Secure logging mechanisms for smart cards

Constantinos Markantonakis

Technical Report
RHUL–MA–2001–6
30 November 2001

Department of Mathematics
Royal Holloway, University of London
Egham, Surrey TW20 0EX, England
http://www.rhul.ac.uk/mathematics/techreports

Secure Logging Mechanisms for Smart Cards

by

Constantinos Markantonakis

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

December 1999

Department of Mathematics
Royal Holloway, University of London

To my grandmother,
Areti Valini

Declaration

These doctoral studies were conducted under the supervision of Chris Mitchell and Dieter Gollmann.

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Mathematics as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

List of Publications

A number of papers resulting from this work have been been presented in refereed conferences and journals.

- Constantinos Markantonakis, "The Case for a Secure Multi-Application Smart Card Operating System", In *Lecture Notes in Computer Science*, volume 1396, pp.188–197, First International Information Security Workshop (ISW), Ishikawa, Japan, September 1997.
- Constantinos Markantonakis, "Secure Log File Transfer Mechanisms for Smart Cards", In *Lecture Notes in Computer Science*, volume 1820, Third Smart card Research and Advanced Application Conference (CARDIS '98), UCL Louvain-La-Neuve, Belgium, September 1998.
- Constantinos Markantonakis, "Java Card Technology and Security", *Elsevier Information Security Technical Report*, Volume 3, No. 2, pp.82-89 (1998).
- Constantinos Markantonakis, "An Architecture for Audit Logging in a Smart Card Environment", In *EICAR' 99 Conference Proceedings*, ISBN 87-987271-0-9, EICAR '99 E-Commerce and New Media: Managing Safety, Security and Malware Challenges Effectively, Aalborg, Denmark, February 1999.
- Constantinos Markantonakis and Konstantinos Rantos, "On the Life Cycle of the Certification Authority key pairs in EMV'96", In *Society for Computer Simulation International (SCS)*, ISBN 1-56555-169-9, pp.125-130, 4th EUROMEDIA Conference 99, Munich, Germany, April 1999.
- Constantinos Markantonakis, "Interfacing with Smart Card Applications (The PC/SC and OpenCard Framework)", *Elsevier Information Security Technical Report*, Volume 4, No. 2, pp.69-77 (1999).
- Constantinos Markantonakis and Simeon Xenitellis, "Implementing a Secure Log File Download Manager for the JavaCard", In *Secure Information Networks, Kluwer Academic Publishers*, volume 23, pages 143-159. Communications and Multimedia Security (CMS '99), Katholieke Universiteit Leuven, Belgium, September 1999.
- Constantinos Markantonakis, "Boundary Conditions that Influence Decisions about Log File Formats in Multi-application Smart Cards", In *Lecture Notes in Computer Science*, volume 1726, pages 230–243, The Second International Conference on Information and Communication Security (ICICS '99), Sydney, Australia, November 1999.

Acknowledgements

This thesis would not have been possible without the guidance, technical and personal support of a number of people.

My initial and foremost thanks go to my supervisors Chris Mitchell and Dieter Gollmann for their continued support, interest, patience and guidance throughout my Master and Doctoral studies at Royal Holloway. Their style of supervision along with their important comments and suggestions for improvements have encouraged me to pursue my ideas.

I want to extend my thanks to Fred Piper for helping me overcome any financial concerns and focus on my research work. The Information Security Group was also very helpful in covering my expenses to attend conferences which has been an important element of my research. Part of the work for this thesis was performed under a contract from Mondex International Ltd to work with the Multos smart card operating system. It is also a pleasure to thank Dave Roberts and Dimitrios Markakis, from Mondex International for their technical support regarding the Multos smart card operating system.

My thanks to my friends and colleagues Keith Howker, Kostas Rantos, Peter Burge, Li Chen, Barry Rising, Kostas Skourtis, Andreas Fuchsberger, Zbigniew Ciechanowicz, Mark Hoyle, Duncan Garret and all of those who have helped to make my stay in Royal Holloway so enjoyable.

I would also like to thank my relatives, friends and doctors back home who helped and supported me in the most difficult time of my life, following a serious traffic accident in Crete.

I cannot thank enough Maria for her love, kindness, support and for putting up with many lost evenings and weekends over the last few years.

My parents have set the cornerstone for my studies. They have guided, motivated and supported me through my academic career and my life. Without their initiation and support I would not have started my studies. Mum and Dad thanks ($M\alpha\mu\alpha\ \kappa\alpha\iota\ M\pi\alpha-\mu\pi\alpha\ \sigma\alpha\varsigma\ \epsilon\upsilon\chi\alpha\rho\iota\sigma\tau\omega$).

Abstract

This thesis investigates the implementation of secure log file handling mechanisms in the light of recent smart card improvements.

Initially, we examine how smart cards evolved from single application cards into true multi-application cards. Additionally, we present the most recent architectures (client application interfaces) that enable client applications to interface with smart card applications.

Previous proposals for maintaining log files in smart cards are very limited and mostly theoretical. We examine those most related to smart cards along with presenting the very few real world examples of log files.

We go on to examine the new events that require logging along with the requirements of the entities involved. Subsequently, we describe an ideal event-logging model for a multi-application smart card environment.

To meet the identified requirements, we describe the details of a smart card entity that is responsible for dynamically updating the smart card log files. In that context, along with providing adequate log file space management, we propose a possible standard log file format for smart cards.

In the core part of the thesis we describe three different smart card log file download protocols, the selection of which depends on the requirements of the entities involved. These protocols download audit data to another entity that does not suffer from immediate storage restrictions.

Finally, we describe implementation details and performance measurements of both the log file download protocol and the standard log file format in two of the most advanced multi application smart cards.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

The aim of this chapter is to provide an introduction to the thesis and also present the overall structure of the thesis.

1.1 Motivation and Challenges

Smart cards are becoming an essential security instrument in the light of the explosive increase in electronic services. In applications where magnetic stripe card technology and its security features are not sufficient (e.g. banking, electronic purses, access control, etc.) smart cards appear to be the most prominent alternative candidate.

Since smart cards are involved in security sensitive applications it is evident that a number of protection layers are required. The required protection is offered either at the hardware level (i.e. as part of the physical security of the chip) or at the software level (e.g. the smart card operating system, or higher at the application protocol level).

During the last three years, smart card technology has developed considerably. Following improvements at the smart card hardware level, smart cards have evolved from multi-functional to multi-application cards. In simple terms this implies that the smart card operating systems are capable of securely hosting multiple applications. Although this might sound simple, given that in computer systems the issues involved have been addressed over many years, in a smart card environment the situation is more complex due to hardware and software restrictions.

Despite these difficulties, smart card technology developers have responded successfully, and, for example, the concepts of secure application download or deletion, secure inter-application communication, and application isolation are no longer an issue. One thing of importance here is that the introduction of multi-application cards introduces new vulnerabilities that require additional security mechanisms which are not yet available. For example, in a smart card environment application execution can easily be interrupted (e.g. by removing the card from the reader); it is then important to have a mechanism that will recover the smart card application to a safe state. Another mechanism could be responsible for logging certain events in order to provide evidence as to whether the security of the card or an application has been compromised. Finally, it

could be the case that some smart card applications rent or sell part of their functionality to other applications. In such cases it will be helpful to have a mechanism that will identify when an application has used the functionality offered by another application.

The above mechanisms have some common characteristics. They have to identify the relevant events and make sure that they are adequately reported. Therefore, smart cards need to store certain information internally in the card. The ideal place to store this information is in dedicated log files.

Although a lot of work has been done in providing logging architectures for traditional computer systems, the literature on smart card log file handling mechanisms is scarce. Log file mechanisms in computer systems are regarded as one of the essential instruments for monitoring system activity along with helping a system to recover to a "safe state" after a crash. Log files can be used to:

- record security critical events to enable users to be held accountable for their security related actions,
- resolve possible disputes about events,
- detect security breaches after they have happened, and
- recover the system to a safe state after fatal failures.

However, smart card log file handling mechanisms have not yet received the necessary attention mainly due to the smart card's technological limitations and its previous single application architecture. Recent advances in smart card technology — more memory and increased processing power, advanced operating systems — suggest that log file mechanisms could be implemented as an enhancement to the overall smart card security.

In a smart card environment with limited storage and processing resources (unlike

most modern computer systems) the use of log files needs very careful thought and investigation. For example, whatever space is allocated for the smart card log files will eventually be used up. The obvious question is what happens in such a case. The currently favoured approach suggests that smart card log files should be overwritten in a cyclic mode. When following this approach and when the smart card is frequently used (i.e. a lot of information is written to the log files) it could be the case that certain valuable information might be lost. In any case, access to the log files should be controlled in order to ensure that the log file space is not overused by a single application or that the log file information is not disclosed to unauthorised entities.

In addition to the usual design problems when providing solutions to the above, one has to take into account the limitations of smart card micro-processors. Among the most notable limitations are the following:

- limited memory in the card, thus the complexity of any algorithms and the amount of data to be stored should be minimised,
- restricted communication bandwidth when receiving or sending any information to/from the card,
- processing difficulties when handling large smart card files (e.g. larger than 255 bytes), these are restrictions mainly imposed by the card's file system.

1.2 The Contribution of this Thesis

Currently, a great deal of emphasis is placed on cryptography and security. In other words, great efforts have been placed in improving the performance of smart card cryptographic algorithms. At the same time, similar efforts (e.g. application isolation, etc.) aim to provide adequate security protection so that the concept of multi-application smart cards becomes a reality. Although we are using cryptography we provide an al-

ternative security mechanism that gives the necessary evidence after a security relevant incident has happened.

This thesis addresses the topic of secure log file handling mechanisms in smart cards, a way to securely generate and maintain log files taking into account the characteristics and limitations of current smart card technology.

The main goal of this thesis is to propose new concepts and techniques that lead to smart card log file handling mechanisms that can be applied in practice thereby bridging the gap between research and real-life applications. The results of this thesis may promote the development of improved smart card log file handling mechanisms and serve as a basis for understanding the issues involved in smart card event logging. Therefore, we proceed in the direction of providing answers to the following three questions:

- what are the new events that require to be logged, taking into account the recent smart card improvements,
- how can we efficiently manage the smart card log file space, and
- what happens when the smart card log file becomes full.

In order to highlight the issues involved we present a model for smart card logging that will form a foundation for the rest of the thesis. In this model we identify the entities involved, and define the characteristics of their behaviour. Once the entities are identified, it also becomes clear which types of events are of particular importance for each entity. Having identified the new events that require logging we also define the characteristics of the mechanism that will perform smart card logging.

The significance of our logging mechanism, which also distinguishes it from other proposals, is that the logged events can be used to identify pay-as-you-use details or to dynamically monitor the behaviour of a downloaded application.

Furthermore, in order to allow adequate log file space management we propose a standard log file format. Such a standard format will prevent smart card applications from over consuming valuable log file space and also will simplify the dispute resolution phase, since it will be easier for the arbitrators to access the content of the log files in case of a dispute.

In order to provide an answer to the problem of overwriting log files and avoiding the destruction of valuable evidence we present the core idea of the thesis, i.e. secure log file download protocols. These protocols securely extract the log files from the card and send them to another entity that does not suffer from immediate storage restrictions. Three different protocols are presented according to the different requirements of the entities involved.

Finally, in order to bridge the gap between the theoretical work and real world implementation of the proposals we present experimental implementation details and performance measurements for the standard log file format and the first of the secure log file download protocols. Both prototypes are based on two of the most widely used multi-application smart cards.

1.3 Organization of the Thesis

This thesis is organised as follows. Chapter 2 provides an overview of smart card technology. We explain the characteristics of today's multi-application smart cards in order to identify the new events that require logging, and subsequently demonstrate that a possible real implementation of our proposals is feasible. In this context we also examine the characteristics of programming interfaces that allow client applications to interface with smart card applications. In this chapter we also present clear definitions of smart card concepts that will be used throughout the whole of this thesis.

In Chapter 3, we examine existing proposals for smart card log file handling. Each of

the proposed schemes looks at the log file handling problem from a different angle and thus help to highlight the issues involved. Additionally, in order to maintain a balance between the theoretical log file handling proposals and real world implementation examples, we examine the use of log files in the Mondex purse and the MPCOS cards from Gemplus.

In Chapter 4, we present our ideal model for smart card log file handling. We consider a set of general requirements for the protection of log files. Additionally, we examine the different entities that will benefit from the existence of log files. We also identify the new events that require logging in the light of recent smart card improvements. The issues covered in this chapter will help us to better understand the exact magnitudes of smart card based logging.

In Chapter 5, we introduce a new concept referred to as "dynamic logging". Within this scheme we log the use of certain smart card application primitives dynamically. This operation is performed by the first entity of our smart card model called the Log File Update Manager (LFUM). We also define the operational behaviour of the Log File Browse Manager (LFBM), as the second entity of our model, that will provide controlled access to the smart card log files. Additionally, in order to improve log file space management we propose a possible standard log file format.

