Issues in authentication by means of smart card devices

Ph. D. Thesis

by

Tommaso Cucinotta
Scuola Superiore Sant'Anna, Pisa, Italy
cucinotta@sssup.it

July 14, 2004

Issues in authentication by means of smart card devices

Ph. D. Thesis

by

Tommaso Cucinotta
Scuola Superiore Sant'Anna, Pisa, Italy
cucinotta@sssup.it

July 14, 2004

Candidato

Dott. Tommaso Cucinotta _____

Relatori

Prof. Paolo Ancilotti _____

Prof. Marco Di Natale _____

Dott. Giuseppe Lipari _____

Dott. Paolo Bizzarri _____

Contents

Preface

Data security is currently one of the basic requirements in computer software design, and it is most commonly achieved in the software industry by means of cryptographic algorithms and protocols. 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. Such bit strings constitute the weakest point of the overall "security chain", where compromise of even a single cryptographic key related to an application may lead to the compromise of the entire data managed by that application.

Management of the cryptographic keys is thus a crucial point to be addressed in the design and development of a secure system/application, the most effective means for protecting them, today, being the adoption of smart card technology [49]. This thesis discusses issues in realizing secure solutions based on the adoption of smart card devices for the purpose of securely managing a user cryptographic keys, with a clear focus on the availability of the techology on open systems. Despite the growing number of applications, libraries and software components that are available on such platforms and are already used in the daily life by every user, and the relatively low cost of smart card devices today, such devices are struggling in being supported by applications and secure software components, especially on open systems. This happens essentially because integration of smart card technology implies dealing with a high number of reader and card devices different in nature and capabilities. This situation results in the presence on each computer of a certain number of cryptographic keys which, even

if protected by passphrases or similar means, are every day at risk of being compromised due to the great number of malicious software programs overpopulating the Internet, today.

The situation is unacceptable, especially when looking to one of the most interesting application of the technology today, namely law compliant digital signatures. More and more countries, today, embed into their legal framework rules and technical regulations which allow an electronic document digitally signed to have the same legal value as the corresponding paper document with a handwritten signature would have. This constitutes a great potential for the increase of the efficiency of public and private administrations, potential that is hindered in the first place by the availability of such technology on only those platforms which are of interest for the smart card market, i.e. the Microsoft operative systems. In order for such technology to have a great impact on the citizens, it is of paramount importance that they are given the possibility to digitally sign documents by substaining a very little cost, as it is the case for a handwritten signature, which required a sheet of paper and a pen.

The open smart card architecture which is being proposed in this thesis, in the author's opinion, will help both in simplifying the development cycle of a smart card driver, and in easying engagement of this technology by applications with basic security requirements, especially on open platforms, so to increase the chances of availability and finally of use of smart card technology on these systems.

Background

This chapter introduces common issues in smart card interoperability, discussing how such issues have been faced with by standard organization in the past, and giving a hint on why standard documents issued by those organizations have not completely succeeded in solving the problem. Also, the JavaCard standard by SUN is introduced, which has constituted one of the latest step towards interoperability among different-vendor devices, and which has been largely adopted by card manufacturers, thanks to the platform independence inherited by the design of the Java language.

Introduction

Different protocols have been developed to allow interoperability among smartcard devices within various application contexts [27, 30, 31, 22, 19, 20]. Although they provide a common set of commands for exposing smartcard services, the goal of a common, multi-application, device-independent and application-independent card protocol was not achieved. It is not possible to write a host-side software that accesses basic storage and cryptographic facilities of most widely used crypto cards only relying on the standardized commands. The protocol which is being introduced in Chapter has been designed around most widely needed card services among commonly used applications. Its voluntary limited complexity and completeness at the same time make it a suitable minimal protocol for using programmable crypto cards in a generic, multi-application, and secure fashion.

The smartcard world has some kind of intrinsic complexity due to the existence of varying kinds of both card readers and card devices. The ISO standards establishing physical and electrical characteristics and those defining the T=0 and T=1 low-level communication schemes have been widely accepted and adopted by the industry. No further documents standardizing smartcard cryptographic commands [30, 31, 22] have had the same success so far.

This has happened for two reasons: excess of complexity and deficiencies in completeness. To cite an example, these were common problems of the ISO standards for storage and cryptographic services [27, 30]. These documents defined commands to access files and keys on a card, ranging from simple commands like those for PIN management to sophisticated hierarchical filesystem browsing and certificate verification, but there was no specification of how to create such resources on the card. This pushed card manufacturers to embed custom, proprietary commands in their devices for the purpose of creating resources. As a result, a standard compliant software can use the resources already on the card, but has no standard way of creating them. Usually only a card manufacturer provides the software needed to initially "personalize" the card and "manage" it afterwards.

Furthermore, smartcard manufacturers have seen interoperability as an enemy, because of their will to tie their customers as long as possible. So card vendors also embed their own features, only accessible with their own custom commands (or a variation on the standard commands), only implemented in their own software. This is done to give some kind of added value to their solutions with respect to the concurrent ones. This situation discourages wide adoption of smartcard technology, because developers do not have a direct way to write an interoperable, card independent, smartcard plugin for their crypto-aware programs and they don't want to cope with proprietary extensions of each and every card and token manufacturer.

Application developers today can use standard interfaces like PKCS-11 [45] for accessing crypto services in a generic way. Installation of the proper implementation of such interfaces provided by a card manufacturer auto-

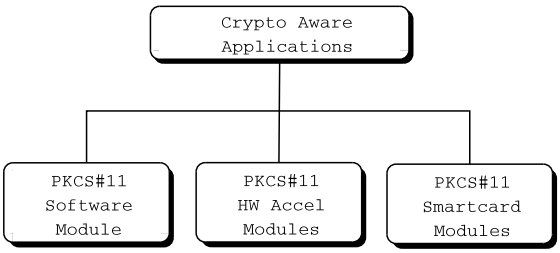

Figure 1: The common API approach for embedding cryptography into applications.

matically enables a card to be used by those applications (see Figure 1). Unfortunately, most of such implementations are commercial products and are only available on Windows platforms. Public specifications for smartcard operating systems or cryptographic applets are rare. Even with some open specifications it is common for the manufacturer to require you to license some piece of software or documentation in order to make use of the device. This behavior disrupts and discourages a wide adoption of these devices. It also breeds poor public perception of these devices in a world where open standards such as USB and others exist.

After all the advances in technology there still does not exist an interoperable, pluggable, portable, and freely useable smartcard solution that achieves the basic goals of the smartcard existence: user authentication and digital signature. We believe that the basic design around the MuscleCard Card Edge protocol, together with the open source, freely available, portable host-side middleware [15] that uses it, will widen smartcard technology adoption in applications that handle sensitive data. It must be noted that the proposed solution is not actually suitable for all possible applications of smartcards. Prepaid cards, electronic cash and mobile phone are all examples of applications that could require additional specific [20] operations to be performed on the card side.

The need for smart cards

Smart card technology allows to enclose into the physical boundaries of an Integrated Circuit Card (ICC) one or more cryptographic keys, which are directly used by the on-card circuits, logics, operating systems and applications. Furthermore, smart cards are built in such a way to be physically tamper proof, i.e. it is very difficult and expensive to try to recover the on-board memory contents by physical inspection (at what degree such property is applicable depends on a device-by-device basis). The on-board logics is still enough simple so to allow to have a real control on what software runs onto a smart card device, and the on-board software may be done functionally correct, what is practically impossible within traditional computer systems, where the much higher software complexity led to a situation in which evey PC has flawn which may be used by viruses, computer pirates and hackers to get remotely total control of the machine. Antivirus and intrution detection systems are, of course, just a medicine which is useful after the intrution/break-in has occurred, rarely they manage to prevent a new attack. Last, but not least, a smart card owner has physical protection on the device, which resides almost always into his/her pocket. This greatly reduces the chances for such devices to fall into unauthorized hands. This is not the case for traditional computer systems, unless physical surveillance procedures are undertaken which usually lead to unsustainable costs.

The problem of smart card interoperability

In spite of the growing need for smart card integration into applications and the great benefit these devices incorporate, smart card devices are struggling for a wider use in network security applications. This is mainly due to the complexity inherent to their integration within software applications, which exists for several reasons.

- There exist different types of card devices, as it will be underlined in Section : storage only, crypto-enabled, with general purpose CPU,

programmable in Java, Assembler, Basic, etc...

- Various card devices talk to the outside world using different protocols. In spite of the effort made by standard organizations [27, 29, 30, 31, 22], still card devices have many restrictions, uncommon filesystem approaches, and different cards have typically different ways to accomplish the same function, i.e. a file creation.

- Many smart cards have closed protocols and functionalities, what makes their use within open solutions impossible.

- Some of the existing standards for card interoperability addressed from the beginning only the usage issue, leaving the personalization of a card to proprietary extensions. As an example, the ISO 7816-4 [27] standard described the commands to browse an on-card filesystem, saying nothing about its creation and management. Only later, the ISO 7816-9 [31] standard fixed the hole, when tenths of card devices were already on the market with proprietary protocol extensions. Furthermore, these standards described a too complex protocol to be implemented on programmable devices commonly available today.

- The card life cycle is short. By the time a card is supported on lower priority systems like Unix-like ones, it ceases to be manufactured.

To further the argument for on-card interoperability let's look at today's trend in computer hardware. In nearly every hardware sector, manufacturers are unifying devices' interfaces by using open standards, like PCI or USB. Keyboards, scanners, mice, etc... no longer exist in a proprietary fashion, but conform to the USB HID class specification, thus requiring only one driver for all the devices which meet the specification. The smart card world still ties to a "proprietary" approach in which every manufacturer deliberately deviates from standards in order to give to their products some sort of added value and to link their customers to the company as long as possible. Application developers are forced to link their applications to a specific card

in order to use smart card services. If that card is no longer manufactured in six months, they can just hope the manufacturer has a new card which inter operates with the old one at the software level. The other alternative is to go to another card manufacturer and replace the entire software stack.

This situation discourages smart card integration and has the consequence of a reduction in the overall smart card usage hindering their evolution in security software and frameworks.

The world of smart card devices

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 solutions for PCs, 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 any need of inserting the card into a slot.

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 a few on-card logics is used to perform basic operations on the stored information such as data retrieval or decrease of some on-card counters. A typical application is constituted by a pre-paid card, where the information stored onto the card corresponds to the amount of money a user paid for accessing some service, such as telephone calls or access to the services of an Internet Point, and the user is required to insert the card into a terminal as long as he uses the service. The only operations the terminal can request to the card are typically retrieval of the residual pre-paid service time and its decrease of a prefixed amount. Usually, such cards have no way to authenticate users before the card use, or in some cases a PIN code verification is requested before the on-card data can be read.

A *cryptography-enabled* smart card, instead, is able to perform sophisti-

cated 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 to the system possession of it during a *challenge-response* cryptographic protocol that is run between the target system and the card device. The cryptographic key is securely stored onto the tamperproof smart card device and is directly used by the card itself. It is never revealed 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, where the signature operation is computed on the card device itself. In the digital signature application, sometimes the card is also requested to perform other operations such as data digesting and padding, for the purpose of increasing security of the system. Sometimes cryptographic smart cards may also have a means to authenticate the host system in a cryptographic way, so to prevent unauthorized use of the device. In such cases, the card runs a cryptographic challenge-response protocol with the host PC, where this time the host PC has to prove to the card device possession of its own authentication cryptographic key.

GSM-enabled smart cards are a special kind of cryptographic smart cards which are widely used in the Global System Mobile (GSM) telephony world, as Subscriber Identity Modules (SIMs) which authenticate users to the mobile telephony provider for the purpose of accounting the telephone calls made by a mobile phone. Such cards expose the required cryptographic capabilities by means of the GSM standard protocol [20].

A *programmable* smart card is usually a cryptographic smart card with a general-purpose CPU on-board, so that a program can be dynamically loaded onto the device for the purpose of implementing a custom application through the implementation of a custom protocol for interacting with the host PC. This kind of devices offer the highest flexibility to applications, which may be designed so to delegate the most security-critical operations to be performed onto the protected on-card environment. These devices are usually programmed by using a subset of a well-known programming language, such as Java, Assembler or Basic.

Smart cards and open systems

Even if smart cards have been widely adopted and supported on proprietary platforms, they are not being used at all on open platforms. On these systems many Open Source libraries and software applications exist embedding cryptography for protecting their data, but the achieved security level is strongly limited because of the use of software-only cryptography. Only a few solutions exist supporting just one or a poor set of smart card devices.

Furthermore, on open platforms a strong demand by the developers' community exists for the use of unrestricted libraries and applications. Open source software and open solutions are probably the right match for this demand, where smart card solutions manufacturer do not usually provide open source components, nor they make public any of the protocols used in their products. This makes integration of such devices quite difficult, as witnessed by the fact that, although many Open Source programs exist which embed cryptographic services, most of them still lack the support for external cryptographic smart cards.

The Java Card standard

One step towards interoperability of smartcards has been done with the issue of the Java Card$^{(TM)}$ standards [50, 51, 52]. These documents define a standard API for a Java Applet which runs onto a smartcard to access crypto services onboard. Because of the intrinsic complexity of the Java language, a standard subset of it and of the Java Runtime Environment has been defined for implementation onto smartcards. This way it is possible to write a program that runs on any compliant smartcard, implementing a custom protocol for communicating with the host. Fortunately this standard is being adopted by card manufacturers.

A classic way of handling Java Cards is by designing a specific protocol and writing a Java Applet implementing it for the particular application that the card has to cope with. The resulting solution is not suitable for other applications or contexts. Instead the protocol presented in Chapter has

been designed to allow use of the card by the most widely used applications, remaining as generic and application independent as possible.

The proposed approach

The open approach which is being introduced in the next chapters constitutes a step toward openness in smart card middleware design and implementation. In the author's opinion, the protocol introduced in Chapter , along with the open middleware components and prototype applications discussed in Chapter and Chapter which have been developed on the host-side, will promote adoption of these devices on open platforms, increasing the chance for their worldwide adoption in the daily life within computer systems.

Smart card middleware

This chapter aims to overview basic concepts about smart card architectures and protocols. First, Section depicts what is the typical architecture of a smart card middleware, and introduces basic concepts around standard protocols and Application Programming Interfaces (APIs) widely used in the world of smart cards. Then, for the sake of completeness, Section briefly surveys such standards, focusing specifically on the International Standard Organization (ISO) 7816-x set of standard protocols, on the RSA Labs' PKCS#11 API and on the PCSC smart card standards by the PCSC Workgroup.

Background on smart card middleware

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.

In spite of the effort made by standard organisations [28, 22], card devices have many restrictions and non-standard filesystem structures. Different cards have typically different ways to accomplish the same function, i.e. a file creation, thus interoperability at the software level is usually achieved through common, high-level, application programming interfaces that sup-

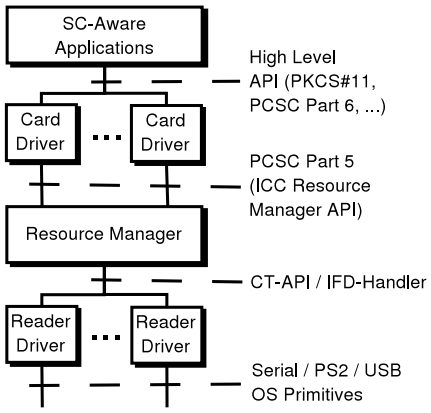

Figure 2: Architecture of a generic smart card middleware

port on-card operations in a manner that is independent of the card and reader devices.

Two APIs that have been defined for this purpose are PKCS#11 [45] by RSA Labs, and PCSC [37], 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 2.

At the bottom layers, a resource manager component is required for managing the smart card 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 connected to the system. This is done through the PCSC ICC Resource Manager interface [37, 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 exchanges with a card. The reader driver takes care of translating the

requests into the low-level Protocol Data Units (PDUs) to be transmitted to the reader through the low level serial OS primitives. Reader drivers implement the CT-API or the PCSC IFD-Handler API [37, 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 APDUs compliant to the ISO T=0 or T=1 protocols [26] (see Section for details).

The top level component of the middleware is traditionally a monolithic component, provided by card vendors, that implements the PKCS#11 or PCSC Part 6 interfaces. These APIs 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 range of similar card devices provided by the same vendor. Furthermore, it must comply with the higher level API, what requires additional tasks to be performed in the component, such as session management and transaction handling. Such tasks are quite similar in the driver implementations provided by different vendors, where the only changes regard the specific way information is exchanged with the card by means of APDU exchanges. This is why an investigation has been made on the possibility of introducing a further abstraction layer, breaking the traditional driver architecture through the use of a middle-level API.

In fact, in the architecture that is being introduced in Chapter , functionalities are grouped into separate components: a lower level (LL) driver, which formats and exchanges command APDUs with the card device, and 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 two functionalities just cited are split. As it will be shown in Chapter , the main benefit of such an approach is that it is possible to write the high level API specific man-

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

Standard protocols and APIs

The T=0 and T=1 protocols

The T=0 and T=1 protocols define, respectively, an asynchronous half duplex character oriented transmission protocol and a block oriented one for exchanging data between an interface device (i.e. a smart 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 3). Only an overview of how the T=0 protocol works is provided, in order to allow a better understanding of the architecture that is being introduced. The complete specifications can be found in [26]. The header is composed 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 data that eventually follows, are used to feed input parameters to the command. The `P3` 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 `0x9000`) or not.

Note that, usually, for each command-response APDUs only one device, either the card or the host, are allowed to send data, not both. Basically, four modes of operation are allowed:

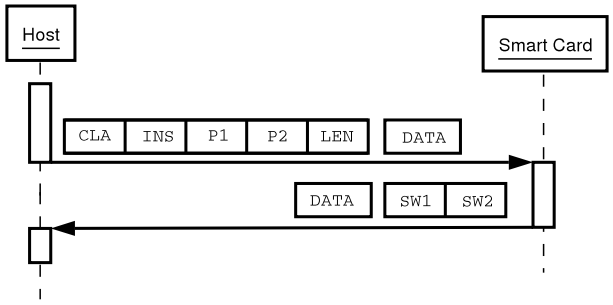

Figure 3: Generic invocation of a smart card command by exchanging ISO APDUs.

- the host sends no data, the card responds with no data, i.e. only with a satus word indicating weather the operation was successful or not
- 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 of exchanges may happen within a single command-response APDU pair. However, whenever the card device needs to provide some data as a response to a command APDU containing also some data, it responds with a special status word (`0x61XX`) containing, in the second byte, the length of the data to be transmitted to the host. The host is then 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. Though, not all card devices comply with such extensions.

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. Main goals of the API design have been, among others, independence from the specifics of a security device and resource sharing, so to allow multiple applications to share access to a single device, as well as to access multiple devices, presenting to applications a common, logical view of the device called a *cryptographic token*.

Cryptoki provides an interface to one or more cryptographic devices that are active in the system through a number of *slots*. Each slot, which corresponds to a physical reader or other device interface, may contain a *token*. Typically, a token is "present in the slot" when a cryptographic device is present in the reader or interface device.

The kinds of devices and capabilities supported will depend on the particular Cryptoki library and supported devices. The standard specifies only 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 whether an object is private or public, and there are also attributes that 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 secret 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 tied 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 and destroy objects, manipulate them, and search for them. 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 and for setting the normal user's PIN and possibly to manipulate 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 the latter platforms, it is rare that a vendor provides a PKCS#11 module for one or more of their smart-card devices.

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 other ICC-relevant resources within the system and for supporting controlled access to Interface Devices (IFDs) and, through them, individual Integrated Cicruit Cards (ICCs). The ICC Resource Manager solves three basic problems in managing 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 IFD resources across multiple applications. It does this by providing mechanisms for attaching to specific IFDs in shared or exclusive modes of operations. Finally, it supports transaction primitives on 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 functional interface exposed by the ICC Resource Manager is described, in an object-oriented fashion, in terms of object *classes* and *methods* on object instances of those classes, 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 communicating with the ICC Resource Manager. The class `ScardTrack` encapsulates functionality that supports determination of 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. It provides methods for managing the connections, controlling transactions, sending and receiving commands, and determining card state information. 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[1], it is available on many open Unix-like platforms too, such as Linux, OpenBSD and Mac OS-X.

