrictions on the set of commands that can be sent:
- MANAGE_CHANNEL commands are not allowed.
- SELECT by DF Name (P1=04) are not allowed.
- CLA bytes with channel numbers are de-masked.

Parameters:

command - the APDU command to be transmitted, as a byte array.

Return value:

The response received, as a byte array. The returned byte array contains the data bytes in the following order: [<first data byte>, …, <last data byte>, <sw1>, <sw2>]

Errors:

IOError - if there is a communication problem to the reader or the Secure Element.
IllegalStateError - if the channel is used after being closed.
IllegalParameterError - if the command byte array is less than 4 bytes long.
SecurityError - if the command is filtered by the security policy.

6.8.7 Method: boolean selectNext()

Performs a selection of the next Applet on this channel that matches to the partial AID specified in the openBasicChannel(byte[] aid) or openLogicalChannel(byte[] aid) method. This mechanism can be used by a device application to iterate through all Applets matching to the same partial AID.
If selectNext() returns true a new Applet was successfully selected on this channel. If no further Applet exists with matches to the partial AID this method returns false and the already selected Applet stays selected.

Since the API cannot distinguish between a partial and full AID the API shall rely on the response of the Secure Element for the return value of this method.

The implementation of the underlying SELECT command within this method shall use the same values as the corresponding openBasicChannel(byte[] aid) or openLogicalChannel(byte[] aid) command with the option:
P2='02' (Next occurrence)

The select response stored in the Channel object shall be updated with the APDU response of the SELECT command.

Return value:
True if a new Applet was selected on this channel.
False if the already selected Applet stays selected on this channel.

Errors:
IOError - if there is a communication problem to the reader or the Secure Element.
OperationNotSupportedError - if this operation is not supported by the card.
IllegalStateError - if the Secure Element is used after being closed.

7. Service Layer APIs

The Service APIs as part of the Open Mobile API provides a framework to access Secure Elements available in the mobile device with high level interfaces.

7.1 Overview

The Open Mobile API contains a Service API on top of the Transport API for providing high level API methods for different purposes. The Service API relies on the Transport API for accomplishing the communication to the Applets within the Secure Element. The Service API consists of different APIs specialised for different purposes (e.g. File Management API for file operations). Normally each specialised API requires a counterpart on SE side (e.g. an SE Applet providing a defined set of APDUs).

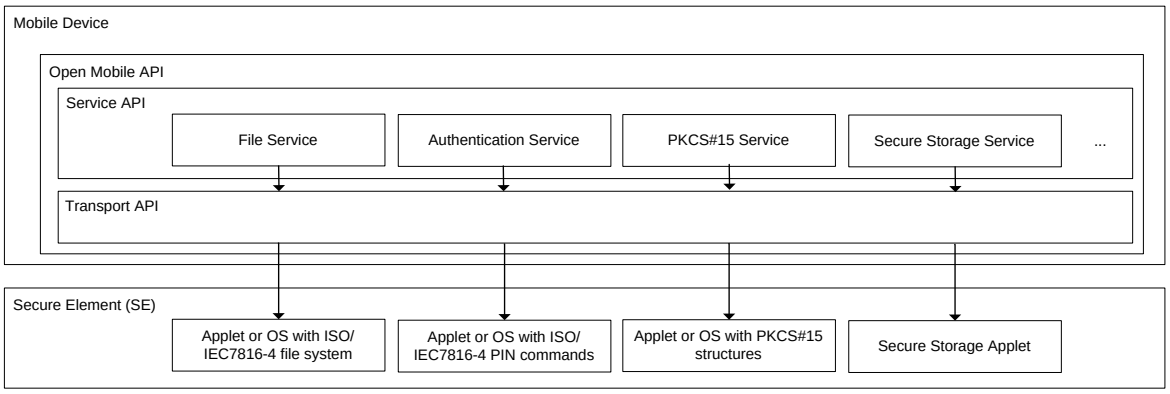

Figure 7-1: Service API overview

Note:
The use of the service layer API requires SE access rights (see chapter 10 for more information on SE access rights).

Since all operations within the Service API layer are based on the Transport API, all error conditions of the corresponding transport classes can be thrown in the service layer although not explicitly named. E.g. a call to AuthenticationProvider::verifyPin() can cause an IOError because the implementation uses the Channel::transmit() API call internally.

7.2 Class diagram

This class diagram contains the Service API besides the Transport API. The Service API consists of a set of classes derived from the base class Provider. No part of this diagram but also a component of the Service API is the Crypto API. The Crypto API relies on already existing APIs in the OS and realises the SE communication via the Transport API.

org.simalliance.openmobileapi

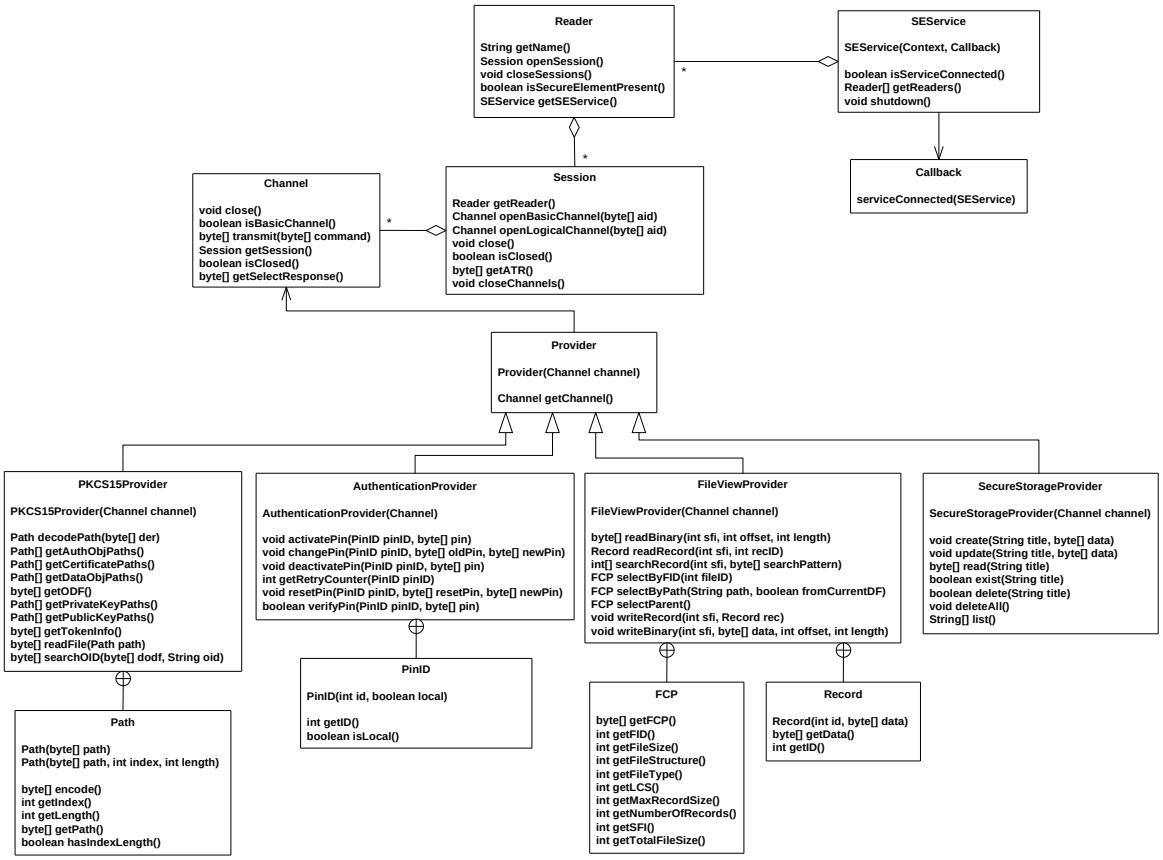

Figure 7-2: Service API class diagram with Provider classes

Besides these Provider classes the Service API contains an SEDiscovery class providing a discovery mechanism which can be used for an SE selection by defined criteria.

org.simalliance.openmobileapi

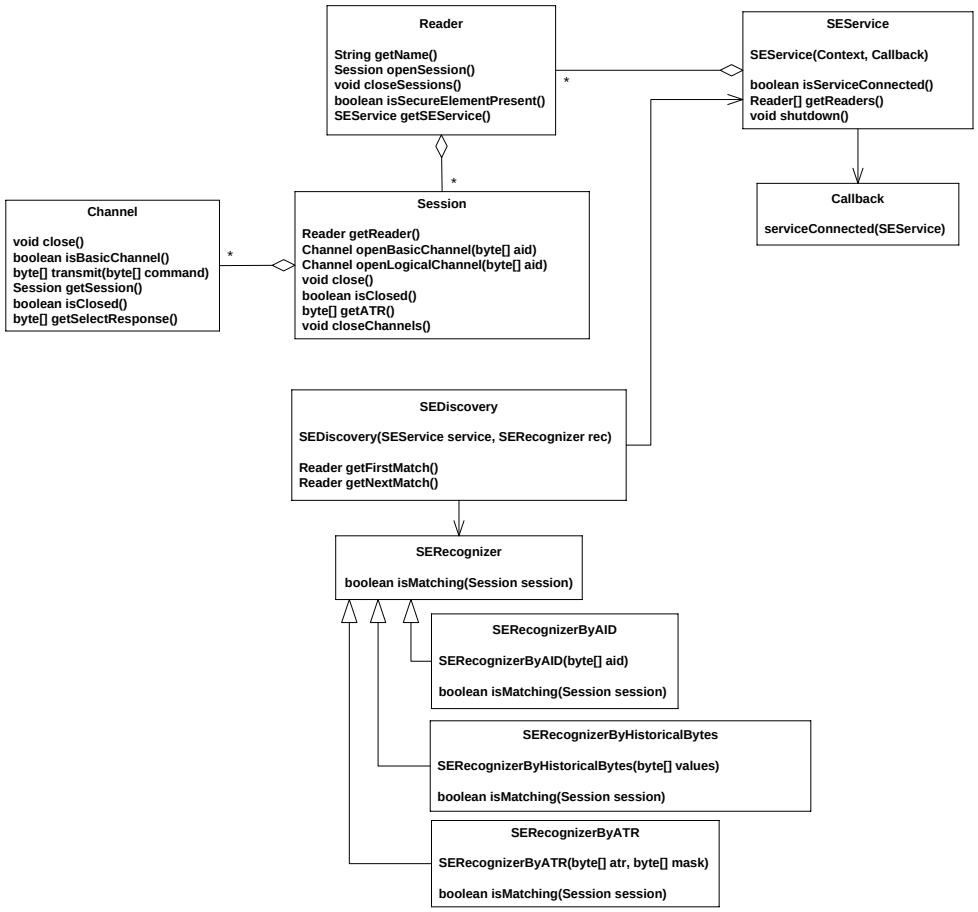

Figure 7-3: Service API class diagram with SEDiscovery classes

7.3 Usage pattern

The usage pattern of the service API is as follows:

1. The application gets access to the Secure Element service(s):
   It creates an SEService class, passing an object implementing the SEService.Callback interface, whose serviceConnected method is called asynchronously when the connection is established. This doesn't represent a connection with the SE itself, but with the subsystem implementing the Secure Element access functionality.

2. The application enumerates the available readers.
   Readers are the slots where SEs are connected (in a removable or non removable manner). Once the user or an application-specific algorithm has selected a Reader, then the application opens a Session on this reader. The right Reader can also be chosen by using the Discovery API. The Discovery API allows defining different criteria (ATR, AID, …) for finding an appropriate SE and Applet in an SE. Finally, the Discovery API allows iterating through the Readers containing an SE with the defined criteria.