In Chapter 6, the core part of the thesis, we describe the third entity of our model, the Log File Download Manager (LFDM). This entity is responsible for securely downloading the log files from the card to an external entity that does not suffer from immediate storage restrictions. We define three different log file download protocols, the choice of which will depend on the different requirements of the entities involved.

In Chapter 7, we provide the experimental implementation details and execution timings results both for the LFDM and the standard log file format. The significance of the results are discussed and further work is suggested. Our implementation remarks should

be considered as a reference point when implementing smart card applications since they present certain features and limitations of the smart card application development phase.

Finally, Chapter 8 gives the conclusions of this thesis.

Chapter 2

Smart card Technology

Contents

Over the last three years the smart card scene has changed considerably, both at the hardware and software level. More powerful micro-processors and new software technologies (e.g. application code interpretation and dynamic application downloading) have made multi-application smart cards a reality. The aim of this chapter is to outline the main characteristics of smart card technology.

2.1 Introduction

Smart cards are already playing a very important role in information technology (IT). The public at large encounter smart cards as public telephone cards, bank cards or Secure Identity Modules (SIMs) in mobile phones.

Among the key issues in the acceptance of smart card technology are security, upgradability and programmability. These features are the consequences of improvements both in the performance of current smart card microprocessors [14, 33] and in more advanced operating system architectures [10, 13, 53].

In smart cards, as in many computerised devices, the operating system is a security critical system component that determines the internal and external behaviour of the whole system. The development of Smart Card Operating Systems (SCOSs) offered the same benefits as the development of operating systems in the early computers. The main benefit is that application developers are free from any concerns about the specific hardware constraints of their device, and users benefit from a variety of new applications (e.g. transport, banking, retail, health care, etc.).

Among the most notable changes in smart card technology is that smart card operating systems have developed from multi-function [10] into multi-application [19, 31].

We open with an overview of "traditional" smart card technology. We then describe the current status of the true multi-application smart cards that we consider for testing our techniques, explaining their main characteristics, advantages and disadvantages. Finally, we follow this by clarifying the techniques used for communicating (interfacing from an application running in a PC or a general purpose terminal) with the smart card applications.

2.2 Smart card Technology

In this section we describe the fundamental characteristics of smart card technology.

2.2.1 Smart card Microprocessors

Smart cards are the latest and most advanced member of the identification card (ID-1) family [41]. Certain aspects of smart card technology are standardised in the form of ISO standards described in §2.2.2.

The heart of the smart card consists of a microprocessor chip. The chip contains four main functional blocks:

I. The Central Processing Unit (CPU). Most of today's smart cards are based on 8 bit micro-controllers. These controllers operate within the clock range of 4Mhz–32Mhz and their operating voltage varies from 3V to 5V.

II. The Read Only Memory (ROM) contains all the information that will remain unchanged during the life-cycle of the card. The ROM typically contains the operating system of the card, the transmission protocols, security algorithms, etc.

III. The Random Access Memory (RAM) is the volatile memory the contents of which are lost when power is removed from the card. It is used for storing transmission data and intermediate results.

IV. The Electrically Erasable Programmable Read Only Memory (EEPROM). This type of memory is used for storing information that will be used between power on/offs. It can hold certain information that will be necessary for the card's lifecycle, personalized data like PIN numbers and smart applications.

Certain smart card microprocessors contain additional entities e.g. cryptographic co-

processors, intrusion detection mechanisms (e.g. low/high power supply voltages or changes in clock operating frequencies).

The aforementioned smart card chip is bonded in a circuit board and is connected to electrical contacts on the metallic board. This metallic board is in turn glued to the base of the plastic card.

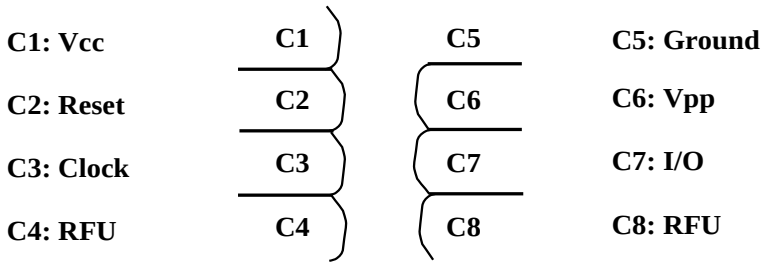

Figure 2.1: Naming and numbering of the smart card contacts according to ISO 7816-2.

The eight contacts (C1–C8) that provide the chip's interface to the outside world are shown in figure 2.1. In accordance with ISO/IEC 7816-2 [43] two of the eight contacts (C4 and C8) are reserved for future use. This is the reason why certain cards are produced with only 6 contacts in order to slightly reduce production costs. The rest of the contacts are as follows: C1-Supply Voltage, C2-Reset, C3-Clock Frequency, C5-Ground Mass, C6-External Programming Voltage, C7-Input/Output.

2.2.2 Smart card Standards

The most important and basic ISO standards for smart cards are now complete.

- ISO/IEC 7810 [41] defines the characteristics of the plastic card (card dimensions, toxicity of plastic, etc.) on which the smart card chip is bonded.
- ISO 7816-1 [47] defines the physical dimensions of the contacts along with their electromagnetic radiation and mechanical stress.

- ISO 7816-2 [43] defines the location, purpose and electrical characteristics of the card's metallic contacts.
- ISO 7816-3 [45] defines the operating voltage and the transmission protocol requirements.
- ISO 7816-4 [42] establishes a set of commands that provide file access, basic security functionality and transmission of card data.
- ISO 7816-5 [40] defines the rules governing the numbering and naming of smart card applications.
- ISO 7816-6 [44] describes the details of the physical transportation of device and transaction data, answer to reset (ATR) and transmission protocols.

In addition to the above, there are further ISO smart card specific standards, e.g. specific security related inter-industry commands [49], structured card query language (SCQL) commands [48].

Currently, major companies producing smart card related products are forming consortia in order to provide unified and interroperable smart card products (e.g. MULTOS [19], Java Card [50]). Since these proposals are still evolving, it will be some time before they are adopted as complete cross-industry standards.

2.2.3 Smart card Life Cycle

From the manufacturer to the application developer and the card holder, the production of the smart card is divided in different phases. There are five main phases that aim to test the functionality of the smart card chip along with ensuring the secure initialisation and personalisation of the chip's different functions. We briefly discuss each of them below.

I. Chip and card manufacturing phase is carried out by the chip manufacturers. In this phase the silicon chip is manufactured and tested. Certain fabrication data will be written to the chip and it is then ready to be delivered to the card manufacturer.

II. Pre-Personalisation. This phase is carried out by the card suppliers. The main operation of this phase is that the chip is mounted on the plastic card. The chip is connected to the printed circuit board and the smart card operating system is also installed. If it is required, at this stage the functionality of the smart card can be tested.

III. Personalisation. This phase is carried out by the card issuers. It is regarded as the main operation since the logical data structures (e.g. data files, directories and applications) are written to the card. Additionally, information about the card holder identity will be written to the card.

IV. Utilisation Phase. This phase includes the use of the card by the card holders.

V. Termination Phase. There are two ways that the card can be moved into this phase. Firstly, the smart card operating system or an application may block access because certain security features (e.g. PIN numbers) are blocked. Secondly, the smart card may become physically damaged, e.g. the chip is broken.

A slightly different smart card life cycle (dedicated for financial transaction cards) is presented in ISO 10202-1 [39].

As we examine the multi-application smart card technology, in section §2.3.1 we point out that the the above life cycles are not absolute. For example, it could be the case that the smart card operating system is not installed in the pre-personalisation phase but in the chip and card manufacturing phase. Similarly, with the feature of dynamic smart card application download the card can accept or remove further applications while it has moved into the utilisation phase.

Although the actual chip and the smart card operating system provide the required security, it is the personalisation operation which creates and enables the card's functionality. Therefore, personalisation is the central process in the smart card production phase.

2.2.4 Multi-function Smart cards

In the following subsection we outline the main characteristics of one of the most well-known multi-function smart card operating systems. In doing so, we highlight the characteristics and inadequacies of smart cards before the advent of multi-application cards.

MPCOS Smart card Operating System

The Multi-application Payment Chip Operating System (MPCOS) [10], as its name implies, is an operating system adapted to multi purpose and payment applications. The fact that the MPCOS name contains the word "multi-application" does not imply that applications can be downloaded at any stage of the card's life cycle. It rather means that applications can coexist on the same card, but they have to be installed during the personalisation phase.

The MPCOS smart card operating system comes with the matching MPCOS smart cards. It is compatible with the ISO 7816-4 standard data structures and commands (one difference is that it supports a single level of sub-directories). It is also compatible with its predecessor, the Multi-application Chip Operating System (MCOS) [9].

The MPCOS cryptographic security features can be summarised as follows:

- Secure messaging, which implies protection of the administration command transmissions between cards and terminals, as defined in ISO 7816-4. This is actually

achieved either by encrypting the data sent to the card, or by sending 3 bytes of a cryptographic checksum along with certain smart card commands. Thus, the receiver can verify the integrity of the transmission.

- Card/Terminal authentication, is a two-way authentication process. The process is achieved with a simple challenge-response system and a shared key between the two entities.

- Administration command transmission verification using secure messaging, employing cryptographic checksums.

- Sensitive command monitoring, which generates a cryptographic certificate based on a sensitive command counter and the previously executed sensitive command header, details of which are stored internally in the card.

- Payment command cryptograms are issued during payment transaction sessions. The MPCOS cards and terminals generate cryptograms that can be used to verify the integrity of the transmission.

Another multi-function SCOS which offers very similar functionality to the MPCOS is called OSCAR [13].

2.3 Multi-Application Smart card Technology

In this section we describe the multi-application smart card technology. We open our discussion by explaining the need for multi-application smart cards. Subsequently, we describe two of the most advanced and well known smart card technologies, the Java Card and Multos.

2.3.1 The Need for Multi-application Smart cards

Most SCOSs are largely concerned with the management and protection of data files along with the provision of cryptographic algorithms. Although the ISO 7816-4 standard claims to provide an abstract view of a multi-application operating system, it is obvious that SCOSs built around this proposed methodology will not be true multi-application SCOS. This is true since if more than one application is to reside in the same card, application developers will have to agree and define their application structures and inter-application relationships in advance.

In the past, before the emergence of multi application smart cards, developing a smart card application was often a difficult process. Although the ISO 7816 series of standards attempted to standardise certain aspects at the smart card application level, the smart card's internal working varied between different smart card vendors. This is the main reason for the large number of application programming interfaces (APIs) dictated by the specific hardware characteristics of the different smart card microprocessors.

Given that smart card manufacturers were reluctant to provide information and programming tools for their products, it was very difficult for smart cards to be utilised by anyone outside a small numbers of experts, mainly working for the large smart card manufacturing companies. Furthermore, due to the different steps involved in the personalisation phase (section §2.2.3), smart card developers had to wait for a period of time (from a couple of weeks to a couple of months) in order for their application to be delivered within the smart card mask.

Since the applications were designed to run on proprietary smart card micro-processors they were not able to run on different smart cards. This implies that there was no platform independence and applications had to be completely re-written in order to become portable between different smart card platforms.

It is obvious that application developers require new techniques that will enable smart cards to host multiple applications securely. For example, these applications might originate from different companies, share certain information and more importantly they could be installed at any later stage in the card's life cycle without reference to the existing applications. Finally, strong guarantees should be present to make sure that applications will not interfere with each other (i.e. they should be restricted within their own application space) and that resources should be accessible only through well defined operating system calls.

2.3.2 Java Card APIs

The Java Card Application Programming Interface (API) 1.0 was released in October 1996. This was the initial attempt to bring the benefits of Java to the smart card world. The minimum smart card environment needed to run the Java Card API 1.0 is a 300 KIP (kilo instructions per second) CPU, 12 Kbytes ROM, 4 Kbytes EEPROM and 512 bytes of RAM. The Java Card API supports the following data types needed for Java Card applications: Boolean, byte and short data types, all object-oriented scope and binding rules, all flow control statements, all operators and modifiers, uni-dimensional arrays. The API consists of the `java.iso7816` package which defines the basic commands and error codes as specified in ISO 7816-4.

In September 1997, the Java Card API 2.0 was released by JavaSoft with more advanced features and extended functionality. The API 2.0 consisted of three different documents:

I. The Java Card Virtual Machine Specification [29], that defines the behaviour of the virtual machine, e.g. Java Card supported and unsupported features, how exceptions are caught, etc.

II. The Java Card 2.0 Programming Concepts [30], contain information about the Java Card 2.0 classes and how they can be used in smart card applets. Some of

the concepts covered are: transaction atomicity, ISO 7816-4 file system, applet life time, etc.

III. The Java Card API 2.0 Specification [31], describes all the Java Card packages, classes and methods.

Among the most notable supported features of the API 2.0 [25] are the following:

- Packages are used exactly the way they are defined in standard Java.
- Dynamic Object Creation of both class instances and arrays is supported.
- Virtual Methods and Interfaces, may be defined as in standard Java.
- Exceptions are generally supported, apart from some exceptions and error subclasses which are naturally unsupported (e.g. exceptions for multi-dimensional arrays, etc.).

