Comput Syst Sci & Eng (2005) 6: 439–450

International Journal of
Computer Systems
Science & Engineering

An open middleware for smart cards

Tommaso Cucinotta*, Marco Di Natale* and David Corcoran†

*Scuola Superiore Sant'Anna, Pisa, Italy. Email: {cucinotta,marco}@sssup.it
†Identity Alliance, Austin, TX, USA. corcoran@identityalliance.com

This paper presents an open and modular middleware for smart cards, providing a simple abstraction of the device to application developers. The software is interoperable across multiple card devices, and portable across various open platforms. The architectural design is centred around the definition of a new API that allows protected access to the storage and cryptographic facilities of a smart card. In the envisioned architecture, a smart card driver architecture is partitioned into a lower card-dependent component, that formats and exchanges APDUs with the external device, and a higher card-independent component, that implements more sophisticated services and interfaces, such as the well known PKCS-11 standard. Each layer can focus on a smaller set of functionality, thus reducing the effort required for the development as well as the testing and maintenance of each component. The proposed architecture, along with a set of pilot applications such as secure remote shell, secure web services, local login and digital signature, has been developed and tested on various platforms, including Open BSD, Linux, Solaris and Mac OS X, proving effectiveness of the new approach.

Keywords: smart cards, middleware, architecture

1. INTRODUCTION

Security of applications and services is becoming increasingly important for today's software applications. The design and development of complex software systems is much better off if security issues are addressed since the early stages of the design and development, rather than applied as a late patch to an existing application. In order to achieve an adequate level of security while exchanging information or running transactions onto an open network, such as the Internet, cryptographic mechanisms need to be used. The adoption of such techniques can only guarantee protection of the application data as long as the cryptographic keys are securely created, stored and managed. Key bit strings are often the weakest point of the overall security trust chain, where compromise of a single cryptographic key related to an application may lead to the compromise of the entire set of data managed by that application. Management of cryptographic keys is thus a crucial point in the design and development of secure systems.

Smart card (SC) technology is, among others, an enabler for guaranteeing the secure management of cryptographic keys. Card devices have a high degree of trustworthiness, for many reasons:

- a card owner always has physical control of the card;
- on-card architecture is very simple, hence on-board code and logic can easily be made functionally correct;
- SC hardware is designed to be tamper-proof, so that it is very hard and expensive to recover the contained data by means of physical inspection.[1]

Smart cards are sufficiently powerful to perform cryptographic operations on-board, without the need to reveal cryp-

1 Cheap timing and power analysis attacks have been shown to be effective against some kind of devices, as shown in [20]. Security assurance for these devices is an active research topic [21,22], but clearly beyond the scope of this paper.

tographic keys to the outside world. These operations are only allowed after a proper user identity verification, through the use of Personal Identification Numbers (PINs), or even biometrics information. In conclusion, smart cards possess a sufficiently simple architecture, and the right amount of processing capability. They have been standardized at the hardware level and are easily available at cheap prices, therefore, they can be trusted in managing on-board resources, more effectively than traditional computer systems.

1.1 Smart cards and open platforms

Despite the growing need for smart card integration inside systems for protecting user data, such devices are struggling in being supported by secure applications and software components, especially on open systems. On these platforms, open source libraries and applications allow the use of cryptography for protecting data, but the security level that can be possibly achieved is strongly limited because of the use of software-only cryptography. This means that most computer systems, today, store cryptographic keys onto hard-drives, sometimes protected by weak passwords quickly chosen by careless users, thus exposing crucial pieces of information to the risk of being compromised due to the great number of malicious software programs and other kind of menaces overpopulating the Internet.

This situation is essentially due to the difficulty inherent to the integration of smart card technology inside applications, beacuse of the high number of reader and card devices, different in nature and capabilities, programmers have to deal with.

Smart cards differ significantly with respect to other hardware components, which are being supported not only on proprietary platforms, but also on open ones (albeit with a reduced set of functionality). Smart card manufacturers often deliberatly deviate from standards in order to provide products that differentiate themselves from others, possibly claiming enhanced security features. Standard APIs for interoperability do exist [1,2], but only a few vendors provide their implementation on open platforms, and, even when they do it, it is only for one or a limited set of devices. Furthermore, due to the short life cycle of a smart card device and to the low priority an open platform is usually assigned with respect to commercial ones, a card driver for an open system is likely to be released when manufacturing of the device is about to cease.

On a related note, protocol specification for card devices do not exist, apart from a few vendors who provide them for a limited set of devices. This further reduces the chances of having software components that support such devices on open systems. This situation discourages smart card integration and has the consequence of an overall reduction in their use, hindering the development of more secure computer applications and services.

The MUSCLE[2] Card middleware, which is being introduced in this paper, constitutes a step toward openness and simplification in smart card middleware design and implementation. Architectural solutions like the one we are proposing can ease adoption of these devices by speeding up the development stage of middleware components.

The paper is structured as follows. In the next section, after introducing common concepts about SC middleware, other open architectures for smart cards are overviewed. Section 3 introduces the proposed architecture and features an overview of the new API. Finally, we draw our conclusions in Section 4.

2. BACKGROUND ON SMART CARD MIDDLEWARE

This section recalls the basic concepts of smart card architectures and protocols, and introduces some terminology. These concepts will be extensively used in Section 3 for describing the proposed architecture.

2.1 The world of smart cards

The world of smart cards is characterised by various card-reader and card device types. Card readers can be connected through the serial, PS/2 or USB ports. Some of them have multiple slots for the insertion of multiple cards at the same time. Others have an on-board pin pad allowing the user to enter the PIN code in a more secure way than the traditional solution where the user is required to enter the PIN code onto the PC keyboard. Some readers are also capable of wireless communications with the card, that is without need of physical card insertion.

Different types of card devices exist as well. Storage-only cards are traditionally used for storing, in a protected and mobile way, some kind of information, and on-card logics is used to perform basic operations such as data retrieval or decrease of internal counters. A typical application for this family of products is a pre-paid card, where the information stored onto the card corresponds to the amount of money that is available to the user for accessing some service, such as telephone calls or Internet connectivity. The user is required to insert the card into a terminal during service operations. The terminal for retrieves the residual (pre-paid) service time from the card and decreases the available credit. Usually, such cards only require PIN code verification or possibly do not authenticate users at all.