The PCSC Standard, Part 6

The part 6 of the PC/SC Workgroup's architecture defines the Service Provider (SP) component, composed 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 exposure is expected to include common interfaces, defined in the specification, for managing connections to a specific ICC, as well as access to file and authentication services. In addition, the ICCSP may implement interfaces that the vendor defines for features specific to the application domain. The ICCSP interface provides mechanisms for connecting and disconnecting to an ICC. In addition, the ICCSP exposes file access and authentication services which encapsulate 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.

The CSP, in contrast to the ICCSP, isolates cryptographic services because existing regulations imposed by various governments affect import and export. The CSP allows applications to make use of cryptographic services in a manner that compartmentalizes the sensitive elements of cryptographic support into a well-defined and independently installable software package.

[1] Movement for the Use of Smart Cards in a Linux Environment, http://www.musclecard.com

The CSP encapsulates access to cryptographic functionality provided by a specific ICC through high level programming interfaces. Its purpose is to expose general-purpose cryptographic services to applications running on a PC, like key generation, key management, digital signatures, message digesting, bulk encryption services, and key import and export.

Relevant classes defined in this part of the standard include:

- the `FileAccess` class, which defines a high level interface to a ISO 7816-4 based on-card 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) or allow the user to change a CHV code;
- 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, instead, the few vendors who provide a high level API, usually provide a PKCS#11 module.

A protocol for programmable smart cards

This chapter presents an open protocol for interoperability across multi-vendor programmable smart cards, which allows programmable card devices to expose storage and cryptographic services to host applications in a unified, card-independent way. Its design, inspired by the standardization of on-card Java language and cryptographic API, has been kept as generic and modular as possible, allowing to embed future extensions for additional cryptographic primitives and user identification schemes. As a proof of concept, a biometric extension has been recently developed allowing on-card fingerprint verification for the purpose of authenticating the owner before use. The protocol security model has been designed in order to allow multiple applications to use the services exposed by a same card, with either a cooperative or a no-interference approach, depending on application requirements.

With respect to other existing protocols for smart card interoperability, requiring the cards to implement sophisticated services, the presented protocol exposes a reduced set of functionalities, achieving a simpler management of the on-card resources, resulting in a size contained code when implemented on programmable cards. The reduced functionalities suffice to most smart card enabled applications that use card devices for authentication and digital signature purposes, comprising PKI based applications,

constituting a better solution to be implemented with programmable card devices due to the on-board resource constraints.

An open source card-side implementation of the protocol has been developed as an Applet for Java Card 2.1.1 compliant cards. Also an open source, pluggable, host-side middleware has been developed, that is portable on a multitude of open source and UniX-like platforms (see Chapter). The middleware includes a full protocol implementation and exports a new smart card Application Programming Interface to the upper smart card middleware layers or applications. Even if the API original design aimed at a one to one mapping with the protocol commands, the resulting interface is enough generic to be implemented on any card supporting storage and cryptographic APDUs. The resulting smart card middleware allows for dynamic loading of the correct smart card plug-in, based on the card Answer To Reset. The host-side implementation of the cited protocol is currently just one of the available plugins, supporting any Java Card compliant card, once loaded with the Applet. Other plug-ins have also been developed for some non programmable smart cards.

Motivations

Today many smart card aware applications exist on closed platforms that use smart cards for the only purpose of securely storing and managing user cryptographic keys and a few related data. Some examples are PKI[2] based applications, like digital signature programs, secure on-line web services, secure e-mail. These applications face with the interoperability problem by adopting common interfaces at a software level, like the PKCS#11 [45] or PCSC [37] ones. Unfortunately, the modules implementing these interfaces are usually provided by card vendors only for those platforms that are considered of commercial interest. Rarely they provide an implementation for open platforms. This situation discourages smart card integration and has

[2]Public Key Infrastructures.

the consequence of a reduction in the overall smart card usage, hindering their evolution in security software and frameworks.

On the other hand, programmable smart cards have always been used in the context of secure solutions with different and application specific tasks to be performed by the external device. Usually a custom program is loaded on the card implementing a custom protocol to exchange data with the specific application. A common example is a prepaid card, where the on-board program is used to manage an electronic wallet.

In this chapter an hybrid approach is introduced, where a program is loaded onto a programmable card allowing exposure of cryptographic and storage services to generic applications by means of an open protocol. The advantage of this approach is that it is possible to both use the generic services provided by the program, and to implement custom commands in order to satisfy specific application requirements. Existing standard protocols are too much complex to be implemented on such devices, where the on-card program must be of contained size in order to leave enough space on the card for cryptographic keys, user data and other extensions.

In the author's opinion, the introduction of an open, well-designed, unrestricted protocol for programmable devices, along with an open source card-side implementation and an open host-side smart card middleware using it, will lead to a wider use of these devices.

Protocol overview

This section features a technical overview of the protocol, underlining how the project goals have been accomplished. The discussion only addresses protocol's main features, and explains main design choices. The complete protocol specification [16] is available for download at the URL: *http://www.musclecard.com*.

Objectives and design choices

ISO standards impose a high level of complexity on data storage and cryptographic services exposed by a smart card, constituting a powerful and flexible solution that is suitable for a wide range of applications. Though, an implementation of these standards onto a programmable device existing today, like a 32K Java Card, would result in a so big program that a few on-board memory, if any, would be left for application data and cryptographic keys.

The new protocol introduced in this thesis, instead, has been designed with the aim of satisfying the requirements that a smart card stands for: protecting user keys.

The project has been focused from inception on the release of an open, simple, card independent, complete and freely available card protocol that allows a host application to talk to any programmable smart card, in order to access cryptographic and storage facilities on the card. The main goal in protocol design was retaining enough generality to catch and satisfy requirements of a multitude of target applications, comprising digital signature, secure e-mail, secure login, secure remote terminal and secure on-line web services, both PKI based and not. These requirements have been identified in having a means of generating, importing, exporting, and using cryptographic keys on the card. Also required is having a means of creating, reading, and writing generic data on the card in separate "containers". This is useful, for example, to store a public key certificate associated with a private key on the card. The access to some of these resources needs to be granted only after host application and user authentication. Another requirement is the independence of the managed data chunks from the lower level T=0 APDU size limitations, so to preserve the ability to handle large keys and data chunks that will be needed in a near future. The fundamental constraint on the protocol design was due to the limited card memory of todays' programmable devices (ranging from 16 to 64 KBytes), resulting in the requirement of a size-contained code for the on-board protocol implementation. This resulted in serious constraints on the protocol complexity,

that needed to be as simple as possible.

The result has been a simple and light protocol that is more suitable than already existing ones to be implemented on programmable card devices, given the limited amount of available memory and computational resources. As a remark, the developed Applet, implementing the entire protocol, has a code size of around 10 KBytes. On a Schlumberger Cyberflex Access 32K card, this leaves enough free space on the card for keys, certificates, additional application data, and further Applets to be loaded on the card for additional services to be used in a joint or alternative fashion.

The protocol design explicitly addresses initialization issues, such as how data or key objects are created on the card, and what authorizations are needed for these operations to succeed. The protocol does not address sophisticated card services that can be required by some specific applications. For example applications for "digital money" or "pre-paid cards" can require special operations to be performed on stored data. Multi-key digital signatures and authentication schemes can require specific cryptographic protocols to be performed on multiple cards. These applications can still benefit of the exposed protocol and open implementation, by extending them with the required functionalities.

Protocol command set

With respect to the T=0 and T=1 protocols (see Section), standing at the *transport* layer according to the terminology defined in [21], section 7, our protocol stands above, at the *application* layer, identifying a set of commands that a smart card program should support. The protocol specification exactly defines what class, instruction, parameter and data bytes must be provided by the host for each command, and what data is expected in response, if any, from the card, along with the possible error codes that identify abnormal conditions during command execution.

A general overview of the commands available in our protocol specification is reported in table 1, while specific details about various commands are

Data Storage	CreateObject, DeleteObject WriteObject, ReadObject ListObjects
Cryptographic Key Management	GenerateKeyPair, ComputeCrypt ImportKey, ExportKey ListKeys
PIN Management	CreatePIN, ChangePIN UnblockPIN, ListPINs
Security Status Management	VerifyPIN, ISOVerify GetChallenge, ExtAuthenticate GetStatus, LogOutAll

Table 1: MUSCLE Card Protocol command set.

reported in the following sections.

Data storage services

The protocol encapsulates applications' data into simple containers, called *objects*, identified by means of a 32 bit object identifier (OID). Access control is enforced on a per-object and per-operation basis, distinguishing among create, read, write and delete operations. More details on this are given in section . The defined data storage service suffices to the target applications cited above, by allowing them to store, retrieve and manage data onto a card in a secure and controlled way. This is a minimum set of operations and access constraints that is needed from host applications and suffices to securely store, retrieve and manage data onto a smart card. This does not preclude a hierarchic organization of applications' data into a filesystem-like fashion. That could be achieved on the client side on a per-application basis, or in an inter-application fashion if further documents came up standardizing particular objects to be used for filesystem information. This way consistency of the achieved hierarchy would be up to the host-side applications and could not be enforced by the card. Still this approach would be suitable for environments where the hierarchy is created statically and needs not to

be changed dynamically.

The protocol does not provide hierarchic arrangement of objects, nor typed objects, conversely to other approaches [22] in which both special file types and special file contents have been standardized for a particular application context. However, a range of object identifiers has been reserved for future use and cannot be used by applications. This could be used in the future to support extended features, like file or certificate directories, that could be managed by the card with a set of extension commands. The protocol does not define specific object contents, leaving to applications total freedom on what to store onto a card: user private information, application specific data, public key certificates, etc.... This is highly dependent on the application itself and cannot be established on a document like our protocol specification, that instead still leaves space for other documents to come up standardizing object identifiers to be used to store special information with an inter-application relevance.

The protocol specification does not address issues like how objects should be created and managed on the card, how many objects are allowed to exist due to management constraints (i.e. allocation tables), how free object memory is to be handled by applications (i.e. by use of compaction or full defragmentation of free blocks). Applications have only a view of the total available memory, and whether an object of that size can be really created or not depends on the specific on-board memory management that is performed on the card.

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 sometimes 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 a PIN verification. The approach introduced in this chapter is completely different in that we have a unique memory space, where single created objects are associated with access control rules specifically customized for each object and operation. So our model allows reconfiguration of the memory space as public or private according to application

requirements. The section will explore in more detail the adopted security model.

Cryptographic services

The protocol allows up to 16 keys to be stored and managed on the card, identified by means of a numeric key identifier. A full key pair can also be stored using two key identifiers. Key types are those provided by the Java Card 2.1.1 API: RSA, DSA, DES, Triple DES, Triple DES with 3 keys. The protocol is designed in such a way to allow further key types to be easily added in the future.

Operations provided on cryptographic keys are import/export from/to the host, calculation of cryptograms, and listing of keys, that provides size and type information. All key operations but key listing can be allowed only after proper host application/user authentication. The protocol allows asymmetric key pairs to be directly generated on board guaranteeing the private key can never be exposed outside of the card. In this case the public key can be obtained by the host application with an *ExportKey* operation to be performed after the key pair generation (see figure 22). When a key pair is created on-board, the host application specifies under what conditions subsequent reading, overwriting and use operations are allowed for each of the keys in the pair. The same rules can be specified when importing a new key from the outside world by means of an *ImportKey* command. Further details on access control and security model enforced by the protocol will follow in paragraph .

Input and output objects

Objects have also been used to overcome the T=0 protocol's limitation of 256 bytes per APDU exchange[3]. This is due to the fact that the LEN field in the APDU header is a single byte and indicates either the length of the data

[3]Extended data length fields and the Envelope command, as defined in [27], are not implemented on all smart cards

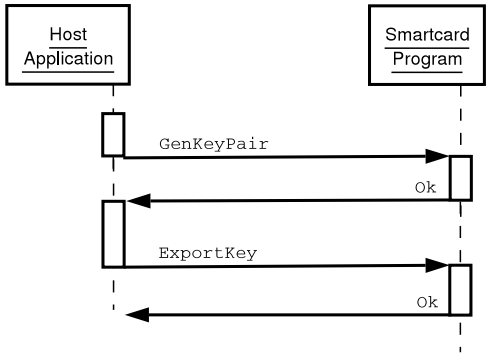

Figure 4: Sequence of commands for an on-board key pair generation with the subsequent export of the generated public key value, to be read after the ExportKey command from the output object.

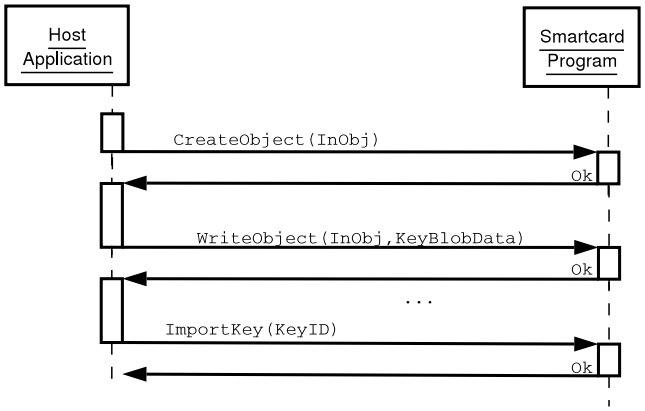

Figure 5: Use of the input object during a key import operation.

to be transmitted to the card with the command APDU or the maximum length of the data to be received from the card with the response APDU, limiting such data to have a size between 0 and 256 bytes. The protocol allows commands that need to transfer more than 256 bytes to or from a card, like key exchange. When dealing with reading or writing an object contents, the problem has been solved by introducing an *offset* field as a parameter to the *ReadObject* and *WriteObject* commands, so that reading or writing of a long data chunk can be performed by invoking multiple times the commands with increasing offset values. When dealing with exchange of cryptographic keys or cryptograms, instead, this limit was overcome by reserving two object identifiers for an *input object* and an *output object*. These are used for providing and retrieving long data to and from other commands. For example, in order to import a key into the card, the key data must be provided into the import object, then an *ImportKey* command simply reads the key data from that object (see figure 5). Similarly, to export a key, the *ExportKey* command calculates the key data and leaves it into the export object to be retrieved in subsequent commands by the host application (see figure 6). In the latter case it's also up to the application to delete the output object after retrieval of contained information. A few commands have been specified in such a way that it is possible to provide the data parameter bytes either within a single APDU or by using the I/O objects, depending on the length of exchanged data. In these cases a command option field, called *data location* byte, is provided by the application to specify if the variable size part of the command data bytes is provided "inline" in the command APDU, or if it must be read from the input object. For example, this paradigm has been used in the ComputeCrypt and ExtAuthenticate commands for a "quick" exchange of cryptograms, usually shorter than 256 bytes due to the current size of on-card cryptographic keys. In fact a 64 bit DES key requires computation of 8 data bytes at a time, a 128 bit 3-DES key requires computation of 16 data bytes at a time, a 1024 bit RSA key requires computation of 128 data bytes at a time. By setting the data location properly, larger cryptograms can still be managed by the protocol

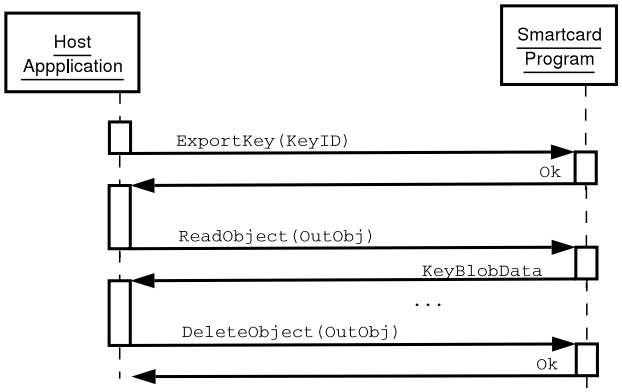

Figure 6: Use of the output object during a key export operation.

with key sizes that a smart card will have to deal with within a near future.

I/O objects can contain sensitive information like key values or application plain text data, so special attention must be paid to their management. In fact operations using the I/O objects must be split in 3 or more protocol commands, where only the last one deletes the involved I/O object. If execution of the command sequence is interrupted for any reason, this object would not be deleted, retaining its contents. These problems have been avoided by requesting that the I/O objects should be deleted as soon as possible, and at the card reset. For example, each operation that uses the input object must delete it before returning. When granting security of operations involving the export object, instead, it's up to the host application to read the contained data as soon as possible, and to finally delete the object from the card. In both cases security of the composite operation is granted both by the operative system resource manager that does not allow other applications to interfere with the current multi-command operation, and by the object deletion at card reset that avoids attacks relying on a sudden extraction of the device by the user before the object has been deleted. In order for these mechanisms to achieve the desired security level, the application must acquire access to the smart card reader in an exclusive mode before starting any composite operation. This is also required before

issuing any authentication command to the card, like a *VerifyPIN* or an *ExtAuthenticate* command, in order to avoid that other applications access protected on-card resources.

Security model and access control enforcement

A simple Access Control List (ACL) is defined, allowing operations to be performed only after proper host application and user authentication. This may be performed by means of a PIN code verification, a *challenge-response* cryptographic protocol, or a combination of both of these methods. Furthermore the protocol has been designed to allow future support for other identification schemes like fingerprint verification or generic biometric verification. As a proof of concept, a prototype implementation has been recently developed for on-board fingerprint verification. Even though additional commands have been added to the protocol for biometric template management, the new authentication mechanism fits well into the protocol, allowing, in example, the restriction of a key use or an object reading only after a successful fingerprint verification.

Access rules for on-card resources are specified in terms of the authentication needed to access each operation on each key or object. This has been achieved by defining the concept of *identity*. This term refers to one of several authentication mechanisms that host applications and users can use to be authenticated to a smart card. Identities, PINs, and cryptographic keys are referred to by means of numeric identifiers. Different types of identity are defined: (see also table 5) identities n.0-7 are said *PIN-based* and are associated, respectively, with PIN codes n.0-7; identities n.8-13 are said *strong* and are associated, respectively, with cryptographic keys n.0-5 for the purpose of running challenge-response cryptographic authentication protocols; identities n.14-15 are *reserved*, their behavior is not defined by the actual version of the protocol and is reserved for other authentication schemes to be incorporated in the future[4].

[4] The fingerprint verification mechanism recently developed uses identity n.14.

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

Table 2: Association between identity, PIN and cryptographic key numbers.

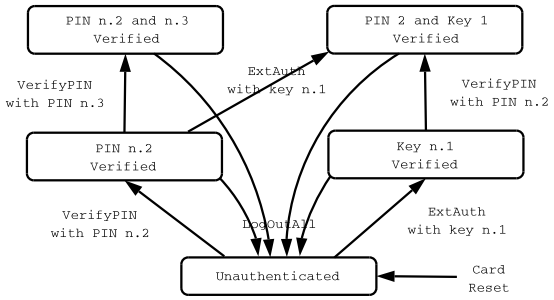

Figure 7: Subset of possible security state transitions allowed by the protocol.

A successful run of one of the authentication mechanisms causes the *log in* of the associated identity, in addition to identities already logged in. This way a host application can gradually switch to a higher security level that grants access to more and more of the card's capabilities, as it runs additional authentication mechanisms. Furthermore the *LogOutAll* command allows a host application to return back to the unauthenticated security status. A little subset of possible security states and transitions due to successful authentication commands is shown in figure 23.

Logged identities control which operations are allowed on an object or

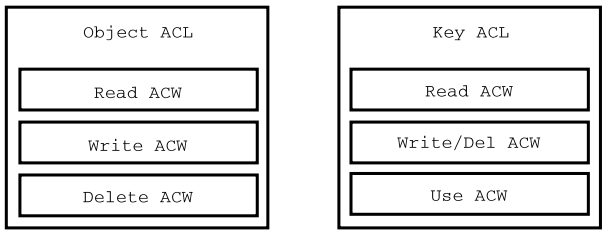

Figure 8: Composition of the Access Control List for objects and keys.

on a key by means of an ACL specifying which identities are required to be logged in to grant access to each operation of each object or 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 relating to an operation (see figure 24). An ACW has each bit corresponding to one of the 16 total identities that can be logged in. An all-zero ACW means that the operation is publicly available, that is a host application can perform it without any 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 means that the operation is disabled and cannot be performed, independently of the connection security status. This is useful to disable reading of private keys, for example. Table summarizes the situation.