3. With this session, the application can retrieve the ATR of the SE, and if it matches with one of the known ATRs, it can start opening channels with Applets in the SE.

4. To open a channel, the application will use the AID of the Applet or use the default Applet on the newly opened channel. The application is in charge of selecting an Applet which fits to the specific Provider class that will be chosen in the next step for SE operations.
5. The application creates an instance of a certain Provider class depending on the application's intention. If the intention is to read files from the SE's file system the FileViewProvider has to be chosen. The application has to consign the communication channel (established in the step before) to the Provider before the Provider instance can be used for SE operations.
6. Then the terminal application can start performing operations on the selected Applet in the SE with the help of the provider's methods. If a FileViewProvider instance was created for reading files from the SE's file system the application can use the FileViewProvider's methods readBinary() or readRecord() therefore.
7. The terminal application can also use several Provider instances alternately on the same channel or it can use the transmit method of the Transport Layer to send any APDUs directly to the SE on that channel. This option is especially useful if the application needs to perform different operations on the same channel. This could happen for example if the application needs to read a file from the SE's file system that requires a successful PIN verification on the same channel before. In this case the terminal application can instantiate an Authentication class which provides the verifyPin() method. After the successful PIN verification via the AuthenticationProvider the FileViewProvider can be used to read the file from the SE's file system. Since the SE usually manages the PIN verification individually for each logical channel it is important to apply the AuthenticationProvider and FileViewProvider on the same channel.
8. Once done, the application can close any existing channels or even sessions, and its connection to the SEService.

7.4 Service API Framework

The Open Mobile API provides a set of service layer classes with high level methods for SE operations.

7.4.1 Class: Provider

This Provider class (realised as interface or abstract class) is the base class for all service layer classes. Each service layer class provides a set of methods for a certain aspect (file management, PIN authentication, PKCS#15 structure handling, …) and acts as a provider for service routines. All Provider classes need an opened channel for the SE communication. Hence before a certain Provider class can be used for SE operations the channel has to be consigned. For performing different operations (PIN authentication, file operation, …) the Provider classes can be easily combined by using the same channel for different Provider classes and calling alternately methods of these different providers. It has to be considered that each provider class needs a counterpart on SE side (e.g. an applet with a standardised APDU interface as required by the Provider class). The application using a Provider class for SE interactions is in charge of assigning a channel to the Provider where the Provider's SE counterpart Applet is already preselected.

(a) Constructor: Provider(Channel channel)

Encapsulates the defined channel by a Provider object that can be used for performing a service operations on it. This constructor has to be called by derived Provider classes during the instantiation.

Parameters:

channel - the channel that shall be used by this Provider for service operations.

Errors:

IllegalStateError - if the defined channel is closed.

(b) Method: Channel getChannel()

Returns the channel that is used by this provider.
This returned channel can also be used by other providers.

Return value:

The channel instance that is used by this provider.

7.5 Crypto API

The crypto API that is provided natively by the mobile operating system shall be used.

For the Secure Element vendors, it guarantees that the crypto functionality offered by the Secure Elements will be seen at the same level as others.

For the application developers, they have to be careful to choose the crypto functionality provided by the Secure Element when enumerating the available crypto providers.

For example, in a Java environment, it is recommended to use the Java Crypto Extension as the Java binding of the crypto API. More information on the JCE can be found in [7].
[7] serves two purposes:

- It contains information about the API side of the Java Cryptography Architecture, that may be used by developers to learn how to use this API
- It contains information about the SPI side of the Java Cryptography Architecture, i.e. how to add new cryptography capabilities to a Java platform, by implementing a Provider, and how to declare and use the new providers.

In a native environment, where C/C++ is used as the programming language, it is recommended to use PKCS#11 as the native binding of the crypto API. More information on PKCS#11 can be found in [5].

In a mixed environment where Java and C/C++ are cohabiting, the PKCS#11 implementations should be made available to the Java applications. This is typically done by defining a JCE Provider for PKCS#11, that behaves as a Java binding for PKCS#11.

The following is an example of such a mixed architecture.

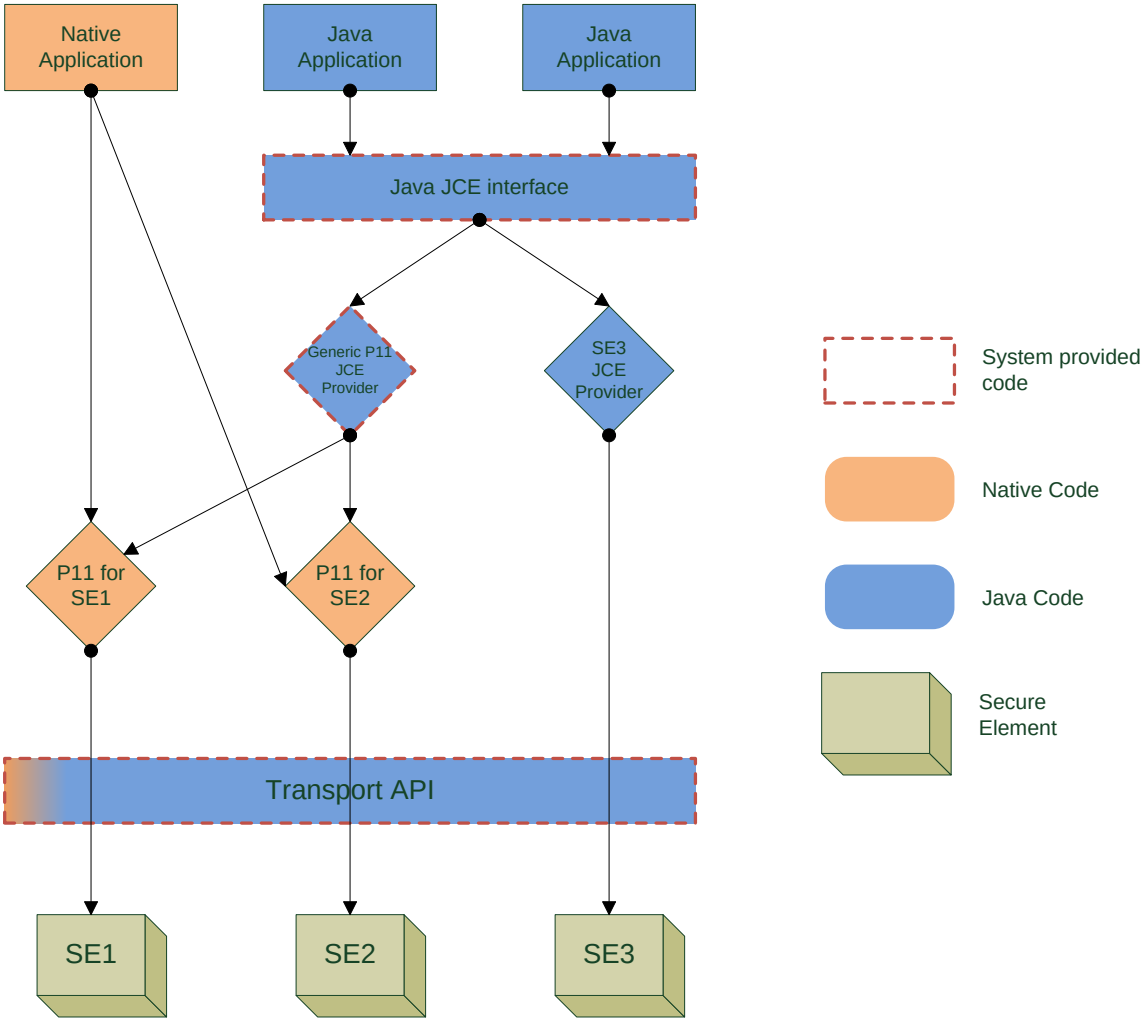

Figure 7-4 Crypto API architecture

7.5.1 Extensibility

It is required that this API provides the functionality to add system wide Crypto providers during runtime (without flashing the device) to support the different card implementations in the field.

For example, in a Java environment, the JCE provider architecture should be used to declare new providers (by Secure Element vendors) and to lookup for JCE providers (by application developers).

In a native environment, a mechanism to declare new PKCS#11 implementation (typically as shared libraries) must be available (to be used by PKCS#11 implementers), and reciprocally a mechanism to choose between the available PKCS#11 libraries must be available (to be used by application developers).

7.5.2 Extending by Shared Libraries

On systems that support the use of shared libraries, this mechanism can be used to provide new implementations of crypto providers. For example, PKCS#11 implementations can be provided as shared libraries. The mechanism to register and discover these shared libraries is yet to be defined (it is not part of PKCS#11).

7.5.3 Extending by Applicative plugins

On systems whose extensibility can be achieved only by installing new application and that provide a way to perform inter-application communication (e.g. IPC), a plugin mechanism can be used.

A generic provider based on the IPC capabilities can be offered by the system, to be used by the applications. The role of this generic provider is to lookup for plugin applications implementing a specific crypto service interface, list them to the crypto-aware applications, (enumerated as regular crypto providers) and when a crypto-aware application selects a particular provider, establish the connection to the plugin application actually implementing the crypto operations and forward all the requests thanks to the IPC mechanism.

Here's an illustration of such an implementation on a Java-based environment, using a JCE provider (the same could apply to a native-based environment, using a PKCS#11 library).

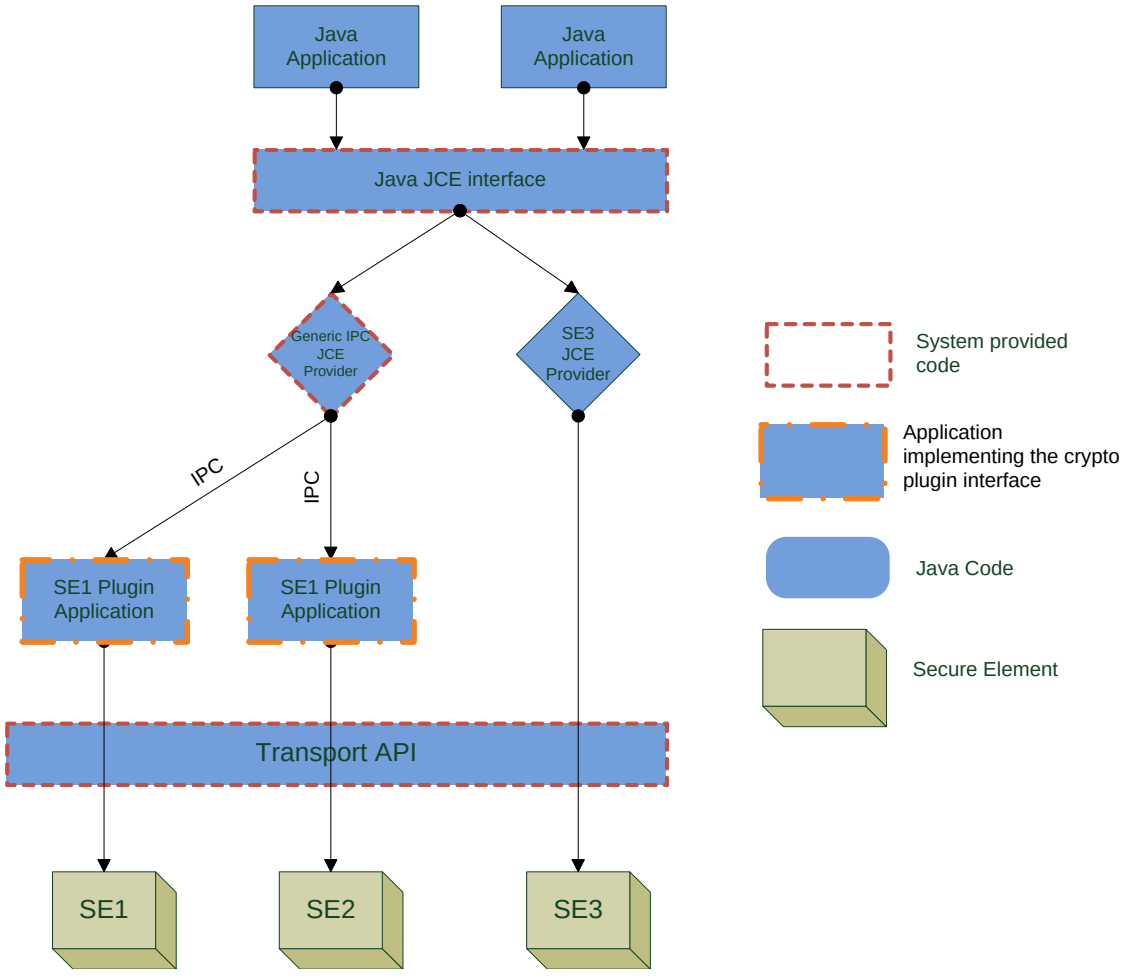

Figure 7-5: Crypto API architecture with plugin Applications

With this architecture, when support for a new type of Secure Element providing crypto services has to be added to the system, it's just a matter of installing a new plugin application implementing the protocol to interact properly with this new type of crypto provider.