A *cryptography-enabled* smart card, instead, is able to perform sophisticated cryptographic operations, usually for the purpose of authenticating to a system on behalf of the legitimate owner. In such cases, the card stores the user authentication cryptographic key, and proves possession of it during a *challenge-response* cryptographic protocol that is run between the target system and the card device. The cryptographic key is stored onto the tamperproof smart card device and is used by the card itself without exposing it to the outside world. A different kind of application is *digital signature*, where a document is signed by using the on-card user's signature private key. The signature is computed on the card device, which . may be required to perform other operations such as data digesting and padding, for the purpose of increasing security. Sometimes smart cards may authenticate the host system in a cryptographic way, to prevent unauthorized use of the device. In these cases, the card

2 Movement for the Use of Smart Cards in a Linux Environment.

runs a cryptographic challenge-response protocol with the host PC, where it is the host PC that must prove possession of its own authentication cryptographic key.

GSM-enabled smart cards are a special kind of cryptographic smart cards widely used in Global System Mobile (GSM) telecommunication systems, as Subscriber Identity Modules (SIMs). These cards authenticate users to the mobile telephony provider for the purpose of accounting the calls made by the phone, and possibly other services. Cryptographic capabilities are made available by means of the GSM standard protocol [4].

A *programmable* smart card is usually a cryptographic smart card with a general-purpose CPU on-board, allowing dynamic loading and execution of programs. These devices allow implementation of custom applications and possibly custom protocols for interaction with the host PC. These devices are usually programmed by using a subset of a well-known programming language, such as Java, Assembler or Basic and offer the highest flexibility to applications, allowing delegation of security-critical operations to the protected on-card environment.

In spite of the effort made by standard organisations [5,6], card devices have many restrictions and non standard filesystem structures. Different cards have different ways to accomplish the same function, such as, for example, file creation.

The simplest way of increasing an application or system security through the use of smart card technology is by delegating management and use of one or more cryptographic keys to the card device. For PKI applications, one or more public key certificates can be stored on the card for easing mobility of the card among various physical locations. This is usually achieved through common, high-level, application programming interfaces that support on-card operations independently of the card and reader devices. The two most common interfaces at this level are briefly overviewed in the following subsection.

2.2 Smart card middleware architecture

Two APIs that have been defined for this purpose are PKCS-11 [1] by RSA Labs, and PCSC [2], Part 6, by the PCSC Workgroup. While the former has been widely adopted on various systems and platforms, most of them proprietary, the latter is only used on Microsoft platforms. Such high level APIs are made available to applications through a smart card middleware that requires various drivers to be installed on the system, depending on the actual reader and card devices that are going to be used.

A generic smart card middleware architecture is depicted in Figure 1. At the bottom layers, a resource manager component is required for managing the SC readers that are available on the system, and making their services available to higher level components, in a way that is independent of the hardware. This is done through the PCSC ICC Resource Manager interface [2, Part 5], which provides function calls for listing the available readers, querying a reader about the inserted card(s), enabling or disabling the power to an inserted card, and establishing an exclusive or shared communication channel for data exchange with a card. The reader driver takes care of translating the requests into the low-level

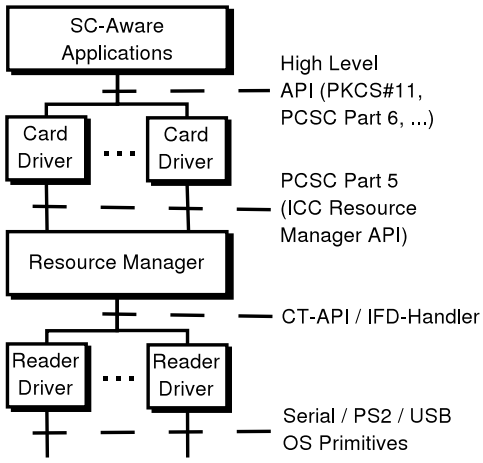

Figure 1 Architecture of a traditional smart card middleware

Protocol Data Units (PDUs) to be transmitted to the reader through the low level OS primitives for serial communication. Reader drivers implement the Card Terminal API [3] or the PCSC IFD-Handler API [2, Part 3 Appendix A], and the resource manager translates calls to the PCSC Part 5 interface to the appropriate lower level API calls. The higher software stack, once a communication channel with a card device is established, performs data exchanges through command Application PDUs (APDUs) compliant to the ISO T = 0 or T = 1 protocols [6] (see later).

The top level component of the middleware is traditionally a monolithic component, provided by card vendors, implementing the PKCS-11 or PCSC Part 6 interfaces. These have calls that allow the application to locate, manage and use cryptographic keys and public key certificates that are available on the card. The card driver translates such requests into the appropriate lower level ISO T = 0 or T = 1 command APDUs to be exchanged with the card. Typically, it supports a family of card devices provided by the vendor. Furthermore, it must comply with the higher level API, which requires additional tasks to be performed in the component, such as session management and transaction handling. These tasks are quite similar in the driver implementations provided by different vendors, where the only changes relate to the specific way information is exchanged with the card by means of APDUs. This is why we investigated the possibility of introducing a further abstraction layer, breaking the traditional driver architecture through the use of a middle-level API.

In fact, in our architecture functions are grouped into two separate components:

- a lower level (LL) driver, which formats and exchanges command APDUs with the card device;
- a higher level (HL) one, which performs the additional management tasks required for the compliance with the higher level interface.

This is done through the introduction of a middle-level API, clearly identifying the boundary and commitments of the two sublayers around which the cited functionalities are split. As it will be shown in Section 3.1, the main benefit of such an approach is that it is possible to write the HL-API-specific

management code only once. Interoperability among card devices is achieved by writing, for each card, a specific LL driver implementing the common middle-level API.

2.3 Standard protocols and APIs

2.3.1 The T = 0 and T = 1 protocols

In order to allow a better understanding of the proposed architecture, we provide an overview of how the T = 0 protocol works, while the complete specifications can be found in [7]). The T = 0 and T = 1 protocols define, respectively, an asynchronous half duplex character oriented and a block oriented transmission protocol for exchanging data between an interface device (i.e. a card reader) and a smart card. These protocols require that each action be started by the host by sending a *command APDU* to the card, composed of a mandatory *header* and an optional, variable length, *data field*. After having performed some internal computations, the card sends back a *response APDU* to the host, composed of an optional, variable length, data field and a mandatory *status word* (see Figure 2). The T = 0 header consists of five bytes. The class byte CLA and the instruction byte INS identify the command to be performed, while the bytes P1 and P2, along with the optional following data, are the input parameters to the command. The fifth byte contains either the length of the optional data sent by the host after the header in the command APDU, or the expected length of the optional data sent by the card before the status word in the response APDU. The status word in the response APDU is a two bytes sequence used to notify if the command completed successfully (usually this corresponds to a value of `0 x 9000`) or not.