The discussed 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 usually disabled. Public objects may always be readable, but their modification could be PIN protected. Private objects could require PIN pro-

Binary ACW	Meaning
0000 0000 0000 0000	Operation always allowed
0000 0000 0000 0010	Operation allowed only if PIN n.1 has been verified
0000 0000 0000 0101	Operation allowed only if both PIN n.0 and PIN n. 2 have been verified
0000 0001 0000 0000	Operation allowed only if a successful External Authenticate command has been performed using key n.0
0000 0001 0000 0010	Operation allowed only if both a successful External Authenticate command has been performed using key n.0 and PIN n.1 was successfully verified
1111 1111 1111 1111	Operation never allowed

Table 3: Some Access Control Word example values and their meaning.

tection for reading and possibly strong protection for writing. The model has enough flexibility to allow all of these access policies, and more, to be enforced.

PIN management

APDUs have been defined for PIN management, enabling to create, verify, change and unblock PINs. Several PIN codes are allowed to exist and be managed onto a single card. A special PIN (*transport PIN*, n.0) is assumed to already exist right after the program has been loaded and instantiated on a card, and it must be verified to allow a host application to create further resources on the card. This has been imposed to prevent allocation of card resources without the user knowledge. It is highly suggested that applications *format* a smart card by creating an application specific PIN code, and using it for protecting application specific data and keys. With a proper setting of the ACLs on the card, an application can "format" the card in such a way that further usage of the card is dependent on a different PIN than the transport one or on a different authentication scheme, like a challenge-

response cryptographic protocol. This way after the user has entered the PIN code from an untrusted terminal, 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 that terminal to interfere with other applications.

Multiple applications, one single card

The protocol has been designed to allow multiple applications to use the same card and, on that card, the same program instance, without interfering each other. While on a Java Card device this could be allowed in a simple way by creating multiple instances of the same Java Applet, such an approach would suffer of a static allocation of card resources. In fact the total memory to be reserved for an Applet instance must be specified at instantiation time. By allowing multiple applications to use the same Applet instance, we allow a dynamic allocation of card resources to single applications as needed[5]. The general idea is that each application must be able to have and manage its own PIN, data objects and keys. This has been accomplished in two ways: requiring verification of the *transport* PIN to allow creation of new PINs, objects and cryptographic keys, and allowing an application 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 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 only 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.

[5] A static pre-allocation of part of the object space can still be performed by an application by creating a "fake" object with the required size and properly resizing it when additional objects must be created.

Transactions and related issues

Different kind of error conditions can occur during a smart card operation. It is possible that the host provides an incorrect class code for the currently inserted card, an incorrect instruction code for the specified command class, or incorrect parameters to the specified command. Furthermore, a host can cause an access violation when trying to access a resource/operation on the card that is forbidden with privileges of the actual session. It can happen that the software component that is currently either providing data to or retrieving data from the card suddenly interrupts its operation (i.e. an application crash or a system shutdown). Finally, the card can be suddenly extracted by the user during a command execution.

The protocol specification explicitly deals with first four conditions, by specifying, for each command, what error codes must be returned by the device when some of these conditions occur. These can be regarded as "graceful" failures because they assume that both the host and the card are still operating correctly and can run the needed error recovery procedures to handle the condition. Last two error types are also very important with respect to a smart card life, because they raise transactions issues due to a sudden reset or power down of the device. In fact after a host-side application crash, usually, the device is again under control of the resource manager, that should issue a "reset" or "power down" command (see Figure 25) to the reader in order to guarantee security of data and keys that were handled by that application (if it was using the device in exclusive mode). Furthermore a card extraction always powers down the card. If the device was updating its internal data during the command execution, it is very important that this is done in a *transactional* way, so to guarantee consistency of data in such cases. The really important property that must be guaranteed is consistency of internal "directories" of objects and keys, including access control information. Data contained in objects does not need to be handled in a transactional way. Consider the case of a power-down during execution of a command that is overwriting an object contents. First of all, other objects

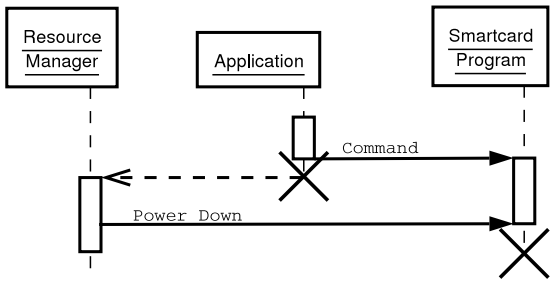

Figure 9: Sudden application crash causing a reset or power-down of the card during a command execution.

must not be affected anyway by the event. Second, the access control information related to this object must not be affected anyway by the event, otherwise it would be possible to set-up attacks based on this kind of events. A desirable property is also the transactional update of object contents, that is either the contents are updated or the old contents are recovered, but it is not really required, as it should be anyway considered at an application level.

The provided implementation of the protocol is a Java Card Applet running into the Java Card Runtime Environment [51], that already implements transactions on every updates to permanent card data during a single command execution, up to a maximum amount of changed bytes. This allowed to write the Applet without any additional code for implementing the transactional behavior.

When implementing the protocol onto a programmable, non Java Card, device, it is of fundamental importance that the program explicitly addresses transactional issues, guaranteeing consistency of at least internal key and object "directories", and access control data.

Extendibility

Our protocol does have limitations. These are due to the main purpose of its design: to allow new generation programmable cards to expose *basic*

cryptographic and data storage facilities to host applications in a way that does not depend on the specific card. So 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 software that has already been written and will eventually be written, the protocol 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 still 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 done 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 as a means for adding on-card biometric pattern matching without affecting the original protocol.

Implementation notes

The introduced protocol has been implemented and used with various applications. On the card-side, an open source Java Card Applet has been developed, fully compliant to the protocol specification, and tested both on Schlumberger Cyberflex Access 32K and Gemplus P11/PK cards. On the host-side, a new smart card middleware has been developed, exposing to upper layer software an open smart card API that almost maps one to one with the protocol itself. The API resulted to be enough generic to allow development of plug-ins for different types of cards, within the same middleware. On the top of this layer, an open source PKCS#11 module has been developed, allowing integration of all available applications supporting this

standard on open platforms. Mozilla and Netscape Communicator are example softwares now able to perform secure access to web sites (by means of the HTTPS protocol) and to sign e-mail messages using the exposed Applet and protocol. Furthermore, the new smart-card API has been used to directly integrate smart card technology into the OpenSSH software, an open source implementation of the Secure Shell protocol [58] for secure remote terminal. An open source Pluggable Authentication Module[47] (PAM) has also been developed, allowing smart card based secure login, and smart card based access to all applications using this mechanism. A command line application for digital signatures has also been developed directly with this new API. XCardII, a GUI based smart card manager, and MuscleTools, a command line one, have also been developed directly on the top of the new smart card API. All software components have been developed and tested on a variety of open platforms, including various Linux distributions and Mac OS/X, and are available for free download either from the Muscle Card web site (http://www.musclecard.com), or from the Smart Sign web site (http://smartsign.sourceforge.net).

As a proof of concept, the protocol extension mechanism has been used recently for providing a biometric extension to the Applet. This allows management of a new identity type, that *logs in* after a successful run of an on-card fingerprint verification algorithm. The extended Applet allows, for example, to use an on-board private key or to read an on-board object contents, only after the user has been authenticated by matching the fingerprint template provided by the host against the on-board stored one. A scheduled task is integration of other applications with this biometric extension.

Final software architecture is depicted in figure 10.

In this section we briefly expose some implementation issues around the MUSCLE Card Protocol, showing how they have been managed in our implementation.

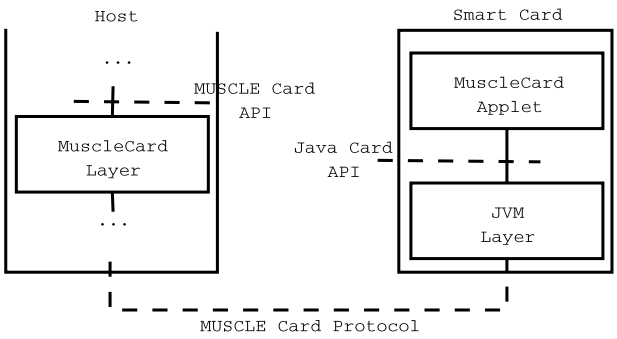

Figure 10: MUSCLE Card software architecture.

Memory Management

The protocol specification only gives to applications a view on the total available free memory on the card, leaving maximum freedom to developers to implement the most appropriate memory management scheme for specific cards. This means that applications and users must be aware of the fact that it could not be possible to allocate a single object of that size.

In our implementation we chose to pre-allocate a single Java byte-array at the Applet instantiation time. The array is used to store all of the object related information: OIDs, ACL, size, contents. Also a compaction of consecutive free memory blocks is performed on object deletion, but memory defragmentation has been retained to be too much complex to be performed and not really required. In fact memory fragmentation is typically the result of many block allocation and deallocation operations. Most smart card applications, instead, "format" the card during the issuance process that typically stores a user certificate on the card. In most cases such information does not need any further changes, or changes happen very rarely.

In the future we expect an increase on the available memory to smart card devices, as well as on the transfer speed with the hosts. This could result in a more dynamic management of data on the cards, and an on-board memory defragmentation operation could become a requirement. This

could be implemented either in a "transparent" and automatic fashion during execution of other commands (i.e. an object creation command that fails), or by means of an explicit invocation of a new protocol command to be added as a protocol extension.

Resource reclaim

Our Applet has been developed according to the Java Card 2.1.1 specifications. This implies that we do not have a means of actually freeing memory resources used by Java objects after they have been created. A drawback of this behavior of the Java Card Runtime Environment is that, once a cryptographic key has been instantiated into the Applet, even if the object reference was overwritten, really the associated memory would still be unusable garbage. For this reason we chose to recycle existing key objects as much as possible, allowing overwriting of existing keys with imported keys or re-generated keys of the same size and type. Our Applet does not allow to import or re-generate a key object with another one with a different size or type, actually. There is a similar problem with deletion of on-card PIN codes, once they have been created.

PIN policies

Our Applet implements some example on-board *PIN policies*. These are useful to force users to avoid weak PIN codes for protecting their keys, leading to a slightly higher security level of the protected data or services. Another application of PIN policies is within environments with, for example, numeric pin-pad terminals, in which the user must not be allowed to set up a literal PIN code from his home PC. This feature has been implemented in our Applet essentially as a proof of concept, and like other Applet features, can be entirely disabled from the code compilation process.

PIN policies are embedded in a transparent way with respect to the protocol, that does not currently contain any concept of "PIN policy". The Applet is able to perform various checks on a PIN value when the user tries

to change it by means of an ChangePIN command. The implemented checks include, among others, a check on the minimum length of the code, a check on the allowed character sets (i.e. numeric only, lowercase only, etc...) and a check on the mixture of character sets (i.e. at least 1 digit and a letter, etc...). When the code does not satisfy the policy, the command fails giving to the application an invalid parameter return value.

Selective disabling

Given the limited resources on the card, we the Applet sources have been given the ability to exclude some parts of the code in the final compiled bytecode, if not required by applications. This allows a smaller program size, leaving more space on the card for applications' data with respect to the full Applet. Features that can be selectively enabled or disabled include the availability of specific cryptographic key types (RSA, DSA, etc.) and modes of operation (encryption, signing/verification), the ability to import/export keys and the ability to perform external authentications. For example, disabling the DSA cryptographic algorithm, not implemented yet on most Java Card devices, leads to the elimination, from the compiled Applet, of one third of the key management code, that would remain unused on those devices that do not support it. A capability feature, that will be embedded in the protocol in a near future, will allow applications to know in advance what are the features supported by a particular smart card/Applet. At the moment, trying to use a disabled feature leads to an unsupported feature error.

Existing solutions

In this section some existing protocols for smart card interoperability are described. These protocols define protocol data units (APDUs) that are exchanged between a host and a smart card, relying on the T=0 or T=1 [26] lower level protocols, which have been described in Section . Proba-

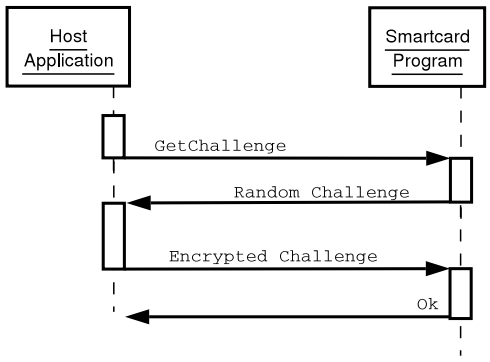

Figure 11: A challenge-response authentication protocol.

bly the most commonly implemented standard protocol for smart cards is the ISO 7816-4 [27]. In short, this document defines commands to browse an on-board filesystem, read/write data from/to files, allow reciprocal authentication of the card and external users and applications, manage multiple logical communication channels with a card, and perform authenticated and/or encrypted APDU exchanges (*secure messaging*). Different types of files are defined: *dedicated* files store directory information, while *elementary* files store application data. Elementary files are also distinguished in *transparent* if the content is merely a sequence of bytes, *linear* if the content is a sequence of records, with the possibility to have both fixed-size and variable-size records, and *cyclic* if the records are to be handled in a cyclic fashion.

Authentication of external users/applications can be performed by means of a PIN code verification, or cryptographic *challenge-response* protocol. In the first case the user or host application is required to prove knowledge of a PIN code, typically a short alphanumeric string, that is compared by the card with the on-board one. In the second case, a host application is required to prove knowledge of a cryptographic key, by using it to encrypt a random sequence of bytes, called *challenge*, generated by the card itself (see figure 11). The card decrypts the encrypted challenge using the on-board key, then compares it with the original generated challenge. A challenge-

response authentication protocol can be performed by using both symmetric and asymmetric cryptography. In the first case the host-key and the on-board key are the same, while in the second case they are respectively the private and the public key of a key pair. Also authentication of the card to external applications can be performed by means of challenge-response protocols.

The ISO 7816-4 standard addressed from inception only issues related to card use, while commands for creating the on-card filesystem, as well as the ones for loading, using and managing cryptographic keys on the card, were completely missing. Only later the ISO 7816-8 [30] and ISO 7816-9 [31] standards fixed the missing specifications, when tenths of card devices were already on the market with proprietary protocol extensions. The final protocol arising from ISO standards is very powerful and flexible. It has command APDUs for: calculation of cryptographic primitives and hash functions; calculation and verification of digital signatures; verification of on-board public key certificates; data encryption and decryption; and creation and management of *security environments* (SE) and *security associations* (SA), allowing the definition, for each card resource and operation, of complex access control rules (ACRs). It is possible to require multiple authentication mechanisms, with an *and* or *or* semantics, and, in the *expanded format*, ACRs can be combined into arbitrarily complex boolean expressions. Furthermore, an inheritance mechanism is defined that allows ACRs associated to directories to hold for all the contained elements.

On a related note, the PKCS#15 standard [44] defines, in the context of an ISO on-card filesystem, a file and directory format for storing security-related information on cryptographic tokens, like digital certificates, cryptographic keys, and authentication data (i.e. PIN codes).

The ISO 7816-7 standard [29] defines a set of command APDUs that allow a smart card to expose advanced data retrieval facilities to applications. This way an application can specify a SQL-like search query, and retrieve only those records that match the query. With incoming smart cards with more and more on-board memory, this is supposed to leverage the needed transfer

bandwidth between the card and the applications by performing searches on the card-side, and transferring to the host only the required data.

A further protocol for smart cards is the US Government SC Interoperability Specification [22], defining specific commands for an interoperable use of smart cards in the US Government context. In example, file formats are defined for the *general information* file, containing personal user information like name, surname, title, etc..., for the *protected personal information* file, containing Social Security Number, date of birth, etc..., and for the X.509 certificate files. The standard defines a set of ISO 7816-4 compliant commands for on-card filesystem, PIN verification and host/card authentication, plus additional commands for computing RSA digital signatures and encryption operations, and for retrieval of a public key certificate associated with an on-board key. The card access control model allows a predefined set of protection modes for card resources: always allowed, allowed after PIN verification, allowed after strong external authentication, a simple *and* or *or* combination of last two modes, allowed only when a secure channel is used, and never allowed. This protocol is tied to a specific context, and does not provide extension mechanisms for allowing, in example, different key types than RSA to be used for public key operations in the future.

Interoperability issues among cryptographic smart card devices are faced with in a different way by the Java Card$^{(TM)}$ standards [50, 51, 52]. These documents refer to cards with an on-board Java Virtual Machine (JVM), that are able to execute custom Java programs, called Applets. The standards define a subset of the Java language and Runtime Environment (JRE) that must be supported by the on-card JVMs, and a standard API that must be exposed to the Applets in order to allow access to on-board crypto facilities. This way it is possible to write a program that runs on any compliant smart card, implementing a custom protocol for communicating with the host. Fortunately this standard is being adopted by different card manufacturers. Both for its success, and for the well designed on-card cryptographic API, this platform has been chosen for implementation of the protocol introduced in this thesis.

Instead, the MuscleCard Card Edge protocol has been designed to allow use of the card by the most widely used applications, remaining as generic and application independent as possible.

Comparing protocols for smart cards

In this chapter the protocol introduced in Chapter is compared to other existing protocols for smart card interoperability, which have been defined by standard organization in the past. Because of the high level of complexity that such standards impose on data storage and cryptographic services exposed by a smartcard, a compliant protocol would fail to be the best one to be implemented on board in software, because of the size constraints to which the program is subject to.

First, a requirements analysis is presented in Section , catching what are the basic features that need to be really exposed by a smart card in order to achieve its primary goal: protecting user keys. Then, a technical comparison is made in Section between the data storage, cryptographic and PIN management services exposed by the new protocol, and the corresponding features as defined by the existing standards, highlighting features and limitations of the new approach. Finally, Section summarizes main results of the comparison.

Requirements analysis

In this section we make a brief survey on the requirements that a smartcard must meet in order to allow a host application to provide secure user

authentication, digital signature and data secrecy services. Our attention is focused on these services only, so our discussion does not consider card requirements in different contexts, like electronic cash or mobile phone applications, where additional operations or services could be needed.

A smartcard provides a portable, reasonably tamper proof mechanism for protecting user's keys. It provides the most effective way of protection, in that it can never reveal the key value[6] to the outside world. In order to achieve this, smartcards must be able to manage cryptograms internally, so they must implement on-board algorithms for using the cryptographic keys they are responsible for. Furthermore these devices need to store additional information in a permanent way. Such information is completely dependent on the application and should be regarded by the card itself as "generic data". In order to protect the card from misusage by unauthorized users or applications, access control must be enforced by the card itself.

Basically a smartcard needs to expose to host applications the following services (see fig.12):

cryptographic services to store and manage crptographic keys

data storage services to store and manage specific application data

access control services to specify and enforce access control rules on the card resources, so to prevent information leakage in cases of attack by an untrusted host or misusage by an unauthorized user.

Further details are discussed separately for each card service.

Data Storage Services

The most common example of data that is stored onto a smartcard is a public key certificate associated with the private signature key on the card. This is very useful to avoid scenarios where each application has to retrieve

[6] If such devices are really tamper resistant and what level of security they realize are topics out of the scope of this thesis.

Figure 12: Set of services that a smartcard protocol must expose

such a certificate from an online public repository. In case of a hierarchical arrangement of multiple certification authorities, a chain of certificates could be needed to reside on the smartcard, if enough space exists. Even if a card has not enough space for a digital certificate, the user's Common Name should be stored, so that an application can easily browse an on-line certificate directory for getting the user's public key certificate. However, application of smartcard technology should not be restricted or tied to the concept of Certification Authority. Infact in a PGP like approach [12] such a concept would not exist, and public key certificates would not exist. Other examples of user information that can be stored onto a smartcard include personal data about the user, numeric identifiers used by applications to uniquely associate the card with external data or processes[7], user preferences,[8] and others. Finally we should consider use of the same smartcard for multiple applications, where each of them could require different data and/or shared data to be stored on the card. So we realize that a smartcard must have a means for storing and managing multiple data in separate "containers" (see fig.13).

We define the following operations in the data storage facility:

Creation of a data container This operation creates a new data container

[7]Think of the certificate/key identifier that is temporarily stored onto a smartcard during the online interaction between the card and the certification authority web site.