The minimum smart card microprocessor requirements for supporting the API are 16 Kbytes of ROM, 8 Kbytes of EEPROM and 256 bytes of RAM. The Java Card API 2.0 is compliant with ISO 7816 parts 6, 7 and 8 [40, 44, 42]. A more detailed coverage of the API is provided in section §2.3.4.

In February 1999 Sun released the latest version of the Java Card API 2.1 [32]. The major enhancements and changes from Java Card API 2.0 are the following:

- The cryptography extension package has been reconstructed. This implies that the `javacard.security` and `javacard.crypto` packages provide extended functionality for security primitives. Additionally all the export control classes are transferred in `javacard.crypto`.
- The applet firewall is more robust and more restrictive. From now on applets are

allowed to customize access control via a very well defined sharable interface for object sharing.

- Particular importance has been placed on the design of the specification in order to allow applets from different smart card application developers to become more portable.

- The ISO7816-4 file system extension package `javacardx.framework` has been deleted.

- Since Java Card API 2.1 classes, i.e. the `java.lang` package, are re-defined as a strict subset of Java and since transient objects are created using factory methods, the model can be conveniently simulated on a workstation.

2.3.3 The Benefits of Java on Smart cards

When smart card manufacturers decided to enhance smart card technology they also realised that they needed to overcome the limitations mentioned in section §2.3.1. This led to the conclusion that: "the problem faced is similar to that of loading code into the World Wide Web browsers, a problem that Java attempted to solve" [5].

When using the Java programming language the development of smart card applications is simplified and improved. The primary reason behind this improvement is that Java Card developers are presented with a variety of standard off-the-shelf integrated Java development environments along with their supporting documentation. Java Card applications could be written much more quickly since the details of smart card complexities are hidden.

The Java programming language paradigm provides a secure model that prevents programs from gaining unauthorised access to sensitive information. This implies that if Java Card applications adhere to the general Java programming model they will be restricted within their own operational environment.

Since Java is based on a runtime byte-code interpreter the portability issue is successfully addressed. This implies that with the usage of interpreters in the cards, Java Card applications would be portable between different smart card microprocessors. Upgrading the smart card's functionality could take place with new improved applications that could be installed at any time during the life cycle of the smart card.

Finally, the major problem that had to be solved was the migration of the general Java scheme into a smart card environment (taking into account the smart card hardware and memory constraints).

2.3.4 Java Card Technology

The internal architecture of the Java Card specification is illustrated in the right hand side of figure 2.2. At the bottom layer of the smart card architecture there is the smart card microprocessor. Immediately above the hardware layer we observe the smart card operating system (SCOS). On top of the SCOS we have the Java Card Virtual Machine (JCVM). Both the JCVM and the SCOS are written in the native language of the microprocessor. The JCVM hides the manufacturer's proprietary technology with a common language interface. It is actually a reduced subset of the standard Java Virtual Machine (VM), as a result of memory and hardware constraints.

In contrast with the normal Java virtual machine the JCVM does not support garbage collection. Application developers should not assume that objects which are allocated are automatically de-allocated. This feature, along with the limited memory space of the card, implies that application developers should be very careful when they develop Java Card objects. They have to make sure that their applications are small enough to fit within the memory of the card. Additionally, the Java Card does not support object cloning. This is mainly a restriction imposed by the card's limited storage memory.

In the Java Card there is no separate Security Manager which enforces the policy

decisions. The security policy decisions are embodied within the JCVM. A security policy decision could be, for example, the mechanism which defines the inter-application communication (i.e. *Gateways* as described in the following paragraphs). Threads are not supported and a Java Card system is not able to load classes dynamically. The latter is the result of the external (not in the card) byte code verification phase.

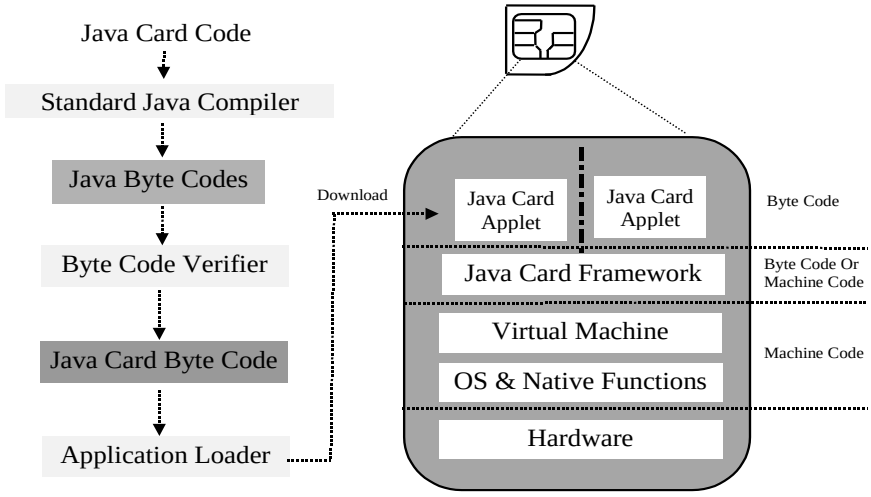

Figure 2.2: The Java Card application development and the Java Card architecture.

The system classes are either masked in the ROM of the card during manufacture or they can be loaded through a controlled installation process (after verifying the necessary application certificates) at any later stage of the card's life cycle.

The lifetime of the JCVM is, in a sense, equivalent to the life time of the card. Since information must be preserved even when power is removed from the card, most of the persistent information is stored in EEPROM memory.

Some of the most important features of the Java Card technology are transaction atomicity, exceptions, and inter-application communication. Transaction atomicity guarantees that any updates to a single persistent object or class will be atomic (either fully performed or not performed at all). Automatic transaction integrity describes how the virtual machine shall behave in case power is lost during the update of a field in an object. On the other hand block transaction integrity describes the virtual machine behaviour when the application programmer defines a specific part of his code which

must be executed in one piece.

Exceptions and their subclasses represent serious errors usually thrown up by the JCVM (when internal runtime problems are encountered) or programmatically (Checked exceptions or Unchecked exceptions). The exceptions that are not caught by the application are caught by the JCVM. The Java Card application programmers can define their own exceptions by declaring subclasses of the class `Exception`.

Among the most notable improvements of API 2.0, compared to API 1.0, is the inter-application communication. For example, applications can communicate and exchange information with each other in a controlled manner. According to the Java Card API this inter-application communication is achieved via a complement to the Java *Sandbox* security model called the *Gateway* model. This model allows flexible bilateral agreements between applets willing to share information. The *Gateway* model is based to a large extent on Capabilities as it maintains a table of shared objects. The model is implemented as a system class which allows specific objects to be declared sharable, and access to these objects is then controlled. To make an object sharable, the sharing application must create an authorisation object called a *TicketChecker*. The details of the sharing application, sharable objects, and the *TicketChecker* are stored in the *Gateway*. An application asking for access to a shared object must provide to the *Gateway* a valid *Ticket* object. The *Ticket* objects are in turn checked by the *Gateway* model in order to grant or refuse access to the corresponding object.

On top of the JCVM we have Java Card `framework`. The Java Card API 2.0 defines four main packages:

- `javacard.framework`, as the core package present in all Java Card implementations which defines the following classes (Applet, PIN, Util, System, AID, APDU). The `javacard.framework` package also provides an object oriented view of ISO 7816-4.

- `javacardx.framework` might contain specific commands for specific applications (e.g. EMV, GSM, etc.).

- `javacardx.crypto` and `javacardx.cryptoEnc` provide support for cryptographic functionality (e.g. RSA, DES, SHA-1) that might be required by smart card applications.

The `javacardx.*` packages are considered as extensions and they might not always be present in all Java Card implementations. In addition to these packages, a subset of the `java.lang` package is also included. Detailed explanations of each package, are given in the API 2.0 specification [31].

The steps for creating a Java application, downloading it and executing it on the smart card are presented in the left hand side of figure 2.2 and described below. Firstly, the application programmer must conform to the Java Card API, develop and compile a Java Card application by using a standard Java development environment. Due to the fact that there is no byte code verifier in the card, the newly created Java code classes must be verified externally producing signed loadable byte code. This is checked by the JCVM before accepting code from an external source. As soon as the Java classes are verified the application code is ready for loading onto the card via the external application loader.

2.3.5 Multi-Application Operating System (MULTOS) Technology

Multos is offered as a high-security Multi-Application Operating System (MAOS) for smart cards. It enables a number of different applications or products to be held on the smart card at the same time, separately and securely.

Multos has been designed with security in mind. Thus, issues like secure application download and application isolation are a major concern. In Multos each application

carries an application certificate that is verified prior to the application download.

The Multos operating system checks the validity of the application it has received, allocates the program a protected and — through the use of special "firewall" programs — isolated area in memory, and locks the new program into place. Subsequently, each new application is kept separate by these firewalls from those already installed on the card.

Mondex International has developed a smart card optimised language: MEL (Multos Enabling Language). MEL applications interface with the Multos operating system via the MAOS-API (Application Programming Interface). The operation of writing applications for Multos is similar to writing Java Card applications as described in figure 2.3.

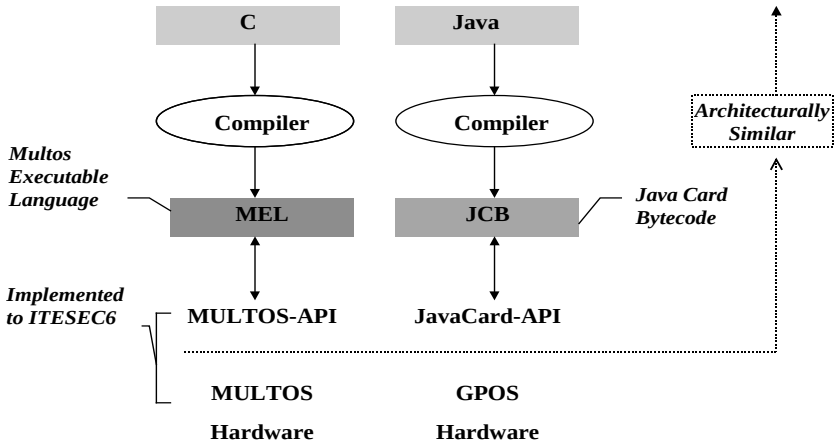

Figure 2.3: Contrasting Java Card technology and Multos technology.

Applications to run on Multos can be programmed in C which is then translated into MEL. Finally, Multos Ver. 3 on Hitachi H8/3112 smart card micro-processor was certified, in September 1999, at ITSEC [37] level E6.

2.4 Interfacing With Smart card Applications

Smart card application programming interfaces form one part of smart card technology. We will also cover the APIs that will allow "client" applications to communicate with smart card resident applications.

We start by describing a traditional but effective method, i.e. the smart card manufacturer specific drivers. Subsequently, we examine the most recent and popular card interface APIs (PC/SC and OCF) that aim to enable interoperability between smart cards and smart card readers. We aim to highlight the main characteristics of each different proposal in order to justify our selection for the implementation described in chapter 7.

2.4.1 Manufacturer Specific Drivers and DLLs

Until very recently there were no card reader independent application programming interfaces. Two specific reasons for this are:

- The smart card and the card reader device were very closely coupled; there was no need for a card to be used with a different card reader and vice-versa.
- Card reader programming interfaces were not standardised, whereas smart card interfaces were standardised.

Thus, the most common method employed when smart card programmers want to communicate with a smart card application via a smart card reader is the following: obtain the drivers specific for the smart card reader, install them in the system and subsequently integrate them within the client applications. This architecture requires that each manufacturer provide a device driver that will transport Application Protocol Data Units (as described in ISO 7816-4) with its proprietary host-reader protocol.

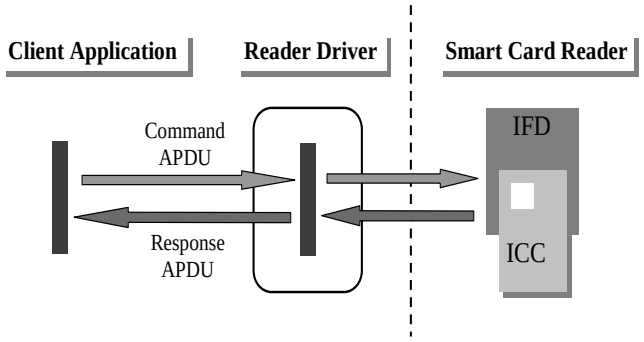

Figure 2.4: Communicating with the card-reader via manufacturer specific drivers.

The whole procedure is summarised in figure 2.4. The client sends an APDU via the card reader driver to the card reader and in turn to the card. The reverse steps are followed when an APDU is send from the card to the client.