For each command-response APDUs only one device is allowed to send data, and it may be either the card or the host, , but not both. Four modes of operation are allowed:

- the host sends no data, the card responds with no data, i.e. only with a status word indicating whether the operation was successful;
- the host sends some data, the card responds with no data;
- the host sends no data, the card responds with some data;
- both the host and the card send some data.

All but the last mode may happen within a single command-response APDU transaction. However, whenever the card device needs to provide some data as a response to a command APDU containing data, it responds with a special status word (0 x 61XX) containing, in the second byte, the length of the data to be transmitted to the host. The host is supposed to retrieve such data by using a special command APDU, namely the GetResponse APDU. Also, note that various extensions are standardized that allow each command-response APDU to be more flexible in the way data is transmitted, especially when more than 256 bytes need to be transmitted at each exchange. However, not all card devices comply with such extensions.

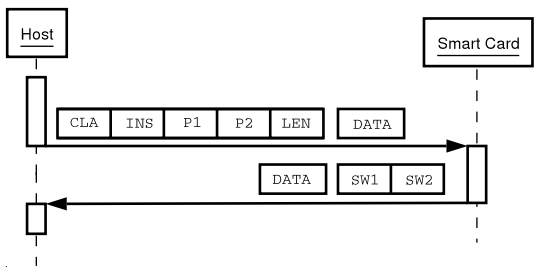

Figure 2 Generic invocation of a smart card command by exchanging ISO APDUs

2.3.2 The PKCS-11 Standard

The Cryptographic Token Interface Standard specifies an application programming interface (API), called *Cryptoki*, to mobile devices which hold cryptographic information and perform cryptographic operations, such as smart cards, PCMCIA cards and smart diskettes. The main goals of the API design are, among others, independence from the security device and resource sharing, so to allow multiple applications to share access to a single device, as well as allow access to multiple devices, presenting applications with a common, logical view of the device called a *cryptographic token*.

Cryptoki also provides an interface to cryptographic reader devices through the *slot* abstraction. Each slot, corresponding to a physical reader or other interface device, may contain a *token*. Typically, a token is 'present in the slot' when a cryptographic device is inserted into the reader or the interface device.

The kinds of supported devices and capabilities depend on the particular Cryptoki library and on the supported devices. The standard only specifies the interface to the library, but not all libraries support all the mechanisms (algorithms) defined in the interface (since not all tokens are expected to support all the mechanisms).

The logical view of a token is a device that stores typed objects and performs cryptographic operations. Each object type, or *class* in the Cryptoki terminology, is associated a set of metadata information, available as a set of *attributes*, i.e. name-value pairs. Some attributes are general, such as the private or public nature of objects, some are specific to a particular type of object, such as the modulus of an RSA key. Classes are arranged in a hierarchical fashion, where each class inherits attributes of the parent class. Cryptoki defines three main classes of objects: *certificates, keys* and *data*. A certificate object stores a public-key certificate. A key object stores a public key, a private key, or a symmetric key. Each of these types of keys has subtypes for use in specific *mechanisms*. A data object is a container whose contents is application-dependent.

Objects are also classified according to their lifetime, visibility, and access control. *Token objects* are persistently stored onto the token, even after token extraction, and are visible to all applications that connect to the token. *Session objects* are temporary objects which are only visible to the application which created them, and their lifetime is limited to the session in which they were created.

Public objects may be accessed without any prior authentication of the application. *Private objects*, on the other hand, require the application or user to authenticate to the token through the use of a PIN code or some other token-dependent method (for example, a biometric device).

Cryptoki defines functions to create, destroy, manipulate and search for objects. It also defines functions to perform cryptographic functions with an object.

Cryptoki recognizes two user types: the *security officer* (SO) and the *normal user*. The SO is responsible for initialization of a token, for setting the normal user's PIN and possibly for manipulating (some) public objects. Only the normal user is allowed to access private objects on the token, and the access is granted only after the normal user has been authenticated. Some tokens may also require that a user be authenticated before any cryptographic function can be performed on the token, whether or not it involves private objects.

The PKCS-11 standard also defines how the API behaves with respect to concurrent accesses by multiple tasks and threads to the same slot or token.

This interface is used both on proprietary platforms and on open systems like Linux. Unfortunately, on latter platforms, very few vendors provide PKCS-11 modules for at least some of their smart-card devices.

2.3.3 The PCSC Standard, Part 5

The part 5 of the PC/SC Workgroup's architecture defines the *ICC Resource Manager* component, which is responsible for managing the ICC-relevant resources within the system and for supporting controlled access to Interface Devices (IFDs) and, through them, individual Integrated Circuit Cards (ICCs). The ICC Resource Manager performs three basic functions related to access to multiple IFDs and ICCs. First, it is responsible for identification and tracking of resources. Second, it is responsible for controlling the allocation of FD resources across multiple applications. It does this by providing mechanisms for attaching to specific IFDs in shared or exclusive mode. Finally, it supports transactional access to services available within a given ICC. This is extremely important because current ICCs are single-threaded devices that often require execution of multiple commands to complete a single function. Transactions allow multiple commands to be executed without interruption, ensuring that intermediate state information is not corrupted.

The interface exposed by the ICC Resource Manager is described in an object-oriented fashion, in terms of *classes* and *methods*, along with required parameters and expected return values. The interface definition is language and system independent.

The class *ResourceManager* provides the methods necessary to create and manage *Contexts*, which are needed for communication with the ICC Resource Manager. The class *ScardTrack* encapsulates functions that determine the presence or absence of specific card types within the available readers. This information is made available based on selection criteria provided by the calling application. The class *ScardComm* encapsulates a communication interface to a specific card or reader and provides methods for managing the connections, controlling transactions, sending and receiving commands, and extracting information on the card state. A fundamental method of this class is the *ScardTransmit* function, which allows exchange of ISO/IEC T = 0 and T = 1 APDUs with an ICC device.

This API is a standard component on Microsoft platforms, and, thanks to the MUSCLE project[3], it is available on many open Unix-like platforms too, such as Linux, OpenBSD and Mac OS-X.

3 Movement for the Use of Smart Cards in a Linux Environment, more information available at the URL: http://www.musclecard.com.