7.5.4 Integration with the Transport API

The crypto API(s) should be implemented on top of the Transport API, and if it is not, the system should behave as if it was. For example, if the crypto API requires access to the basic channel of a Secure Element, then it is subject to the same rules as the other applications: access to the basic channel is provided only if the application calling the crypto API is authorised to access the basic channel, and if it is currently not locked. In the same way, if the basic channel is locked by the crypto API, it is not available to other applications.

If the PKCS#11 libraries are implemented on top of a PC/SC layer or an equivalent layer that gives a reader-access level, then it is recommended that the APDUs sent over this layer are filtered and process to be translated into commands for the transport API.

For example, if a native application sends a MANAGE_CHANNEL command and then a SELECT_BY_DF_NAME command, this should be translated into a call to openLogicalChannel.

7.6 Discovery API

This API provides means for the applications to lookup for a Secure Element, based on a search criterion. The rationale behind such an API is to factorise this lookup code in a system-provided API, reducing the development cost of this part of each application.

This API relies on an object-oriented approach: the lookup method is shared, but the criterion is implemented as a separate class, that can be derived. A set of basic criterion class is provided, they include:

- Search by ATR
- Search by Historical bytes
- Search by AID

If these basic research criteria are not sufficient to fulfil a specific application needs, then the application can provide its own criterion object. This object will have its isMatching() method called for each Secure Element present in the system, and will be able to use the Transport API (or any other service API) to implement a specific matching algorithm. Examples of such specific algorithm are:

- Analysis of EFdir file (if available)
- Analysis of Global Platform Status information (if available)

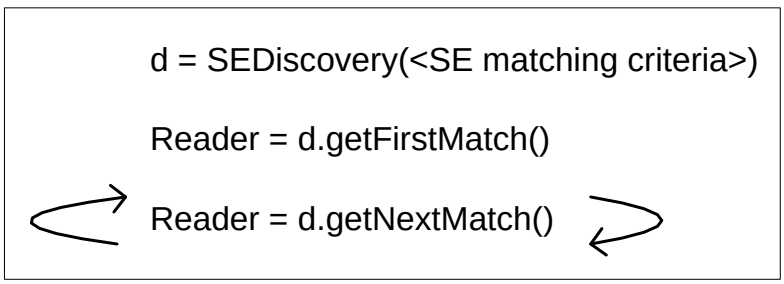

Figure 7-6: Discovery mechanism

7.6.1 Class: SEDiscovery

Instances of this class must be created by the applications to start a discovery process. When created, they are configured with an SEService and an object that will perform the recognition algorithm.

(a) Constructor: SEDiscovery (SEService service, SERecognizer recognizer)

Creates a discovery object that will perform a discovery algorithm specified by the recognizer object, and will be applied to the given SEService.

Parameters:

service - the SEService used to perform the discovery. Cannot be null.
recognizer - an SERecognizer instance, whose isMatching will be called. Cannot be null.

Errors:

IllegalParameterError – if one of the parameters is null.

(b) Method: Reader getFirstMatch()

Returns the first Secure Element reader containing a Secure Element that matches the search criterion.
Actually starts a full discovery process:

- Secure Element readers are enumerated
- For the first reader, if a Secure Element is present, open a session
- On this session, call the isMatching method of the SERecognizer object given at construction time.
- The session is closed.
- If the isMatching method returns false, the process is continued with the next reader.
- If the isMatching method returns true, the reader object is returned

The sessions used by the discovery process are closed to avoid the risk of leaks: if they were opened
and returned to the caller, there would be a risk for the caller to forget to close them.

Calling getFirstMatch twice simply restarts the discovery process (e.g. probably returns the same result, unless a Secure Element has been removed).

Return Value:

The first matching Secure Element reader, or null if there is none.

Errors:

NONE. All errors must be caught within the implementation.

(c) Method: Reader getNextMatch()

Returns the next Secure Element reader containing a Secure Element that matches the search criterion.
Actually continues the discovery process:

- For the next reader in the enumeration, if a Secure Element is present, open a session
- On this session, call the isMatching method of the SERecognizer object given at construction time.
- The session is closed.

- If the isMatching method returns false, the process is continued with the next reader.
- If the isMatching method returns true, the reader object is returned

Return Value:
The next matching Secure Element reader, or null if there is none.

Errors:
IllegalStateError - if the getNextMatch() method is called without calling getFirstMatch() before, since the creation of the SEDiscovery object, or since the last call to getFirstMatch or getNextMatch that returned null.

7.6.2 Class: SERecognizer

Base class for recognizer classes.
Extended by system-provided recognizers, or by custom recognizers.

(a) Method: boolean isMatching(Session session)

This is a call-back method that will be called during the discovery process, once per Secure Element inserted in a reader. Application developers can use the given session object to perform any discovery algorithm they think is appropriate. They can use the Transport API or any other API, conforming to access control rules & policy, like for regular application code (i.e. this is not privileged code).

Parameters:
session - a Session object that is used to perform the discovery. Never null.

Return value:
A boolean indicating whether the Secure Element to which the given session has been open is matching with the recognition criterion implemented by this method.

Errors:
NONE. All errors must be caught within the implementation of the method, and must be translated in a "false" result.

7.6.3 Class: SERecognizerByATR

Instances of this class can be used to find a Secure Element with a specific ATR (or ATR pattern).

(a) Constructor: SERecognizerByATR (byte[] atr, byte[] mask)

Parameters:
atr - a byte array containing the ATR bytes values that are searched for.
mask - a byte array containing an AND-mask to be applied to the Secure Element ATR values before to be compared with the searched value.

Errors:
IllegalParameterError – if ATR is invalid.

7.6.4 Class: SERecognizerByHistoricalBytes

Instances of this class can be used to find a Secure Element with a specific value in their historical bytes.

(a) Constructor: SERecognizerByHistoricalBytes (byte[] values)

Parameters:
values - byte array, to be checked for presence in the historical bytes.

Errors:
IllegalParameterError – if historical bytes are invalid.

7.6.5 Class: SERecognizerByAID

Instances of this class can be used to find a Secure Element implementing a specific Applet, identified by its AID. The presence of such an Applet is verified by trying to open a channel to this Applet. The opened channel, if any, is closed before the end of the isMatching method.

(a) Constructor: SERecognizerByAID (byte[] aid)

Parameters:
aid - byte array holding the AID to be checked for presence.

Errors:
IllegalParameterError – if AID is invalid.

7.7 File management

API for file management to read and write the content of files in an ISO/IEC 7816-4 compliant file system provided by the Secure Element's Operating System (OS) or Applet (installed in the Secure Element).

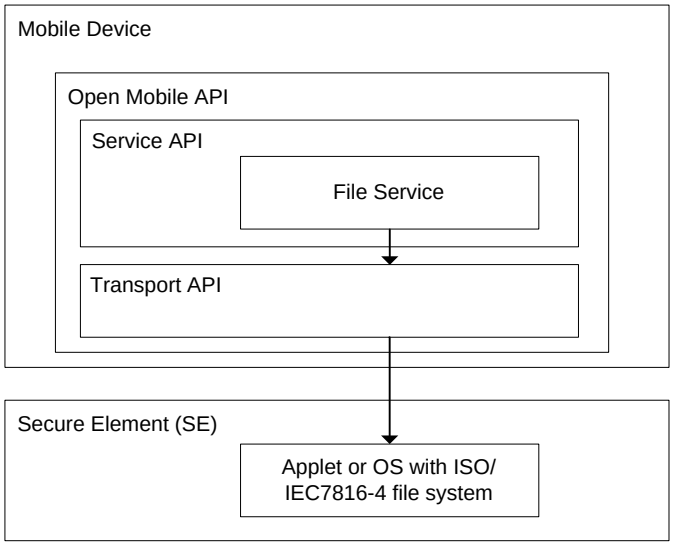

Figure 7-7: File management overview

7.7.1 Class: FileViewProvider

This Provider class simplifies file operations on Secure Elements with a file structure specified in ISO/IEC 7816-4.

Methods are provided that allows reading or writing file content. If the read or write operation is not allowed because security conditions are not satisfied a SecurityError will be returned. It must be considered that a file operation can only be applied onto a file which has a corresponding structure.

Prerequisites:

This Provider requires an ISO/IEC 7816-4 compliant file system on the SE. If this file system is implemented by an Applet within the SE then this Applet must be preselected before this Provider can be used, in case that the applet is not default selected (e.g. the GSM Applet as default selected Applet on UICC)

Notes:

- If used by multiple threads, synchronisation is up to the application.
- Each operation needs an access to the Secure Element. If the access can not be granted because of a closed channel or a missing security condition the called method will return an error.
- Using the basic channel for accessing the file system of the UICC (provided by the default selected GSM Applet) implies the risk of interferences from the baseband controller as the baseband controller works internally on the basic channel and can modify the current file selected state on the basic channel anytime. This means a file selection performed by this FileViewProvider does not guarantee a permanent file selection state on the UICC's basic channel and the application using the FileViewProvider has to take care of having the needed file selection state. The FileViewProvider itself cannot avoid interferences from the baseband controller on the basic channel but the risk could be minimised if the application using the FileViewProvider performs implicit selections for the file operation or performs the file selection immediately before the file operation.

(a) Constant: CURRENT_FILE

Indicates for file operation methods that the currently selected file shall be used for the file operation.

(b) Constant: INFO_NOT_AVAILABLE

Indicates that the demanded information is not available.

(c) Constructor: FileViewProvider(Channel channel)

Encapsulates the defined channel by a FileViewProvider object that can be used for performing file operations on it.

Parameters:

channel - the channel that shall be used by this Provider for file operations.

Errors:

IllegalStateError - if the defined channel is closed.

Note:

A file must be selected before a file operation can be performed. The file can be implicitly selected via a short file identifier (SFI) by the file operation method itself or explicitly by defining the file ID (FID) with selectByFID(int) or path with selectByPath(String, boolean).

(d) Method: FCP selectByPath(String path, boolean fromCurrentDF)

Selects the file specified by a path.

The path references the Secure Element file by a path (concatenation of file IDs and the order of the file IDs is always in the direction "parent to child") in following notation: "DF1:DF2:EF1". e.g. "0023:0034:0043". The defined path is applied to the Secure Element as specified in ISO/IEC 7816-4. Note: For performing read or write operations on a file the last knot in the path must reference an EF that can be read or written.