[8]Think of a login application that identifies a user by means of a smartcard, then retrieves from the card itself information about the preferred graphic environment options.

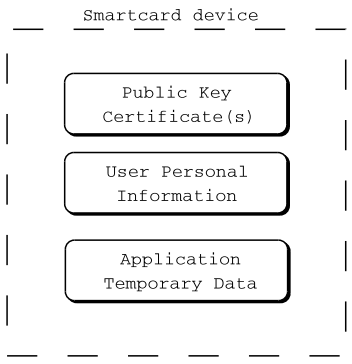

Figure 13: Example data that needs to be stored onto a smartcard

on the card to be addressed in further operations by means of its name. Because of resource constraints in a smartcard, usually the size of the container must be specified at creation time because of the memory that is reserved with such an operation. Access control rules for the container should also be defined in this phase. In order to avoid DoS attacks due to a hacked host, such an operation must be allowed only after a user or application authentication to the card.

Reading of data from inside a container This operation reads data from one of the available containers on the card. Depending on the type of the contained information this operation may be allowed either always or only after an authentication phase.

Modifying data inside a container This operation overwrites data that is already stored in one of the available containers. Usually this operation should only be allowed after an authentication phase, but some application data could be allowed to be always modifiable by a user.

Resizing a container This operation changes a container size, allowing it to grow or shrink according to the application requirements. The constraints in terms of available on-card memory impose applications to have containers whose size fits as much as possible contained data

size. Therefore a resize operation would be useful, even if not really required. This operation should only be allowed after an authentication phase.

Deleting a container This operation deletes a container given its name. This is useful when an application needs to create a container just to store temporary data and can also be used to resize a container on the card, fitting actually used space or increasing its size for additional data. This operation should only be allowed after an authentication phase.

For a proper specification of these operations, a naming scheme for containers needs to be defined. When using smartcards for authentication or digital signature applications, considering the constraints in terms of available memory, a simple and flat naming scheme seems to be appropriate. From this point of view, a smartcard protocol shouldn't require exposition of a hierarchical arrangement of the contained data, as this would lead to a need for a more sophisticated code for handling the hierarchy. We recommend using a flat filesystem for the management of data containers. An additional note on this can be found in section .

In order to complete a the data storage service specification, we need to define how access control rules can be specified and how they should be enforced by the card. This will be discussed in section .

Cryptographic Services

Multiple keys are required if different applications use the same smartcard or different type of keys need to be protected by the card. For example, a user could have an identification keypair, used for authentication only, and a signature keypair, used for signing electronic documents [9]. In a well designed protocol the set of allowed cryptographic algorithms must be expandable in

[9] This is required, for example, in the "Electronic Identification Profile" defined in the PKCS-15 standard [44].

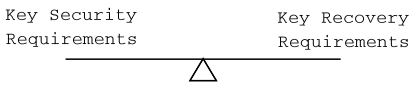

Figure 14: Usually there is a trade-off between key security and recovery requirements.

order to allow future versions to cope with new crypto algorithms that could be developed. Some applications need to protect their data at such a high level of a security that they require the assurance that the private key relative to a user's keypair is not known by any other entity than the smartcard itself. So enhanced security is gained if a card is able to generate onboard a keypair, where the public generated key can be exported after generation, while the private key can never be revealed to the outside world.

On the other hand, some applications want to give users the assurance that even if their smartcard is damaged there exists a copy somewhere of their private keys, and a new smartcard can be issued for the user[10] (see fig. 14).In some contexts such a behavior could be enforced by laws (cfr. key escrow). This requires for full key import/export facilities to be supported by a smartcard protocol, where the imported/exported data can optionally be encrypted using another key on the card so that the local host that is driving the card cannot gain information about such keys.

Often a smartcard's private key is not used to authenticate to the local machine where the card is actually inserted. Instead it is used to authenticate the user to a remote system. In these situations, it should be possible for a remote application to access smartcard resources without necessarily trusting the machine that is local to the user. This is achieved by establishing a secure communication channel between the remote application and the card, where every command to the card and exchanged data are secured through a cryptographically secure channel[11] (see fig. 15).

[10]Usually one or more trusted third party can store the user's private keys, entirely or partitioned by means of a secret splitting scheme.

[11]This is traditionally referred to as "secure messaging".

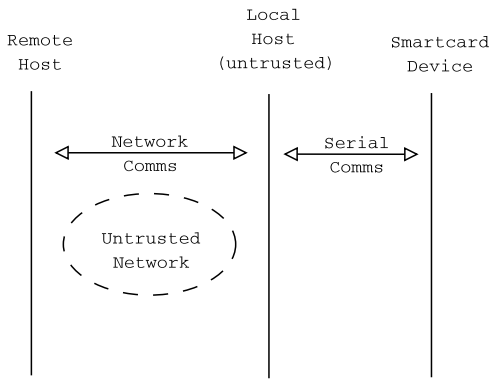

Figure 15: In this scenario the machine local to the smartcard acts only as an untrusted gateway to the remote host to which authentication is performed.

Finally we can require the following cryptographic related operations to be supported by a smartcard protocol[12]:

Key import This operation allows the card to import an externally provided key. This is useful when the card has no on-board generation facilities and when it needs to load a specific key value on the card. This operation could also overwrite an already existent key on the card. We recommend that a key import is allowed only after user/application authentication.

Key export This operation allows the card to export a key value to the outside world. For what stated above is should be possible to do both a plain export and an encrypted export, where the exported blob is encrypted using another cryptographic key. We recommend that a key export be wither forbidden or protected by means of requiring a strong application authentication or exporting an encrypted keyblob.

Key generation This operation allows generation of a new keypair on the card. On-board generation of symmetric keys is still possible but is not

[12]In this discussion, we include the key storage facility in the cryptographic service definition. Other approaches place it in the data storage facility [44], defining special data containers reserved for the purpose of storing cryptograhic keys.

really required, as the key has to be shared with other entities. This operation could also overwrite an already existing key on the card. It should be allowed only after a strong application authentication.

Key usage This operation computes a cryptographic operation on externally provided data, returning the output to the host. Operations to be performed can be defined as encrypt and decrypt for symmetric keys, and encrypt, decrypt, and sign for asymmetric keys. The verify operation is not required to be available on the card, as it usually relies on the asymmetric key that is publicly available. Different modes of operations are available for different crypto algorithms and operations. It is recommended that the set of available modes is easily expandable in future releases of the protocol. In order to prevent unauthorized use of a cryptographic key, this operation should require at least a user authentication.

key replacement This operation overwrites a key with either a new internally generated one or an externally provided one. This operation is just a special case of the "generate" and "import" operations. We can notice that a key replacement can require additional authentications, depending on the access control rules associated with the key being replaced.

key deletion This operation deletes a cryptographic key from the card memory, freeing all of the additional resources that are tied to it, like intermediate buffers needed during encryption operations. This operation also should require application authentication.

A smartcard stores and manages secret keys and private keys, so improper or unauthorized invokation of key operations could lead to a weakness of the entire system. Therefore an application must be able to selectively restrict operations on certain keys, so that either they are not allowed at all or they are allowed after a proper user and/or application authentication.

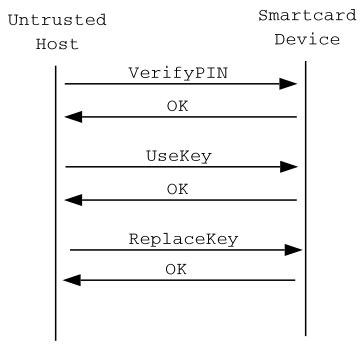

Figure 16: Successful attack by a hostile terminal to a single-PIN smartcard allowing key replacement.

Access Control Specification and Enforcement

A basic requirement of a smartcard is that it must reasonably protect the contained resources even if it is exposed to a "hostile" machine. This is due to the fact that, being a portable device, a smartcard is very likely to be connected to a host machine that is out of the control of the cardholder. In such cases, the information leakage must be as low as possible, while the key material leakage must be zero. The effectiveness of such an attack depends on the access control model that the card enforces. For example, if the card protects everything inside with a single PIN code, once that the code has been inserted by the legitimate user, a hostile machine can take control of the card (see fig.16). A well designed card must have at least multiple PIN codes granting different access levels on the resources on the card, so that an attack like above would be limited to granting access to only a restricted set of allowed operations (fig.17).

Furthermore, we require that different operations on a single resource are protected differently. For example, public user data could be always readable but never modifiable, or modifiable only after user and/or application authentication. A private key usage should be allowed only after user authentication by means of a PIN code verification, while key overwriting or exporting should either be always forbidden or only allowed after a strong

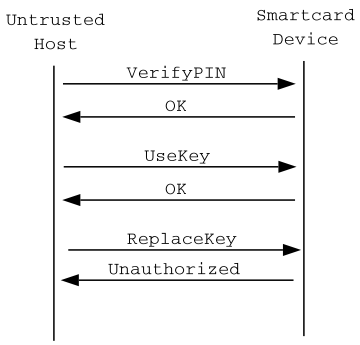

Figure 17: Unsuccessful attack by a hostile terminal to a smartcard allowing key replacement only after verification of a second PIN.

application authentication.

We can define the following schemes for authenticating external entities to a smartcard:

PIN code This is typically an ASCII encoded numeric value that is entered on a pinpad by the cardholder and compared by the the card with the on-board stored one. Some unsuccessful attempts are allowed to happen, as this code is interactively entered by a human, but consecutive failures in the verification must cause a block of the operation so that exhaustive searching of the code is not possible by an untrusted host or user.

Strong authentication This is achieved by means of a cryptographic challenge response protocol where the private key is known to a host application while the public one is stored onto the smartcard. As such an authentication is run by a program it would make no sense to tolerate unsuccessful attempts.

Biometrics This is achieved by means of a biometric information (commonly a fingerprint scan) that is grabbed by the host by means of a biometric device and is sent to the card, where an on-board matching algorithm compares it with the data stored on the card. Due to

the lack of standards in this field, we require that a smartcard protocol supporting biometrics has enough "freedom" to allow this kind of authentication, where the single details could vary according to the specific adopted algorithm.

For those cards not supporting on-board biometrics algorithms, it is still possible to use the card for simply storing the biometric data, that is retrieved by the application each time it is required, with a decrease of the achieved security level. Such an approach can rely only on simple storage facilities on the smartcard and nothing else is required on the protocol side. Furthermore in this case the biometrics verification does not affect the security level of the session with the smartcard, as only the host application is aware that a biometric identification of the user has been performed.

On a further note, we must notice how a common requirement in the smartcard security model is the availability of different levels of authorizations for different operations. For example, a public key certificate could always be readable from the outside world, but it should not be writable. In order to prevent improper card use in case it has been stolen, computations using private keys should only be allowed after a user authentication, by means of a PIN verification. Furthermore it is often required that private keys' use is restricted depending on their purpose. For example, for improved security, a key for digital signatures could be enforced to only process data after a hashing and padding phase, so to avoid choosen plaintext attacks. This leads to a possibility to specify more complex key management rules, also known as "key policies".

Both the certificate data and the user's private key could need to be replaced in case of a renewal process. Rewriting such objects could be either denied at all, or protected by means of a strong authentication of the C.A. application.

This takes to different access control rules to be enforced on different resources and on different operations (or operation classes) on the same resource. Access control rules must be able to require that exter-

nal users/applications have been authenticated at a certain security level.

In common applications, it is easy to think of a key whose usage requires a preliminary PIN verification by the cardholder. A more sophisticated application could require that the user uses a key only with that specific application or with an application class. This could lead to an additional application authentication to be performed by the card before allowing the operation. A second PIN verification could be sufficient for some applications, where a challenge/response scheme could be required for a stronger security level.

A well designed smartcard protocol must allow the session to start at the lowest authorization level ("public level"), then to gradually switch towards higher authorization levels as the card runs more and more successful authentications. It must be possible to exactly specify what authentications are required before each operation is allowed on each resource (key or data container).

It is evident that access control rules and authentication mechanisms constitute additional data that has to be stored and managed by the card, so they require additional commands in the protocol. We suggest specifying access control rules within the command creating the resource to which the rules apply. Authentication mechanisms, instead, require a separate set of commands for their management:

Mechanism creation This command creates a new authentication mechanism providing all of the necessary data. Each mechanism requires specific parameters to be specified. Cryptographic challenge-response based mechanisms would require as parameter a reference to the stored public cryptographic key. The set of allowed mechanisms should be expandable with future releases of the protocol

Mechanism verification This command runs the authentication mechanism by providing necessary data to the card. Challenge-response based mechanisms require a preliminary retrieval of random data from the card, then a second command performs the authentication itself.

Mechanism parameter change This command modifies parameters of an authentication mechanism. This is commonly required when a user PIN code or an authentication private key have been someway compromised; for strong authentication mechanisms this can be performed with an import operation on the cryptographic key to which the mechanism is tied.

Mechanism unblock This command unblocks an authentication mechanism that has been blocked by the card after multiple successive failures of the authentication (to avoid exhaustive searching of the secret).

Technical comparison

In this section we analyze how the new M.U.S.C.L.E. Card Edge protocol satisfies the requirements defined in the previous sections, also making a comparison with some of the standards existing today in the smartcard industry world. Remainding to the specification document for details about the protocol, our attention in this chapter is focused on a technical comparison between different approaches to interoperability across card devices.

Cryptographic services

For what discussed in the previous sections, we argue that the set of crypto related commands that a smartcard protocol incorporates must be as generic as possible, being untied to particular applications or particular concepts defined inside applications. This approach leads to a protocol where the smartcard is seen as a crypto processing unit with a complete set of commands to access its functionalities, where proper access control rules can limit the allowed operations depending on the accessed resource and operation and on the authorization level of the host side application.

This is not the case for the ISO set of cryptographic commands [30], for example, where at a first glance we immediately notice that the concept of digital certificate is heavily embedded into the protocol. Infact one of the

possible operations of the "Perform Security Operation" command is a certificate verification. This operation has also an operation mode that allows extraction of the contained public key from the certificate for a successive use. While such an operation could turn out to be useful in PKI based applications, we push towards not having such operations on board. First of all a certificate parsing operation is not trivial, so it can take several resources on the card to be accomplished. Second, a good certificate verification would need not only a digital signature verification, but also a temporal validity check, then a check that the type of certificate and eventual attribute extensions match the intended use of the contained public key, and so on. All of these operations are unlikely to be performed by a smartcard device, and most of all only the final application is aware of the purpose of a public key or certificate. Furthermore, public key operations can usually be conducted on an external host without diminishing security level of the involved protocols.

In the protocol specified in the Government Smartcard Interoperability Specification [22], on the other hand, we can find an "RSA Compute" command, making the protocol strictly tied to a very specific cryptographic algorithm. We strongly discourage such practices, where a more generic, algorithm independent approach should be followed. In spite of the existence of many cryptographic algorithms, only a few of them has enough credits to be widely adopted. This is the case of the RSA [46] 2-primes algorithm, DSA [55] and of triple DES [54], that are supported by most common crypto-aware applications. Elliptic Curves [41] are also gaining much interest in the smartcard world. This has been reflected by smartcard manufacturer, that basically implement just a few algorithms in their devices. However, it is not a good design choice to tie a smartcard protocol to a particular algorithm just because it seems to be widely adopted and accepted. Situation could change in a near future.

The MuscleCard Card Edge protocol defines a "Compute Crypt" command that is able to process data using multiple cryptographic algorithms and operation modes. Actually supported algorithms are only those considered by the JavaCard standard, while extensions for allowing additional

algorithms and/or modes of operations (i.e. padding or hash schemes) can be easily achieved by expanding the set of allowed values for the command parameters.

On the other hand, a limitation of the cryptographic service defined in current version of our protocol is that it is currently missing a key policy specification and enforcement mechanism. This is one of the extension that will be integrated in the next release.

Data Storage Services

Most smartcard manufacturers have cards complying more or less with the storage service as standardized by ISO in [27]. This document defines the concept of a filesystem-like hierarchic arrangement of files onto a smartcard, and provides a set of protocol commands that allow a host application browse, retrieval and change of contained data. Unfortunately the initial set of commands was not complete, because the fundamental commands for creating resources were completely missing. Infact the standard specified how to verify a PIN code, but there was no specification of a command to create it. Similarly it contained commands for navigation inside the filesystem, but there were no standard commands to create directories or files. Resource creation commands were only addressed lately by ISO [31]. In the meantime, card manufacturers had to embed non-standard, proprietary commands for such tasks with the result that most smartcards today can be "formatted" and managed only by using proprietary software.

An evident advantage of the ISO hierarchical naming scheme is certainly the possibility to allocate file names in each subdirectory independently from other subdirectories (figure 18). This allows to have much more freedom in allocating file names even in presence of a limited filename length like the typical 2 bytes identifiers. We have to observe that such a feature would be indispensable if we had to store hundreds of files onto a smartcard, but this is not the case, given the actual memory size of these devices. The 4 bytes numeric identifiers of the MUSCLE protocol are enough wide to allow freedom

Figure 18: Hierarchical Filesystem defined in the ISO standard for data storage.

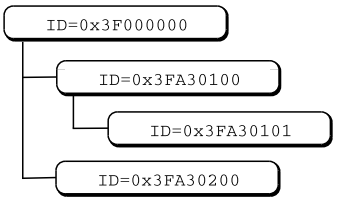

Figure 19: Hierarchical arrangement of object identifiers in the flat M.U.S.C.L.E. object naming scheme.

in their allocation by applications. Furthermore, if needed, they can allow a kind of hierarchical arrangement by splitting the 32 bits into substrings, into a fashion that resembles allocation of 4 bytes IP addresses (fig. 19). The "directory" containers, in such a case, could contain additional information about the hierarchy or be absent. In order to increase interoperability among applications, it is encouraged that a further document comes up as soon as possible standardizing how inter-application data should be arranged onto a card implementing our protocol, resembling what the PKCS-15 standard [44] has done for filesystem cards. For example a specific object ID could be reserved for storing a certificate directory, containing also the required associations between on-board certificates and private keys. Finally we have to observe that an application absolutely requiring a filesystem can always provide, as a last resort, an "emulation layer" on the host side. This would be of course an untrusted filesystem, in that its consistency could not be enforced by the card itself.

Maybe the main purpose of hierarchical arrangement of data onto a smartcard is the ability to separate the contained data files into different "security domains". Different access control rules, external authentication secrets and cryptographic key sets can be associated to a directory and have them inherited by all of the contained files. This is a good feature as long as there exist a complex scheme for specifying access control, like in the ISO standards, so that there is no need of repeating the complex rules for each of the resources to which the rules apply. Next session will analyze this in more detail.

As a final note, we can observe that another advantage of directories is to have a means for an application to statically reserve a certain amount of memory on the card at the time of creation of the directory entry. This can be easily achieved also with a flat filesystem, simply allocating a "fake" data container that is resized (or deleted and created again) each time new containers need to be created or existing ones need to be deleted by the application.

As more and more storage space is allowed to exist onto a smartcard, more and more data is going to be stored onboard by applications. If only a small subset of such data is needed during a session with the host, and the available bandwidth for communications is seriously limited (like in 9600 bps smartcards), then a need for search facilities to be implemented directly on the card itself arises. An application could just retrieve the data that matches its search criteria, without having to download the entire stored material on the card. There exist today smartcards with 32K bytes storage size, and 64K cards are to be issued. So this kind of problems will become a reality in a near future. We have to observe that also transmission speed between smartcards and hosts is going to improve, as we migrate from serial to USB reader devices.

However, even if not really required today by most applications, the command set standardized by ISO [29] for complex data storage and retrieval is fully justified. This approach allows definition of tables onto a smartcard in a database-like fashion. Complex data storage services like this are missing

from our protocol, that focuses mainly on basic services that need to be exposed in order to allow an interoperable use of the card for authentication, digital signature and encryption purposes.

Access Control

The ISO security model is very complex. It requires storing of multiple "security environment" and "security attribute" definitions, managing of a global, file-specific and command-specific "security status", storing of associations between resources and security attributes. Quite everything is tied to the hierarchical structure of the files, where access rules defined for a directory automatically apply to all of the contained elements, unless overridden by the element specific rules.