2.3.4 The PCSC Standard, Part 6

Part 6 of the PC/SC Workgroup's architecture defines the Service Provider (SP) component, consisting of two fundamental subcomponents: the ICC Service Provider (ICCSP) and the Cryptographic Service Provider (CSP).

The ICCSP is responsible for exposing high-level interfaces to non-cryptographic services. This includes common interfaces, defined in the specification, for managing connections to a specific ICC, as well as access to file and authentication services. The ICCSP interface provides mechanisms for connecting and disconnecting to an ICC and it exposes file access and authentication services encapsulating functionality defined by ISO 7816-4, along with natural extensions for functionality such as file creation and deletion. The file access interface defines mechanisms for locating files by name, creating or opening files, reading and writing file contents, closing a file, deleting files, and managing file attributes. The authentication interface defines mechanisms for cardholder verification, ICC authentication, and application authentication to the ICC. In addition, the ICCSP may implement interfaces for features specific to the application domain.

CSP (in contrast with ICCSP) isolates cryptographic services in response to regulations issued by governments upon import and export. CSP compartmentalizes the sensitive elements of cryptographic support into a well-defined and independently installable software package and it encapsulates access to cryptographic functionality provided by a specific ICC through high level programming interfaces. Its purpose is exposing general-purpose cryptographic services, like key generation, key management, digital signatures, message digesting, bulk encryption services, and key import and export to applications.

Relevant classes defined in this part of the standard include:

- the *FileAccess* class, which defines a high level interface to a ISO 7816-4 on-card based file system;
- the *CHVerification* class, which provides an application with the ability to force a *Card Holder Verification* (CHV, i.e. a PIN code in the PCSC terminology) and allows user modifications of the CHV;
- the *CardAuth* class, which exposes the interfaces to the authentication services that may be supported by an ICC, i.e. it allows to run cryptographic challenge-response protocols for the authentication of the host PC application or the card;
- the *CryptProv* class, which exposes the primary methods for accessing cryptographic services.

Unfortunately, this API is only available on Microsoft platforms, and it constitutes the standard way a smart card vendor integrates its own devices with widely known applications such as Internet Explorer and Outlook. On open platforms, the few vendors providing a high level API usually provide a PKCS-11 module.

2.4 Related projects

This subsection provides a quick overview of existing open architectures for smart cards, highlighting how these projects compare with respect to the proposed architecture.

The OpenSC [8] project provides a library and a set of

utilities for accessing ISO 7816 [5] and PKCS-15 [9] compliant card devices. Specifically, the project features a programming interface with functionality for: ISO 7816-4 [5, Part 4] filesystem browsing and file reading/writing; ISO 7816-9 [5, Part 9] filesystem management; ISO 7816-8 [5, Part 8] cryptogram computation for cards complying with the PKCS-15 standard for storing certificate and key information. It provides a good set of middleware components, as well as modules for their integration within widely used secure applications, constituting an effective solution for integration of ISO 7816-4 and PKCS-15 compliant, pre-formatted devices. Though, various cards exist today with custom, proprietary APDUs for filesystem management, which adhere to ISO 7816-4 in a read-only fashion, and/or do not comply with the PKCS-15 standard for managing information about the on board cryptographic data. Such devices cannot be directly supported within this architecture, especially on the side of card-personalisation.

The SecTok [10] project provides a library for the management of files onto an ISO 7816-4 compliant device. The library includes functions for initialisation, reading and writing of files. It does not support cryptographic functionality of the devices, thus it cannot be used in the context of cryptographic smart cards.

The Open Card Framework (OCF) [11] is a Java based development platform for smart card development. It aims at reducing dependence from card terminal vendors, card operating system providers and card issuers, by means of a consistent and expandable framework. The core architecture of OCF features two main parts: the *CardTerminal* layer, providing access to physical card terminals and inserted smart cards, and the *CardService* layer, providing support for a wide variety of card operating systems and the different functions they offer. Examples of CardServices are the FileAccessCardService, providing a fairly complete set of interfaces and classes abstracting an ISO file system, and the SignatureCardService, offering methods to create and verify digital signatures. Further, the problem of card issuer independence is addressed separately by the OCF's *Application-Management* component, supporting listing, selecting, installing, uninstalling, blocking and unblocking of applications. OCF is a promising framework for smart card integration within Java applications. Despite the modular and expandable design, its main limitations are due to the lack of support of some readers because of the way I/O is managed at the lowest levels of the architecture, and the inherent difficulties and overheads to access functionality from programs written in different programming languages than Java.

The GPKCS-11 project [12] aims at providing support for the development of a PKCS-11 driver for cryptographic tokens. It contains a complete software token library, based on the OpenSSL project[4], as well as an automated testing environment for PKCS-11 modules. The framework provides basic services for managing PKCS-11 session handles, object handles, and object attributes through the use of internal lookup tables that map handles to C structures, and vice-versa. A GPKCS-11 driver implements an internal API which resembles the original PKCS-11 API, where all handles have been substituted with the looked up C structures, and some mandatory parameters checking dictated by the standard are embedded within the framework.

Furthermore, the framework aims at leaveraging the programmer from the support of concurrent applications, by implementing the necessary locking mechanisms within the framework[5]. Even if most of the concepts that inspired the GPKCS-11 project are clearly valuable, documentation of its features is currently insufficient, and it has not been maintained since 2000. This, in our opinion, hinders its adoption.

The Common Data Security Architecture (CDSA) [13] is an open standard introducing an interoperable, multi-platform, extensible software infrastructure for providing high level security services to C and C++ applications. It features a common API for authentication, encryption, and security policy management. As far as smart card technology is concerned, the CDSA standard supports external cryptographic devices through the use of PKCS-11 modules, while the overall architecture is designed and focused around higher level security services, such as certificate and CRL management, verification of signatures, authentication, and others. Initiated by Intel in 1995, CDSA v2.0 was adopted by The Open Group at the beginning of 1998. Intel also initiated a reference implementation, later moved to open-source[6] [14], targeting both Microsoft and various Unix-like platforms.

With respect to our, and to the other cited architectures, CDSA is essentially at a higher software level, aiming at providing complex security services, rather than low-level cryptographic services interoperable among different security devices.