Parameters:
path - the path that references a file (DF or EF) on the Secure Element. This path shall not contain the current DF or MF at the beginning of the path.
fromCurrentDF - if true then the path is selected from the current DF,
if false then the path is selected from the MF.

Return value:
The FCP containing information to the selected file.

Errors:
IllegalStateError - if the defined channel is closed.
IllegalReferenceError - if the file couldn't be selected.
IllegalParameterError - if the defined path is invalid.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
OperationNotSupportedError - if this operation is not supported.

Notes:
- A file must be selected before a file operation can be performed.
- This method is based on the ISO/IEC 7816-4 command SELECT.

(e) Method: FCP selectByFID(int fileID)
Selects the file specified by the FID.

The file ID references the Secure Element file (DF or EF) by a FID. The FID consists of a two byte value as defined in ISO/IEC 7816-4.

Parameters:
fileID - the FID that references the file (DF or EF) on the Secure Element. The FID must be in the range of (0x0000-0xFFFF).

Return value:
The FCP containing information to the selected file.

Errors:
IllegalStateError - if the defined channel is closed.
IllegalReferenceError - if the File couldn't be selected.
IllegalParameterError - if the defined fileID is not valid.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
OperationNotSupportedError - if this operation is not supported.

Notes:
- A file must be selected before a file operation can be performed.
- This method is based on the ISO/IEC 7816-4 command SELECT.

(f) Method: FCP selectParent()
Selects the parent DF of the current DF.

The parent DF of the currently selected file is selected according to ISO/IEC 7816-4. If the currently selected file has no parent then nothing will be done.

Return value:
The FCP containing information to the selected file.

Errors:
IllegalStateError - if the defined channel is closed.
IllegalReferenceError - if the File couldn't be selected.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
OperationNotSupportedError - if this operation is not supported.

Notes:
- A file must be selected before a file operation can be performed.
- This method is based on the ISO/IEC 7816-4 command SELECT.

(g) Method: Record readRecord(int sfi, int recID)
Returns the record which corresponds to the specified record ID. If the record is not found then null will be returned.

Parameters:
sfi - the SFI of the file which shall be selected for this read operation. CURRENT_FILE can be applied if the file is already selected. The sfi must be in the range of (1-30).
recID - The record ID that references the record that should be read.

Return value:
The record which corresponds to the specified record ID.

Errors:
IllegalStateError - if the used channel is closed.
IllegalReferenceError - if the file couldn't be selected via SFI.
IllegalParameterError - if the defined sfi is not valid.
IllegalParameterError - if the defined record ID is invalid.
IllegalStateError - if no file is currently selected.
IllegalStateError - if the currently selected file is not a record based file.
IllegalStateError - if the record couldn't not be read.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command READ RECORD.

(h) Method: void writeRecord(int sfi, Record rec)
Writes a record into the specified file.

Parameters:
sfi - the SFI of the file which shall be selected for this write operation. CURRENT_FILE can be applied if the file is already selected. The sfi must be in the range of (1-30).
rec - The Record that shall be written.

Errors:
IllegalStateError - if the used channel is closed.
IllegalReferenceError - if the file couldn't be selected via SFI.

IllegalParameterError - if the defined Record is invalid.
IllegalParameterError - if the defined sfi is not valid.
IllegalStateError - if no file is currently selected.
IllegalStateError - if the currently selected file is not a record based file.
IllegalStateError - if the record couldn't be written.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command APPEND RECORD and UPDATE RECORD (which replaces existing bytes).

(i) ***Method: int[] searchRecord(int sfi, byte[] searchPattern)***
Returns the record numbers that contains the defined search pattern.

Parameters:
sfi - the SFI of the file which shall be selected for this search operation. CURRENT_FILE can be applied if the file is already selected. The sfi must be in the range of (1-30).
searchPattern - The pattern that shall match with Records.

Return value:
A list of record numbers (position 1..n of the record in the file) of the records which match to the search pattern. If no record matches then null will be returned.

Errors:
IllegalStateError - if the used channel is closed.
IllegalReferenceError - if the file couldn't be selected via SFI.
IllegalParameterError - if the defined sfi is not valid.
IllegalStateError - if no file is currently selected.
IllegalStateError - if the currently selected file is not a record based file.
IllegalStateError - if the search pattern is empty.
IllegalStateError - if the search pattern is too long.
IllegalStateError - if the data couldn't be searched.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
OperationNotSupportedError - if this operation is not supported.

Note
This method is based on the ISO/IEC 7816-4 command SEARCH RECORD with simple search.

(j) ***Method: byte[] readBinary(int sfi, int offset, int length)***
Reads content of the selected transparent file at the position specified by offset and length.

Parameters:
sfi - the SFI of the file which shall be selected for this read operation. CURRENT_FILE can be applied if the file is already selected. The sfi must be in the range of (1-30).
offset - Defines the start point of the file where the data should be read.
length - Defines the length of the data which should be read.

Return value:
The data read from the file or null if no content is available.

Errors:
IllegalStateError - if the used channel is closed.
IllegalReferenceError - if the file couldn't be selected via SFI.
IllegalParameterError - if the defined sfi is not valid.
IllegalParameterError - if the defined offset and length couldn't be applied.
IllegalStateError - if no file is currently selected.
IllegalStateError - if the currently selected file is not a transparent file.
IllegalStateError - if the data couldn't be read.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command READ BINARY.

(k) Method: void writeBinary(int sfi, byte[] data, int offset, int length)

Writes the defined data into the selected file at the position specified by offset and length.

Parameters:
sfi - the SFI of the file which shall be selected for this write operation. CURRENT_FILE can be applied if the file is already selected. The sfi must be in the range of (1-30).
data - The data which shall be written.
offset - Defines the position in the file where the data should be stored.
length - Defines the length of the data which shall be written.

Errors:
IllegalStateError - if the used channel is closed.
IllegalReferenceError - if the file couldn't be selected via SFI.
IllegalParameterError - if the defined sfi is not valid.
IllegalParameterError - if the defined data array is empty or too short.
IllegalParameterError - if the defined offset and length couldn't be applied.
IllegalStateError - if no file is currently selected.
IllegalStateError - if the currently selected file is not a transparent file.
IllegalStateError - if the data couldn't be written.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command UPDATE BINARY.

7.7.2 Class: FileViewProvider:FCP

File control parameter contain information of a selected file. FCPs are returned after a file select operation. This class is based on the ISO/IEC 7816-4 FCP returned by the SELECT command as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects).

(a) Method: byte[] getFCP()

Returns the complete FCP as byte array.

Return value:
The complete FCP as byte array.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects).

(b) Method: int getFileSize()

Returns the file size of the selected file (Number of data bytes in the file, excluding structural information).

Return value:
The file size depending on the file type:
- Transparent EF: The length of the body part of the EF.
- Linear fixed or cyclic EF: Record length multiplied by the number of records of the EF.

INFO_NOT_AVAILABLE if the information is not available.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects).

(c) Method: int getTotalFileSize()

Returns the total file size of the selected file (Number of data bytes in the file, including structural information if any).

Return value:
The total file size depending on the file type:
- DF/MF: the total file size represents the sum of the file sizes of all the EFs and DFs contained in this DF plus the amount of available memory in this DF. The size of the structural information of the selected DF itself is not included.
- EF: the total file size represents the allocated memory for the content and the structural information (if any) of this EF.

INFO_NOT_AVAILABLE if the information is not available.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects).

(d) Method: int getFID()

Returns the file identifier of the selected file.

Return value:
The file identifier of the selected file.
INFO_NOT_AVAILABLE if the FID of the selected file is not available.

Note:

This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects).

(e) Method: int getSFI()

Returns the short file identifier of the selected EF file.

Return value:

The short file identifier of the selected file.
INFO_NOT_AVAILABLE if selected file is not an EF or an SFI is not available.

Note:

This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects).

(f) Method: int getMaxRecordSize()

Returns the maximum record size in case of a record based EF.

Return value:

The maximum record size in case of a record based EF.
INFO_NOT_AVAILABLE if the currently selected file is not record based or the information can not be fetched.

Note:

This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects).

(g) Method: int getNumberOfRecords()

Returns the number of records stored in the EF in case of a record based EF.

Return value:

The number of records stored in the EF in case of a record based EF.
INFO_NOT_AVAILABLE if the currently selected file is not record based or the information can not be fetched.

Note:

This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects).

(h) Method: int getFileType()

Returns the file type of the currently selected file.

Return value:

The file type:
- (0) DF
- (1) EF

INFO_NOT_AVAILABLE if the information can not be fetched.

Note:

This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects). The file type is based on the definition in table 14 (File descriptor byte).

(i) Method: int getFileStructure()

Returns the structure type of the selected EF.

Return value:

The structure type of the selected file:

- (0) NO_EF
- (1) TRANSPARENT
- (2) LINEAR_FIXED
- (3) LINEAR_VARIABLE
- (4) CYCLIC

INFO_NOT_AVAILABLE if the information can not be fetched.

Note:

This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects). The file structure is based on the definition in table 14 (File descriptor byte).

(j) Method: int getLCS()

Returns the life cycle state of the currently selected file.

Return value:

The life cycle state:

- (0) NO_INFORMATION_GIVEN
- (1) CREATION_STATE
- (2) INITIALISATION_STATE
- (3) OPERATIONAL_STATE_ACTIVATED
- (4) OPERATIONAL_STATE_DEACTIVATED
- (5) TERMINATION_STATE

INFO_NOT_AVAILABLE if the information is not available.

Note:

This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects). The LCS is based on the definition in table 13 (Life cycle status byte).

7.7.3 Class: FileViewProvider:Record

Record class serves as container for record data. The created Record (as immutable object) can be used to read record data from a file or to write record data to a file.

(a) Contructor: Record(int id, byte[] data)

Creates a Record instance which can be used to store record data.

Parameter:

id - the record id that shall be stored.
data - the data that shall be stored.

(b) Method: int getID()

Returns the record ID of this record.

Return value:
The record ID of this record.

(c) ***Method: byte[] getData()***
Returns the data of this record.

Return value:
The data of this record.

7.8 Authentication service

Provide an API to perform a PIN authentication on the SE for enabling an authentication state.

Use cases:

- Performing an operation on the SE which requires a User authentication.
- e.g. for reading a file from the SE's file system which is PIN protected.
- e.g. for using a key from the SE which requires a PIN authentication.

Moreover the API allows the management of SE PINs with management commands like reset, change, activate and deactivate.

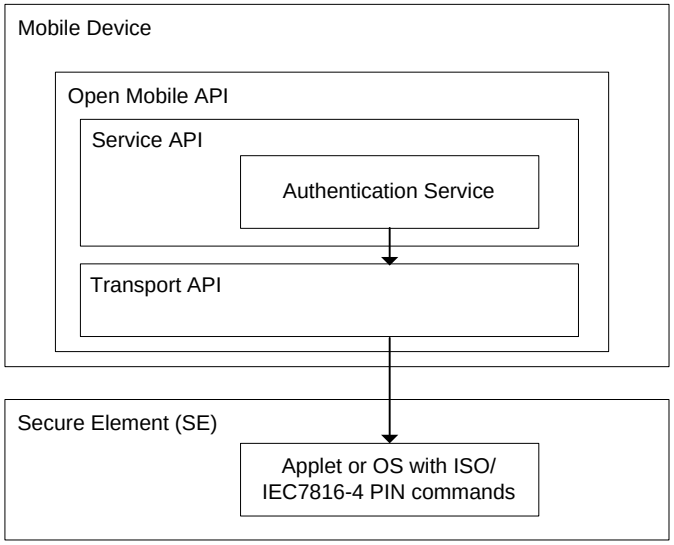

Figure 7-8: Authentication service overview

7.8.1 Class: AuthenticationProvider

This Authentication class can be used to privilege a certain communication channel to the Secure Element for operations that requires a PIN authentication. Besides the PIN verification for authentication this class provides also PIN management command for changing, deactivating or activating PINs.