Specification of access control rules is powerful and flexible. It is possible to state, for each possible operation on a resource, with a fine granularity, what authentications must have been performed in order for that operation to be allowed. Authentications can be required in both an "and" and an "or" fashion. In the "expanded format" a rule can apply to a custom operation, specified by means of the CLA, INS, P1 and P2 parameters of the command invoking it, and can be built combining multiple sub-rules into complex boolean expressions. A similar but less powerful approach is followed in the PKCS-15 document, where the notation syntax for a "Security Condition" allows to specify a boolean expression arbitrarily combining required authentication mechanisms.

In the ISO model, while an ACR definition can generically require an "external authentication" or a "PIN verification" for an operation, it does not contain references to what key or PIN are to be used for the purose. This is dependent on the position where the resource is located inside the filesystem hierarchy. Infact files can be implicitly associated with local authentication secrets. This adds extra value to the hierarchical filesystem, making it not only an arrangement of names, but especially an arrangement of security domains inside the card.

Such a scheme, in order to be well defined, requires non trivial management protocol commands and a complex enforcement engine onboard, that must be able to scan the security rules in order to establish if an operation can be granted or not. For this reason it is not the best scheme to be implemented in software on a Java Card or generically on a programmable card, today. We really argue that such a scheme is over-sized for most widely used applications, that often require just a couple of keys to be stored on the card, with some additional information associated to them (i.e. a certificate), all protected by a PIN code and/or an additional strong external authentication. Also there's not a real need for arbitrarily complex boolean expressions in specifying access control rules for on-board resources and operations.

From this point of view, we argue that the Muscle Card security model [17], being much simpler and retaining the needed level of complexity, is much more suitable for being implemented on a programmable card.

The Muscle Card model defines multiple authentication mechanisms, called "identities", available for the outside world to authenticate to the card. There exist different types of mechanisms: PIN verification, cryptographic challenge-response authentication protocol, and application defined. Everytime one of these mechanisms is successfully run, the corresponding identity is said to be "logged in" in the current session. For each resource on the card and for each operation class on a resource, it is possible to define what are the minimum set of identities that must already be logged in order for the operation to be allowed. When multiple identities are specified they are always required in an "and" fashion. Operations on which the protocol allows a separate access control specification are: read, write, delete for objects and export, overwrite, use for keys.

Examples of achievable access control rules are:

- operation is always allowed
- operation is allowed after a PIN verification
- operation is allowed after a double PIN verification

- operation is allowed after a strong authentication
- operation is allowed after a strong authentication and a PIN verification
- operation is allowed after a custom application mechanism (i.e. biometrics)
- operation is allowed after a PIN verification and a custom application mechanism (i.e. biometrics)

This scheme is very simple but has enough power to express all of the access conditions specified in . Basically, in our protocol, we are actually missing the alternative specification possibility, that could allow access to an operation on a resource by means of an authentication by either the user or the application (and similar). This possibility could come in handy, for example, to allow a user not to enter her PIN code once a particular application has been authenticated to the card. Another situation in which alternatives in ACRs would help arises when two applications need to access a particular operation on a shared resource on the card, but they don't want to share the same private authentication key. Basically the possibility of specifying multiple user authentication and external authentication mechanisms has been designed thinking of a multi application use of a smartcard, where the accent is much more on application indipendence than on their sharing or cooperation. So we defined a security model that allowed an application to define its own authentication mechanism, so that once it is run by the user, data owned by other applications cannot be corrupted or accessed. In our protocol an application is allowed to set up proper access control words so to guarantee such an independence property.

So we recognize that adding an "or" behaviour in ACR specification would add power to our scheme, that is actually sacrificed to the easiness of implementation on the card itself. Infact a simple arithmetic AND operation between the security status word and an operation ACL word, then a com-

parison, is all what is required in order to check if an operation is granted or not.

Another security model that is enough simple is specified in the Government Smartcard Specifications [22]. Basically that document defines 4 access control modes:

- operation is always allowed
- operation is allowed after a PIN verification
- operation is allowed after a strong authentication
- operation is allowed after a strong authentication and a PIN verification
- operation is allowed after a strong authentication or a PIN verification
- operation is allowed only if a secure channel is used for data exchange
- operation is never allowed

This variety in access modes to card resources satisfies requirements of most smartcard applications. Unfortunately the security model defined in this document is restricted to a single application context, missing the possibilities of having multiple PIN codes or authentication keys on the same card.

Conclusions and future work

In this chapter a set of requirements has been defined which a smart card protocol must meet in order to be generically usable by common applications that need to protect user's data with cryptographic services. Basic requirements have been identified allowing storage, cryptographic, and access control facilities of the card protocol. Also, it has been shown how existing standards for smartcard interoperability satisfy such requirements,

comparing their approach to the recently issued MuscleCard Card Edge protocol.

It has been underlined how the constraints in terms of storage and computational resources onto a programmable smartcard make it quite unfeasible to have complex operations and services directly implemented on-board, like some of the existing standards suggest.

On the contrary, the new protocol combines enough simplicity and completeness at the same time to be suitable for a software implementation on programmable devices. MuscleCard protocol does not pretend to be exhaustive with respect to existing smartcard applications. It has been designed around user authentication, digital signature and data secrecy services, where a technical comparison with other approaches evidenced how some additional features could turn out to be useful for expanding the set of supported applications.

An investigation needs to be done in order to integrate into the protocol some minimal missing features that could allow a wider set of applications to take benefit of it, still retaining the simple and clean design that allows a feasible implementation on programmable devices without using too much memory for the program implementing the protocol.

Specifically, features that need to be addressed in a near future for inclusion into the protocol are a secure messaging service for allowing remote card administration and a key policy specification and enforcement mechanism for improved key protection. Lower priority features to be added could be a searching facility to allow selective retrieval of application data, a little more complex security model that increases the flexibility in specifying access control rules and a set of minimal additional operations needed to allow integration with common payment system applications. The versioning mechanism already featured by the protocol would allow such extensions maintaining backward compatibility.

The host-side architecture

This chapter overviews the open smart card middleware which has been developed around the new protocol introduced so far. The architectural design is centred around the definition of a smart card API that allows protected access to the storage and cryptographic facilities of a smart card. The proposed API constitutes a new interoperability layer that allows partitioning of a smart card driver architecture into a lower *card-dependent* level, that formats and exchanges APDUs with the external device, and a higher *card-independent* level, that uses the API for implementing more sophisticated 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 development 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. Smart cards that have been tested with the new architecture include Schlumberger Cyberflex Access 32K and Cryptoflex, Gemplus GemXPresso 211/PK and FIPS 64K, US DoD card and IBM JCOP 32K, thus the development stage has proved effectiveness of the new approach.

The chapter is structured as follows. Next section makes a brief overview of the new architecture, while Section features an overview of the new API. Finally, Section presents other existing open architectures for smart cards,

highlighting their advantages and disadvantages with respect to the proposed solution.

Architecture overview

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 20. 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 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 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 can manage 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).

The API 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 [17], in that most function calls are directly mapped into the APDUs of the protocol. This layer has been implemented in various card drivers for card 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 the MUSCLE Card Applet has been loaded on-board ; the Schlumberger Cryptoflex 16K

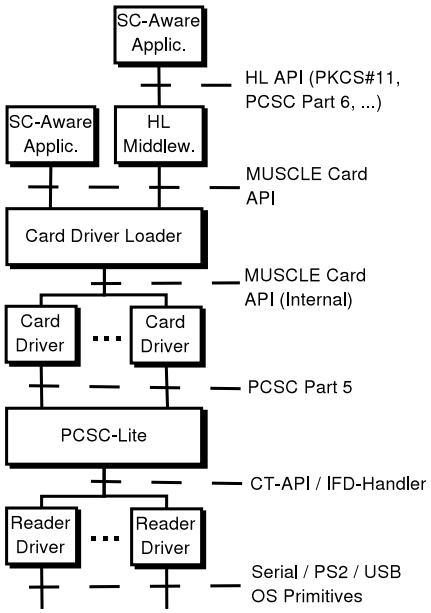

Figure 20: General architecture of the proposed smart card middleware.

card, which exposes a set of ISO 7816-4 APDUs for filesystem management, and custom commands for cryptographic operations; 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, further application and middleware layers have been developed. Specifically, an open source PKCS-11 module, mapping the PKCS-11 API calls into the appropriate sequences of MUSCLE Card API function calls, has been developed.

As an alternative, applications can directly use the proposed API in order to talk to smart card devices at a lower level, and 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 project (http://smartsign.sourceforge.net): a command line digital signature application (sign-mcard), a variant of the OpenSSH software (openssh-mcard). A PAM [47] module has been directly developed using

this API, allowing smartcard based user authentication for applications using PAM on Unix like systems, like the Unix login. Finally, 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.

API overview

This section features a technical overview of the proposed API. The discussion is focused on the introduction of the API main features, and explanation of the main design choices behind its development. The complete API specification [15] is available for download at the URL: *http://www.musclecard.com*.

Objectives and design choices ISO standards impose a high level of complexity on data storage and cryptographic services exposed by a smart card, constituting a powerful and flexible solution that is suitable for a wide range of applications. Though, an implementation of these standards onto a programmable device existing today, like a 32K Java Card, would result in a so big program that a few on-board memory, if any, would be left for application data and cryptographic keys.

The new protocol introduced in this chapter, instead, has been designed with the aim of satisfying the requirements that a smart card stands for: protecting user keys.

Main aim of the API design is to provide higher layer software components with an open, simple, 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 services, both PKI based and not.

These requirements have been identified in having a means of generating, importing, exporting, and using cryptographic keys on the card. Another requirement is to have a means of creating, reading, and writing generic data on the card in generic “containers”. This is useful, for example, to store

on the card a public key certificate associated with a private key. Access to some of these resources needs to be granted only after host application and user authentication.

The result is a simple and light interface that has been proved to be effective in allowing integration of smart card technology into secure applications, as shown by our sample application cases. 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 [13].

The API does not address 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.

API function set

The set of functions available in the proposed API is summarised in Table 4. API functionality has been divided into 5 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 further 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 [15].

Session management The API has a minimal set of functions allowing the enumeration of connected readers and inserted smart cards, and management of the connections to the card devices. Establishment of a connection to a card device is a prerequisite for the use of any of the other functions of the API. Specifically, the `ListTokens` function is able to enumerate readers connected to the system, readers which have a card inserted,

Session mgmt	ListTokens, EstablishConnection, ReleaseConnection WaitForTokenEvent, CancelEventWait BeginTransaction, EndTransaction
Data storage	CreateObject, DeleteObject, ListObjects WriteObject, ReadObject
Cryptography	GenerateKeyPair, ComputeCrypt ImportKey, ExportKey, ListKeys
PIN mgmt	CreatePIN, ChangePIN UnblockPIN, ListPINs
Access ctrl	VerifyPIN, GetChallenge ExtAuthenticate, GetStatus, LogOutAll
Miscellaneous	WriteFramework, GetCapabilities, ExtendedFeature

Table 4: MUSCLE Card API function set

along with the type of inserted device, and the list of all supported card devices in the system. Furthermore, an application is able to block and wait until a card insertion or removal by using the `WaitForTokenEvent` function. Once a card is inserted into a 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 21(b), in the case in which the device is not yet inserted, and the application waits for its insertion.

Data storage services Our middleware allows the definition of simple containers for The API specification encapsulates applications' 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 will be given later). The data storage services suffice for the target applications cited in the beginning of Section , by allowing them to store, retrieve

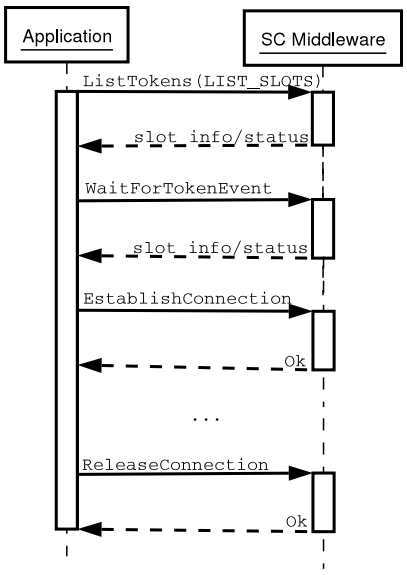

Figure 21: Sequence of calls needed for establishing a connection with a card device

and manage data onto a card in a secure and controlled way. This is a minimum set of operations and access constraints that is needed from host applications and suffices to securely store, retrieve and manage data onto a smart card. This does not preclude a hierarchic organization of applications' data into a filesystem-like fashion. That could be achieved on the client side on a per-application basis, or in an inter-application fashion if further documents came up standardizing particular objects to be used for filesystem information. This way consistency of the achieved hierarchy would be up to the host-side applications and could not be enforced by the card. Still this approach would be suitable for environments where the hierarchy is created statically and needs not to be changed dynamically.

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 about this). The same information may be visioned by applications for all existing on card objects through subsequent calls to the `ListObject`

function[13] . Reading and writing of data to and from objects is performed, respectively, through the `ReadObject` and `WriteObject` functions. Execution of these functions may be restricted on a per-object and per-operation basis. The API does not provide, at the moment, any facility for the definition of a hierarchic structure of objects, and it does not even provide typed objects, conversely to the intrinsic nature of ISO 7816-4 compliant devices, which possess an on-board hierarchic filesystem. However, such functionality could be added in the future, if needed. 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 still leaves space for other documents to come up standardising OIDs to be used to store special information with an inter-application relevance. As far as the card storage capacity is concerned, the interface specification gives only 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 specific on-board 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 made free 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 sometimes 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 a PIN verification. Our approach is completely different in that we have a unique memory space, where single created objects are associated with access control rules specifically customised for each object and operation. So our model allows reconfiguration of the memory space as

[13]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

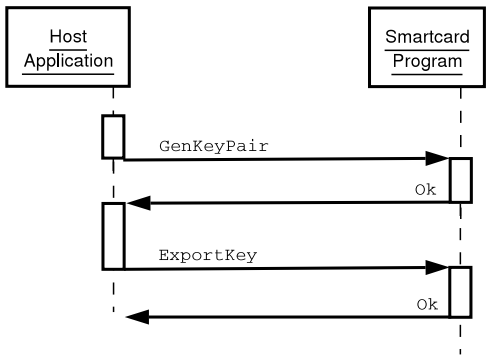

Figure 22: Sequence of commands for an on-board key pair generation with the subsequent export of the generated public key value, to be read after the ExportKey command from the output object.

public or private according to application requirements. Section will explore in more detail the adopted security model.

Cryptographic services The API allows up to 16 keys to be managed on the card, identified by means of a numeric key identifier. A full key pair can be stored by using two key identifiers. Key types are those provided by the Java Card 2.1.1 API: RSA, DSA, DES, Triple DES, Triple DES with 3 keys. The interface is designed to allow further key types in the future. Operations provided on cryptographic keys are import/export from/to the host, computation of cryptograms, and listing of keys, which provides size and type information. All key operations except key listing are allowed only after proper host application or user authentication. The API allows key pairs to be directly generated on board guaranteeing the private key is not exposed outside the card. In this case the public key can be obtained with a call to `ExportKey`, right after the key pair generation. When a key pair is created on-board, the host application specifies under what conditions subsequent reading, overwriting and use operations are allowed for each of the keys in the pair. The same rules can be specified when importing a new key from the outside world.by means of the `ImportKey` function.

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

Table 5: Association between identity, PIN and cryptographic key numbers.

Security model and access control enforcement A simple Access Control List (ACL) based model is defined to protect on-board objects, allowing operations to be performed only after proper 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 several authentication mechanisms that host applications and users can use to be authenticated to a smart card. Identities, PINs, and cryptographic keys are referred to by means of numeric identifiers. Different types of identity are defined (see also table 5): identities n.0-7 are labelled as *PIN-based* and are associated, respectively, with PIN codes n.0-7; identities n.8-13 are said *strong* and are associated, respectively, with cryptographic keys n.0-5 for the purpose of running challenge-response cryptographic authentication protocols; identities n.14-15 are *reserved* [14].

A successful run of any of the authentication mechanisms causes the *log in* of the associated identity, in addition to identities already logged in. The use of multiple identities allows a host application to switch to a

[14] The fingerprint verification mechanism recently developed uses identity n.14.

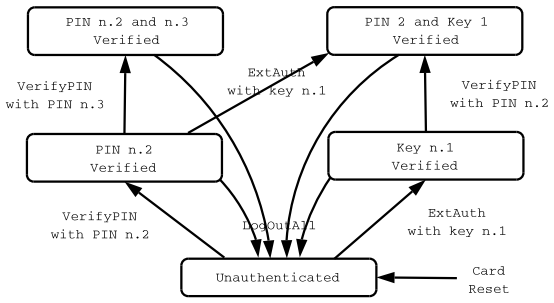

Figure 23: Subset of possible security state transitions allowed by the API specification.

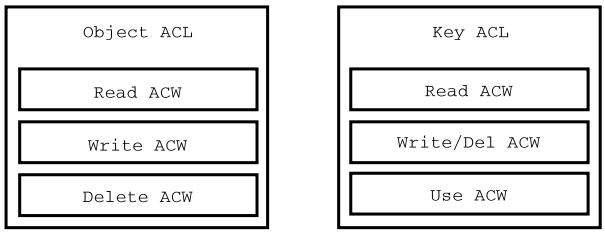

Figure 24: Composition of the Access Control List for objects and keys.

higher security level that grants access to more of the card's capabilities, as it runs additional authentication mechanisms. Furthermore the *LogOutAll* command allows a host application to return back to the unauthenticated security status. A little subset of possible security states and transitions due to successful authentication commands is shown in Figure 23.

An ACL specifies which identities are required to grant access to each operation of each object or 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 relating to an operation. (see figure 24). An ACW consists of 16 bits. Each bit corresponds to one of the 16 identities that can be logged in. An all-zero ACW means that the operation is publicly available, that is a host application can

perform it without any 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, means that the operation is disabled and cannot be performed, independently of the connection security status. This is useful to disable 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 usually 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.

PIN management services 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. Up to eight PIN codes are allowed in principle to 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. A special PIN (*transport PIN*, n.0) is assumed to exist right after the program has been loaded and instantiated on a card, and it must be verified to allow a host application to create further resources on the card. This has been imposed to prevent allocation of card resources without user knowledge. It is highly suggested that applications *format* a smart card by creating an application specific PIN code, and using it for protecting application specific data and keys. With a proper setting of the ACLs on the card, an application can "format" the card in such a way that further usage of the card is dependent on a different PIN than the transport one or on a different authentication scheme, like a challenge-response cryptographic protocol. This way after the user has entered the PIN code from an untrusted terminal, 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 that terminal to interfere with other applications.

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 PIN, data objects and keys. This has been accomplished in two ways: requiring verification of the *transport* PIN to allow creation of new PINs, objects and cryptographic keys, and allowing an application 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 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 only 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.

Transactions and related issues.

Different kind of error conditions can occur during a smart card operation. It is possible that the host provides an incorrect class code for the currently inserted card, an incorrect instruction code for the specified command class, or incorrect parameters to the specified command. Furthermore, a host can cause an access violation when trying to access a resource/operation on the card that is forbidden with privileges of the actual session. It can happen that the software component that is currently either providing data to or retrieving data from the card suddenly interrupts its operation (i.e. an application crash or a system shutdown). Finally, the card can be suddenly extracted by the user during a command execution.

The protocol specification explicitly deals with first four conditions, by specifying, for each command, what error codes must be returned by the device when some of these conditions occur. These can be regarded as "graceful" failures because they assume that both the host and the card are still operating correctly and can run the needed error recovery procedures to handle the condition. Last two error types are also very important with respect to a smart card life, because they raise transactions issues due to a sudden reset or power down of the device. In fact after a host-side application crash, usually, the device is again under control of the resource manager, that should issue a "reset" or "power down" command (see figure 25) to the reader in order to guarantee security of data and keys that were handled by that application (if it was using the device in exclusive mode). Furthermore a card extraction always powers down the card. If the device

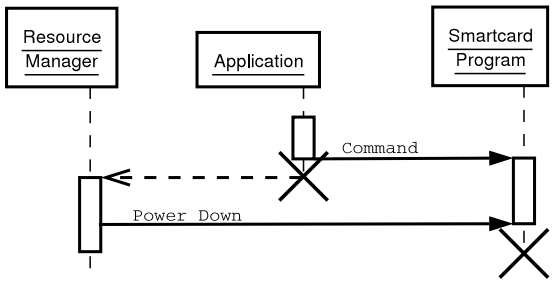

Figure 25: Sudden application crash causing a reset or power-down of the card during a command execution.

was updating its internal data during the command execution, it is very important that this is done in a *transactional* way, so to guarantee consistency of data in such cases. The really important property that must be guaranteed is consistency of internal "directories" of objects and keys, including access control information. Data contained in objects does not need to be handled in a transactional way. Consider the case of a power-down during execution of a command that is overwriting an object contents. First of all, other objects must not be affected anyway by the event. Second, the access control information related to this object must not be affected anyway by the event, otherwise it would be possible to set-up attacks based on this kind of events. A desirable property is also the transactional update of object contents, that is either the contents are updated or the old contents are recovered, but it is not really required, as it should be anyway considered at an application level.