Table 1 provides a brief summary of the cited open architectures, along with their limitations or relationship with respect to our new approach.The architecture framework introduced in this paper, (at least by the authors' knowledge), is the only open architecture completely modular that allows support for multiple heterogeneous devices through the implementation of a common lower level API. The API exposes sufficient functionality as needed by most PKI applications from the issuing of the card by a CA, up to the use of the device by applications, such as management of on-card memory, cryptographic keys and PIN codes. Adoption of the middleware limits the effort needed for the implementation of drivers, at least with respect to full implementation of well known standards, such as PKCS-11 or PCSC level 6.

Table 1 Summary of open architectures for smart cards

Project name	Language	Limitation
OpenSC	C	Only ISO 7816-4 and PKCS-15 compliant devices
SecTok	C	Only ISO 7816-4 storage functionality
OCF	Java	High overhead for non Java application
GPKCS-11	C	Lack of documentation and maintenance
CDSA	C	Higher level of abstaction

4 More information at the URL: http://www.openssl.org.

5 The implementation is available at the URL: http://sourceforge.net/projects/cdsa

6 It is available at the URL: http://sourceforge.net/projects/cdsa.

Connectivity with existing standards is still possible through implementation of the higher level API exploiting the MUSCLE Card API. Mapping MUSCLE services to higher level APIs can be confined in a separate module, to be implemented only once and for all. As an example, our architecture provides a PKCS-11 module that suffices for all card devices for which a lower level plugin has been implemented. The module, for example, can be plugged at the lowest levels of the CDSA architecture.

3. PROPOSED ARCHITECTURE

The MUSCLE Card project proposes an open SC middleware that is both interoperable across multi-vendor card devices, and portable across a multitude of open platforms. The middleware architecture of the MUSCLE Card project is shown in Figure 3. At the bottom layers, the PCSC-Lite project provides an open and stable daemon for managing the SC-related hardware resources of the PC (e.g. serial/USB ports, connected readers). Various readers are supported through reader drivers, most of which are open source, implementing either the CT-API or the IFD-Handler interface. Devices connected to serial and PS2 ports need to be already connected when the daemon starts, while USB devices can be plugged in at run-time, provided that the drivers are installed onto the system.

At the above layer, independence from the card is achieved by using a common API. Specifically, the *Card Driver Loader*, at the time the card is inserted, identifies the inserted device through the Answer To Reset (ATR) bytes, then loads dynamically the driver that manages the card. Differently from traditional approaches, in which higher level APIs such as PKCS-11 or PCSC Level 6 are implemented by card drivers, in the proposed architecture a card driver implements a simpler API (see Section 3.1). This exposes basic storage, cryptographic and access control functionality to the host machine, independently of the kind of card device the host is using. This interface is inspired by the protocol introduced in [15], in that most function calls are directly mapped into the APDUs of the protocol. The layer has been implemented in various card drivers for devices that are different in architecture and nature. Examples are Schlumberger Cyberflex Access 32K and Gemplus 211/PK cards, two programmable cards based on the JavaCard platform, which are supported once an appropriate JavaCard Applet (MUSCLE Card Applet) has been loaded on-board; the Schlumberger Cryptoflex 16K card, which exposes a set of ISO 7816-4 APDUs for filesystem management, and custom commands for cryptographic operations; and the US Department of Defence (DoD) card, which exposes a custom data model. Details on the proposed API follow in the next subsections. On top of our API, an open source PKCS-11 module has been developed, mapping the PKCS-11 API calls into the appropriate sequences of MUSCLE Card API function calls. This allows PKCS-11 compliant applications to use our architecture for communicating with the supported devices, for accessing the on-card services as exposed through the PKCS-11 model.

As an alternative, applications can directly use the proposed API in order to communicate with smart cards at a lower level, and to take advantage of the exposed functionality, like access control mechanisms based on multiple PINs or other authentication means.

The API has been directly used for embedding smartcard technology into a set of target applications, within the Smart Sign[7] and MUSCLE projects:

- a command line digital signature application (sign-mcard);
- a variant of the OpenSSH software (openssh-mcard);
- a Pluggable Authentication Module (PAM) [16] , directly developed using this API, allowing smartcard based user authentication for applications using PAM on Unix like systems, like the Unix login;
- a CSP module for Windows platforms has also been developed, integrating functionality of the exposed architecture into applications like MS Outlook, Internet Explorer and Windows login.

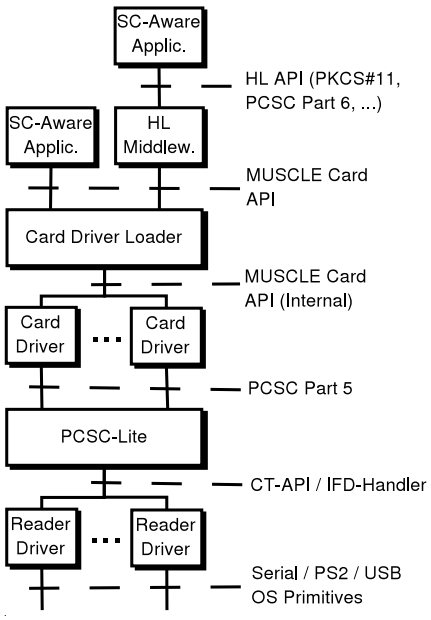

Figure 3 Architecture of the proposed smart card middleware

3.1 The new smart card API

This section, after explaning main design choices behind the development of the new middle-level smart card API, introduces main features of the new interface, showing how these have been provided through the various functions available in the API.For a complete interface specification, the reader is referred to [17].

3.1.1 Objectives and design choices

The new API has been designed with the aim of providing higher layer software components with an open, simple, and card independent framework which exhibits sufficient generality to meet the requirements of a multitude of target applications, including digital signature, secure e-mail, secure login, secure remote terminal and secure on-line web ser-

7 More information available at the URL:http://smartsign.sourceforge.net.

vices, both PKI based and not. These requirements have been identified in having a means for generating, importing, exporting, and using cryptographic keys on the card. Another requirement has been identified in having a means for creating, reading, and writing generic data on the card in generic "containers". This is useful, for example, to store a public key certificate associated with a private key on the card. Access to the on-card resources needs to be granted only after host application and user authentication. The API design allows future extensions, like the use of alternative key types or authentication mechanisms, as proved by the biometrics extensions that have recently been added [18]. The resulting interface has been proved to be enough simple and light, so to allow an easy integration of the new architecture into secure applications, as shown by our sample application cases.

Our interface does not target sophisticated card services that might be needed by specific applications. Multi-key digital signatures and authentication schemes may need specific functionalities to be provided through the use of multiple cards. These applications can still benefit from the exposed middleware by extending it with the required functionality, given the open nature of the project.