2.4.2 The Personal Computer Smart Card Specification (PC/SC)

The Personal Computer Smart Card (PC/SC) specifications [24, 66] were created by the PC/SC workgroup (formed in mid 1996), as a joint effort between Bull CP8, Gemplus, Hewlett-Packard, IBM Corporation, Microsoft, Schlumberger, Siemens Nixdorf, Sun Microsystems, Toshiba and Verifone. The first documents became public in December 1996 and are formally known as "Interoperability Specifications for Integrated Circuit Cards (ICCs) and Personal Computers". They are more commonly referred to as PC/SC. Currently, the PC/SC workgroup retains the ownership of the specification until it is accepted by a formal standards body.

The main goal of the PC/SC architecture is to allow interoperability of smart card readers and cards in a PC environment running the Windows operating system. According to the PC/SC consortium the main reasons for creating the specifications were:

- Current lack of interoperability of smart card technology (software tools and hardware devices) in a PC environment.

- No widely accepted application programming interface (API) for developing "client" applications that interface with smart card applications.
- The mechanisms that permit multiple applications to share and serialise access to a single smart card simply do not exist.

The main components of the PC/SC architecture are presented in figure 2.5 and explained in the following subsections.

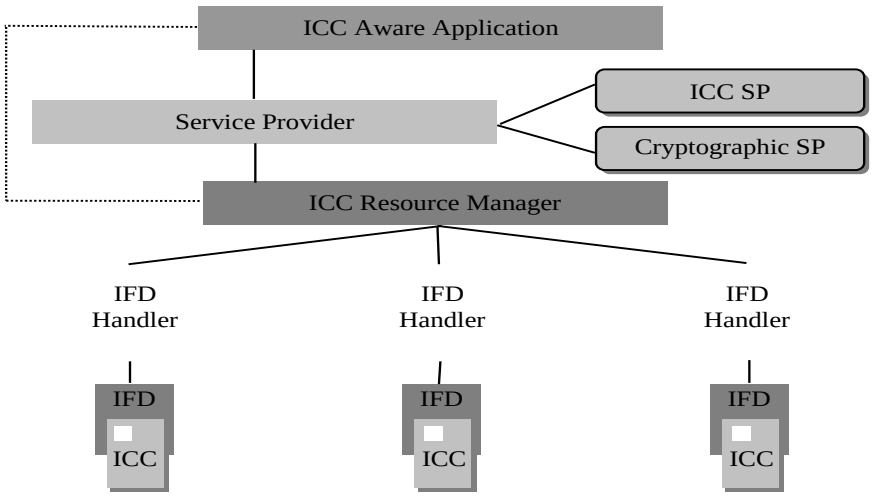

Figure 2.5: The PC/SC architecture.

The Smart card Resource Manager

The central entity of the PC/SC architecture is the smart card Resource Manager. It is responsible for controlling all accesses to the smart card relevant resources within the system. The resource manager is considered as the privileged component of the architecture and thus is very likely to be provided as part of the operating system. There should be only one smart card Resource Manager within a system. The smart card Resource Manager is responsible for the following two tasks:

- Tracking the smart card relevant resources: This implies tracking all the available and installed Interface Devices (IFDs) and smart cards and making this informa-

tion available to other applications. Similarly, it has to identify which cards are inserted or removed from specific readers.

- Control allocation of IFDs and resources: This is done by providing the means for attaching specific IFDs in shared or exclusive modes of operation.

As the smart card Resource Manager is considered a privileged component, its installation or removal must be a carefully controlled process.

The Smart card and Interface Device Components

A PC/SC compliant smart card must adhere to the smart card standards defined by ISO/IEC. Similarly, the IFD is the actual interface between the smart card and the outside world.

In order to enable the smart card Resource Manager to perform its tasks it is expected that each IFD will be identified to the smart card Resource Manager as part of the IFD's installation process. At this stage the Resource Manager IFD database will be updated in order to reflect the latest installed device. Similarly, each smart card manufacturer should provide a setup utility that will permit initial installation of the smart card within the Resource Manager.

The IFD handler component is the device driver for a specific reader. This driver maps the functionality of the specific reader to the native Windows services.

The Service Provider

The Service Provider is responsible for making available the functionality of a specific smart card to an application through a high level programming interface. The current version of the specification divides the Service Provider into two independent compo-

nents, namely the smart card Service Provider and the Cryptographic Service Provider. In the future these interfaces may be extended in order to meet further specific application requirements.

The Cryptographic Service Provider exists as a separate entity because of its specialised task of making available the smart card's cryptographic functionality to PC applications and because of the export controls associated with cryptographic devices.

The smart card Service Providers interface directly with an application. They make available a predefined set of services and further assumptions/restrictions regarding these services to PC applications.

Smart card Aware Application

The smart card aware application is a software program that requires access to the smart card. It can be written in many high level languages (Visual Basic, Java, or C++) that can access the COM service providers [18]. Although the PC/SC specification is platform independent there are some general assumptions about the functionality of the operating system for which the application is targeted:

- Multi-threading capability,
- Asynchronous event and message handling,
- A shared library mechanism with dynamic linking to shared code.

Assuming that this functionality is present, the application could determine which resources are available (either at installation or at run time) and proceed accordingly.

2.4.3 The OpenCard Framework (OCF)

The OpenCard consortium [24, 38] was founded in March 1997 by computer and smart card technology manufacturers. Initially the consortium's focus was on the use of smart cards with networked computers (NCs). Later, they realised that the main areas that needed addressing were independence from the host operating system and transparent support for a number of different multi application smart cards.

OCF Overview

OCF was specified as an open standard providing an architecture (i.e. an API) that helps terminal application providers to build their smart card aware applications in any terminal that supports Java and subsequently OCF. This is actually achieved via two distinct interfaces. Firstly, there is a high level API that hides the characteristics of a smart card or terminal component from application/service developers. Secondly, there is a common provider interface that enables the seamless integration of smart cards and readers from different vendors.

The core architecture of the OCF [24, 38] consists of two components: the CardTerminal and the CardService. The OCF components are presented in figure 2.6.

The CardTerminal component contains all classes that allow access to smart card terminals and their slots. For example, using certain classes of this component, smart card insertion or removal could be tracked.

The CardService component contains all the infrastructure required in order to access the functionality of a smart card. Each card service implements a high level API through which applications gain access to a specific smart card's functionality. Examples include the file access card service which enables applications to access an ISO 7816-4 style file system. Both the CardTerminal and the CardService components are designed using

the abstract factory and singleton pattern [8].

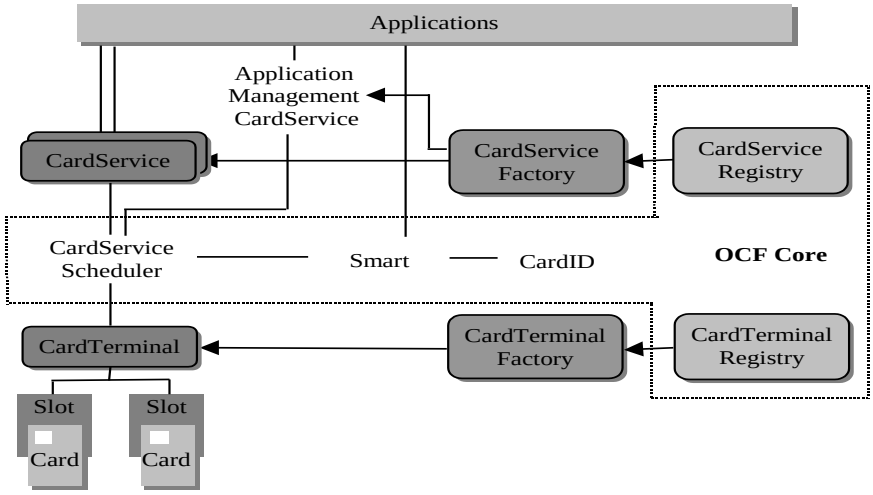

Figure 2.6: The OpenCard Framework Architecture.

In OCF, the PC/SC's central resource manager is divided into two distinct objects, the registry and factory objects. Each respective manufacturer supplies a factory object that contains the manufacturer's product specific details. To determine which factory to use, OCF uses the singleton registry. This registry object contains the configuration of an OCF component and creates the respective factory objects as required. The rest of the OCF components are described in more details in the following sections.

The CardTerminal Component

The CardTerminal provides access to card terminals and subsequently to smart cards. All this functionality is encapsulated in the CardTerminal class, the Slot class and the CardID class.

Each CardTerminal object can contain one or more Slot objects. Each Slot represents the physical card slots of that card terminal. Whenever a smart card is inserted into a slot, a SlotChannel gate object is activated. Access to the smart card occurs through this specific object. The CardTerminal object ensures that at any given point in time at most one SlotChannel object exists for that particular slot. Similarly, at most one

object will obtain access to that SlotChannel.

The CardTerminal class is also responsible for checking the card presence or card absence within a reader along with sending and receiving Application Protocol Data Units (APDUs).

The CardTerminal Factory and CardTerminal Registry objects are responsible for handling the attachment (and subsequent removal) of a reader to a computer (or similar device). Each terminal manufacturer that supports OCF will provide a CardTerminal Factory subclass which holds all the characteristics of the specific terminal. The CardTerminal Registry object keeps track of the installed card terminals. It also provides methods for enumerating, registering and de-registering CardTerminal objects.

The CardService Component

The CardService component is an abstraction of the different Smart Card Operating Systems (SCOSs). The main sub-components of the CardService component are the CardService class, the CardServiceRegistry class, the CardServiceScheduler, and the SmartCard class.

The core of the CardService component implements an API that maps smart card commands onto specific APDUs. An example of a card service currently included in OCF is the FileSystemCardService.

The CardServiceScheduler is responsible for serialising access to a given smart card and maintaining the state of that card. For example, the CardServiceScheduler class allows multiple instances of card services to gain access to a specific smart card.

Access to OCF methods is mainly through the SmartCard class. This class along with the CardID class are used by the application programmer to identify a given card and

associate it with a slot in a reader.

The main advantage of OCF is that it is completely independent of the underlying operating systems since it is implemented in Java and thus all its components become available on any platform (PCs, NCs, etc.) supporting Java.

2.4.4 Security Requirements for the PC/SC and OCF Architectures

Both architectures have certain entities or components that have significance from a security perspective. For example in PC/SC the main entity is the Resource Manager. Thus, in order to give increased protection to this entity, it is suggested that it will be part of the operating system. However, this component holds the databases of all the installed IFDs and smart cards along with their cross-references and their availability status. If the entries in the databases are modified then at least one can mount denial of service attacks. Similarly the mappings of specific smart cards to their associated Service Providers or interfaces could be mixed up, etc. If the PC/SC is running under Windows 95 or Windows 98 then there is little that can be done. When PC/SC is installed in an operating system that offers basic security functionality (e.g. Windows NT or Linux) then the Resource Manager could be more adequately protected.

In OCF the functionality found within PC/SC's Resource Manager is divided into two sets of registry and factory objects. Taking into account the fact that OCF is "completely" independent of the underlying operating system does not minimise the chances of the above security breaches taking place. Therefore, the obvious countermeasure would be to use OCF or PC/SC in an environment that restricts logical accesses to certain components.

As one would expect, both architectures provide a range of cryptographic services. Both specifications include cryptographic services (public key and secret key algorithms) along with identification and authentication mechanisms (authentication to remote en-

tities or to the smart card; cardholder or application verification).

The notion of exclusive access to a smart card is defined in both PC/SC and OCF. This feature is of significant importance since under certain circumstances an uninterrupted sequence of smart card operations (e.g. "sensitive" or payment applications) must be executed.

When developing smart card aware applications under OCF, the general security model of the Java language applies. In particular, when executable code is downloaded from the Internet (applets) extra security concerns arise. For that reason the specification describes the steps required for writing "secure" applets with OCF and the Java Plug-In or the standard Java Platforms, Version 1.2.

2.4.5 Card Applications as Remote Objects

This development methodology is promoted by GemXpresso [12] and considers smart card applications as distributed objects where communication between them and client applications is abstracted from low level protocols. This implies that client applications are regarded as remote objects in a distributed system. The procedure is summarised in figure 2.7.

Smart card application functions are called through a client stub (application proxy) handling the communication interface.

Each application defines an application interface that describes the functionality of the application. This interface is used as a basis on which to construct the application proxy. The proxy will be marshaling and un-marshaling processes on the method invocations. Finally, a Remote Procedure Call (RPC) protocol [6] will be used for transporting, via the smart card reader driver, the calls between the terminal and the reader.