The provided implementation of the protocol is a Java Card Applet running into the Java Card Runtime Environment [51], that already implements transactions on every updates to permanent card data during a single command execution, up to a maximum amount of changed bytes. This allowed to write the Applet without any additional code for implementing the transactional behavior.

When implementing the protocol onto a programmable, non Java Card,

device, it is of fundamental importance that the program explicitly addresses transactional issues, guaranteeing consistency of at least internal key and object "directories", and access control data.

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 the 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 done 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 as a means for adding on-card biometric pattern matching without affecting the original protocol.

Card specific behaviour The API which has been just introduced provides a unified means, for higher level middleware components as well as applications, to access the described 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 is able to support the full set of functionality available through this API. Each specific card generally sup-

ports only a subset of such 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.

Related projects

This subsection provides a quick overview of existing open architectures for smart cards: OpenSC, SecTok, OCF, GPKCS#11 and CDSA. , highlighting how these projects are placed with respect to the proposed architecture. The OpenSC [33] project provides a library and a set of utilities for accessing ISO 7816 [28] and PKCS#15 [44] compliant card devices. Specifically, the project features a programming interface with functionality for: ISO 7816-4 [28, Part 4] filesystem browsing and file reading/writing; ISO 7816-9 [28, Part 9] filesystem management; ISO 7816-8 [28, 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 only in a read-only fashion, and/or do not respect the PKCS#15 standard for managing information about the on board cryptographic material. Such devices cannot be directly supported within this architecture, especially on the side of card-personalisation.

The SecTok [14] 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) [36] is a Java based development platform for smart card development. It aims at reducing dependence among card terminal vendors, card operating system providers and card issuers, by the adoption 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 the wide variety of card operating systems. in existence and the various different functions they may offer. Example CardServices are the FileAccessCardService, providing a fairly complete set of interfaces and classes abstracting a ISO file system's functionality, and the SignatureCardService, offering methods to create and verify digital signatures. Further, the problem of card issuer independence is addressed separately by OCF's ApplicationManagement 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 due to the way I/O is managed at the lowest levels of the architecture, and the inherent difficulties and overhead needed in order to access such functionality from programs written in different programming languages than Java.

The GPKCS#11 project [53] aims at providing support functionality that ease the development of a PKCS#11 driver for cryptographic tokens. It contains a complete software token, based on the OpenSSL library, 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. Then, a PKCS#11 driver is supposed to implement 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 already 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. Despite the optimal principles that inspired the GPKCS#11 project, Unfortunately, the project lacked detailed documentation about its features, and it has not been maintained since year 2000.

The Common Data Security Architecture (CDSA) [35] 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. Intel provides an open source implementation [9] of the standard for both Windows and different flavours of Unix[15]. 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. It is now available for both Windows and different flavours of Unix from http://sourceforge.net/projects/cdsa The main site for CDSA is http://developer.intel.com/ial/security/.

The architecture that is being introduced in this chapter, at the authors' knowledge, is the only open architecture completely modular that allows multiple heterogeneous devices to be supported through the implementation of a common lower level API, which exposes sufficient functionality needed by most PKI applications , such as management of on-card memory, cryptographic keys and PIN codes, since the time of issuing of the card by a CA, up to the final use of the device by applications. The efforts needed for the implementation of such drivers is limited, with respect to the full implementation of one of the well known standards for smart cards, such as PKCS#11 or PCSC level 6. Still connectivity with such standards is possible

[15]The implementation is available at the URL: http://sourceforge.net/projects/cdsa.

Project	Lang	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 applications
GPKCS#11	C	Lack of documentation and maintenance
CDSA	C	Architecture stands at a higher level

Table 6: Summary of open architectures for smart cards

through the implementation of the higher level API through the MUSCLE Card API, what can be done in a separate module, and once and for all. As an example, our architecture provides a single PKCS#11 module that works with all the card devices for which a plugin has been implemented. This module, for example, can be plugged at the lowest levels of the CDSA architecture.

On-board fingerprint verification

This chapter presents a hybrid fingerprint matching algorithm combining two heterogeneous schemes, namely the texture-vector and minutiae-based methods. The proposed technique has been designed in order to run on a programmable smart card, with image processing and feature extraction performed on the host, and matching performed by the card device. The two matching algorithms have been carefully tuned in order to achieve an acceptable performance despite the computation and memory constraints. Given the high level of intrinsic security that smart cards already have, and the interactive nature of target applications, the complexity of the problem has been greatly reduced, making such an approach feasible. This is validated by the experimental results we show, gathered from an implementation onto a Java Card device, where acceptable false acceptance and rejection rates are achieved at the cost of a reasonable response time of the device.

Introduction

User authentication is one of the most important issues when designing a secure system. Traditional password based solutions, relying on the concept that a user is authenticated by proving knowledge of a secret information, usually offer an unacceptable security level. In fact, the secret information can easily be revealed to (or stolen by) unauthorised users. If the password is not strong, it can also be easily guessed by an attacker. Use of smart

cards, along with cryptographic authentication protocols, increases security by requiring a user to prove both possession of a physical card, containing a cryptographic key, and knowledge of a secret information, usually a Personal Identification Number (PIN) protecting the card (two factor authentication). This raises the security level with respect to remote attackers, but still it is subject to the problem of voluntary delegation, or card stealing / PIN extortion. Biometrics based authentication techniques solve this problem, by requiring the user to prove possession of a unique, characteristic property of his own body, such as fingerprints ridges, hand shape, retina, etc... When such a technique is used in conjunction with smart card technology, a high security level is achieved since users are required to prove, at the same time, knowledge of a secret information, possession of a physical token, and possession of their own physical body (three factor authentication), before access to a system is granted.

This work is focused on systems where the authentication mechanism relies on the cryptographic capabilities of the card, and *fingerprint verification* is used by the card, in addition or alternative to PIN code verification. An alternative target is a secure application running entirely or in part onto a smart card, where the card itself authenticates users. A typical target application is smart card based digital signature, where the non repudiation property, usually established only at a jurisdictional level by dictating card owner responsibilities, can be technically enforced by requiring a biometrics authentication by the card, before the signing operation takes place.

This chapter is organised as follows. Section briefly reviews works found in literature related to fingerprint verification. Section features an overview of the proposed technique, with a detailed description of the matching mechanism that has been implemented on the card device. Evaluation results for the proposed algorithm are reported in Section . Specific notes about the algorithm implementation are reported in Section . Finally, Section draws conclusions and presents possible areas of future investigation.

Related work

In recent years, the problem of merging smart card technology and biometrics for the purpose of authenticating users has gained more and more attention from research and industry altogether. Smart card based authentication has been widely used whenever user authentication was required, though the result has always been the authentication of the plastic card itself, not the user. Biometrics promise a final solution to this problem, achieving an integrated authentication system in which not only a user is authenticated by proving possession of a physical token and knowledge of a secret information, but by showing to the system some unique biological characteristics of its own body.

Correct use of biometrics and smart cards is not as immediate as it could seem at a first glance. Recent works [23, 40] focused on the possible attacks a system integrating such technologies could be subject to. In [40] eight types of attacks to a biometric authentication system are identified, targeted either to the components themselves, or to the communication protocols among them. Recently, the European Union has also focused attention on feasibility of matching-on-card technologies, as in [48], where it is underlined that, in the context of electronic signatures, the possibility of identifying people based on biometric characteristics is of fundamental importance due to the non-repudiation security requirement, and the need for on-card matching is also outlined.

Feature extraction from fingerprint images has been widely studied, as shown in [32], where a good overview is made on the general structure of automatic fingerprint identification systems (AFIS), emphasizing the main challenges such a system has to face with. In [24] the authors demonstrate how an image enhancement algorithm based on Gabor filters can significantly improve the performances of an AFIS thanks to the greater reliability and precision gained by the minutiae extraction process, which leads to a reduced False Rejection Rate (FRR) for a given False Acceptance Rate (FAR). In [39], Prabhakar proposes an innovative approach to fingerprint

analysis & matching, based on the use of a Gabor filter bank to extract from the fingerprint image statistical information, which have been proven to degrade much more smoothly with image quality than classical minutiae-based algorithms. This approach has further been developed in [43] in order to achieve comparable performances even in conjunction with small sensors, which offer to the analysis system only a limited portion of the fingerprint. In the same work, the authors opened a relatively new investigation direction inspired from the so called *multi modal* biometric verification techniques, where multiple kinds of a person biometric characteristics are used at once for the purposes of authentication. Due to the independence between the different kind of biometric information that is matched in such techniques, a combination of them results in a higher performance, as shown in [42].

The specific problem of combining two fingerprint matching algorithms in order to improve performance is addressed in [34], where the authors compare three different ways of combining the scores obtained from distinct matching algorithms (a *linear* combination, a *multiplicative* combination and a combination based on the *logistic* function) and demonstrate that the best performance is achieved with the logistic function.

With respect to previous investigations on hybrid fingerprint matching, the approach which is being introduced in this chapter is specifically focused on the problem of implementing such techniques onto programmable smart card devices. It does not aim at achieving the highest possible performance, but achieving an acceptable performance for the cited usage context, while keeping a sufficiently low complexity level so to allow implementation onto a programmable card device. We give an extensive description of the adopted algorithms, and a precise specification of how various parameters have been tuned in the implementation. Furthermore, we present an on-card architecture for the matching algorithm, realised as a consistent extension to the protocol and JavaCard Applet introduced in [17], and report experimental timings gathered from the execution of the proposed algorithm onto a JavaCard device.

On a related note, fingerprint matching on JavaCard devices is not novel,

as industrial products already exist based on the same kind of technology, like the one from Precise Biometrics [38]. However, implementation details and extensive description of the experimental setup from which such measurements arise are not available, making it impossible to perform a comparison with other approaches.

Hybrid matching

Our system uses both Prabhakar's *fingercode* and minutiae information in order to perform a multi modal biometric verification of the user. Both techniques have been split into the two fundamental steps of feature extraction and feature matching. Thus, the live-scanned fingerprint image is first analysed on the host machine in order to extract the features using the two relatively complex feature extraction algorithms; such features are then transmitted to the smart card device, which performs the matching phases of both algorithms, comparing the received features with the templates previously stored into its internal memory during enrollment.

In the following, we report a description of the extraction and matching algorithms adopted for the two techniques.

Features Extraction and Representation

Fingercode

Fingercode extraction has been implemented following the method described in [39], where an exhaustive description of the algorithm can be found. Briefly, it consists of the following steps (see Figure 26). First, the fingerprint image is normalised to a constant mean and variance, then a reference point (*core*) is determined, defined as the topmost point on the innermost upward ridge. A circular *region of interest* around the core is then tessellated and filtered using eight Gabor filters, tuned over eight different directions. For each of the filtered images, and for each tessel, the intensity absolute deviation from the mean is computed. The complete list of such absolute

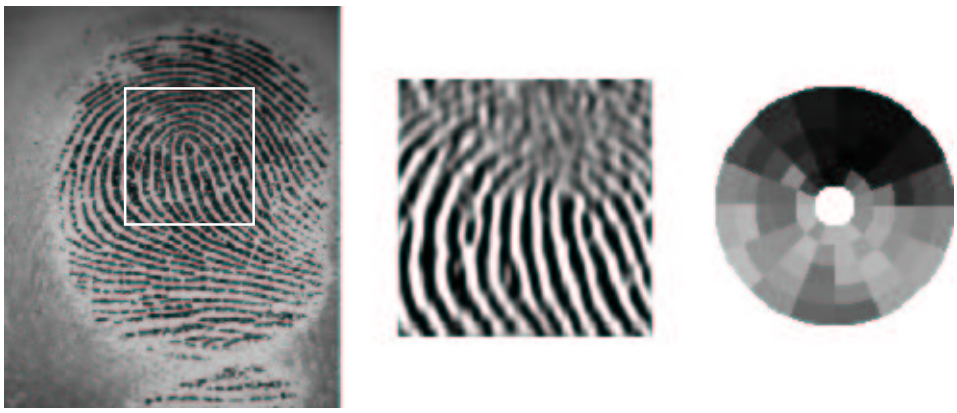

Figure 26: Example of Fingercode computation: a *region of interest* is determined around the core, then it is directionally filtered (only vertical filtering is showed), tessellated and intensity absolute deviations (represented in gray scale) are computed.

deviations, normalised in the range [0..255], constitutes the *fingercode* of the original image.

Minutiae

In order to extract the minutiae from the fingerprint image, we adopted the algorithm supplied by the National Institute of Standards and Technology (NIST) as implemented in the NIST Fingerprint Image Software (NFIS), a public domain software developed for the Federal Bureau of Investigation.

This algorithm can be roughly subdivided into the following phases [16]. First, the gray-scale fingerprint image is reduced to a binary, black and white one, then analysed in order to find every singular point (bifurcations and terminations) which could be a potential minutia. This operation results in false positives (minutiae detected where none exists), due to low-quality image, cuts, bruises or other *noise*. Thus, various heuristics are applied to discover and delete such false positives (for example two facing and aligned minutiae

[16] For further details the reader is referred to NFIS official documentation, freely distributed by NIST (*http://www.nist.gov*).

at small distance are likely to be due to a cut determining two false terminations). Finally, for each of the remaining minutiae the algorithm outputs the position, direction (defined as the main direction of the surrounding ridge flow) and an index of reliability, determined considering multiple factors such as local image quality and proximity to image borders. Our system excludes from further analysis the minutiae with a reliability index under a given threshold. The others are ordered based on increasing distance from the core, where only the nearest num_{m} ones, up to a maximum number of *maxMinutiaeNumber*, are considered, so to limit computation requirements for the matching phase.

Let $\{m_i\}_{0 \leq i \leq num_{\mathrm{m}}}$ denote the set of found minutiae, where m_0 is the fingerprint core. The algorithm builds a graph representation of the minutiae, where each minutia m_i is associated a node n_i in the graph, and the set of outgoing arcs from a node n_i represents the set of minutiae which are considered *neighbours of* m_i for the purpose of matching. The following algorithm builds the graph:

1. determine the bounding-box of the minutiae set;
2. let ref_i be the number of references to m_i, initially 0;
3. let p be the list of pending minutiae; initially p contains m_0;
4. let c be the list, initially empty, of *consolidated* minutiae;
5. extract next minutia m_{curr} from p, and add m_{curr} to c;
6. enumerate m_{curr}'s nearest neighbours, given the following restrictions:
   a) a maximum of max_{neigh} neighbours can be listed, where max_{neigh} is n_{c} if $m_{\mathrm{curr}} \equiv m_0$, n_{m} otherwise;
   b) minutiae in c are ignored;
   c) neighbour's distance from m_{curr} must be in the range $[dist_{\mathrm{min}}, dist_{\mathrm{max}}]$; we do not accept a neighbour too close because at small distances

even light errors in position detection can determine large variations in direction when expressed in polar coordinates; on the other hand we can't accept too far neighbours because at large distances the elastic deformation of finger's skin couldn't be ignored;

7. for each neighbour m_j found

   a) associate to m_{curr} the corresponding *vector* (i. e. distance and direction from m_{curr} to m_j and the index j);

   b) increment ref_j; if $ref_j = max_{\text{ref}}$, add m_j to c;

   c) add m_j to p, if not already present;

8. if p is not empty, continue from step 5.

If $max_{\text{ref}} = 1$, the graph becomes a *spanning tree* touching every minutia in the set; the choice to allow multiple references ($max_{\text{ref}} > 1$) to the same minutia is due to the necessity to give the graph enough redundancy, which (as discussed in Section) reduces the probability of erroneous early abort by the matching algorithm when comparing two corresponding fingerprints. On the other hand, max_{ref} has to be lower than max_{neigh}, otherwise the graph could result in a strongly connected, central cluster of nodes which does not reach outer minutiae[17].

The representation of a fingerprint, as output by the described process, is composed of: the bounding box coordinates; the found minutiae list $\{m_i\}$, including, for each minutia, its Cartesian coordinates (relative to the core), direction and list of vector-distances to its neighbours.

[17] It should be noted that we *do not* guarantee to reach every minutia, but only that the probability of a minutia to be excluded from the graph is sufficiently low.

Matching

Fingercode

Fingercode matching has been implemented as described in [39]: given the two vectors, we compute the sum of absolute differences between corresponding elements and store the result as the score of the process $score_{\text{fc}}$.

Minutiae

The minutiae matching has been inspired by the point-pattern matching algorithm described in [57], with the simplification obtained by computing a common reference point: the core. In the following, we consider two vectors (as defined in Section , step 7a) $v_1(dist_1, dir_1, i_1)$ and $v_2(dist_2, dir_2, i_2)$ to match given the rotation rot and the tolerance parameters th_{dist}, th_{dir} and th_{angle} when:

$$\begin{aligned} |dist_1 - dist_2| &< th_{\text{dist}} \\ |dir_1 - dir_2 + rot|_{360} &< th_{\text{dir}} \\ |m_{i_1}.dir - m_{i_2}.dir + rot|_{180} &< th_{\text{angle}} \end{aligned}$$

where $m_i.dir$ denotes the direction of the i^{th} minutia. These three tolerances have been chosen by performing a statistical analysis of pairs of corresponding fingerprints. This avoids the system to behave too strictly (resulting in high FRR) or too permissively (resulting in high FAR).

The basic task of the algorithm is to find, given the template and the minutiae graphs, a *spanning ordered tree* touching as many nodes as possible, starting from the two cores (which are assumed to be corresponding by hypothesis) and visiting the graphs only via common arches, i. e. the ones corresponding to vectors matching within accepted tolerance.

The algorithm proceeds as follows:

1. Let T_i and S_i indicate respectively the i^{th} minutia of the template and of the proposed set;

2. let m be the list of matches found, composed of couples of indexes, where the presence into m of the couple (i_1, i_2) means that a match has been detected between T_{i_1} and S_{i_2};
3. let p be a list, initially empty, of *pending* minutiae;
4. look for the rotation *bestRot* which gives the maximum number of matches among the two cores' neighbours under tolerances th_{distCore}, th_{dirCore} and th_{angle} (th_{distCore} and th_{dirCore} are less restrictive than their general counterpart to take in account the possible imprecision in core detection); the search has two limitations:
   a) $|bestRot| < max_{\text{rot}}$ (we assume that the user puts his finger approximately vertically);
   b) given two rotations rot_1 and rot_2 which give the same number of matches, the lower one (in absolute value) is preferred;
5. for each minutia S_i for which a corresponding T_j was found during previous step, insert (j, i) into m and S_i into p;
6. extract next pending minutia S_{curr} from p and find into m the corresponding matching template minutia T_{corr};
7. for each vector $v_1(dist_1, dir_1, i_1)$ associated to S_{curr} look for a matching vector $v_2(dist_2, dir_2, i_2)$ associated to T_{corr} with rotation *bestRot* and tolerances th_{dist}, th_{dir} and th_{angle}; for each match found for which (i_2, i_1) is not already into m, add (i_2, i_1) to m and add m_{i_1} to p;
8. if p is not empty, continue with step 5.

Defined $numMin_{\text{t}}$ as the number of minutiae of T lying inside the bounding box of S, $numMin_{\text{s}}$ as the number of minutiae of S lying inside the bounding-box of T, and $numMatches$ as the number of matches found by the algorithm, the score is evaluated as:

$$score_{\text{min}} = 100 \frac{numMatches^2}{numMin_{\text{t}} * numMin_{\text{s}}}$$

Matchers' fusion

Given the two scores $score_{fc}$ and $score_{min}$, the overall score is calculated as a linear combination of them:

$$score = \alpha score_{fc} + \beta score_{min} \quad .$$

If $score$ exceeds a given threshold th_{score} the system considers the proposed fingerprint to be sufficiently similar to the enrolled one and the match succeeds, otherwise the match fails. Coefficients α and β have been determined with an *a posteriori* analysis as the ones which minimise the overall Equal Error Rate (EER) of the system.