3.1.2 API function set

The set of functions available in the proposed API is summarised in Table 2. API functionality has been divided into 6 general function sets, giving access to one or more of our middleware class of services, namely: session management, data storage, cryptographic key management, PIN management, access control, and a set of miscellaneous functions. In the following, we provide detailed information on the intended use of the various API calls. For the complete API specification, the reader should refer to [17].

Table 2 MUSCLE Card API function set

Category	Function name(s)
Session management	ListTokens, EstablishConnection, ReleaseConnection WaitForTokenEvent, CancelEventWait BeginTransaction, EndTransaction
Data storage	CreateObject, DeleteObject, ListObjects WriteObject, ReadObject
Cryptography	GenerateKeyPair, ComputeCrypt ImportKey, ExportKey, ListKeys
PIN management	CreatePIN, ChangePIN, UnblockPIN, ListPINs
Access control	VerifyPIN GetChallenge, ExtAuthenticate GetStatus, LogOutAll
Miscellaneous	WriteFramework, GetCapabilities ExtendedFeature

3.1.3 Session management

The session management functions allow an application to enumerate connected readers and inserted card devices, as well as to manage connections with a smart card. Establishment of a connection to a smart card is a prerequisite for the use of any of the other functions of the API. Specifically, the ListTokens function enumerates the readers connected to the system, the readers which have a card inserted, along with the type of inserted device, and the list of all supported card devices in the system. Furthermore, applications can block and wait until card insertion or removal by using the `WaitForTokenEvent` function. Once a card is inserted into the reader, the `EstablishConnection` and `ReleaseConnection` functions allow to reset the device and prepare it for subsequent commands. When connecting to a card, it is possible to select either exclusive or shared access to the card. In the latter case, it is possible to acquire an exclusive lock on the device with a call to the BeginTransaction function, and release it with the EndTransaction function.

An example sequence of calls needed for the establishment of a session with a smart card device is shown in Figure 4, in the case when the device is not connected, and the application waits for its insertion.

3.1.4 Data storage services

The API specification encapsulates application data into simple containers called *objects*, identified by means of a string identifier (OID). Access control is enforced on a per-object and per-operation basis, distinguishing among create, read, write and delete operations (more details follow). The proposed minimum set of data storage services (operations and access constraints) suffice for the target applications cited at the beginning of Section 3.1, by allowing them to store, retrieve and manage data onto a card in a secure and controlled way. This does not preclude hierarchical organization of application data into a filesystem-like structure, which could be implemented on the client side on a per-application basis, or in an inter-application fashion if further documents came up standardizing classes of objects to be used for filesystem information. This approach would suffer of the limitation that the host-side applications be responsible for maintaining the consistency of the hierarchy without any mechanisms enforced by the card. Though, such an approach would be perfectly suitable in those cases in which the hierarchy is static, i.e. it is never modified after creation.

The `CreateObject` function allows creation of an empty object on the card, providing the object name, size and access control list (see forward for details). The same information may be visioned by applications for all the existing on-card objects through subsequent calls to the `ListObject-` function.[8]

Reading and writing of data to and from objects is performed, respectively, through the `ReadObject` and `WriteObject` functions. Execution of these functions may be

8 Order in which objects are listed is not specified, and may vary depending on the type of device, specifics of the driver, and order of creation

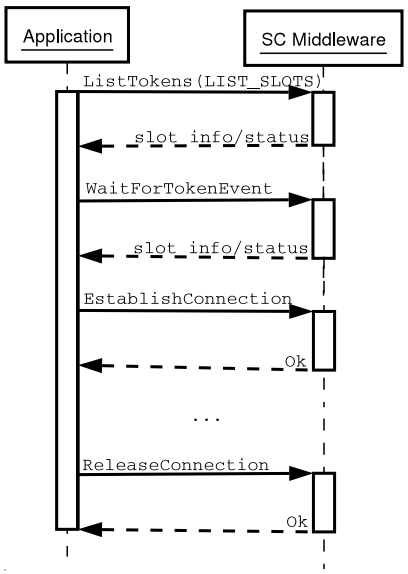

Figure 4 Sequence of calls needed for establishing a connection with a card device

restricted on a per-object and per-operation basis. The API specification does not define specific object contents, leaving the applications total freedom on what to store onto a card, like user private information, application specific data or public key certificates. The nature of the stored data is highly dependent on the application itself, and out of the scope of our interface specification, which leaves space for other documents to come up standardising OIDs to be used to store special information with an inter-application relevance. As an example, the PKCS-11 module defines and manages two objects for each private cryptographic key on the card: one for storing the public key certificate associated to the key, the other for storing the PKCS-11 attributes associated with the key.

As far as the card storage capacity is concerned, the interface specification only gives a view of the total available memory on the device, through the `GetStatus` function. It does not deal with various aspects of on-card memory management, which depend on the allocation strategy performed by the device, such as whether an object with a given size can be created or not, how many objects are allowed to exist due to constraints (i.e. allocation tables), how object memory is freed on the device (i.e. by use of compaction or full defragmentation of free blocks).

Some smart card devices tend to separate among a public and a private memory space. The first one typically contains public information that can be read or possibly changed at any time (like user preferences). The latter is reserved for personal data to be handled by an application and its use is allowed only after PIN verification. Our approach is different in that we have a unique memory space, where individual objects are associated with access control rules specifically customized for each object and operation. Hence, it allows reconfiguration of the memory space as public or private according to application requirements. Section 3.1.6 explores in more detail the adopted security model.

3.1.5 Cryptographic services

The API allows up to 16 keys to be managed on the card, identified by means of a numeric key identifier ranging from 0 to 15. A full key pair can be stored by using two key identifiers. All key types defined in the Java Card 2.1.1 standard are allowed: RSA, DSA, DES, Triple DES, and Triple DES with 3 keys. The interface is designed to allow addition of further key types in the future. Operations related to cryptographic keys are: import of a key from the host, export of a key to the host, encryption and decryption of cryptograms, signing, and listing of keys, which provides size and type information. All key operations except key listing are protected by an access control mechanism, as described in the next paragraph, in order to forbid unauthorized use of a key. Key pairs may be directly generated on the card device, through the `GenerateKeyPair` function, or imported from the outside world, through the `ImportKey` function. These calls allow to set up the access control information related to each new key, specifying under what conditions subsequent reading, overwriting and use operations are allowed for each of the generated or imported keys. For example, it is possible to generate a key pair guaranteeing that the private key can never be exposed outside the card. After a key pair generation, the public key can be obtained with a call to `ExportKey`, as highlighted in Figure 5.