Prerequisites:

The PIN operations performed by this AuthenticationProvider class are based on the ISO/IEC 7816-4 specification and require a preselected applet on the specified communication channel to the Secure Element that implements ISO/IEC 7816-4 compliant PIN commands.

Notes:

- If used by multiple threads, synchronisation is up to the application.
- Each operation needs an access to the Secure Element. If the access can not be granted because of a closed channel or a missing security condition the called method will return an error.

(a) Constructor: AuthenticationProvider(Channel channel)

Encapsulates the defined channel by an AuthenticationProvider object that can be used for applying PIN commands on it.

Parameters:
channel - The channel that should be privileged for operations that requires a PIN authentication.

Errors:
IllegalStateError - if the defined channel is closed.

(b) Method: boolean verifyPin(PinID pinID, byte[] pin)
Performs a PIN verification.

Parameters:
pinID - The PIN ID references the PIN in the Secure Element which shall be used for the verification.
pin - the PIN that shall be verified.

Return value:
True if the authentication was successful.
False if the authentication fails.

Errors:
IllegalReferenceError - if the PIN reference as defined couldn't be found in the Secure Element.
IllegalParameterError - if the PIN value has a bad coding or a wrong length (empty or too long).
IllegalStateError - if the used channel is closed.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command VERIFY.
When the PIN is blocked the method returns false. Clients have to use getRetryCounter to check for a blocked PIN.

(c) Method: void changePin(PinID pinID, byte[] oldPin, byte[] newPin)
Changes the PIN.

Parameters:
pinID - The PIN ID references the PIN in the Secure Element which shall be changed.
oldPin - the old PIN that shall be changed.
newPin - the PIN that shall be set as new PIN.

Errors:
SecurityError - if old PIN does not match with the PIN stored in the SE. The PIN is not changed.
IllegalReferenceError - if the PIN reference as defined couldn't be found in the Secure Element.
IllegalParameterError - if the value of oldPin or newPIN has a bad coding or a wrong length (empty or too long).
IllegalStateError - if the used channel is closed.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command CHANGE REFERENCE DATA.

(d) Method: void resetPin(PinID pinID, byte[] resetPin, byte[] newPin)

Resets the PIN with the reset PIN or just resets the retry counter.

Parameters:

pinID - The PIN ID references the PIN in the Secure Element which shall be reset.
resetPin - the reset PIN that shall be used for reset.
newPin - the PIN that shall be set as new PIN. Can be omitted with null if just the reset counter shall be reset.

Errors:

SecurityError - if resetPIN does not match with the "reset PIN" stored in the SE. The PIN or reset counter is not changed.
IllegalReferenceError - if the PIN ID reference as defined couldn't be found in the Secure Element.
IllegalParameterError - if the value of resetPin or newPIN has a bad coding or a wrong length (empty or too long).
IllegalStateError - if the used channel is closed.
OperationNotSupportedError - if this operation is not supported (e.g. PIN is not defined).

Note:

This method is based on the ISO/IEC 7816-4 command RESET RETRY COUNTER.

(e) Method: int getRetryCounter(PinID pinID)

Returns the retry counter of the referenced PIN.

Parameters:

pinID - The PIN ID references the PIN in the Secure Element and its retry counter.

Return value:

The retry counter of the referenced PIN.

Errors:

IllegalReferenceError - if the PIN reference as defined couldn't be found in the Secure Element.
OperationNotSupportedError - if this operation is not supported.
IllegalStateError - if the used channel is closed.

Note:

This method is based on the ISO/IEC 7816-4 command VERIFY.

(f) Method: void activatePin(PinID pinID, byte[] pin)

Activates the PIN. Thus a deactivated PIN can be used again.

Parameters:

pinID - The PIN ID references the PIN in the Secure Element which shall be activated.
pin - the verification PIN for activating the PIN if required. Can be omitted with null if not required.

Errors:
SecurityError - if the defined pin does not match with the PIN needed for the activation. The PIN state will not be changed.
IllegalStateError - if the used channel is closed.
IllegalReferenceError - if the PIN reference as defined couldn't be found in the Secure Element.
IllegalParameterError - if the PIN value has a bad coding or a wrong length (empty or too long).
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command ENABLE VERIFICATION REQUIREMENT.

(g) Method: void deactivatePin(PinID pinID, byte[] pin)

Deactivates the PIN. Thus the objects which are protected by the PIN can now be used without this restriction until activatePin() is called.

Parameters:
pinID - The PIN ID references the PIN in the Secure Element which shall be deactivated.
pin - the verification PIN for deactivating the pin if required. Can be omitted with null if not required.

Errors:
SecurityError - if the defined pin does not match with the PIN needed for the deactivation. The PIN state will not be changed.
IllegalStateError - if the used channel is closed.
IllegalReferenceError - if the PIN reference as defined couldn't be found in the Secure Element.
IllegalParameterError - if the PIN value has a bad coding or a wrong length (empty or too long).
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command DISABLE VERIFICATION REQUIREMENT.

7.8.2 Class: AuthenticationProvider:PinID

This PIN ID uniquely identifies a PIN in the Secure Element system. The PIN ID is defined as specified in ISO/IEC 7816-4 and can be used to reference a PIN in an ISO/IEC 7816-4 compliant system.

(a) Constructor: PinID(int id, boolean local)

Creates a PIN ID (reference) to identify a PIN within a Secure Element. The created PIN ID (as immutable object) can be specified on all PIN operation methods provided by the AuthenticationProvider class.

Parameters:
id - the ID of the PIN (value from 0x00 to 0x1F).
local - defines the scope (global or local). True if the PIN is local. Otherwise false.

Errors:
IllegalParameterError - if the defined ID is invalid.

Note:
This constructor is based on the P2 reference data for PIN related commands as specified in ISO/IEC 7816-4 in chapter 7.5 (Basic security handling). Local set to true indicates specific reference data and local set to false indicates global reference data according to ISO/IEC 7816-4. The ID indicates the number of the reference data (qualifier) according to ISO/IEC 7816-4.

(b) Method: int getID()
Returns the PIN ID.

Return value:
The PIN ID.

Note:
This method is based on the P2 reference data for PIN related commands as specified in ISO/IEC 7816-4 in chapter 7.5 (Basic security handling). The ID indicates the number of the reference data (qualifier) according to ISO/IEC 7816-4.

(c) Method: boolean isLocal()
Identifies if the PIN is local or global.

Return value:
True if the PIN is local. Otherwise false.

Note:
This method is based on the P2 reference data for PIN related commands as specified in ISO/IEC 7816-4 in chapter 7.5 (Basic security handling). Local set to true indicates specific reference data and local set to false indicates global reference data according to ISO/IEC 7816-4.

7.9 PKCS#15 API

The PKCS#15 standard is a structured way to store and organise data. The PKCS#15 service API simplifies the access to PKCS#15 file systems according to the version 1.1 of the PKCS#15 specification. Classes and methods are provided to retrieve the elementary PKCS#15 data structures, such as ODF (Object Directory File) and TokenInfo.

PKCS#15 file systems can be used to store cryptographic data, but also, any kind of data using application-specific files and OIDs. For example in the OMA-DM use case, the bootstrap data can be stored in the Secure Element's file system using a dedicated PKCS#15 file structure. This PKCS#15 API can be used by an OMA-DM client application to retrieve the bootstrap data from a Secure Element.

The preferred way to select a PKCS#15 file system is through the PKCS#15 AID (A0 00 00 00 63 50 4B 43 53 2D 31 35), however, a legacy file system can reference a PKCS#15 data structure though the EF(DIR).

Example of typical PKCS#15 file system:

```
ADF(PKCS#15)          : AID = A0 00 00 00 63 50 4B 43 53 2D 31 35
 |-EF(ODF)            : FID = 5031
 |-EF(TokenInfo)      : FID = 5032
 |-EF(PrKDF)          : optional, referenced by EF(ODF)
 |-EF(PuKDF)          : optional, referenced by EF(ODF)
 |-EF(CDF)            : optional, referenced by EF(ODF)
 |-EF(DODF)           : optional, referenced by EF(ODF)
 |-EF(AODF)           : optional, referenced by EF(ODF)
```

Example of legacy file system with a PKCS#15 structure:

```
MF                    : FID=3F00
 |-EF(DIR)            : FID=2F00
 |-DF(PKCS#15)        : referenced by EF(DIR)
    |-EF(ODF)         : FID = 5031
    |-EF(TokenInfo)   : FID = 5032
```

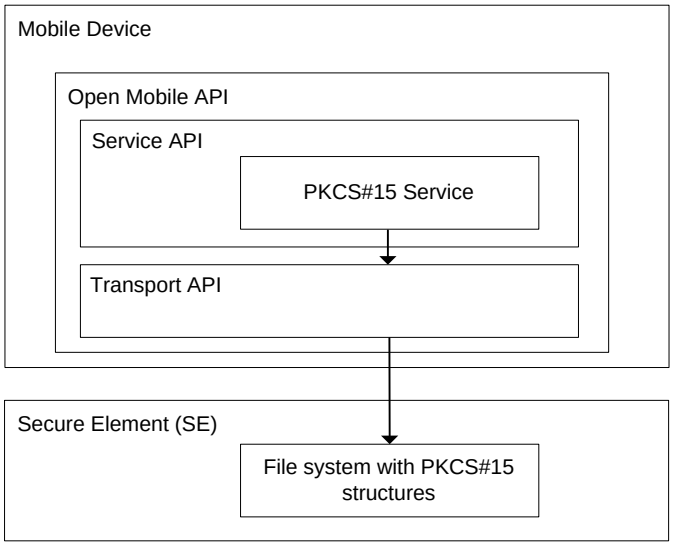

Figure 7-9: PKCS#15 service overview

7.9.1 Class: PKCS15Provider

This Provider class offers basic services to access a PKCS#15 file system. This Provider requires a PKCS#15 data structure on the Secure Element and a Channel instance allowing the access to this PKCS#15 data structure.

(a) Constant: byte[] AID_PKCS15

Default PKCS#15 AID (A0 00 00 00 63 50 4B 43 53 2D 31 35).

(b) Constructor: PKCS15Provider(Channel channel)

Encapsulates the defined channel by a PKCS#15 file system object. This method checks the presence of the EF(ODF) (Object Directory File) with file identifier 5031 and of the EF(TokenInfo) with file identifier 5032. Both files are mandatory and must be present in a valid PKCS#15 file system.

This method must first try to select EF(ODF) and EF(TokenInfo) on the provided channel. If the select fails, this method must try to locate a DF(PKCS#15) in the legacy file system using the EF(DIR) according to the data structure described in chapter 5.4 of the PKCS#15 specification (v1.1).

Parameters:

channel - the channel that shall be used by this Provider for file operations.

Errors:

IOError - if no PKCS#15 file system is detected on the provided channel.

(c) Method: byte[] getODF()

Returns the raw content of the EF(ODF) (Object Directory File).

Return value:

The EF(ODF) as a byte array. Must not be null.

(d) Method: byte[] getTokenInfo()

Returns the raw content of the EF(TokenInfo).