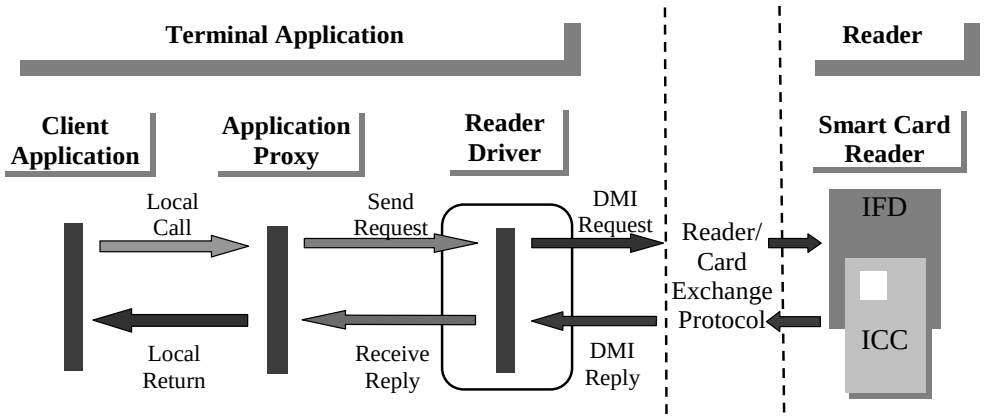

Figure 2.7: Communication modules in more object oriented architecture.

2.5 Summary

Without doubt the Java Card API and the Multos API revolutionise both the way smart card applications are designed and implemented, and also the smart card technology itself.

With the above technologies it becomes easier to develop smart card applications. Similarly, the important issue of securely isolating the smart card applications is successfully addressed. Additionally, smart card applications can also be downloaded or deleted at any stage of the smart card's life cycle.

The procedure for developing smart card client applications has also been simplified. Both PC/SC and OCF hide the underlying IFD architecture with a more abstract programming interface.

Although currently PC/SC runs on the Windows 95, 98 and NT platforms efforts by individual developers aim to migrate the PC/SC architecture on other platforms (Linux) [34]. This will increase PC/SC's availability on different computer platforms. On the other hand OCF is completely independent of the underlying operating system and thus it could be used on any platform that supports Java.

Chapter 3

Related Work

Contents

The aim of this chapter is to introduce the reader to related work in the area of log file mechanisms in smart cards. We must make clear that both theoretical work on, and real implementation examples of, log files in smart cards are very limited.

3.1 Introduction

In the previous chapter we reviewed the current status of smart card technology. We now examine proposals for log file use in smart card environments or more generally in untrusted machines or machines with limited memory. The work presented in this chapter is related to the new techniques presented in this thesis. This chapter highlights certain aspects of the problem along with presenting the relevant work which inspired the development of the new techniques presented in this thesis.

We open with the problem of providing support for secure logs on untrusted machines. We then describe a proposal related to the subsequent work in this thesis which introduces a property called "forward integrity" for log files. Finally, we describe a real world example of how log files are used both by Mondex International in the Mondex electronic purse and in the MPCOS cards from Gemplus.

3.2 Secure Logs on Untrusted Machines

In [59] Schneier and Kelsey consider the dual problem of an untrusted machine (U) generating and maintaining log files, and a trusted machine (T) periodically checking the logs in U. Although U is "untrusted" it is not generally expected to be compromised.

The authors of [59] state that applications that will benefit from such a mechanism abound. For example, U could be an electronic wallet, a smart card or even a PC. On the other hand T could be a bank Automated Telling Machine (ATM) or a server in a secure location.

It is clearly stated that their system aims to provide strong security guarantees about the authenticity of the logs on U. In particular if we suppose that an attacker gains control of U at time t, she/he will not be able to alter or delete log entries made before

time t in a way that these manipulations will not be detected when U next interacts with T.

The need for such an architecture is explained by the fact that there are many systems in which the owner of a device is not the actual owner of the secrets within the device. In that case auditing can help to determine if there was an attempted fraud. Note that, as in the majority of the logging architectures, this proposed system aims to detect possible fraud after it happened and not to prevent possible attempts.

3.2.1 A Brief Description of the Method

We provide a few of the most characteristic assumptions and statements that the above authors mention in their work. We believe that they will highlight the exact dimensions of the issues involved, acting as a reference point for our proposals:

- "... no security measure can protect the audit log entries written *after* an attacker has gained control of U. All that is possible is to refuse an attacker the ability to read, alter or delete log entries made *before* the machine was compromised."
- "If there is a reliable, high bandwidth channel constantly available between T and U, then this problem won't come up. U will simply encrypt each log entry as it is created and send it to T over this channel."
- "No cryptographic method can be used to actually prevent the deletion of log entries: solving that problem requires write-only hardware such as CD-ROM disk, etc."
- "The long term storage of U (holding the logs) is sufficiently large and filling it up is not a problem."

The behaviour of the proposed architecture is divided into four steps. Firstly the two entities exchange some information in order to enable U to create the log file. The main

information exchanged is an authentication key A_j which is used in the generation of the encryption key K_j that will encrypt the j^{th} entry in the log file. The initial value of the authentication key is denoted as A_0 and contains some fixed values as defined in the paper.

The second step involves the creation of each log file entry according to some predefined procedure. This procedure requires that the data to be inserted in the j^{th} entry will be encrypted with the encryption key K_j. This key is derived from A_j along with some information which is used as a permission mask for a verifier entity V. Entity V is a moderately-trusted verifier, who will be trusted to review certain kinds of records, but not trusted with the ability to change records.

In the third step the log file is closed. This is done simply by writing a final-record message D_f (which includes the data and a timestamp along with a text message "NormalCloseMessage"). Additionally the values of A_f and K_f are irretrieveably deleted.

The last step addresses the verification of the log files. There are two variations in this phase. In the first, T can receive a closed log file and validate it since it knows all the encryption keys and thus can read the whole audit log. In the second, V may need to verify the log file records while they are still stored on U. This last case involves V generating a list with an index of each entry to which it requests access, and sending it to T. T then verifies that the log file has been properly created on U and that V is authorised to work with the log. If there are no problems T forms a list or responses to the requests of V. Note that T computes the required keys based on information provided by V. Thus, if this information is incorrect then the response keys will be incorrect. With this architecture the protocol guarantees that V will get only the log entry decryption keys which he/she is authorised to decrypt.

In addition to the above verification methodologies the authors propose off-line or voice line variants of the verification protocol.

3.2.2 Discussion on the Secure Logs on Untrusted Machines

In this subsection we review the above methodology. The major assumption in order for the proposed architecture to work is that the log file is sufficient large and that filling up is not a problem. This can be true if U is a PC or even a smart wallet, but it is relatively unreasonable (due to the restricted memory space) if U is a smart card.

A particular problem arises from the following observations:

- apparently, the authors assume that the owner of U is a legitimate user, and
- it is nowhere stated that A_0 should only be known internally in U.

Since the owner of U knows A_0 he/she can create a whole sequence of fake log files. Thus, in this case the legitimate users are provided with the capability of attacking the system. Having said this, the authors are partially covered against this case by stating the first assumption presented in §3.2.1.

Another potential problem arises from the requirement that U must be available for the verification of the log files. Potentially the user can destroy U and as a result destroy the logs (i.e. all the available evidence). This type of attack is described as the "Watergate Attack" [58].

Finally, in their second assumption (refer to §3.2.1) the authors assume that if there is a constant and reliable channel between the entities, then the problem of maintaining log files is simplified. In a smart card environment this is not always a valid assumption. Thus, in our proposals in chapter 6 we examine the scope of the problem when the smart card periodically interacts with a trusted server in order to download the log files. In our proposed architecture we try to overcome some of the above problems and limitations.

3.3 Forward Integrity For Secure Logs

In [3] Bellare and Yee introduce a security property referred to as forward integrity (FI). The method involves generating message authentication codes (MACs), mainly for audit logs, in such a way that even when the MAC key is compromised it will not be possible to forge past "log" file entries. Similarly to the previous proposal this implies that an attacker can erase log entries, but cannot modify existing entries without being detected.

More formally the FI scheme works as follows: the time is divided in "epochs" (E). If a machine is compromised at time T_c during an epoch E_j i.e. $T_c \in E_j = \{t : T_j \leq t < T_{j+1}\}$, then the attacker cannot forge log entries that appear to be generated at times $t < T_j$. This implies that no guarantees are provided for any log entries produced after T_j.

FI is achieved since the MAC key is variable, i.e. it evolves over time periods. For example, K_i in epoch i is obtained from a non-reversible function (hash) of the key K_{i-1} of the previous epoch. Additionally, upon starting epoch i the key K_{i-1} is deleted. Meanwhile K_0 can be used to verify the MAC of all log entries, regardless of the epoch.

3.3.1 Discussion on the Forward Integrity Property

With this proposal the authors claim that they avoid remote logging, continous logging or log replication.

The authors also state that among the key factors for the success of the system is the need to be able to quickly change epochs. Thus, they suggest different options for changing epochs: either every 100ms or after a certain number of log entries.

The most notable characteristic of the above work is that it actually contains specific

implementation details and results. The authors provide a web page reference to the source code for their implementation along with presenting experimental figures on the performance of their proposed system.

3.4 The Mondex Log Files

The Mondex purse is an electronic cash scheme. The smart card holds a certain amount of a currency which is used for payments or transfers between other cards (e.g. merchant cards or normal user cards). The Mondex electronic purse scheme uses a secure value transfer protocol and along with a number of log files maintains the integrity of the system. The three types of log files maintained by the Mondex purse are described below:

- The Pending log maintains details of the current payment. Thus, this log file consists of a single entry (record). The log record information can be used to resume an interrupted payment (payment recovery). After certain commands the payment log record is moved to the Exception log or the Payment log.

- The Payment log keeps details of the ten most recent successful payments. It is a circular log, thus the oldest entry is overwritten with the most recent one. As soon as the balance in a purse changes, a payment log record is created. For example, this happens during the processing of the `Payment Request` command in the payer purse, or `Payment Value` command in the payee purse. In the case of the payer purse, the payment log record is marked as incomplete, since the payee purse has not yet acknowledged receipt of the value. When the payer purse receives a valid `Payment Ack` command, the payment is complete. Once the payee purse balance changes, the payment is considered complete and no further messages are expected.

- The Exception log stores the details of payments that failed to complete successfully. This log file will eventually ensure that lost value can be restored. The entries in this log file are never overwritten but they can be erased by a purse provider after being transferred elsewhere. The log file holds three records and payments are not permitted when the exception log is full.

3.5 The MPCOS Log Files

Log files are used in MPCOS cards in their simplest form. Sensitive commands (e.g. creation of files, payment commands, etc.) can be monitored by computing cryptographic *certificates* (e.g. `pMonitor` command) that can be transmitted to the smart card terminal for future reference. This feature is used to record and trace the aforementioned commands.

The cryptographic *certificate* is based on the sensitive command counter value (3 bytes) and the previously executed sensitive command header. Thus, a very simple file that will hold this sensitive counter value is required. The Sensitive Command Counter is stored in a Transaction Manager File and it is updated automatically every time a sensitive command is used.

Details on where and how this file is created or maintained do not appear in the MPCOS reference manuals. It seems that this log file is maintained by the MPCOS smart card operating system and it is automatically updated and provided to the `pMonitor` command whenever is required.

3.6 Summary

The main concluding remarks of this chapter are the following. Firstly, there are a few applications using smart card log files. Secondly, there is limited theoretical work

on handling log files and each proposal examines the problem from a different point of view.

For example, Schneier and Kelsey assume that the card can be compromised and that there should be a trusted entity that will periodically check the validity of the log files. This scenario is not unrealistic but not very suitable for smart cards. The work of Bellare and Yee is important, since, apart from its theoretical contribution it also provides details and performance measurements from an experimental implementation of the proposed architecture.

As we describe in the next chapters our proposal examines what happens when the smart card log files fill up and need to be downloaded to another entity which does not suffer from storage restrictions. We know of no previous work in this area.

Chapter 4

A Model for Handling Log Files in Smart Cards

Contents

Traditionally smart card log files are mainly used for storing receipts for the successful or otherwise completion of certain events. In today's multi-application smart cards a whole range of new events require logging. This last point introduces significant changes not only in the nature of the events to be logged but also in the definition and implementation of the actual event logging mechanisms. This chapter highlights the issues involved when maintaining different types of log files in smart cards in the light of the recent technological advances. It also introduces a proposed event logging model for smart cards.

4.1 Introduction

Audit log mechanisms are very powerful tools for monitoring a system's activity. Although much work has been done in the area of audit log design and implementation for databases and general computer systems [35, 51, 36], the distinctive nature and technological characteristics of current smart card technology impose specific constraints that demand careful investigation.

Audit logs for smart cards are potentially of great importance, since they increase the evidence available in the event of fraud, while at the same time providing evidence of the completion of important events. This is particularly true nowadays, as smart cards are moving away from their traditional focus based on single application smart card operating system [9, 13, 42] towards multi-application environments [12, 55].

As in the past the usage of log files in smart cards was limited, obvious questions can be raised as to why log files were of limited use and why the situation has changed.

The answer to the first question is rather simple: the limited storage capacity of the smart cards, along with the restricted functionality of the smart card operating systems, made audit logs virtually impossible to implement. These restrictions forced the smart card application developers to seek alternative methods in order to avoid providing the log file functionality.

The answer to the second question is more complicated and will be answered in the following paragraphs: Recent improvements in existing smart card operating systems [19, 31, 61, 62] along with related theoretical work [16, 20, 52, 54] and further improvements at the hardware level [11, 14, 17, 33] have made the whole idea more feasible. At the same time this new type of technology, as described in chapter 2, introduces further complexities which require revised and more dynamic log file handling mechanisms. The need for new techniques primarily arises because there are new events to

be logged, both in the operating system and the application level.

