A Protocol for Programmable Smart Cards

T. Cucinotta, M. Di Natale
Scuola Superiore Sant'Anna, Pisa (Italy)
E-mail: {cucinotta,marco}@sssup.it

D. Corcoran
Schlumberger Smart Card Research, Austin (TX)
E-mail: corcoran@linuxnet.com

Abstract

This paper presents an open protocol for interoperability across multi-vendor programmable smart cards. It allows exposition of on-card 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. The protocol security model has been designed with the aim of allowing multiple applications to use the services exposed by a same card, with either a cooperative or a no-interference approach, depending on application requirements.

Existing protocols for smart card interoperability define powerful and sophisticated card services, intended to be hard-coded into the device hardware. The presented protocol, instead, is intended to be implemented in software on programmable smart cards. By defining simple functionalities, it allows to achieve a small management code that, once loaded onto a card, leaves enough free memory for application data, cryptographic keys or further programs.

A card-side implementation of the protocol has been developed as an open source Applet for Java Card 2.1.x compliant cards. On the host-side, the protocol has been implemented into an open-source, modular smart card middleware, portable among Unix like platforms, that exposes a new smart card API to the upper software layers. Various open source programs have been developed using the new middleware, including digital signature, console login, remote terminal, and card management tools, proving effectiveness of the new protocol in the context of widely used applications, despite its reduced functionalities.

1. Introduction

Data security is one of the basic requirements in computer software design, and it is most commonly achieved by means of cryptographic algorithms and protocols. These techniques do not suffice to guarantee security of applications unless cryptographic keys are securely managed. The most effective means for protecting keys, today, is the adoption of smart card technology, that delegates management of cryptographic keys to external trusted devices. Even if smart cards have been widely adopted and supported on proprietary platforms, they are not being used on open platforms due to the lack of open solutions. On these systems open source libraries and applications allow the use of cryptography for data protection, but the achieved security level is strongly limited because of the use of software-only cryptography. A few solutions exist supporting only one or a limited set of smart card devices.

In our opinion, the reasons for this limitation have been, among others, the differences among devices from different vendors, trying to differentiate products with respect to concurrency, and the lack of open standards for interoperability. The MUSCLE[1] Card protocol, which is being introduced in this paper, constitutes a step toward openness in smart card middleware design and implementation. We believe that this protocol, along with the open framework that is being developed around it for smart card development, will promote adoption of these devices on open platforms.

This paper is structured as follows. Next section introduces common issues involved in smart card interoperability, and section 3 briefly discusses how existing standards approached them. Section 4 presents the new protocol, underlining and motivating the main choices at the base of its design. Section 5 introduces some practical aspects related to protocol implementation, showing how they have been dealt with in the implementation for Java Card devices. Section 6 shows some of the extensions that could be embedded in the protocol to make it suitable for a wider range of applications.

2. The problem of smart card interoperability

In spite of the growing need for smart card integration into applications and the advantages arising from their use, smart card devices are struggling for a wider use in network security applications. This is especially true for open

[1] Movement for the Use of Smart Cards in a Linux Environment.

platforms, where 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. Although many open source programs exist which embed cryptographic services, most of them still lack the support for external cryptographic smart cards due to the complexity in integrating such devices. This complexity exists for several reasons: different types of card devices exist, including storage only, crypto-enabled, GSM enabled, with general purpose CPU, programmable in Java, Assembler, Basic, etc...; in spite of the efforts made by various organizations in defining common standards [3, 1, 5, 8, 7, 6] for interoperability at a protocol level, different card devices have partial implementations with many restrictions, proprietary extensions and uncommon filesystems; many smart cards have closed protocols and functionalities: this makes their use within open solutions impossible; the card life cycle is short, and by the time a device is supported on less common systems like most Unix variations, it ceases to be manufactured.

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 [13] or PCSC [4] ones. Unfortunately, the modules implementing these interfaces are usually provided by card vendors only for those platforms that are considered of interest. Rarely they provide an implementation for open platforms. 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.

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 paper 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.

[2] Public Key Infrastructures.

3. 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 [2] lower level protocols. T=0 and T=1 define, respectively, an asynchronous half duplex character transmission protocol and a block one for exchanging data among an interface device (a smart card reader), and a smart card. Most times the interface device only acts as a gateway between the host and the card, so in the following we will improperly say that protocol data units are exchanged between the host and the card. These protocols require that each action is 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 a response APDU to the host, composed of an optional variable length data field and a mandatory *status word* (SW), reporting success or possible error conditions occurred during the operation.

Probably the most commonly implemented standard protocol for smart cards is the ISO 7816-4 [3]. 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. 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 [5] and ISO 7816-9 [8] 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 [6] 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 [1] 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 [7], 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 [9, 10, 11]. 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 paper.

4. 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 [14] is available for download at the URL: *http://www.musclecard.com*.

4.1. Objectives and design choices

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.

4.2. Protocol command set

With respect to the T=0 and T=1 protocols, standing at the *transport* layer according to the terminology defined in [12], 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 reported in the following sections.

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

Table 1. MUSCLE Card Protocol command set.

4.3. 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 4.6. 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.

The protocol does not provide hierarchic arrangement of objects, nor typed objects, conversely to other approaches [7] 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 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.