Return value:

The EF(TokenInfo) as a byte array. Must not be null.

(e) Method: Path[] getPrivateKeyPaths()

Returns an array of EF(PrKDF) paths (Private Key Directory Files). The PKCS#15 file system may contain zero, one or several EF(PrKDF).

Return value:

The array of EF(PrKDF) paths. May be null if empty.

(f) Method: Path[] getPublicKeyPaths()

Returns an array of EF(PuKDF) paths (Public Key Directory Files). The PKCS#15 file system may contain zero, one or several EF(PuKDF).

Return value:

The array of EF(PuKDF) paths. May be null if empty.

(g) Method: Path[] getCertificatePaths()

Returns an array of EF(CDF) paths (Certificate Directory Files). The PKCS#15 file system may contain zero, one or several EF(CDF).

Return value:

The array of EF(CDF) paths. May be null if empty.

(h) Method: Path[] getDataObjPaths()

Returns an array of EF(DODF) paths (Data Object Directory Files). The PKCS#15 file system may contain zero, one or several EF(DODF).

Return value:

The array of EF(DODF) paths. May be null if empty.

(i) Method: Path[] getAuthObjPaths()

Returns an array of EF(AODF) paths (Authentication Object Directory Files). The PKCS#15 file system may contain zero, one or several EF(AODF).

Return value:

The array of EF(AODF) paths. May be null if empty.

(j) Method: byte[] readFile(Path path)

Selects and reads a PKCS#15 file. The file may be a transparent or linear fixed EF. The 'index' and 'length' fields of the Path instance will be used according to chapter 6.1.5 of the PKCS#15 specification (v1.1). In case of transparent EF, 'index' is the start offset in the file and 'length' is the length to read. In case of linear fixed EF, 'index' is the record to read.

Parameters:

path - path of the file.

Return value:

The file content as a byte array. Or null if the referenced path does not exist.

Errors:

IOError - if the PKCS#15 file cannot be selected or read.
SecurityError - If the operation cannot be performed if a security condition is not satisfied.
OperationNotSupportedError - if this operation is not supported.

(k) Method: byte[] searchOID(byte[] dodf, String oid)

Parses the raw content of an EF(DODF) and searches for a specific OID Data Object. This method is a convenience method to simplify the access to OID Data Objects by applications, as described in chapter 6.7.4 of the PKCS#15 specification (v1.1). In many cases, the EF(DODF) contains a simple OID Data Object with a Path object, in order to reference an application-specific EF. For example, the OMA-DM specification requires a EF(DODF) containing the OID 2.23.43.7.1, followed by a Path object, referencing the EF(DM_Bootstrap).

Parameters:

dodf - the raw content of an EF(DODF) to parse.
oid - the searched OID value (e.g. OMA-DM bootstrap OID is 2.23.43.7.1).

Return value:
The raw object value if Oid has been found, null if not found.

Errors:
IllegalParameterError - If the OID is not correct.
OperationNotSupportedError - if this operation is not supported.

(l) Method: Path decodePath(byte[] der)
Builds a Path object using a DER-encoded (see ITU X.690 for DER-Coding) buffer.

Parameters:
der - the DER-encoded Path object as a byte array.

Return value:
The Path object.

Errors:
IllegalParameterError - If the defined path is not a correctly DER-encoded buffer.
OperationNotSupportedError - if this operation is not supported.

7.9.2 Class: PKCS15Provider:Path

This class represents a Path object as defined in chapter 6.1.5 of the PKCS#15 specification (v1.1).

(a) Constructor: Path(byte[] path)
Builds a Path object without index and length (the path can be absolute as well as relative).

Parameters:
path - the path as a byte array.

Errors:
IllegalParameterError - If the path is not correct.

(b) Constructor: Path(byte[] path, int index, int length)
Builds a Path object with index and length (the path can be absolute as well as relative).

Parameters:
path - the path as a byte array.
Index - the index value.
length - the length value.

Errors:
IllegalParameterError - If the path, index or length is not correct.

(c) Method: byte[] getPath()
Returns the path field of this Path object.

Return value:
The path field.

(d) Method: boolean hasIndexLength()
Checks whether this Path object has an index and length fields.

Return value:
True if the index and length field is present, false otherwise.

(e) Method: int getIndex()
Returns the index field of this Path object. The value of this field is undefined if the method hasIndexLength() returns false.

Return value:
The index field.

(f) Method: int getLength()
Returns the length field of this Path object. The value of this field is undefined if the method hasIndexLength() returns false.

Return value:
The length field.

(g) Method: byte[] encode()
Encodes this Path object according to DER (see ITU X.690 for DER-Coding).

Return value:
This Path object as a DER-encoded byte array.

7.10 Secure Storage

The Secure Storage Service can be used to store and retrieve sensitive data on the SE. This API requires a Secure Storage Applet on the SE with an APDU interface as defined below.
Data are stored in a dictionary format (String, value). It is a simpler way to store data in the Secure Storage than with a PKCS#15Provider or FileViewProvider, which can, in principle, also be used to store data securely but in a more elaborate way.

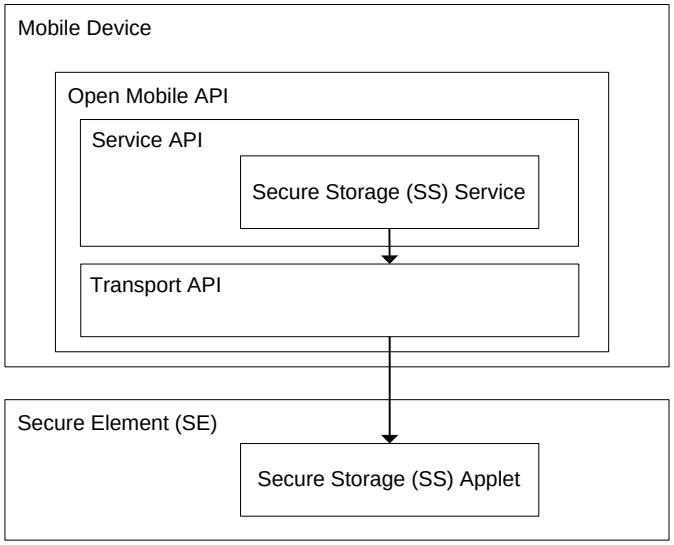

Figure 7-10: Secure Storage service overview

7.10.1 Class: SecureStorageProvider

This class provides an API to store and retrieve data on the SE which is protected in a secure environment. A default set of functionality that is always provided on every platform enables application developers to rely on this interface for secure data storage (e.g. credit card numbers, private phone numbers, passwords, ...). The interface should encapsulate any SE specifics as it is intended for device application developers who might not be familiar with SE or APDU internals.

Security Notes:

A PIN verification is required to grant access to the Secure Storage Applet where the Authentication Provider can be reused for the PIN operations. The Secure Storage Applet must separate the PIN verification on all logical channels to ensure that each device application needs to verify the PIN individually.

Prerequisites:

The Secure Storage operations performed by this Provider class are based on a Secure Storage located in the SE. The Secure Storage is usually realised by an Applet (providing the defined SS APDU interface) that must be preselected on the specified communication channel to the Secure Element before this Provider can be used.

Notes:

- If used by multiple threads, synchronisation is up to the application.
- Each operation needs an access to the Secure Element. If the access can not be granted because of a closed channel or a missing security condition the called method will return an error.

(a) Constructor: SecureStorageProvider(Channel channel)

Creates a SecureStorageProvider instance which will be connected to the preselected SE Secure Storage Applet on defined channel.

Parameters:

channel - the channel that shall be used by this Provider for operations on the Secure Storage.

Errors:

IllegalStateError - if the defined channel is closed.

(b) Method: void create(String title, byte[] data)

This command creates a Secure Storage entry with the defined title and data. The data can contain an uninterpreted byte stream of an undefined max length (e.g. names, numbers, image, media data, ...).

Parameters:

title - the title of the entry that shall be written. The max. title length is 60. All characters must be supported by UTF-8.
data - the data of the entry that shall be written. If data is empty or null then only the defined title will be assigned to the new entry.

Errors:

IllegalStateError - if the used channel is closed.
IllegalParameterError - if the title already exists. All entry titles must be unique within the Secure Storage.
IllegalParameterError - if the title is incorrect: bad encoding or wrong length (empty or too long).
IllegalParameterError - if the data chain is too long.
IOError – if the entry couldn't be created because of an incomplete write procedure.
SecurityError - if the PIN to access the Secure Storage Applet was not verified.

(c) Method: void update(String title, byte[] data)

This command updates the data of the Secure Storage entry referenced by the defined title. The data can contain an uninterpreted byte stream of an undefined max length (e.g. names, numbers, image, media data, ...) .

Parameters:

title - the title of the entry that must already exist. The max. title length is 60. All characters must be supported by UTF-8.
data - the data of the entry that shall be written. If data is empty or null then the data of the existing entry (referenced by the title) will be deleted.

Errors:

IllegalStateError - if the used channel is closed.
IllegalParameterError - if the title does not already exists.
IllegalParameterError - if the title is incorrect: bad encoding or wrong length (empty or too long).
IllegalParameterError - if the data chain is too long.
IOError – if the entry couldn't be updated because of an incomplete write procedure.
SecurityError - if the PIN to access the Secure Storage Applet was not verified.

(d) Method: byte[] read(String title)

This command reads and returns the byte stream of a data entry stored in the Secure Element referenced by the title.

Parameters:

title - the title of the entry that shall be read. The max. title length is 60. All characters must be supported by UTF-8.

Return value:

The data retrieved from the referenced entry. If the data do not exist in the Secure Storage entry referenced by the title then an empty byte array will be returned.

Errors:

IllegalStateError - if the used channel is closed.
IllegalParameterError - if the title is incorrect: bad encoding or wrong length (empty or too long).
IOError – if the entry couldn't be read because of an incomplete read procedure.
SecurityError - if the PIN to access the Secure Storage Applet was not verified.

(e) boolean exist(String title)

This command checks if the Secure Storage entry with the defined title exists.

Parameters:

title - the title of the entry that shall be checked. The max. title length is 60. All characters must be supported by UTF-8.

Errors:

IllegalStateError - if the used channel is closed.
IllegalParameterError - if the title is incorrect: bad encoding or wrong length (empty or too long).
SecurityError - if the PIN to access the Secure Storage Applet was not verified.

Return value:

True if the entry with the defined title exists. False if the entry does not exist.

(f) Method: boolean delete(String title)

This command deletes the Secure Storage entry referenced by the title. If the entry does not exist nothing will be done.

Parameters:

title - the title of the entry that shall be deleted. The max. title length is 60. All characters must be supported by UTF-8.

Errors:

IllegalStateError - if the used channel is closed.
IllegalParameterError - if the title is incorrect: bad encoding or wrong length (empty or too long).
SecurityError - if the PIN to access the Secure Storage Applet was not verified.

Return value:

True if the entry with the defined title is deleted. False if the entry does not exist.

(g) Method: void deleteAll()

This command deletes all Secure Storage entry referenced. If no entries exist nothing will be done.

Errors:
IllegalStateError - if the used channel is closed.
SecurityError - if the PIN to access the Secure Storage Applet was not verified.

(h) Method: String[] list()

This command returns an entry list with all title-identifiers. The title is intended for the users to identify and to reference the Secure Storage entries.

Return value:
A list of titles of all entries located in Secure Storage. An empty list will be returned if no entries exist in the Secure Storage.

Errors:
IllegalStateError - if the used channel is closed.
SecurityError - if the PIN to access the Secure Storage Applet was not verified.