3.1.6 Security model and access control enforcement

On-card resources are protected by a simple Access Control List (ACL) based model, so that operations are allowed only after appropriate host application and user authentication. This may be performed by means of a PIN code verification, a *challenge-response* cryptographic authentication protocol, or a combination of both methods. Furthermore, the API has been designed to allow future support for other identification schemes, like fingerprint verification.

Access rules for on-card resources are specified by using the concept of *identity*. This term refers to one of the 16 authentication mechanisms that host applications and users can use to be authenticated by the cryptographic device. Identities, PINs, and cryptographic keys are referred to by means of numeric identifiers. Different types of identity are defined (see also Table 3): identities n.0-7 are labelled as *PIN-based* and are associated, respectively, with PIN codes n.0-7; identities n.8-13 are *strong* identities associated, respectively, with cryptographic keys n.0-5 for the purpose of running challenge-response cryptographic authentication

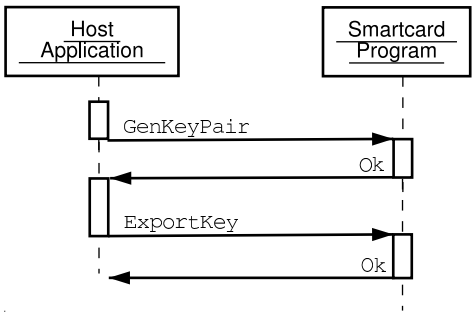

Figure 5 Typical sequence of commands for an on-board key pair generation

9 The fingerprint verification mechanism recently developed uses identity n. 14.

Table 3 Association between identity, PIN and cryptographic key numbers

Identity number	Identity type	Linked to
0	PIN-based	PIN n.0
1	PIN-based	PIN n.1
...	...	...
7	PIN-based	PIN n.7
8	Strong	Key n.0
9	Strong	Key n.1
...	...	...
13	Strong	Key n.5
14	Reserved	Undefined
15	Reserved	Undefined

protocols; identities n.14-15 are *reserved* for future use[9].

At each moment, a session with a smart card is associated to a set of *logged* in identities, representing the set of authentication mechanisms which have successfully run since the start of the session. A successful run of any of the authentication mechanisms causes the *log* in of the corresponding identity, in addition to the identities already logged in. The use of multiple identities allows a host application to switch to a higher security level that grants access to more capabilities by running additional authentication mechanisms. The `LogOutAll` command allows a host application to return back to the unauthenticated security status. A subset of the possible security states and transitions due to successful authentication commands is shown in Figure 6.

An ACL specifies which identities are required to grant access to operations of each object and key. Object operations are *read, write* and *delete*. Key operations are *overwrite* (either by means of regeneration or by means of import), *export*, and *use*. An ACL associated with an object or key is specified by means of three Access Control Words (ACW), each one related to an operation (see Figure 7).

An ACW consists of 16 bits. Each bit corresponds to one of the 16 identities. An all-zero ACW means that the operation is publicly available, that is, host applications can execute it without prior authentication. An ACW with one or more bits set means that all of the corresponding identities must be logged in at the time the operation is performed. An all-one ACW has the special meaning of completely disabling the operation, meaning that the operation can never be performed, independently of the connection security status. This

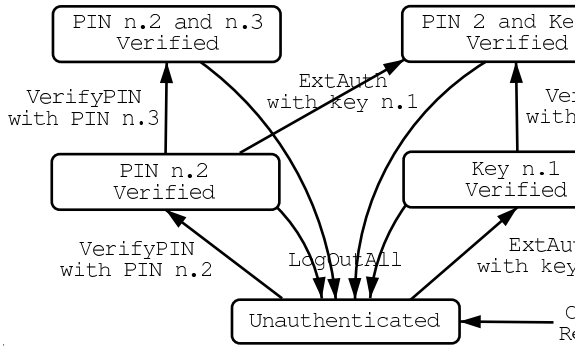

Figure 6 Subset of possible security state transitions allowed by the API specification

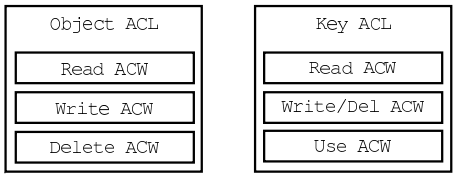

Figure 7 Composition of the Access Control List for objects and keys

ACW code comes into use for disabling reading of private keys. The security model has enough freedom to allow at least four levels of protection for card services. An operation can be *always allowed* if the ACW requires no authentication, *PIN protected* if the ACW requires a PIN verification, *strongly protected* if the ACW requires a strong authentication, and *disabled* if the ACW is all-ones, forbidding its execution. As an example, use of a private key onto a smart card is usually PIN protected, but some applications could require a strong protection. Reading of a private key is normally disabled. Public objects may always be readable, but their modification could be PIN protected. Private objects could require PIN protection for reading and possibly strong protection for writing.

3.1.7 PIN and security status management

Functions have been defined for PIN management, allowing to create, verify, change and unblock PINs. Specifically, the `CreatePIN` function allows to create a new PIN on the card, provided that the transport PIN has already been verified, and the `ListPIN` function allows listing of the existing PIN codes. In principle, up to eight PIN codes can be created onto a single card, though the actual maximum number depends on the underlying device, and may be queried by using the `GetCapabilities` function. The `VerifyPIN` function allows verification of a PIN code, and, if successful, logs in the corresponding identity. The identities logged in at a time may be queried by using the `GetStatus` function. The API defines a unique way of logging out of the device, through the use of the `LogOutAll` function, which logs out all identities at once, returning the session to the unauthenticated state. Finally, the `ChangePIN`function may be used to change the current PIN value, and the `UnblockPIN` function to unblock it after it blocked due to several verification tries with the wrong code.

3.2 Multiple applications, one single card

The API has been designed to allow multiple applications to use the same card without interfering with each other. In fact, each application can create its own PINs and/or cryptographic keys, and require their verification for accessing its own data and keys through the use of appropriate settings for the ACLs of such objects. As an example, on JavaCard devices this can be easily supported through the interaction with the MUSCLE Card Applet, or with different resident Applets. On ISO 7816 compliant devices, each application could define its own Directory File (DF) in which to keep certificates, keys and PINs relative to that application. Each application must be able to manage its own PINs, data objects and keys.