Results

Effectiveness of our verification algorithm has been tested by submitting to it pairs of fingerprint images and by measuring its ability to discriminate between corresponding and non-corresponding ones in terms of FAR/FRR curves.

Tests have been conducted on a database of 55 live-scan fingerprints taken on a group of volunteers, with each fingerprint scan repeated ten times for a total of 550 images. The volunteers were completely unaware of biometrics related technology and scanner use, so they have been subject to a training phase of one minute with visual feedback, so to allow them to understand what was the right position and pressure of the finger for a good scan. Then, they have been asked to pose ten times the finger on the scanner in a natural way.

The obtained images have been analysed and matched in pairs using our algorithm, distinguishing matches between corresponding fingerprints (different images of the same finger) from matches between non-corresponding fingerprints. Figures 27(a) and (b) show the obtained joint (based on minutiae and on fingercode) scores' distributions in the two cases. In the two graphs, the X axis reports the score obtained with minutiae matching, which

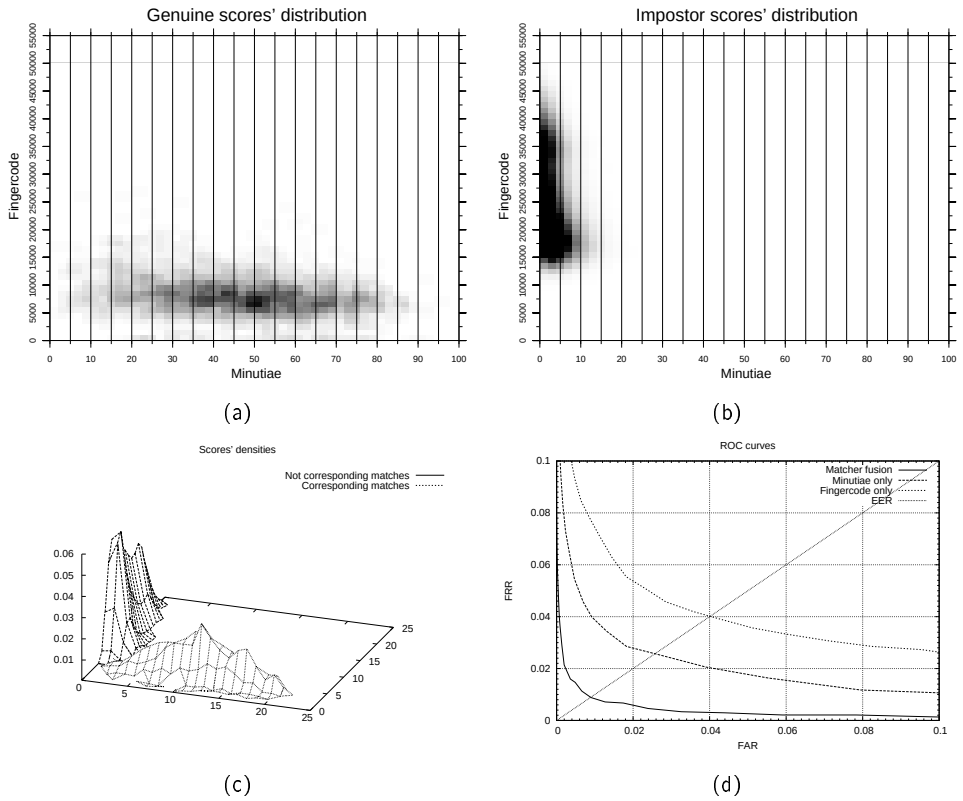

Figure 27: (a)-(c) Joint score distributions for genuine and impostor matches. (d) ROC curves

is higher when a higher number of matching minutiae is detected, while the Y axis reports the fingercode score, which is lower when the live-scan fingerprint is more similar to the on-card template. The same distributions are reported in the 3D plot of Figure 27(c) for convenience.

In Fig. 27(d) we compare the Receiver Operating Curves (ROCs) relative to each matcher separately and to their combination. These curves represent the FAR/FRR pairs that are obtained by continuously varying the score threshold of the matching algorithms. As the picture highlights, the hybrid technique results in a considerable increment of performance when compared

to the results achieved singularly by the two matchers. In fact, in the hybrid ROC curve, the FRR value, for each possible FAR, is consistently lower than those obtained singularly by the two matchers. Furthermore, while the minutiae based and fingercode matchers obtain, respectively, an EER of about 2.3% and 4%, the combined matcher obtains an EER of about 0.8%. The combined matching algorithm requires an on-board computation time of about 11–12 seconds, 4 of which due to the upload of the biometric data onto the card.

Implementation notes

The described biometric authentication system has been developed, on the host side, as an extension to the MUSCLE Card [18] middleware, and on the card side as an extension to the MUSCLE Card JavaCard Applet. This framework defines a high level API that smart card aware applications can use to access smart card storage, cryptographic and PIN management services in a unified, card independent way. The framework also includes a JavaCard Applet allowing the middleware to use the on-card services by means of the protocol described in [17]. Briefly, the framework allows applications to manage on-board data containers (objects), cryptographic keys, and PIN codes. A security model allows to protect, on a per object and a per operation basis, objects and keys, by means of Access Control Lists (ACLs).

An extension mechanism has been embedded in the framework so to allow applications to enhance the basic protocol and Applet in order to support application specific extensions. Biometrics based authentication has been embedded in this context by allowing the access to on-card resources (e.g. reading an object or using a cryptographic key) only after a successful on-board fingerprint verification. Furthermore, the existing access control mechanism allows, by using ACLs, the possibility to combine the new authentication mechanism with traditional PIN based or challenge-response cryptographic based authentication. The used fingerprint scanner is FX2000

USB, by Biometrika s.r.l. (http://www.biometrika.it), providing a portable development kit and API for access to the acquired biometric data. The development platform has been a RedHat 7.3 Linux system.

Conclusions and future work

In this chapter, a hybrid fingerprint matching mechanism has been introduced, designed with the aim of running onto a programmable smart card. Experimental results showed that, by taking advantage of the simplifications inherent to the application context and using ad-hoc designed data representations, it is possible to realise an on-board hybrid fingerprint matcher with an acceptable performance, even into such scarce-resource devices as programmable smart cards, maintaining reasonable execution times. In a short future, it is scheduled to undertake investigations related to the feasibility of on-board multi modal authentication based on alternative means of biometrics.

A middleware for digital signatures

Preface

This chapter presents an open approach to the design of a modular middleware for digital signature (DS), aiming at easing integration of such technology within applications. Despite the simplicity behind the pure concept of digital signature, namely the use of public key cryptography for verifying authenticity of documents, implementation of that concept involves knowledge and perfect adherence to various standards and technical regulations. This is especially true for applications that need to integrate a DS scheme in compliance to the legal framework of a country, where also national rules, laws and directives need to be considered in the process of computing and verifying signatures. Thus, adoption of one of the Software Development Kits (SDK) for DS provided by specialised software houses is a perfectly reasonable choice for the application developer. Unfortunately, the software components provided by different vendors are not interchangeable, because they are based on proprietary middleware architectures and Application Programming Interfaces (API).

The approach that is being introduced in this chapter is based on the design of an open middleware for DS services, which can be used by application developers with the benefit of being leveraged from the burden of a

detailed understanding of a DS policy, still being an interoperable solution that will work with other DS providers. In our opinion, such an approach has a potential in encouraging and speeding up integration of law compliant DS services within document and work flow management systems, sustaining DS technology to demonstrate its full potential in eliminating paperwork from public and private agencies.

Introduction

Today various applications embed functionality for the use of digital signatures in order to authenticate user data. For example, most e-mail clients, in addition to the traditional MIME format, are able to manage the S/MIME format for exchange of signed and/or encrypted messages. Though, the support for such functionality is usually very limited, in that the process of signature computation is just limited to the use of any PKCS#11 [45] or Cryptographic Service Provider [37, Level 6] (CSP) module that is installed onto the system, and the process of verification is limited to the cryptographic verification of the message and public key certificate, plus a check on its expiration period. A few applications perform a correct management of the Certificate Revocation Lists (CRL) made available by Certification Authorities (CA), use the On line Certificate Status Protocol (OCSP), check the key usage or any other extensions that a CA has embedded within a certificate.

Even if such operations suffice for a general use of DS for the purpose of authenticating the incoming and outgoing messages and documents, it is far from a perfect adherence to the security policy that a CA may want to enforce, whenever its certificates and keys are used. This is especially true when DS technology needs to be used in conformance to what dictated by national or international laws in order for the computed signatures to possess a legal value. In fact, depending on the national law framework, additional operations are needed by DS computation and verification software, in order to be perfectly compliant to the legal system in force. For example,

during signature computation, it is commonly required that the software verifies the validity of the user certificate. Another requirement could be the verification that the document complies to particular standards, e.g. it is in one of the recommended formats, and does not contain dynamic contents. Furthermore, verification of signatures may involve several complex operations to be performed, such as: downloading of up to date CRLs, running OCSP protocols with CA servers, retrieving and verifying public key certificates of other CAs, verifying specific extension fields, checking that signatures have been computed using the recommended hash algorithms, etc.... Furthermore, when verifying the digital signature of an electronic document long after the signing operation, it is common to find out that the subscriber's certificate has expired. In such cases, whether a verify operation should reject the signature or not, depends on the particular security policy enforced by the CA and, if the CA is a national law compliant one, by technical regulations issued by the government. In fact, in such a case, correct verification of the digital signature onto a document constitutes a proof of its authenticity and a pre-requisite for its legal validity. Sometimes the verification software may need to verify additional time stamps that may be present in the document, increasing its legal validity beyond the certificate expiration date, thus increasing the number of certificates, certificate issuers, and digital signatures involved during the verification process.

In this chapter, we propose an open architecture for digital signatures, aimed at providing an open API for the access to digital signature services from applications. In our opinion, its adoption will ease the use of such services from within applications, decoupling the functionality that is specifically tied to the management of digital signatures from the one that is application specific. An open source implementation of the middleware is currently under way, along with a digital signature application, and a plugin for integrating italian law compliant DS.

Conversely to existing open middleware for security services, such as the Common Data Security Architecture [35] or the Open Card Framework [36], the introduced approach focuses on issues introduced by the use of DS

services, expecially in relation to PKIs for digital signatures supported by a legal framework for the purpose of guaranteeing the long term conservation of a signed electronic document.

From now on, the set of software requirements dictated by the technical rules of a legal framework, PKI or CA security policy, on the signature computation and verification processes, will be referred to as *digital signature policy*.

The need for open architectures

From what stated so far, it is clear that even a simple document management system, used inside a small PA office, needs complex functionality when dealing with digitally signed documents. This is especially true when the signature on the documents is not merely used as a means for authenticating the sender in a communication, but it is used as a proof to a third party of the will of the subscriber to sign the document contents. This usually requires adherence of the signature computation and verification processes to specific technical regulations enforced by legislative decrees, in order to recognise to digitally signed documents the same legal strength as handwritten signed ones.

Usually, delegating external software components developed by specialised software houses is the optimal choice for application developers, who are leveraged from the burden of dealing with detailed and specific operations that may be needed by the DS policy the application needs to comply. The current way of achieving this separation of functionality is, today, the adoption of one of the existing Software Development Kits (SDK) for DS, made available by the CAs themselves or other software houses. These solutions are usually characterised by the provision of components which enable an application to integrate use of DS services through the use of a simple API, and hiding as much as possible the specifics of a DS services implementation to the final developer. Also, such solutions are usually characterised by the availability of the APIs in various programing languages, such as C/C++,

Java and Visual Basic, through the use of Microsoft COM+/ActiveX technology, which usually also make it possible an easy integration of DS services inside web based applications.

Despite the good principles of inspiration of such solutions, the major drawback in their adoption is the fact that they are based on proprietary architectures, and provide DS services through proprietary interfaces. This ties the final application to the specific DS-SDK provider. Furthermore, a recurrent issue is that the provided solution is usually dependent on a specific CA, and does not inter operate with other authorities[18]. This results in tying the final application to a specific CA or SDK provider. This situation discourages integration of DS services inside applications, what hinders the real potential of such technology in achieving a real *paperwork elimination*, when combined with appropriate legal laws and technical regulations.

The approach proposed in this chapter is centred around the definition of an open and modular architecture for DS services. It mainly provides two APIs, one to be used by application developers, the other by DS policy providers. The former API may be used by application developers for integrating DS services complying to a certain DS policy with no prior knowledge of the specifics of that policy. The latter API is reserved to policy providers, who are supposed to develop pluggable components that fit into the architecture in a transparent way for applications.

National background

In Italy, digital signatures (DS) became equivalent to handwritten ones since year 1997, when the presidential decree n.513/97 [2] stated the fundamental equivalence of the two types of signature, provided that proper technical rules be respected. The first formulation of the DS law, inspired by the great optimism around this new technology that was promising to leverage

[18] This is due sometimes to the usual choice, for each provider, of a specific software stack for connectivity with external smart card devices, which, despite their adherence to the PKCS#11 [45] or PCSC Level 6 [37] standards, have little differences which make hard for higher level software to support all of them

the overall PA document production from the burden of the chapter physicality, was very generic: it contained such simple definitions as certificate, certification authority, digital signature, certificate revocation and suspension, without caring of any technicality involved in generation, storage and management of keys. It also defined the *authenticated* digital signature, when the operation of signing was supervised by a notary official. Immediately, DS technology was embedded inside a technical regulations [3] for archiving of PA documents on optical storage media. The first technical rules for DS were issued by the Authority for Information technologies in Public Administrations (AIPA) two years later, as the DPCM 8 Feb 1999 law [5]. They introduced the concept of *signing device*, defined the algorithms to be used for signing (RSA or DSA) and hash computations (RIPEMD-160 or SHA-1 [56]), along with the minimum acceptable key length (1024 bits); they introduced three different key types (subscription, certification, and temporal validation keys), and dictated constraints on the computer systems dedicated to the generation of key pairs (compliance to ITSEC E3 security level); they dictated formats to be used for storage and exchange of certificate and revocation list data; also, a maximum duration of three years was defined for subscription certificates. Among others, they introduced the possibility of substituting the full personal data of the owner with a *pseudonym*, inside certificates.

Subsequently, in year 2000, the original equivalence was reviewed, integrated and corrected inside the presidential decree n.445/00 [7], known also as *Testo Unico*, a first try in adjusting the national regulations to the European Directive on Electronic Signatures [4]. Also, in year 2000 AIPA issued guidelines for the interoperability of CAs [6], stating, for example, rules to be followed in formatting distinguished names for citizens (Surname/Name/Fiscal Code/ID) and relevant X.509 v3 [25] extensions to be used (Key Usage, Extended Key Usage, Certificate Policies, ...), inside X.509 subscription, certification and temporal validation certificates. The EU directive on electronic signatures has been more consistently integrated in the national legal framework with the actuation legislative decree n.10 23-

January-02 [10], and the presidential decree n.137/03 [11], which highlight a much higher level of complexity with respect to the previous law formulations. Various types of signature are introduced: an *electronic* signature (ES) is defined as any means of ICT based authentication of electronic data; an *advanced* ES is one which "guarantees univocally the connection between the generated signature and the subscriber"; a *qualified* ES is an advanced ES based upon a qualified certificate and created by using a secure device for signature creation; finally a *digital* signature is introduced as a "special type of *qualified* electronic signature". Furthermore, various CA types are introduced: a *certificator* is a generic CA, a *qualified* certificator is one which is compliant to the requirements dictated by national DS laws; an *accredited* certificator is one which is compliant to the EU directive on electronic signatures. *Qualified* certificates are defined as those issued by qualified CAs. Signature computation devices are also subdivided into generic signature creation device and *secure* ones (those meeting requirements dictated by EU directive on ES). Finally, latest upgrade is due to the new technical rules for digital signatures [1], not yet approved, which introduce a few changes: they eliminate the three years limit for the duration of a subscription certificate, they explicitly address the problem of dynamic contents inside digital documents, nullifying the "automatic" legal validity of documents in such cases, even in presence of an "advanced signature based on a qualified certificate, computed using a secure creation device", and others.

Long term validity of digitally signed documents

Another challenge in the use of digital signatures for legally valid documents is the permanence of the legal validity beyond the expiration date of the subscriber certificate, or even beyond the complete out date of the technology used for signing, due to an improbable unpredicted increase in computation power. Clearly, a much powerful attacker, in the future, would be able to recover any private key that she wants, by using a trivial brute force attack, thus being able to create as many false signatures as she wants.

As long as such issues are concerned, laws in enforcement only stated that, even if the user certificate is expired or has been revoked or compromised, the signature is to be considered as valid if there exist a proof of existence of the signed document before the date of expiry or revocation. A directive from AIPA [8] in year 2001 suggested that this proof be obtained by the addition of a time stamp to the signed document, before its expiration date. The same document suggested to repeat the time stamp operation as long as needed, at each expiry of the certificate relative to the previous time stamp. This way, the chain of "updated" time stamps, along with a proper process for verifying them, guarantees authenticity of the signed data. Even in presence of a powerful attacker who manages to recover an "old" signing key, falsification of a signature is impossible due to the impossibility for the attacker to falsify the subsequent time stamps, supposed to be based on much stronger schemes than the original one.

An alternative to such a complicated scheme is supposed to be introduced by the new technical rules [1], if they will be approved. In fact, they state that a single time stamp suffices for guaranteeing long term validity of the signed document, and that the time stamping authority must keep into an archive all computed time stamps for a minimum of five years, and also for longer periods, on request of the citizen. This way, TSAs are delegated for providing the proof of existence of a document cited above.

Project overview

As highlighted in previous section, the current scenario for legal validity of digital signatures within Italy is quite different with respect to the original formulation in year 1997. More importantly, it is quite complex, in that, among others, verification software is supposed to properly check possible time stamps embedded into a signed document, and is supposed to behave properly with respect to signatures created before expiration or revocation of a non valid certificate.

The exact behaviour of the software during computation and verification

of signatures on documents and other types of files is highly dependent on the security policy of the CA that issues the signature certificates. In this chapter, we propose a modular approach in which each CA may provide its own "plugin" that implements its own DS policy specific behaviours. Application developers only need to use a simple API to ask to the middleware if a given signature onto a digitally signed document is valid or not, according to a DS policy of its choice, among the ones available (installed) in the system the application is running into. This approach is not necessarily restricted to legally valid digital signatures: any PKI may provide its own plugin for tuning the exact behaviour of the software providing DS services to applications (for example, in Italy, a user could have installed plugins for both law compliant CAs, and EuroPKI, and OpenCA infrastructures).

This section makes an overview of the proposed architecture and APIs for application and plugin developers.

System architecture

DS functionality that our middleware exposes to applications can be summarised as (see Figure 28): signature computation, possibly through the use of external cryptographic devices; enforcements of mechanisms that allow the long term validity of a digital signature after expiration or revocation of the signing certificate, if allowed by the DS policy; verification of signatures on documents, possibly involving retrieval and verification of certificates, CRLs and CSLs.