7.10.2 Secure Storage APDU Interface

The Secure Storage (SS) Applet has to provide this APDU command interface for adding entries to the SS and deleting entries from the SS. Each SS entry is a container for a Secure Storage data record consisting of a title and data attribute. The attribute title identifies the SS entry with a user readable text and must be unique. The SS entry title shall be defined by the user before the SS entry is created. Thus the user can identify the created SS entry within the Secure Storage afterwards. The data attribute contains the sensitive data which has to be stored into the Secure Storage. Besides the title each SS entry can also be identified with a unique ID which is generated by the SS during the creation and has to be used to reference an SS entry within the Secure Storage. The title and ID must be unique within a Secure Storage as each entry can be referenced by either the title or the ID.

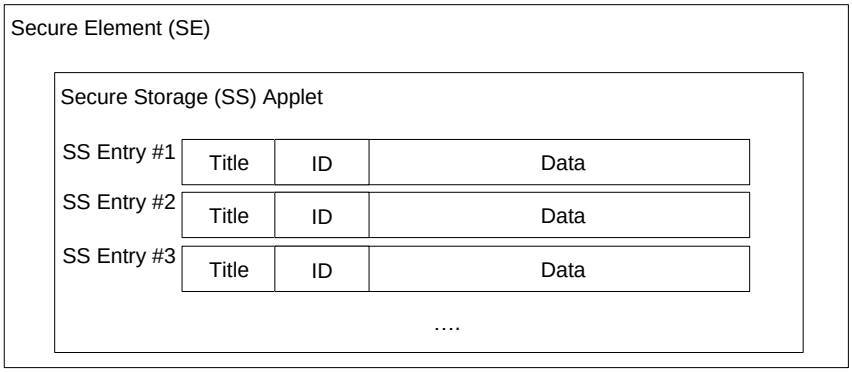

Figure 7-11: Secure Storage Applet overview

(a) CREATE SS ENTRY Command Message

The CREATE SS ENTRY command can be used to create an entry in the Secure Storage. Each entry requires a unique title which must be specified in the command.

The CREATE SS ENTRY command message shall be coded according to the following table:

Table 7-1: CREATE SS ENTRY Command Message

Code	Value	Meaning
CLA	'80'	
INS	'E0'	CREATE SS ENTRY
P1 P2	'00 00'	
LC	Length of title	
DATA	Title	Title of the new entry in UTF-8. This title must be unique within the SS. If the defined title does already exist in the SS then the error code '6A 80' will be returned.
LE	2	

(b) CREATE SS ENTRY Response Message

The CREATE SS ENTRY response shall contain a data field with response code '90 00' (successful operation) or an error response code.

Table 7-2: CREATE SS ENTRY Response Data

Value	Meaning	Presence
ID	The ID of the new entry created in the Secure Storage. This ID can be used to reference an SS entry. The length of this ID is always 2 bytes.	Mandatory

Table 7-3: CREATE SS ENTRY Response Code

SW1	SW2	Meaning
'6A'	'80'	Incorrect values in the command data (if the defined title does already exist)
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'67'	'00'	Wrong length in LC
'6A'	'86'	Incorrect P1 P2
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class
'65'	'81'	Memory failure (if the creation of the entry fails due to memory issues)
'6A'	'84'	Not enough memory space (if not enough memory resources are available)

(c) DELETE SS ENTRY Command Message

The DELETE SS ENTRY command can be used to delete an entry from the Secure Storage. The entry referenced in this command by an ID must exist in the Secure Storage otherwise an error code will be returned.

The DELETE SS ENTRY command message shall be coded according to the following table:

Table 7-4: DELETE SS ENTRY Command Message

Code	Value	Meaning
CLA	'80'	
INS	'E4'	DELETE SS ENTRY
P1 P2	P1: ID high byte P2: ID low byte	The ID of the SS entry which has to be deleted. If the SS entry couldn't be found '6A 88' is returned.
LC	-	
DATA	-	
LE	-	

(d) DELETE SS ENTRY Response Message

The DELETE SS ENTRY response shall contain a response code '90 00' (successful operation) or an error response code.

Table 7-5: DELETE SS ENTRY Response Code

SW1	SW2	Meaning
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6A'	'88'	Referenced data not found (if the referenced SS entry does not exist)
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class
'65'	'81'	Memory failure (if the operation fails due to memory issues)

(e) SELECT SS ENTRY Command Message

The SELECT SS ENTRY has to be used to select an SS ENTRY in the Secure Storage for a write or read operation.

The SELECT SS ENTRY command message shall be coded according to the following table:

Table 7-6: SELECT SS ENTRY Command Message

Code	Value	Meaning
CLA	'80'	
INS	'A5'	SELECT SS ENTRY
P1 P2	P1: Reference parameter P2: '00'	Reference parameter: Select ID('00'): Select the entry referenced by the ID Select First('01'): Select the first SS entry Select Next('02'): Select the next available SS entry
LC	2 or -	The length of the ID of the SS entry (only needed with P1= "Select ID")
DATA	ID or -	The ID of the SS entry which shall be selected (only needed with P1= "Select ID")
LE	'00'	

(f) SELECT SS ENTRY Response Message

The SELECT SS ENTRY response shall contain a data field with response code '90 00' (successful operation) or an error response code.

Table 7-7: SELECT SS ENTRY Response Data

Value	Meaning	Presence
Title	The title of the referenced SS entry in UTF-8.	Mandatory

Table 7-8: SELECT SS ENTRY Response Code

SW1	SW2	Meaning
'6A'	'80'	Incorrect values in the command data (if the data field has not a length of 2 bytes)
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6A'	'88'	Referenced data not found (if the referenced SS entry does not exist)
'67'	'00'	Wrong length in LC
'6A'	'86'	Incorrect P1 P2
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class

(g) PUT SS ENTRY DATA Command Message

The PUT SS ENTRY DATA command message can be used to store sensitive data into the currently selected SS entry. An SS entry can be selected with the command SELECT SS ENTRY DATA. For very long data the command PUT SS ENTRY DATA can be used iteratively by applying the command several times with an appropriate P1 parameter (first) and (next).

Note:
Before data can be stored into the SS entry with PUT SS ENTRY DATA the data size has to be specified. Otherwise an error code will be returned.

The transmitted data will only be stored into the SS entry if all data parts are transferred (this means the sum of all transferred parts fits exactly to the defined data length). As long as the transferred data are not complete the data has to be temporarily buffered within the Secure Storage application. If the succeeding APDU is not a PUT SS ENTRY DATA command or the succeeding PUT SS ENTRY DATA command does not contain the following data as expected then the buffered data has to be discarded.

The PUT SS ENTRY DATA command message shall be coded according to the following table:

Table 7-9: PUT SS ENTRY DATA Command Message

Code	Value	Meaning
CLA	'80'	
INS	'DA'	PUT SS ENTRY DATA
P1 P2	P1: size (0), first (1), next(2) P2: '00'	size(0): The whole size of the data that shall be stored. first(1): DATA contains the first data part. next(2): DATA contains the next data part (append mode).
LC	Data length	
DATA	Data	P1=size(0): Defines the data size. P1=first(1) or next(2): Sensitive data (or a part of the data) which has to be stored to the currently selected SS entry.
LE	-	

(h) PUT SS ENTRY DATA Response Message

The PUT SS ENTRY ID response shall contain a response code '90 00' (successful operation) or an error response code.

Table 7-10: PUT SS ENTRY DATA Response Code

SW1	SW2	Meaning
'6A'	'80'	Incorrect values in the command data
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6A'	'88'	Referenced data not found (if no SS entry is currently selected)
'67'	'00'	Wrong length in LC
'6A'	'86'	Incorrect P1 P2 (if the defined P1/P2 are invalid or cannot be applied)
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class
'65'	'81'	Memory failure (if the defined data exceed the defined size or a size wasn't defined)
'6A'	'84'	Not enough memory space (if not enough memory resources are available)

(i) GET SS ENTRY DATA Command Message

The GET SS ENTRY DATA command message can be used to retrieve data from the currently selected SS entry. An SS entry can be selected with the command SELECT SS ENTRY DATA. For very long data the command GET SS ENTRY DATA can be used iteratively by applying the command several times with an appropriate P1 parameter (first) and (next). If the succeeding APDU is not a GET SS ENRTY DATA command then an outstanding retrieve procedure must be reset by the Secure Storage application.

The GET SS ENTRY DATA command message shall be coded according to the following table:

Table 7-11: GET SS ENTRY DATA Command Message

Code	Value	Meaning
CLA	'80'	
INS	'CA'	GET SS ENTRY DATA
P1 P2	P1: size (0), first (1), next(2) P2: '00'	size(0): Response contains the whole size of the data that shall be read. first(1): Response contains the first data part. next(2): Response contains the next data part.
LC	-	
DATA	-	
LE	'00'	

(j) GET SS ENTRY DATA Response Message

The GET SS ENTRY DATA response shall contain a data field with response code '90 00' (successful operation) or an error response code.

Table 7-12: GET SS ENTRY DATA Response Data

Value	Meaning	Presence
Data	The data (or a part) of the currently selected SS entry. or Whole size of the data stored in the currently selected SS entry.	Mandatory

Table 7-13: GET SS ENTRY DATA Response Code

SW1	SW2	Meaning
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6A'	'86'	Incorrect P1 P2 (if the defined P1/P2 are invalid or cannot be applied)
'6A'	'88'	Referenced data not found (if no SS entry is currently selected)
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class
'65'	'81'	Memory failure (if further data are demanded but no further data exist).

(k) GET SS ENTRY ID Command Message

The GET SS ENTRY ID command message can be used to retrieve the ID of an SS entry referenced by the title.

The GET SS ENTRY ID command message shall be coded according to the following table:

Table 7-14: GET SS ENTRY ID Command Message

Code	Value	Meaning
CLA	'80'	
INS	'B2'	GET ENTRY ID
P1 P2	'00 00'	
LC	Length of title	
DATA	Title	Title of the entry
LE	'02'	

(l) GET SS ENTRY ID Response Message

The READ SS ENTRY ID response shall contain a data field with response code '90 00' (successful operation) or an error response code.

Table 7-15: GET SS ENTRY ID Response Data

Value	Meaning	Presence
ID	The ID of the entry in the Secure Storage referenced by the defined title. The length of this ID is always 2 bytes.	Mandatory

Table 7-16: GET SS ENTRY ID Response Code

SW1	SW2	Meaning
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6A'	'88'	Referenced data not found (if the referenced SS entry does not exist)
'6A'	'86'	Incorrect P1 P2
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class

(m) DELETE ALL SS ENTRIES Command Message

The DELETE ALL SS ENTRIES command can be used to delete all entries from the Secure Storage.

The DELETE ALL SS ENTRIES command message shall be coded according to the following table:

Table 7-17: DELETE ALL SS ENTRIES Command Message

Code	Value	Meaning
CLA	'80'	
INS	'E5'	DELETE ALL SS ENTRIES
P1 P2	'00 00'	
LC	-	
DATA	-	
LE	-	

(n) DELETE ALL SS ENTRIES Response Message

The DELETE SS ENTRIES response shall contain a response code '90 00' (successful operation) or an error response code.

Table 7-18: DELETE ALL SS ENTRIES Response Code

SW1	SW2	Meaning
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class
'65'	'81'	Memory failure (if the operation fails due to memory issues)

7.10.3 Secure Storage APDU transfer

This chapter describes how the Secure Storage APDU interface has to be applied for performing the Secure Storage service operations provided by the SecureStorageProvider.

Note:

All Secure Storage operations have to be realised in an atomic way. This means if an error occurs during a Secure Storage operation (e.g. if an error occurs on a certain

APDU) all modifications made on the Secure Storage (in the previous steps within a Secure Storage operation) have to be reversed.