Additionally, there is often some confusion when the term smart card log file is mentioned. It appears that most of the entities involved (smart card holders, smart card manufacturers, application developers) maintain their own notion of smart card log files, and each expect different functions from such files.

In order to better understand the issues involved, together with the roles, requirements and characteristics of the entities involved, we use the following hypothetical example of a multi-application smart card. This smart card will hold the following applications: an electronic purse application, a loyalty points application used in conjunction with the purse application, a health care application, and an application for digitally signing emails or other information. We will refer to this example in the following sections.

In this chapter we outline the main characteristics of the different types of smart card log files. Subsequently, we present the different entities involved in a smart card logging scenario. Similarly, we present an overview of the new events to be logged. Finally, we describe our ideal log file handling model for a smart card environment.

4.2 Smart card Log File Types

In a real world smart card environment, log files will mainly be used in order to allow the SCOS to recover from fatal failures and to provide evidence regarding the progress of certain events. The latter implies that the log files might also be used in an auditing procedure in order to resolve certain disputes. In this section we also provide a classification of smart card log files.

The uses of log files in a smart card environment may be divided according to the following criteria:

- purpose: audit trail or recovery log,
- architectural level: operating system or application.

The details of each category are presented in figure 4.1 and analysed in the following paragraphs.

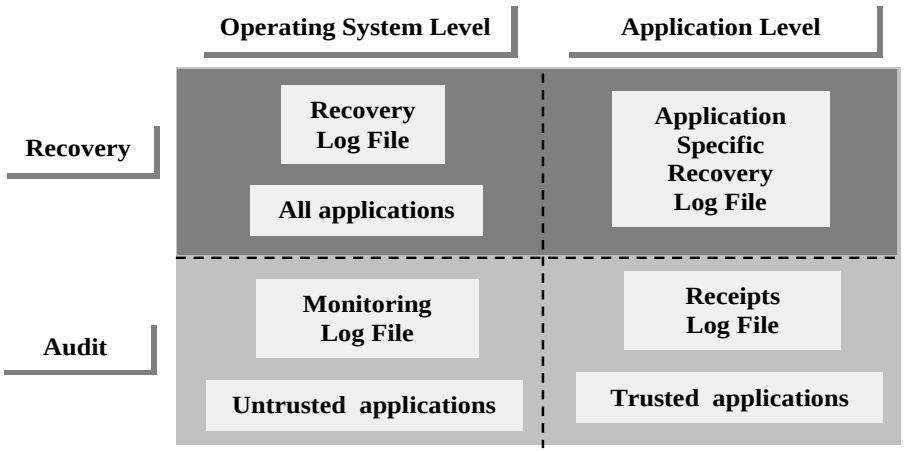

Figure 4.1: The different types of log files.

In the recovery category we come across two different types of log files, depending on whether the log files are handled by the SCOS or by the smart card application. The benefit from the existence of a recovery mechanism at the SCOS level is that smart card applications will not have to replicate their own recovery code. On the other hand the disadvantage is the complexity of such a mechanism since it will have to constantly monitor and identify the events to logged. An example of a recovery log file is the Pending Log maintained by the Mondex purse (refer to §3.4).

An example of a recovery mechanism at the application level involves the scenario that the electronic purse mentioned in §4.1 crashes. Then, a recovery mechanism that will take advantage of the information stored in this log file could help the application to recover the next time that it is selected for execution. The obvious advantage is that since the recovery mechanism is defined by the application programmers it will not rely on the SCOS in order to constantly monitor the application currently executing. The disadvantage is that each application has to replicate its own recovery mechanism

A different type of smart card application recovery is proposed by Trane and Lecomte in [64]. The authors suggest that certain recovery information is partially kept in the smart card terminal, and thus, recovery is achieved the next time the card interacts with the terminal. Clearly, the recovery log files will typically not be used as evidence in cases of dispute. The details of the recovery mechanisms are outside the scope our research.

In the audit trail category, the light grey area in figure 4.1, we also encounter two different types of log files, again depending whether the log files are handled by the SCOS or by the smart card application.

An example of a log file maintained by the SCOS in the audit trail category is the Monitoring log file. The Monitoring log file will hold information for identifying licensing or pay-as-you-use details of the downloaded smart card applications. For example consider the scenario of the electronic purse application mentioned in the previous section, which might offer certain smart card programming primitives which will be used by other applications (i.e. in our example by the loyalty application). It could be the case that these primitives are offered at a pay-as-you-use basis. Therefore, the loyalty application will be charged for the number of times it used the purse's functionality. When the Monitoring log file is examined it could provide all the required information as to which applications used which primitives and how much they should be charged.

The Receipts log file is an example of an audit log file maintained at the application level. It will hold information generated by smart card applications. In our example this log file will hold the payment receipts generated by the electronic purse application. Similarly, the email signing application should also generate evidence on the digital signatures performed. The Receipts log file will remain in the smart card application space. Permission to make entries in this log file will only be granted to authorised and trusted applications (i.e. the applications carrying valid application certificates). Generally, both the Monitoring and the Receipts log files will be subject to auditing.

In the following subsections we propose an architecture that will identify certain events to be placed in the Monitoring and Receipt log files. Additionally, we are interested in what happens when the smart card log files become full. We have excluded provision of smart card application recovery, although in order to get a better understanding of the issues involved we highlight certain recovery aspects, since it would either require knowledge of the application structure or the internals of the SCOS. In the following section we describe our log file handling model.

4.3 The Event Logging Model for Smart Cards

In this section we describe an event logging model and we also highlight certain issues that need to be taken into account when addressing log files in smart cards.

4.3.1 The Entities of the Model

In a smart card environment there are a number of different entities which might benefit from the existence of log file mechanisms [22]. The principal participants and relationships between participants are depicted in figure 4.2.

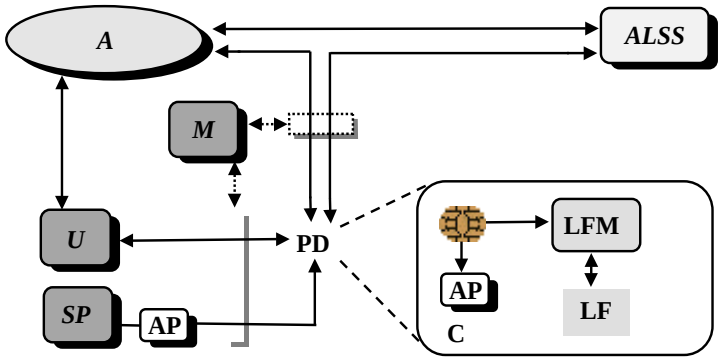

Figure 4.2: Graphical representation of the relationships among the participants.

1. C represents the smart card. Typically this is a sufficiently tamper resistant device which is relatively difficult to compromise and has access to a variety of

cryptographic algorithms.

2. U is a smart card holder, the user.

3. AP is an application running on the smart card.

4. SP is a service provider which offers certain services and provides the relevant applications (APs) for the cards.

5. M is a malicious user with possibly enough knowledge of the internal structure of the card. It can also eavesdrop the exchange of messages during the log file download procedure.

6. ALSS stands for an Audit Log Storage Server. This entity will receive and store the transmitted log files in dedicated locations. Depending on the environment the ALSS could be a "smart wallet", a personal computer used in conjunction with the smart card or even a repository server connected to the Internet.

7. LF is the log file which will be downloaded.

8. LFM is the log file manager, a smart card operating system entity. This is the only authorised entity for updating, browsing and downloading the log files from the card.

9. A is the arbitrator who informs the entities involved and receives the downloaded log files in order to resolve any disputes.

10. PD (Personal Device) is the smart card reader/writer or personal wallet used by the smart card in order to communicate with the outside world.

4.3.2 How to Record Events

Smart card operating system logging is a way of storing data to facilitate transaction atomicity [29] and failure recovery [64]. In [64] Trane and Lecomte propose three kinds of logs that can be used for transaction recovery. These are presented in figure 4.3.

- Value log: this type of log file stores the previous value of a variable in a previous value log and the new value in a current value log. An undo operation requires a simple computation that will recreate the initial value of the variable. The main advantage of this type of log is the relatively low memory consumption which makes it appropriate for smart cards. On the other hand it is not sufficient when extra details of the logged event are required. For example, consider the case that our health care application updates a large amount of card holder medical data. In that case it will be difficult to store the previous and after values.

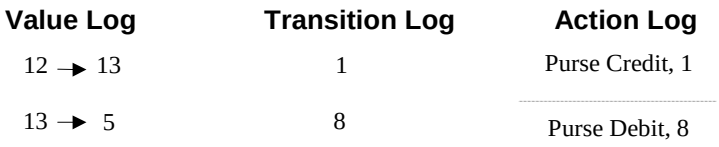

Figure 4.3: Types of recovery log files.

- Transition log: this log stores the "difference" between the old and the new values along with the actual direction of the operation (e.g. plus, minus, etc.). Since the new value is stored in memory and the "difference" in the log file, simply applying the inverse operation will restore the initial value. The main advantage of the transition log file is its low memory consumption compared with the other two kinds. This type of log file works well with numeric values, although it has major drawbacks with other types of logged events, like the previous log file type.

- Action log: this log file contains the names of operations on variables, their arguments, and even sometimes their results. This type of log file is useful when ending transactions after failures. It is more generally useful, e.g. for resolving disputes, since it contains more details of the operations performed. On the other hand it requires large amounts of memory since it stores a significant number of details. Undoing any operations becomes harder since a "reverse" or compensation operation must be logged for each operation.

The notion of smart card transaction atomicity is introduced in [31] and "defines how the card handles the contents of persistent storage after a stop, failure or fatal exception". This is a programming concept in order to enable the application developer to know what values the updated data fields contain, either in case power is lost during any memory updates or after a smart card application protocol is aborted. The aforementioned mechanism rely on the application programmer to define a section within an application that need to be executed atomically.

We realise that the above mechanisms address the recovery issue. On the other hand in respect of the SCOS and auditing there are no mechanisms defined or implemented that will dynamically, as the application is executed, identify the events to be logged.

4.3.3 Structure of the Log File

A smart card holder of the multi-application smart card example mentioned in §4.1 might be interested in logging the successful or otherwise completion of an electronic purse transaction or the calculation of a digital signature by the email signing application. Conceptually, it would be beneficial for the smart card holder to be able to maintain a representation of the log file structure of the type presented in figure 4.4.

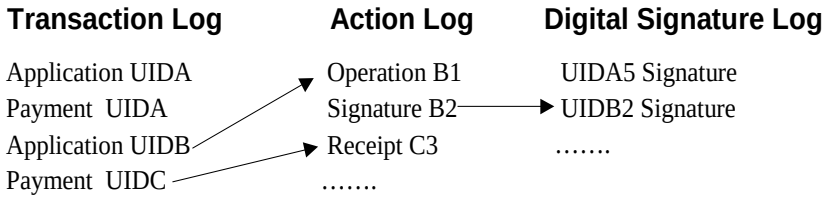

Figure 4.4: The ideal relationship among the card holder smart card log files.

Each entry in the transaction log file would contain the unique identifier for the downloaded application, which will serve as a pointer to the action log file containing the exact operations performed by each application. If for example an application performs a digital signature, a pointer will be created in the digital signature log file.

Some basic knowledge of the internal structure of a smart card will soon reveal that the aforementioned log file architecture is not feasible for the following reasons:

- limited memory space allocated for the log files,
- due to the cross-references between the log files there will be difficulties when the log files are to be downloaded to some other medium [21], or when it is necessary to verify their integrity as described in [59], and
- more importantly some application developers might not be willing to make the internal structure of their applications widely known or exactly when a digital signature is completed within their payment protocols.

Thus, the above observations indicate that a practical implementation of a log file mechanism requires that the events need to be logged in a single log file.

4.3.4 Interested Parties

Naturally, the smart card holders are interested in maintaining some evidence of the transactions performed by their smart cards. Note that, smart card holders might have some interest in the information stored in their cards but on the other hand it could also be in their interest to avoid logging certain details.

Obviously, legitimate service providers need to be protected from any artificial claims against them or against legitimate card holders. Thus, the existence of substantial evidence is considered crucial.

Consider the following scenario where the electronic purse and health care applications perform the following transactions, `(1:Debit, 2:Payphone Debit, 3:Update Medical Data, 4:Credit, 5:Debit)`. The user goes back to the electronic purse service provider and claims that something went wrong between transactions 2 and 4. The

interested parties can then easily examine the log files providing both their customers and themselves with enough evidence of exactly what happened.

Additionally, the service providers might wish to obtain specific feedback concerning the testing of certain functions or applications or even evidence from any intrusion detection mechanisms within the cardholder's smart card.