This has been accomplished in two ways: requiring verifi-

cation of a special (*transport*) identity to allow creation of new PINs, objects and cryptographic keys, and allowing applications to create additional identities by means of creating further PIN(s) or cryptographic key(s). These identities can be required in ACLs of application specific objects and for keys that are "sensitive" for the application. For example, when formatting the card, an application should create a new PIN and require all of its data and keys to be protected by that PIN. This way every time the user interacts with that application, she is required to enter the new PIN value, resulting in the guarantee that the application cannot manipulate other application's resources or create further resources on the card. This is also useful for protecting to some extent the on-card resources from possible attacks made by an untrusted terminal. In fact, after the user has entered a PIN code, the only allowed operations are exactly those specified at the format time (typically use of a key or reading of an object), with no possibility for the terminal to interfere with other applications.

3.3 Extensibility

Our middleware allows connectivity to smart card devices at a lower level than the one that is usually required for the implementation of standard PKCS-11 or PCSC interfaces. The set of functionality that is exposed to applications has been voluntarily kept small, in order to achieve a simple API. Particular attention has been paid to extensions that could be needed in the future. In order to allow such extensions to be performed without compromising previously developed software, the middleware has versioning built into it. The version information is available through the `GetStatus` command, by means of *minor* and *major* version numbers. An increment in the minor version number should retain compatibility with already written software. This could occur, for example, if commands needed to be added to the protocol itself, without changing behavior of already existing ones. An increment in the major version number, instead, would not retain such a compatibility, and would mean a change in some of the protocol core features.

Simple extensions of the first type could be necessary to embed into the protocol alternative user/application authentication schemes, different from the classic PIN verification and cryptographic challenge/response verification. Two identity numbers were reserved in the protocol for this purpose and could serve, for example, as a means for adding on-card biometric pattern matching without affecting the original protocol.

3.4 Card specific behaviour

The API which has just been introduced provides a unified means for higher level middleware components, as well as applications, to access the smart card services in a unified, card-independent way. However, it must be noted that only a JavaCard device with the MUSCLE Card Applet on-board can support the full set of available functionality. Each specific card generally supports only a subset of the prescribed functionality. For example, each card has its own constraints such as: the allowed key types and, for each type, the allowed key length and supported modes of operation. The API provides, through the `GetCapabilities` function, a means for querying what features are supported by the particular device that is connected to the system. This way it is possible to choose the right set of parameters for the specific card that is being used.

4. CONCLUSIONS

In this paper we described an open middleware for smart cards, which is highly modular due to the adoption of a new interface layer that abstracts from the specifics of a card. Such interface has been designed to support minimal functionality needed by applications that use smart card devices to manage cryptographic keys and other kind of data, e.g. public key certificates. With respect to a traditional smart card architecture, in the proposed middleware architecture a driver is split into two sublayers: the lower level one focuses on abstracting the specifics of each single device; the higher level one implements a standard interface, such as PKCS-11, still leaving the applications freedom to use the lower level interface, if needed. For example, a smart card aware, biometrics enhanced, application can directly use the middle level interface for using added functionality.

Target applications, comprising digital signature, secure (local) log-in and secure (remote) shell, have been integrated with smart card technology by using the new middle-level API, proving effectiveness of the new approach.

REFERENCES

1 *PKCS-11 version 2.1.1 Final Draft: Cryptographic Token Interface Standard*, RSA Laboratories, June 2001.

2 Interoperability Specification for ICCs and Personal Computer Systems, *PCSC Workgroup*, December 1997.

3 *Application Indipendent Card Terminal Application Programming Interface for ICC Applications (CT-API 1.1)*, TeleTrustT Deutschland e.V., Juergen Atrott, TUEV Informationstechnik GmbH, October 1998.

4 Global System for Mobile Communications (GSM 11.11) – Digital cellular telecommunications systems – Specification of the Subscriber Ientity Module, ETSI, December 1995.

5 *ISO/IEC 7816-4/7/8/9: Information technology – Identification cards – Integrated circuit(s) cards with contacts*, – Parts 4, 7, 8, 9, International Standard Organization, 1995.

6 *Government Smart Card Interoperability Specification: Contract Modification*, GSA, August 2000.

7 *ISO/IEC 7816-3: Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part 3*, International Standard Organization, 1989.

8 **Olaf Kirch,** OpenSC – Smart Cards on Linux, *Proc. of the 10th International Linux System Technology Conference*, October 2003.

9 *PKCS-15: A Cryptographic Token Information Format Standard*, RSA Laboratories, April 1999.

10 *Sectok library and applications*, Center for Information Technology Integration (CITI), University of Michigan, 2001.

11 *OpenCard Framework General Information Web Document*, OpenCard Consortium, October 1998.

12 *GPKCS11 – GNU PKCS-11 implementation*, TrustCenter, October 2000.

13 *Common Security: CDSA and CSSM, Version 2.3*, The Open Group, May 2000.

14 *Intel Common Data Security Architecture Reference Implementation*, 2001.
15 **Tommaso Cucinotta, Marco Di Natale and David Corcoran**, A protocol for programmable smart cards, Trust and Privacy for Digital Business (TRUSTBUS) Workshop, *Proc. of DEXA 2003*, Prague, Czech Republic, September 2003, IEEE Computer Society.
16 **V. Samar and R. Schemers**, *Request For Comments 86.0: Unified login with pluggable authentication modules (PAM)*, Open Software Foundation, October 1995.
17 **David Corcoran and Tommaso Cucinotta**, *MUSCLE Card API, version 1.3.0*, August 2001.
18 **Tommaso Cucinotta, Marco Di Natale and Riccardo Brigo**, Hybrid fingerprint matching on programmable smart cards, *Proc. of the 1st International Conference on Trust and Privacy for Digital Business (TRUSTBUS 2004)*, Zaragoza, Spain, September 2004, Springer LNCS 3184.
19 Java Card™ 2.1.1 Runtime Environment (JCRE) Specification, Sun Microsystems, Inc., May 2000.
20 **Ross Anderson and Markus Kuhn**, Low Cost Attacks on Tamper Resistant Devices, Security Protocols, *5th International Workshop*, pages 125–136, Sprinter, April 1997.
21 **Pieter H. Hartel**, Formalising Java safety – An overview, *Proc. of the Fourth Smart Card Research and Advanced Application Conference (CARDIS 2000)*, pages 115—134, 2000.
22 **Audun Josang**, *The difficulty of standardising Smart Card Security Evaluation*, September 1994.