(a) Create operation

The create method includes the creation and selection of an SS entry with a succeeding SS entry data update. Following steps are needed:

- CREATE SS ENTRY (tile) creates the SS entry in the Secure Storage with the defined title.
- SELECT SS ENTRY (ID) selects the SS entry in the Secure Storage for the data update.
- PUT SS ENTRY DATA (size) to define the data size.
- PUT SS ENTRY DATA (data) writes the data to the selected SS entry. A long data chain (which cannot be transferred via one APDU command) can be written by applying the command iteratively by using PUT SS ENTRY DATA(P1P2=NEXT) several times after performing PUT SS ENTRY DATA(P1P2=FIRST).

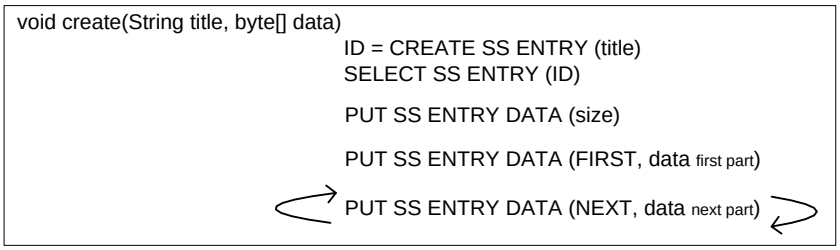

Figure 7-12: Create SS entry operation

Note:

If an error occurs during the create operation all previous steps must be reversed to obtain the same Secure Storage state as before. For example, if the creation of the SS entry was successful but the storage of the SS entry data fails then this newly created SS entry has to be deleted again.

(b) Update operation

The update method includes the selection of an SS entry with a succeeding SS entry data update. Following steps are needed:

- GET ENTRY ID (title) returns the internal SS entry ID to the defined title.
- SELECT SS ENTRY (ID) selects the SS entry in the Secure Storage for the data update.
- PUT SS ENTRY DATA (size) to define the data size.
- PUT SS ENTRY DATA (data) writes the data to the selected SS entry. A long data chain (which cannot be transferred via one APDU command) can be written by applying the command iteratively by using PUT SS ENTRY DATA (P1P2=NEXT) several times after performing PUT SS ENTRY DATA (P1P2=FIRST).

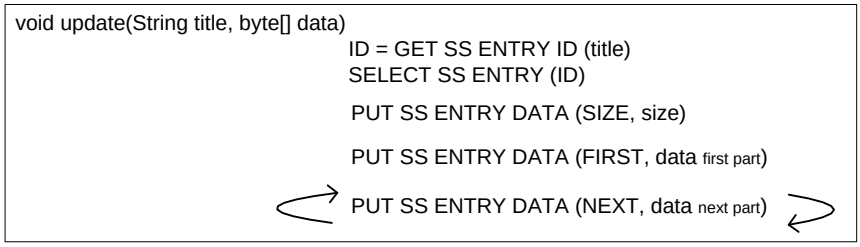

Figure 7-13: Update SS entry operation

Note:
If an error occurs during the update operation all previous steps must be reversed to obtain the same Secure Storage state as before. For example, if the update of some data parts was successful but the update of a following data part fails then the SS entry has to be set to the previous state (e.g. by reassigning the previously stored data to the SS entry).

(c) Read operation

The read method includes the selection of an SS entry with a succeeding read SS entry data operation. Following steps are needed:

- GET ENTRY ID (title) returns the internal SS entry ID to the defined title.
- SELECT SS ENTRY (ID) selects the SS entry in the Secure Storage for the read operation.
- GET SS ENTRY DATA (size) to determine the whole size of the data that shall be read.
- GET SS ENTRY DATA (data) reads the data to the selected SS entry. A long data chain (which cannot be transferred via one APDU command) can be read by applying the command iteratively by using GET SS ENTRY DATA (P1P2=NEXT) several times after performing GET SS ENTRY DATA (P1P2=FIRST).

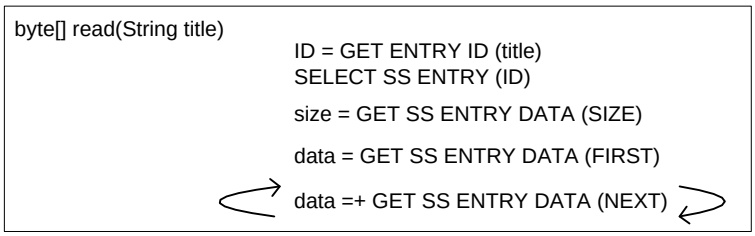

Figure 7-14: Read SS entry operation

(d) List operation

The list method includes an iterative selection of all SS entries. Following steps are needed:

- SELECT SS ENTRY (FIRST) selects the first SS entry and returns its title.
- SELECT SS ENTRY (NEXT) selects the next SS entry and returns its title. This command has to be applied iteratively until all SS entry titles are retrieved.

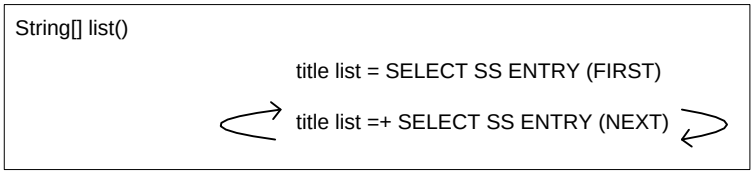

Figure 7-15: List SS entries operation

(e) Delete operation

The delete method includes a delete operation. Following steps are needed:

- GET ENTRY ID (title) returns the internal SS entry ID to the defined title.
- DELETE SS ENTRY (ID) deletes the SS entry referenced by the defined ID.

Figure 7-16: Delete SS entry operation

(f) Delete all operation

The delete all method includes a delete operation which deletes all entries from the Secure Storage. Following steps are needed:

- DELETE ALL SS ENTRIES

Figure 7-17: Delete all SS entries operation

(g) Exist operation

The exist method checks if a certain SS entry exists. Following steps are needed:

- GET ENTRY ID (title) returns the internal SS entry ID to the defined title.
- SELECT SS ENTRY (ID) indicates if the SS entry to the defined ID exists or not.

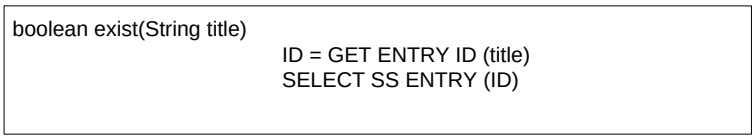

Figure 7-18: Exist SS entry operation

7.10.4 Secure Storage PIN protection

Before a Secure Storage command can be performed a PIN verification has to be performed towards the Secure Storage Applet. The Secure Storage Applet allows the execution of an APDU command only after a successful PIN verification. Therefore the Secure Storage Applet stores an internal PIN verified state for each logical channel. Thus an established logical channel has to be authorised with a PIN verification before

this channel can be used for the SecureStorageService. The internal PIN verified state for a channel is kept up until this channel is closed. If a Secure Storage APDU command is used without being authorised with a PIN the SecureStorage Applet has to return an error code indicating the missing privilege.

For providing the Secure Storage access restriction based on PIN authentication the Secure Storage Applet must provide the ISO/IEC 7816-4 commands VERIFY, CHANGE REFERENCE DATA and RESET RETRY COUNTER for supporting the Authentication Service methods verifyPin(byte[] pin), changePin (byte[] oldPin, byte[] new Pin) and resetPin(byte[] resetPin, byte[] newPin). The Secure Storage can also provide other PIN related ISO/IEC 7816-4 commands like DISABLE VERIFICATION REQUIREMENT, ENABLE VERIFICATION REQUIREMENT for allowing an extended PIN management with the Authentication Service methods deactivatePin(byte[] pin) and activatePin(byte[] pin).

8. Recommendation for a minimum set of functionality

The Secure Element access is a must for secure applications and as a result any mobile device compliant with the Open Mobile API must provide access to all Secure Elements on the device. Therefore the Transport API (with access to all Secure Elements available in the device, e.g. SIM, microSD, eSE, …) is mandatory for Open Mobile API compliant devices. In case a mobile device has a Secure Element with no access to the Transport API this device would be considered as not compliant with Open Mobile API.
The most common Secure Elements today are the SIM cards, secure SD cards or embedded SEs. But new Secure Element can arise in the future. Therefore the Secure Element Provider interface is mandatory to insure that the device can support new Secure Elements in the future.

The Transport API shall support logical channels according to ISO with up to 20 channels. A UICC should indicate the logical channel support in the historical bytes of the ATR, however retrieving the ATR is not valid for non-UICC Secure Elements. Therefore, in case the SE is a UICC, the API should open the logical channels as the ATR indicates. In case the SE is not a UICC or the UICC ATR has not been received the API shall try to open logical channels as long as no error is indicated.

The Transport API shall support extended length APDU commands independent of the coding within the ATR. As low level components (e.g. baseband, CLF, SD host, …) might block extended length APDU commands it cannot be guaranteed that a device implementing the Transport API could work properly when extended lengths are used.

These following components can be provided by device manufacturers or third parties:

- Discovery API
- Crypto API
- Secure Storage
- File Management
- Authentication
- PKCS#15

The Mobile Device should allow the installation of these services as an add-on API.

9. Secure Element Provider Interface

This provides a way to add and remove drivers for Secure Elements at runtime by installing or removing a downloadable application packages. It enables communication to that SE.
The concrete API of such a Secure Element Provider Interface is up to the individual implementer of the Open Mobile API and not defined in this document.

However the following requirements must be fulfilled:

- The implementation has to enforce that it is only used by the transport layer and not directly by mobile applications. (So channel management and security mechanism in the transport layer can not be bypassed).
- A reference implementation needs to be available.
- Existing Secure Element Providers cannot be replaced by dynamically loaded providers.

10. Access Control

Access Control is used by the transport and service layer. It's based on the signature of the mobile application and is enforced when accessing the transport layer.
The permission is based on access policies stored in the SE. These access policies define precisely which mobile application is allowed to get access to an Applet installed in the SE.

The APIs in this specification will be indicating errors (e.g. declaring security errors) when they are subject to the Access Control.
Access Control itself is defined by GlobalPlatform in the SE Access Control Working Group (see [9]).

11. History

Table 11-1: History

Version	Date	Author	Comment
1.0	28.02.2011	SIMalliance	Initial Release 1.0
1.01	16.03.2011	SIMalliance	Minor corrections
1.1	04.05.2011	SIMalliance	Clarifications for several functions in the transport layer
1.2	12.07.2011	SIMalliance	Correction for open Basic Channel
2.0	30.09.2011	SIMalliance	Adding descriptions of the service layer, corrections in the transport layer, adding getSelectResponse() to channel class
2.01	14.10.2011	SIMalliance	Minor corrections
2.02	4.11.2011	SIMalliance	Corrections in diagrams of the transport layer
2.03	19.06.2012	SIMalliance	Clarification on Chapter 10 (since GlobalPlatform SEAC spec is released), on 6.4.4. and 6.7.6 / 6.7.7
2.04	15.07.2013	SIMalliance	Clarification on Chapter 5, 6.2, 6.7.6, 6.7.7, 6.8.6, 7.1, 7.6.1, 7.6.3, 7.6.4, 7.6.5, 7.8.1 and chapter 8. Added 6.4.5, 6.8.7
2.05	28.01.2014	SIMalliance	Chapter 3, clarification on the namespace; Chapter 6.4.2: IllegalStateError added; Chapter 6.6.1 changed definition of reader name according request from GSMA; Chapter 6.8.6: clarification on handling of status words