The envisioned architecture is composed of various components, as shown in Figure 29:

Signer this component deals with the actual computation of digital signatures, possibly through the interface with external devices such as smart cards

Verifier this component is the one that verifies digital signatures, either on public key certificates or on digital documents

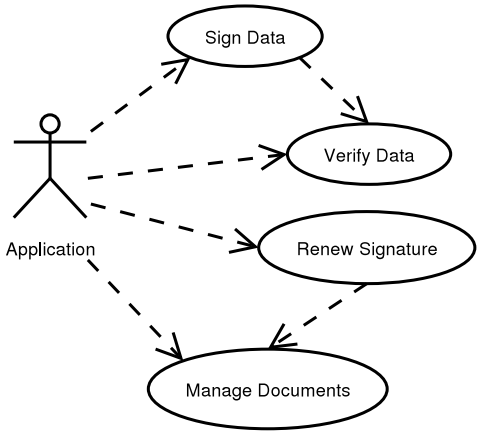

Figure 28: Use case diagram for application requirements

Long term signature validator this component may be used either explicitly by applications, or internally to the middleware, for the purpose of undertaking necessary actions

CRL/CSL/OCSP Manager this component is responsible for retrieving up to date certificate status information by periodic download of Certificate Revocation and Suspension Lists (CRL/CSL), and/or connection to OCSP servers; status information is needed by the Verifier module in order to check the validity of public key certificates, or by the Signer module in order to check validity of the signing certificate

Certificate Manager this is needed by the Verifier component in order to retrieve on the web public key certificates that may be needed for the purpose of verifying a digital signature, and acts as a local cache for such certificates; furthermore this component is required for the initial installation of trusted certificates that are needed for a correct setup of the verification process

Cryptographic Services this is the low level component embedding all cryptographic operations needed by the middleware in order to work properly;

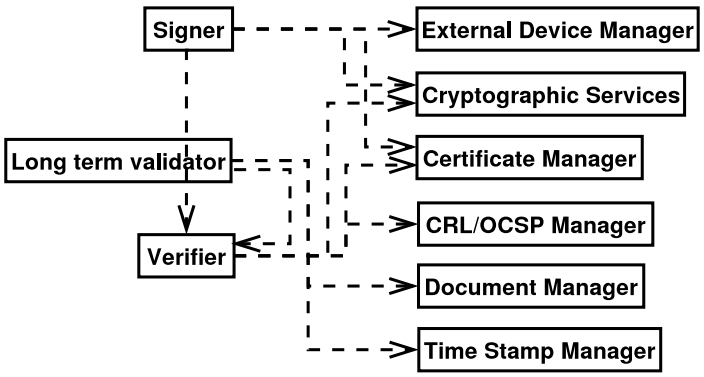

Figure 29: Components of the proposed architecture.

External Device Manager this is the component that directly interfaces with external devices, such as smart cards, through the use of a PKCS#11 standard interface

Document Manager this component may be delegated from the application for the long term storage of digital documents, for the purpose of undertaking necessary actions for guaranteeing long term validity of the attached signatures

Time Stamp Manager this component is responsible for managing time stamps on digitally signed documents, including possible connections to external TSAs for the purpose of getting a time stamp on request by the Long Term Validator module

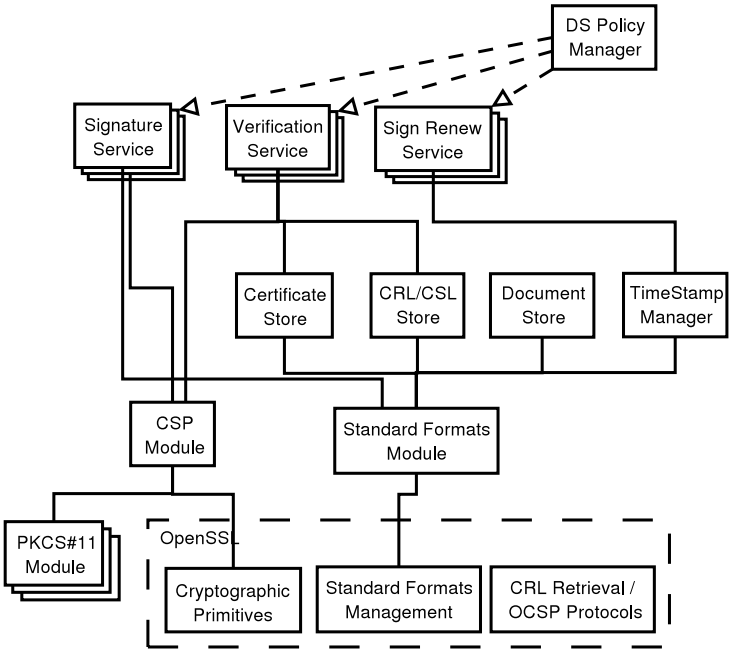

Figure 30: Components of the proposed architecture.

Application of the proposed architecture

This chapter overviews various software components which have been developed in the context of the smart card architecture proposed in the previous chapters. Most of these components have been developed as thesis projects for the Computer Engineering degree at the University of Pisa[19], and they are available as open-source components from the *Smart Sign* project website (http://smartsign.sourceforge.net).

QSign

QSign is an end-user, stand-alone application for digital signatures which has been developed at a high level with the aim of integrating as much as possible the standards commonly used for this kind of application today. In fact, the application is able to compute and verify a digital signature onto a generic file, where the signature is managed according to the PKCS#7 standard. Access to the signature device has been performed by using the most generic and portable API available today for this purpose, namely the PKCS#11 API, which allows access to:

[19] More information at the URL: http://www.unipi.it

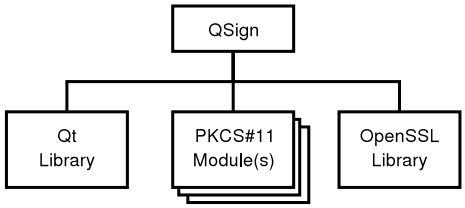

Figure 31: Architecture of *QSign*.

- smart-card devices supported in the introduced architecture, by use of the existing open-source PKCS#11 module developed on the top of the MUSCLECard API;

- other smart-card devices possibly supported onto an open system by use of a vendor-supplied PKCS#11 module, which also allows integration with the vendor-supplied security services, when applicable (as an example, Schlumberger provides a proprietary PKCS#11 module for Linux);

- software-only cryptographic keys, by use of a software-only PKCS#11 library which computes cryptographic operations on the host PC, and manages cryptographic material into some database or files (as an example, the GPKCS#11 project includes a module of this kind).

QSign is a GUI-oriented application (see Figure 31), where the Qt library has been used for developing the graphical user interface, which gives to the application a high degree of portability among open platforms, where such library is available as an open-source project. Furthermore, files in the PKCS#7 standard format have been managed with the help of the OpenSSL library, which is one of the most widely available open-source libraries today onto open platforms.

JMuscleCard

JMuscleCard is a software component allowing the use of the API introduced in Section from the Java language. The smart card middleware introduced so far has been developed entirely using the C language, what gives it the advantage of being a light component of the system, with low resource requirements, in terms of both memory and CPU. *JMuscleCard* has been developed for the purpose of making the services available through this middleware available to Java developers too, what gives the opportunity to easily integrate such services into Java stand-alone and web-based applications (Applets). An example of the use of such component would be within a workflow web-based application developed through Java Applets, where access to the smart-card device by use of *JMuscleCard* could enable smart-card based digital signature services into the application.

As Figure 32 highlights, *JMuscleCard* is basically composed of two bridge subcomponents, one written in the Java language and the other in the C language. The Java Native Interface (JNI) has been used for intra-language communications between the two subcomponents. On the top of *JMuscleCard*, a Java version of the API proposed in Section has been designed, where as few changes as possible have been made to the basic use of the API in order to allow a correct use of it by a Java application. For this purpose, two main Java classes and a few other auxiliary classed have been defined:

- the `JMuscleCard` class encapsulates access to all of the services available through the use of the API; basic functionality of the class is allow the listing of available smart cards accessible on the system
- the `MSCTokenConnection` class encapsulates a connection to a card device, and allows the use of all the on-board available services, i.e. management of PINs, objects and cryptographic keys
- the `MSCTokenInfo`, `MSCStatusInfo`, `MSCObjectInfo`, `MSCKeyInfo` and `MSCGetCapability` provide information on available cards on the

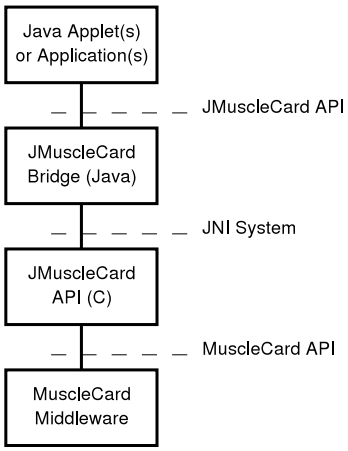

Figure 32: Position of the *JMuscleCard* component with respect to the proposed smart-card architecture

---

system and on the available resources on a card; the `MSCCryptInit`, `MSCGenKeyParams`, `MSCKeyPolicy` classes are needed for parameter passing within the cryptography-related methods of the class `MSCTokenConnection`; `MSCObjectACL`, `MSCKeyACL` and `MSCPinMask` classes are needed for parameter passing within the access-control related methods of the class `MSCTokenConnection`.

JC Emulator

JC Emulator is an emulator for a JavaCard enabled smart-card device that allows to develop a smart-card application interacting with such kind of devices without any need to actually interface with a physical device. With respect to the standard emulator provided by SUN within the Java Card Development Kit, this emulator is designed in suc a way to integrate seamlessly into the smart-card architecture proposed in this thesis, and adds cryptographic functionality that is not available into the package provided by SUN.

JC Emulator is composed of two main components:

- a fake serial smart-card reader, written in the C language, which plugs into the PCSC-Lite projects as a normal reader, but is basically a gateway forwarding all APDUs to and from the second component;

- the emulator server, which is written in Java, so to allow execution of custom Java Card Applets within an emulated Java Card Virtual Machine.

The emulator embeds an implementation of all the standard classes and services available to on-card Java Card Applets through the use of the Java Card API [50], comprising an implementation of the cryptographic classes through the use of the Java Cryptography Extensions (JCE) available on the host machine.

For the sake of simplicity, the Applet loading process has not been emulated. Rather, compiled Java Card Applets must be placed into a particular folder, so to be recognized by the emulator as loaded programs before instantiating them the first time. One of the main advantage of using *JC Emulator* in the smart-card software development cycle is that it is completely transparent to the smart-card solution writer. In fact, no changes are required on the software in order to switch from the testing phase to the production phase, as, from the host side, connecting to the emulator is exactly the same as connecting to an external smart-card device, and, from the card side, a pure Java Card Applet is perfectly suitable for being tested through the emulator, even if its life is constrained within the emulator. However, such constraint has been eliminated by providing a support for serialization of the entire image of a smart-card status onto a file, if wanted. The Applet writer is only required to declare its own classes as implementing the `Serializable` Java interface, in order to take advantage of the feature. Clealy, such declaration must disappear in the final on-card version of the Applet.

Within the development cycle of a Java Card based solution, the use of an emulator is of great benefit due to the fact that it is almost impossible to make a complete and exhaustive test of all the functionality of the secure

solution. In fact, most solutions of this kind require the device to block under certain conditions, and, for the most secure applications, once the physical device has blocked, it can only be thrown away. This makes somewhat cumbersome the testing phase, which is usually conducted by relaxing some blocking features of the solutions, which are again activated only in the final production phase, thus actual testing of the final solution is not always complete as it ought to be. The use of a card emulator greatly simplifies the problem, allowing the virtual creation and destruction of as many card devices as required for a complete test within all the stages of the software development process.

Acknowledgments

This work has partially been the result of a cooperation between the MUSCLE project by David Corcoran and my Ph.D. research project at "Scuola Superiore Sant'Anna" (Pisa, Italy)[20]. It has been sponsored on behalf of Schlumberger Infosec[21] and great contributions were given by SchlumbergerSema. I would like to thank all people who allowed this cooperation, including David Corcoran, William Macgregor and Prof. Paolo Ancilotti. I would like to thank all those people who helped with design and implementation issues, such as Krishna Ksheerabdhi, Jean-Luc Giraud, and many other industry professionals from varying smart card and technical companies.

[20] The PhD has been sponsored by ICube s.r.l. (http://www.icube.it)
[21] Further information at the URL: http://www.slb.com

Table of acronyms

Acronym	Meaning
ICC	Integrated Circuit Card i.e. a smart card
ISO	International Standard Organization
PKCS	Public Key Cryptography Standard by RSA Labs
PCSC	Interoperability Specifications for ICCs and Personal Computer Systems by the PC/SC Workgroup
PC	Personal Computer
MUSCLE	Movement for the Use of Smart Card in the Linux Environment
OS	Open Source
ACL	Access Control List
ACW	Access Control Word
API	Application Programming Interface
PDU	Protocol Data Unit
APDU	Application Protocol Data Unit
PKI	Public Key Infrastructure
DS	Digital Signature
CA	Certification Authority
CRL	Certificate Revocation List
OCSP	Online Certificate Status verification Protocol

Bibliography

[1] *DPCM proposal: Technical rules for generation, computation and verification of digital signatures*.

[2] *Presidential Decree n.513 10-Nov-97: Rules stating criterion for creating, archiving and transmission of documents by using telematic and computer systems, ...*, November 1997.

[3] *Deliberazione AIPA n.24/98: Technical rules for the use of optical storage media*, July 1998.

[4] *Directive 1999/93/CE: European directive on electronic signatures*, December 1999.

[5] *DPCM 8-Feb-99: Technical rules for creation, transmission, ..., of digital documents ...*, February 1999.

[6] *Circolare AIPA/CR/24: Guidelines for interoperability of certification authorities ...*, June 2000.

[7] *Presidential decree n.445/00: Unified set of legislative regulations about document management within administrations*, December 2000.

[8] *Circolare AIPA/CR/27: Use of digital signatures within Public Administrations*, February 2001.

[9] Intel common data security architecture reference implementation. http://developer.intel.com/ial/security/, 2001.

[10] *Legislative decree 23-Jan-2002: Enforcing the 1999/93/CE directive on electronic signature*, February 2002.

[11] *Presidential Decree n.137 7-Apr-2003: Rules for the coordination of electronic signatures, in compliance with art. 13 of legislative decree n.10 23-Jan-2002*, April 2003.

[12] D. Atkins, W. Stallings, and P. Zimmermann. *Request For Comments 1991 – PGP Message Exchange Formats*, August 1996.

[13] Riccardo Brigo. Protecting smart card access by on-board biometrics verification. Computer Engineering Thesis. University of Pisa, 2002.

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

[15] David Corcoran and Tommaso Cucinotta. *MUSCLE Card API, version 1.3.0*, 2001.

[16] David Corcoran and Tommaso Cucinotta. MUSCLE cryptographic card definition for java enabled smartcards. http://www.musclecard.com, August 2001.

[17] Tommaso Cucinotta, Marco Di Natale, and David Corcoran. A protocol for programmable smart cards. In *Proceedings of the* 14^{th} *International Workshop on Database and Expert Systems Application (DEXA 2003) – Trust and Privacy for Digital Business ({TRUSTBUS 2003})*, Prague, Czech Republic, September 2003.

[18] Tommaso Cucinotta, Marco Di Natale, and David Corcoran. Breaking down architectural gaps in smart-card middleware design. In *Proceedings of the* 1^{st} *International Conference on Trust and Privacy for Digital Business (TRUSTBUS 2004)*, Zaragoza, Spain, September 2004. IEEE Computer Society.

[19] EMVCo. *EMV 2000 Integrated cicuit card specification for payment systems*, December 2000.

[20] ETSI. *Global System for Mobile Communications (GSM 11.11) – Digital cellular telecommunications systems – Specification of the Subscriber lentity Module*, December 1995.

[21] European Telecommunications Standards Institute. *ETSI TS 102 221 V4.3.0: Smart cards; UICC-Terminal interface; Physical and logical characteristics (Release 4)*, July 2001.

[22] GSA. *Government Smart Card Interoperability Specification: Contract Modification*, August 2000.

[23] Gael Hachez, Francois Koeune, and Jean-Jacques Quisquater. Biometrics, access control, smart cards: A not so simple combination. In *Proc. of CARDIS 2000*, IFIP, 2000.

[24] L. Hong, A. Jain, S. Pankanti, and R. Bolle. Fingerprint enhancement. In FL Sarasota, editor, *Proc. 1st IEEE WACV*, pages 202–207, 1996.

[25] R. Housley, W. Ford, W. Polk, and D. Solo. *RFC 2459 X.509 Public key infrastructure certificate and CRL profile*. IETF Network Working Group, January 1999.

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

[27] International Standard Organization. *ISO/IEC 7816-4: Information technology - Identification cards - Integrated circuit(s) cards with contacts - Part 4: Interindustry commands for interchange. Part 8: Security related interindustry commands*, 1995.

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

[29] International Standard Organization. *ISO/IEC 7816-7: Information technology - Identification cards - Integrated circuit(s) cards with contacts - Part 7: Interindustry commands for Structured Card Query Language (SCQL)*, 1999.

[30] International Standard Organization. *ISO/IEC 7816-8: Information technology - Identification cards - Integrated circuit(s) cards with contacts - Part 8: Security related interindustry commands*, 1999.

[31] International Standard Organization. *ISO/IEC 7816-9: Information technology - Identification cards - Integrated circuit(s) cards with contacts - Part 9: Additional interindustry commands and security attributes*, 2000.

[32] Anil K. Jain and Sharath Pankanti. Automated fingerprint identification and imaging systems. In *Advances in Fingerprint Technology, 2^{nd} Edition*. Elsevier Science, h. c. lee and r. e. gaensslen edition, 2001.

[33] Olaf Kirch. OpenSC - smart cards on linux. In *Proc. of the 10^{th} International Linux System Technology Conference*, Saarbruecken, Germany, October 2003.

[34] G.L. Marcialis, F. Roli, and P. Loddo. Fusion of multiple matchers for fingerprint verification. In *Proc. of Workshop su Percezione e Visione delle macchine, 8^{vo} Convegno dell'Associazione Italiana per l'Intelligenza Artificiale AI*IA02*, Siena, Italy, September 2002.

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

[36] OpenCard Consortium. *OpenCard Framework General Information Web Document*, second edition, October 1998.

[37] PCSC Workgroup. *Interoperability Specification for ICCs and Personal Computer Systems*, December 1997.

[38] M. Pettersson and M. Harris. Whitepaper: Match-on-card for java cards. Precise Biometrics, November 2002.

[39] S. Prabhakar. *Fingerprint classification and matching using a filterbank*. PhD thesis, Michigan State University, 2001.

[40] N.K. Ratha, J.H. Connell, and R.M. Bolle. Enhancing security and privacy in biometrics-based authentication systems. *IBM Systems Journal*, 40(3), 2001.

[41] J.B. Robshaw and Y.L. Yin. *Elliptic Curve Cryptosystems*. RSA Laboratories, December 1997.

[42] Arun Ross, Anil K. Jain, and Jian-Zhong Qian. Information fusion in biometrics. *Lecture Notes in Computer Science*, 2091:354–359, 2001.

[43] Arun Ross, Salil Prabhakar, and Anil Jain. Fingerprint matching using minutiae and texture features. In *Proceeding of the International Conference on Image Processing (ICIP)*, pages 282–285, 10 2001.

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

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

[46] RSA Laboratories. *PKCS-1 version 2.1: RSA Cryptography Standard*, June 2002.

[47] V. Samar and R. Schemers. Request for comments 86.0: Unified login with pluggable authentication modules (PAM), October 1995.

[48] Dirk Scheuermann, Scarlet Schwiderski-Grosche, and Bruno Struif. Usability of biometrics in relation to electronic signatures. Technical Report GMD-Report-118, GMD - Forschungszentrum Informationstechnik GmbH, 11 2000.

[49] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd Edition*. John Wiley & Sons, October 1995.

[50] Sun Microsystems, Inc. *Java Card™ 2.1.1 Application Programming Interface*, May 2000.

[51] Sun Microsystems, Inc. *Java Card™ 2.1.1 Runtime Environment (JCRE) Specification*, May 2000.

[52] Sun Microsystems, Inc. *Java Card™ 2.1.1 Virtual Machine Specification*, May 2000.

[53] TrustCenter. *gpkcs11 - GNU PKCS#11 implementation*, October 2000.

[54] U.S. Department of commerce/National Institute of Standards and Technology. *FIPS PUB 46-3 – Data Encryption Standard (DES)*, federal information processing standards publication edition, October 1999.

[55] U.S. Department of commerce/National Institute of Standards and Technology. *FIPS PUB 186-2 – Digital Signature Standard (DSS)*, federal information processing standards publication edition, January 2000.

[56] U.S. Department of commerce/National Institute of Standards and Technology. *FIPS PUB 180-2 – Secure Hash Standard*, federal information processing standards publication edition, August 2002.

[57] P.B. Van Wamelen, Z. Li, and S.S. Iyengar. A fast algorithm for the point pattern matching problem. Technical Report 1999-28, Louisiana State University, Dept. of Mathematics, 1999.

[58] T. Ylonen, T. Kivinen, M. Saarinen, and S. Lehtinen. *Internet-Draft: SSH Protocol Architecture*. Network Working Group, January 2002.