4.3.5 The Dispute Resolution Phase and the Arbitrators

In a smart card environment disputes might arise and of course, they need to be resolved. Dispute resolution requires the existence of evidence. The required evidence may reside in the audit logs. The dispute resolution might involve a trusted entity e.g. an arbitrator. Therefore, if an arbitrator is involved then he/she will need to verify that the logs are not tampered with. Similarly, the availability of the log files is another issue. This means that the log files should become available whenever they are needed.

4.3.6 Threat Model

We now describe the assumptions made about security threats to the smart card and its operational environment. These assumptions underlie the definition of the log file handling model and also the definitions of the operation of components of the model which are given in the next two chapters. Note that we only consider those threats which may effect the integrity and/or confidentiality of the log file information. We are not concerned with more general threats to the smart card and its operational environment.

- We suppose that the smart card holder might attempt to modify or delete the log file information. In some cases the smart card holder will wish to protect the content of the log files, although in other cases the cardholder might wish to

prevent the logging of certain events (and/or delete the log information after the event). The log files could be attacked while stored in the card, either by physically attacking the smart card microprocessor or by attempting to bypass the SCOS and Java Card security. Additionally, the log files could be attacked while they are transmitted to the ALSS, e.g. by subverting the log file downloading protocols.

- We suppose that the smart card is a physically secure device with adequate tamper resistance mechanisms. Therefore, we assume that certain information can be securely stored internally to the card. Having said this, we also assume that there are potential threats to the integrity and confidentiality of the log file arising from the need to download portions of it.

- We suppose that smart card applications may be a threat to the log files. Certain rogue applications could potentially attack both the smart card log files in order to modify certain valuable information but also data and application code belonging to other smart card applications.

- We treat the ALSS as a trusted entity. Thus we assume that the ALSS ensures that the log files are properly received, i.e. by following the steps of the log file download protocol. The ALSS will also have to protect the integrity, confidentiality and availability of stored log files. In summary, we assume that the ALSS is trusted by the cardholder to properly receive and store their log files and by arbitrators when they request the log file of a cardholder.

- We suppose that the arbitrators are trusted entities in the sense that they receive the log file information and that they honestly try to resolve any disputes using this evidence.

- The dependence of the components of our model on the smart card reader device is minimal, and therefore we assume that any standardised (e.g. PC/SC compliant) smart card reader device could be used.

Arising from our threat model we can identify a fundamental set of security requirements which need to be met by the log file handling system.

- In order to minimise the chance of the log files being attacked while stored in the card, we suggest that the log files should only be accessible through the LFBM. In order to increase the protection of the log files in transit, careful LFDM design is considered essential. If the log file transmission uses a relatively insecure channel (e.g. the Internet), the assumption underlying the design of the second and the third log file download protocols in Sections 6.3.3 and 6.3.4, we assume that the transmitted data may be subject to passive and active attacks. Where the log files are transmitted in the cardholder's PC, i.e. the assumption underlying the first log file download protocol in Section 6.3.2, we assume that only the cardholder can attack the communications channel. In such a case security measures to protect the transmitted data are unnecessary, since the cardholder will have free access to the log files once they are stored on the cardholder PC.

- Smart card applications should be subject to the general Java Card security model and they should also adhere to the security of the underlying SCOS.

- The log file download protocol must take measures to ensure that it always takes place between a genuine card and a genuine ALSS. The measures must be appropriate to the environment within which the protocol is used. The measures will typically rely on certain information (e.g. cryptographic keys) being maintained as secrets internally to the card.

- All the components of the model internal to the card should be adequately protected from modifications when placed within the SCOS. For example, by placing certain functionality at the SCOS any EEPROM modification attacks are eliminated; however such an approach makes components more difficult to upgrade.

4.4 What Information Should be Logged?

The question as to which information should be logged in a smart card environment is significantly more difficult to answer than it would be for a conventional computer system. The main reasons are the hardware and software characteristics of the smart card processors that seriously restrict both the amount and the nature of the events to be logged. In this section we highlight the issues that influence the decision on which events should be logged.

The decision is highly dependent on the following factors:

- the system's security policy,
- the space allocated for the log file,
- how often the log file is downloaded, and
- who decides which events should be logged.

If, for example, a large number of events are characterised as critical, and thus they should be monitored, then the risk of running out of log file space is increased. On the other hand, if just a few events are to be monitored, there is always a chance that some important events might be missed. A summary of the type of events that might be logged in a smart card environment is given in Table 4.1.

Precisely which events and actions are to be logged depend on the system's security policy. The policy should be sufficiently detailed to enable identification of the types of event which require logging.

Another important factor influencing the nature and the amount of information to be logged is the space allocated for the smart card log files. Ideally, the allocated space should permit logging of all significant events without the need for overly frequent

download operations, which probably means at least 4–5 Kbytes. Although this memory space would allow the logging of sufficient information, more realistically speaking the space offered by the smart card manufacturers and the service providers would be within the range of 1.0–1.5 kbytes. Another difference compared with computer based logging is that it is not possible to have a data reduction tool in a smart card. Thus, the data to be stored should be reduced to the minimum prior to its storage in the smart card log files.

Table 4.1: Suggested events to be logged.

Receipts	**Application Monitoring**
Transaction Type (credit, debit, etc.)	Application Download/Delete
Total Transaction Value	Use of Cryptographic Primitives
Transaction Status	Freeze Operations
Type of Currency	Certificate Updates/Changes
Transaction Time/Datestamp	Intrusion Detection Results
Transaction ID/code	Invalid PIN Presentations
Digital Signatures	Certain EEPROM Updates
Log File Downloads	File Deletions/Updates
	Reading, Browsing Files
	Flushing Memory Space

In a smart card environment it is currently impossible to implement the post-selection method described in [36], i.e. log as many events as possible and decide later which ones should be audited. Obviously, the main restriction is the limited space assigned for the log files. Thus, the events to be logged should be pre-selected. The main advantage of the pre-selection method is that only a number of pre-selected events are logged. As a result, there is more efficient log file space management. The obvious disadvantage is that it is relatively difficult to predict the events which might be of security interest at a future date. Hence, if the table containing the events to be logged is stored in ROM of the card (in order to be more adequately protected) it would be impossible to add new events. If the table is stored in the EEPROM of the card then it becomes easier to update the list of logged events, but on the other hand it becomes more vulnerable. Probably, the most efficient solution would require that certain events are stored ROM and some other in EEPROM.

The supporting infrastructure for downloading log files is another critical factor. For example, if the log files can be downloaded from the card often, then the amount of data to be logged may increase. The log file downloading procedure could be enhanced by offering the facility in secure public terminals or even over-the-air transmission (by using mobile phones).

Finally, another influencing factor is which entity decides the events that should be logged. Is it the card holder, the card issuer or the service providers? Actually in an environment of so many conflicting interests it is not easy to find a solution acceptable to all interested parties, and the answer to the above question should appear in the system's security policy document. For that reason we do not try to explicitly define which events should be logged. Instead, we just provide an indication of the events which are of significance for the above entities.

4.5 The Smart card Log File Manager

In this section we describe an event logging model for the use and operation in a smart card. The main entity of the model is the Log File Manager (LFM) as the only entity authorised to access the smart card log files. The LFM has to perform the following three tasks:

- create and update the log files,
- take control of the log file download procedure, and
- browse the log files while stored in the card.

The functionality of the LFM is divided into the entities presented in figure 4.5.

The above entities are briefly analysed in the following paragraphs.

- the Log File Update Manager (LFUM). The LFUM is responsible for identifying the events to be logged both at the application and the operating system level. Therefore, the LFUM will identify security critical events and will update the audit log files. The details of the LFUM are presented in chapter 5.

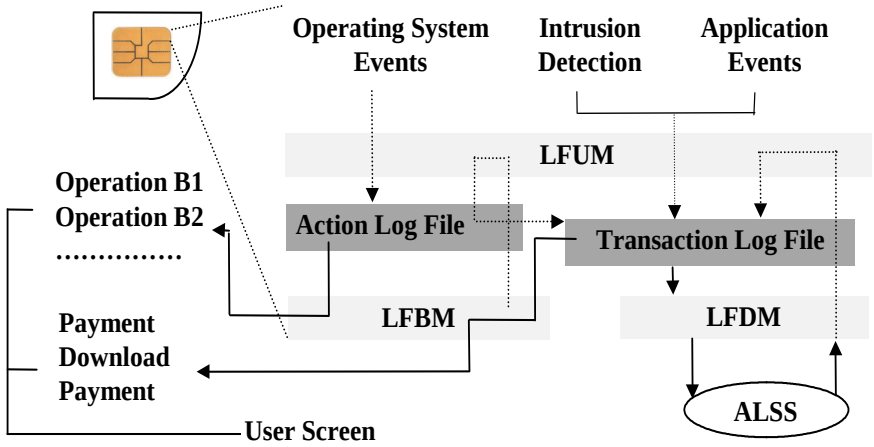

Figure 4.5: Relationships among the entities authorised to access the log files.

- the Log File Download Manager (LFDM). The main role of the LFDM is to securely download the log files from the card to an Audit Log Storage Server (ALSS). Upon the successful completion of the log file download procedure the LFDM will communicate with the LFUM in order to inform about the successful or otherwise result. The exact behaviour of the LFDM is defined in chapter 6.

- the Log File Browse Manager (LFBM). The LFBM is responsible for providing the cardholder with browse functionality to the transaction log file. The details of the LFBM are presented in chapter 5.

4.6 Summary

This chapter serves two objectives: Firstly, it provides a classification of smart card log files. This classification is essential in the light of the recent technological advances, e.g. real multi-application smart cards.

Generally speaking the smart card log files are divided into recovery logs and audit logs. Recovery logs are maintained by the smart card applications or by the SCOS. They are mainly used to recover applications or the SCOS in a consistent state after a "crash". As it becomes evident our work focus in the audit log category. The audit log files are examined by external entities, arbitrators, in order to reconcile any disputes.

Secondly, it introduces our model for smart card based logging. The major three entities of the model are: the Log File Update Manager, the Log File Download Manager and the Log File Browse Manager. The exact behaviour of each of the entities will be described in the subsequent chapters.

Chapter 5

Effective Smart Card Log File Logging

Contents

In this chapter we describe the behaviour of the Log File Update Manager as a tool for implementing dynamic logging. Additionally, we very briefly describe the main characteristics of the Log File Browse Manager. Finally, we propose a standard format for smart card log files in order to make the dispute reconciliation procedure easier and faster and to help efficiently manage the valuable log file space.

5.1 Introduction

The information to be recorded by the LFUM in the log files can be divided into receipts and dynamic logging. In the following paragraphs we analyse the characteristics of each category.

Receipts are log file records generated by the smart card applications, and indicate the successful or otherwise completion of certain events. In this category we mainly encounter receipts from financial transactions, e.g. the information stored in the Payment log file of the Mondex purse. We assume that smart card applications authorised to add entries (receipts) to the log file should carry a valid application certificate. Currently, an application certificate provides assurance of the origin of the application. In our case the application certificates will also indicate that the corresponding applications are "trusted" to append entries in the log file. The application certificates can be compatible with the application certificates as defined in Multos API or the Java Card 2.1 API.

In order to achieve adequate smart card log file space management, the smart card developers should either have defined their own fixed receipt format or they should comply with the proposed standard log file format as defined in §5.5. Since the receipt information can easily be generated by the smart card applications, and appended to the log files, we will not be examining the issue any further.

The main task for the LFUM is to perform dynamic logging or application monitoring. Dynamic logging involves identifying the events to be logged while the smart card application is executing, without having any prior knowledge of the content of the application. For example dynamic logging can be used either to identify pay-as-you-use details or to log the events generated by applications that do not carry application certificates (e.g. less "trusted" applications).

In order to better understand the concept of dynamic logging, consider the case when the electronic purse application introduced in chapter 4 provides certain functionality to be used by the loyalty application, see figure 5.1. In our example, in order for the loyalty application to calculate the loyalty points earned it has to obtain certain information (e.g. `GetAmmount(Y)` the amount spent, and `Sign(Y)` a signature on the amount) from the electronic purse application.

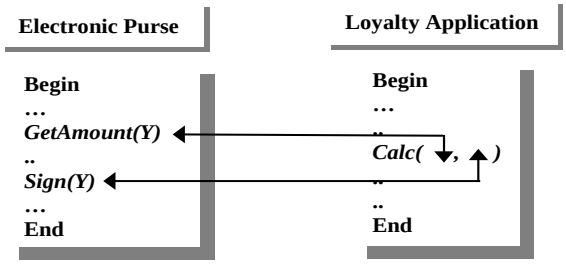

Figure 5.1: An example where certain functionality of an application is shared with another application.

Subsequently, the electronic purse providers might wish to charge, the loyalty application on a pay-as-you-use basis whenever it uses the purse's functionality. In that case, the charging details could be identified (e.g. application monitoring or dynamic logging) by the LFUM and stored in the smart card log files.

We start by presenting the advantages and disadvantages of the candidate mechanisms that will perform smart card application monitoring. We then describe some security properties of these smart card logging mechanisms. Subsequently, we describe the characteristics of both the Log File Update Manager (LFUM) and Log File Browse Manager (LFBM). Finally, we present the details of a proposed standard log file format for smart cards.