4.4. 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 iden-

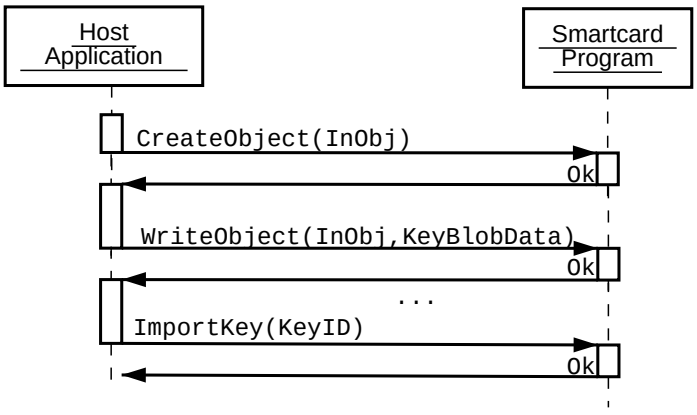

Figure 1. Use of the input object during a key import operation.

tifiers. 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. 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 4.6.

4.5. Input and output objects

Objects have also been used to overcome the T=0 protocol's limitation of 256 bytes per APDU exchange[3]. 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 1). 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. In the latter case it's also up to the application to delete the output object after retrieval of contained information.

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.

4.6. 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

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

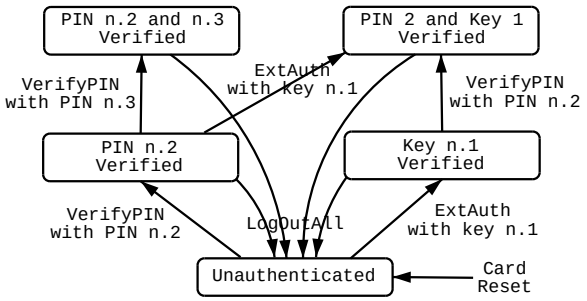

Figure 2. Subset of possible security state transitions allowed by the protocol.

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: 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].

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 2.

Logged identities control which operations are allowed on an object or 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. 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.

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 protection 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.

4.7. 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.

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

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

[5]A static pre-allocation of part of the object space can still be performed

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.

4.8. 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 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 provided implementation of the protocol is a Java Card Applet running into the Java Card Runtime Environment [10], 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.

4.9. 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.

5. 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

by an application by creating a "fake" object with the required size and properly resizing it when additional objects must be created.

integrate smart card technology into the OpenSSH software, an open source implementation of the Secure Shell protocol [16] for secure remote terminal. An open source Pluggable Authentication Module[15] (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.

6. Conclusions and future work

In this paper a new open protocol for smart card interoperability has been introduced, allowing programmable card devices to expose storage and cryptographic services to host applications in a generic way. Design choices and project goals have been described. With respect to existing protocols for smart card interoperability, the exposed one has reduced functionalities, aiming at being implemented in software on programmable devices, like Java Card platforms. 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 on programmable card devices due to the on-board resource constraints. This has been shown by briefly describing some software components developed around the protocol, and showing how some crypto aware applications have been integrated with them and with the protocol, at last. In the authors' opinion, the open protocol specifications, along with the open source components developed, make a step toward simplification in engaging smart card technology into crypto aware applications on open platforms.

Still the efforts have been focused on target applications mainly dealing with user authentication and digital signatures, while security services exist that require special operations to be performed by a smart card and do not actually fit into the protocol's model. One possible investigation direction could be finding those minimal set of operations that could be added as separate commands to the defined protocol in order to extend it and allow deployment of a wider set of security services to applications, while maintaining perfect backward compatibility. Another possibility could be standardization of object identifiers in order to allow applications to share most widely used information onto a smart card, in a fashion that is similar to what PKCS#15 [6] does today with ISO7816-4 filesystem enabled cards. Further, a proper mapping between PKCS#15 file IDs and object OIDs could allow interoperability with PKCS#15 enabled smart cards.

References

[1] 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.

[2] Iso/iec 7816-3: Information technology - identification cards - integrated circuit(s) cards with contacts - part 3: Electronic signals and transmission protocols. 1989.

[3] Iso/iec 7816-4: Information technology - identification cards - integrated circuit(s) cards with contacts - part 4: Interindustry commands for interchange. 1995.

[4] Interoperability specification for iccs and personal computer systems. December 1997.

[5] Iso/iec 7816-8: Information technology - identification cards - integrated circuit(s) cards with contacts - part 8: Security related interindustry commands. 1999.

[6] Pkcs-15: A cryptographic token information format standard. April 1999.

[7] Government smart card interoperability specification: Contract modification. August 2000.

[8] Iso/iec 7816-9: Information technology - identification cards - integrated circuit(s) cards with contacts - part 8: Additional interindustry commands and security attributes. 2000.

[9] Java card^{TM} 2.1.1 application programming interface. May 2000.

[10] Java card^{TM} 2.1.1 runtime environment (jcre) specification. May 2000.

[11] Java card^{TM} 2.1.1 virtual machine specification. May 2000.

[12] Etsi ts 102 221 v4.3.0: Smart cards; uicc-terminal interface; physical and logical characteristics (release 4). July 2001.

[13] Pkcs-11 version 2.1.1 final draft: Cryptographic token interface standard. June 2001.

[14] D. Corcoran and T. Cucinotta. Muscle cryptographic card definition for java enabled smartcards. August 2001.

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

[16] T. Ylonen, T. Kivinen, M. Saarinen, and S. Lehtinen. Internet-draft: Ssh protocol architecture. January 2002.