5.2 Dynamic Logging of Smart card Applications

In this section we describe the issues involved in smart card dynamic logging.

5.2.1 Dynamic Logging Mechanisms

In order to achieve application monitoring or dynamic logging we can use various methods. The advantages and disadvantages of each method are presented in the following paragraphs.

- Compiler Inserted Trace Points: During compilation of the smart card application code the compiler inserts special marks at specific code portions to be monitored. When the application is interpreted by the smart card microprocessor a SCOS based monitoring mechanism will constantly examine and log the events surrounded by these specific trace points. This method reduces the SCOS complexity. On the other hand, the complexity of the smart card application compiler is increased, and thus, a lot of trust is placed in the compiler. The security of this scheme depends on the integrity of the smart card application compiler.

- Re-defining the Smart Card Commands: Re-write the smart card commands that need to be logged. Therefore, every modified command will include the necessary logging statements. For example the `write(x)` command, that writes value `x` in EEPROM, will be replaced by `check.write(x)`. All methods and commands starting with "`check.`" will create the necessary entries (for the relevant events) in the log file in addition to performing their normal operations. The advantages of this approach is that it imposes no further interpreter and compiler complexities, and it is relatively easy to implement. The main disadvantage is a relatively small increase in the size of the smart card primitives.

- Smart Card Application Certificates: If the smart card applications carry application certificates they can be granted permission to append entries in the log file. With this method logging is achieved through minimal changes in the smart card's hardware and software architecture. On the other hand, the application download procedure is restricted only to authorised applications.

- Smart Card Operating System Monitoring Mechanism: Introduce a monitoring

mechanism (LFUM) that will constantly monitor and log the behaviour of certain smart card applications. The main advantage is that the mechanism is effective, since it does not rely on external resources (e.g. the compiler or application certificates) for performing part of the logging procedure, and thus it will be easier to be accepted by the smart card application providers and smart card holders. The obvious disadvantage is that the size and complexity of the in-card interpreter or the SCOS is increased and the application execution becomes slower.

5.2.2 Observations for Smart card Dynamic Logging

In the Orange Book [37] and in the Guide to Understanding Audit in Trusted Systems [36], audit log security and the requirements for logging mechanisms are defined informally in Requirement 4 and section §5.4 respectively. When migrating these requirements into a smart card environment they are no longer entirely applicable due to the smart card hardware and software characteristics.

In our design, as long as the Trusted Computing Base (TCB) — the SCOS — retains its integrity, and the LFUM (when provided as part of the SCOS) is responsible for enforcing the logging policy, the log files should of course, retain their integrity.

In a smart card environment the entries of the log file should be created after the actual completion of the corresponding task. For example, consider the digital signature command `Sign(Y)` of the electronic purse application in figure 5.1. If the command `Sign(Y)` is considered as a security relevant event and should be monitored, then the digital signature will take place first and subsequently an entry will be created in the log file. This design decision will avoid the situation when an event is logged in the log file, and subsequently the task is not completed for various reasons.

Let us consider the following scenario: a smart card command is being executed but when the corresponding event is about to be logged there is an execution error, e.g.

the card is removed form the reader. Although, this is not a very realistic scenario since in order to be planned in advance it will require accurate synchronization, it could always take place by chance. A simple solution will require another file with a "pointer" to the actual command line number or some other indication as to which command is currently executing. Before each command is executed, this pointer will be inserted in the aforementioned file. Subsequently, the command will be executed and then an entry will be created in the log file. If the creation of the log file entry fails the next time the application is selected for execution the current command to be executed will be identified as an entry (i.e. the "pointer"). If the command should have been logged and the corresponding entry does not appear in the log file, then a new attempt will be made. Generally speaking the proposed architecture could be part of a general application recovery mechanism that will be responsible to recover the application to a safe state after a smart card application crashes.

5.3 The Log File Update Manager

In this section we extend the notion of sensitive command monitoring of the MPCOS cards already presented in §3.5. Instead of generating a certificate and transmitting it to the terminal we propose a mechanism (LFUM) which will dynamically identify the events to be logged and update the corresponding log files. Thus, in this section we present the mechanisms that will identify the events to be logged along with their advantages and disadvantages.

5.3.1 Operation of the Log File Update Manager

In the following paragraphs we demonstrate that designing a more dynamic logging mechanism, that will identify the events to be logged "on the fly", is not a very difficult task. A simple and efficient enough implementation would require access to the smart card operating system (SCOS). Placing the LFUM in the SCOS will clearly achieve

faster execution times (since it will be written in the microprocessor's machine language) and will ensure that it is relatively difficult to be modified since it will be masked in ROM. Following the above strategy will also ensure that the logging policy is enforced by all smart card applications. The behaviour of the LFUM is outlined in figure 5.2.

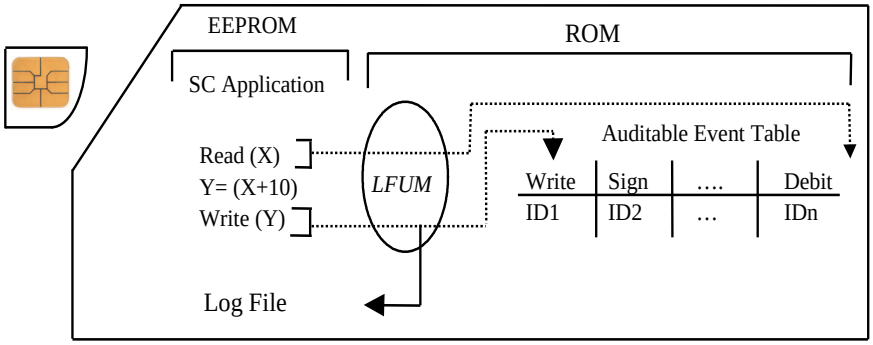

Figure 5.2: The behaviour of the Log File Update Manager.

One major component of the LFUM is the Auditable Event Table (AET), stored in the ROM of the card. If the AET table is placed in the EEPROM memory of the card it will become easier to be updated with new commands that require logging. In our example this table is shown in the right hand side of figure 5.2 and it is stored in the ROM of the card. This table contains all the events, mainly smart card commands, to be monitored along with their corresponding identification string (IDn). The identification string of each command could be stored in the log file instead of the actual commands. The size of the AET depends on the number of events to be monitored, and might typically be 200–300 bytes.

Different applications could be subject to different logging policies according to their logging level. The logging level of an application indicates the number of events that will be logged or otherwise monitored, in each application. The logging level of each application can be stored in a table (refer to table 5.1) maintained by the SCOS, this implies that the SCOS will be the only entity accessing this table, and it will be stored in the EEPROM memory of the card.

Whenever an application is downloaded in the smart card the SCOS, based on in-

Table 5.1: The Application Logging Table (ALT).

Application ID	Logging Level
MON14PURSE477890	1
TIC987KJL64E788F	3
...	...

formation provided by the application loader, will create an entry in the Application Logging Table (ALT). For each application downloaded, the ALT contains an entry of the application's unique identifier [40] along with its corresponding logging level.

For example let us suppose that our electronic purse and loyalty applications, have the following application identifiers "MON14PURSE477890" and "TIC987KJL64E788F" respectively. The logging level of the purse application "1" indicates that it has trustworthy origins. This implies that the application's operations should not be dynamically logged and that the application is granted permission to add entries (mainly receipts) in the log file. On the other hand, the logging level of the second application "3" indicates that the application should be monitored.

Another critical decision concerns how dynamic logging is initiated. When an application is selected for execution, the SCOS will check the ALT in order to identify the logging level of the selected application. As soon as the logging level is obtained it is passed to the LFUM.

The role of the LFUM is to constantly monitor the execution of each smart card command according to the previously identified logging level. If an entry is found in the AET, the LFUM will immediately create a corresponding entry in the log file.

5.3.2 Miscellaneous Characteristics of the LFUM

If the the ALT is stored in the EEPROM of the card, the SCOS developers can define additional logging levels simple by partitioning the AET into smaller sections. For example, they could place a certain set of commands to be monitored (i.e. the first five commands of the AET) under the logging level "2a". Another set of more sensitive commands could be placed under a different logging level. The above model works well when the applications to be downloaded contain unique application identifiers. In any other case the SCOS should allocate a locally unique application identifier. This task is actually simplified due to the global attempts to standardise the smart card application identifiers [40].

Apart from being used for dynamic logging the LFUM can be useful when adding receipt information in the log files. It can act as the entity to receive the receipt information, check if it conforms to a pre-defined receipt format, and finally append the receipt information in the log file or simply reject it along with issuing a relevant message.

5.4 The Log File Browse Manager (LFBM)

The LFBM is responsible for providing the cardholder with the capability to browse the log files while they are still stored in the smart card. Thus, in order to crystallise the features of the LFBM we can think of it as the only entity which can access and present the content of the log files to the card holder.

In order to make it more difficult to modify it would be ideal for the LFBM to be part of the SCOS and masked in the ROM of the card. It will simply read the content of the log file and present it to the user's terminal (smart card reader, PC, etc). In other words the LFBM offers an interface between the smart card log files and the card holder. The LFBM should verify the cardholder's PIN prior to allowing read access to the log file.

5.5 Smart card Log File Format Standardisation

In this section we present a possible standard format [23] for the smart card log files. We start by justifying the need for a standard audit log file format. We then describe the format of the log file and finally, we provide a short discussion on certain practical issues around the proposed architecture.

5.5.1 Why Standardise a Smart card Log File Format?

A similar proposal of a standard audit trail format for a computerised system is presented by Bellare and Yee in [4]. The two main issues that the authors addressed are extensibility and portability of the audit log files.

A standardised log file format in a smart card environment introduces several benefits.

- It makes automated log file analysis and reconciliation easier. Suppose that two smart cards are involved in a financial transaction. Subsequently, one of the smart card holders denies the transaction and contacts an arbitrator. The arbitrator's task when accessing the log files is simplified if the log files conform to the standardised log file format. As a result the dispute resolution procedure can be speeded up.

- It will allow more effective log file storage management. If, for example, an upper limit is set on the amount of data which can be stored in the log file by each application, then the problem of an application over-using valuable log file space is eliminated.

- Similarly, addressing the portability issue is of importance since it will allow the log files to be transferred and handled by various systems (PCs, Electronic Wallets, or large Audit Log Storage Servers). For example, a smart card log file should

have a format that should not make difficult its smooth transportation between the above entities.

- We also believe that if the smart card log files conform to a standardised log file format, the use of log file filter engines [4], which take as input the smart card log files as they are created and translate (post-processing) them into a standard format, is made redundant. Post-processing the smart card log files might be a problem since the whole log file might be encrypted and/or signed before being extracted from the card. Thus, it would be difficult to make the smart card log files conform to a standardised format after they have been extracted from the card. This implies that log file processing has to take place while the log files are still stored in the card.

5.5.2 The Content of the Log File

We start by specifying the characteristics of the smart card log file. The log file can be a *transparent* file (i.e. a simple linear file) [42]. Its size will depend on the number of applications downloaded in the card and the technical characteristics of the card; realistically speaking a smart card log file will probably have size around 1.0–1.5 Kbytes.

We assume that the log file consists of a list of data entries (DEs), which represent a record for the events to be logged by each application, and each data entry is assigned a unique serial number in numerically increasing order. The maximum number of data entries (w) depends on the size of each data entry and the size of the log file. Thus, if we denote the i^{th} data entry in the log file by DE_i, the log file F will have the following form

$$F = (DE_1, DE_2, \cdots, DE_i, \ldots, DE_j)$$

for some $j \leq w$ and $1 \leq i \leq j$.

Ideally each data entry should correspond to multiple auditable "events", and take the form specified below. Note that we assume that the card's LFUM is the only entity responsible for updating the entries in the log file. We further assume that:

$$DE_i = DeBind; f_{K_{AC}}(DE_{i-1} \parallel DeBind) \parallel E$$

where

$$DeBind = AID \parallel N_C \parallel (LE_1; LE_2; \ldots; LE_e)$$

The separator character between fields is ';' and "$\parallel$" represents concatenation of data items. The $DeBind$ variable contains the unique application identifier (AID) field as described in the ISO 7816-5 standard [40], the value of N_C which is a monotonically increasing sequence number, and the actual data field of the logged events is $(LE_1; LE_2; \cdots; LE_e)$, where e is the maximum number of logged events. The value of e will be a variable number depending on the size of the log file and the number of the DEs (see §5.5.3). Each LE_i, where $1 \leq i \leq e$, will either hold data from a transaction receipt or the unique event identification information (EID) for the corresponding event to be logged.

The $f_{K_{AC}}(DE_{i-1} \parallel DeBind)$ field represents the result of applying the MAC function f using as input a secret key K_{AC} and as data the previously generated data entry (DE_{i-1}) along with the current $DeBind$. The secret key K_{AC} could be generated by a trusted entity but will be known only internally in the card. The latter could be securely installed in the card either during the personalisation phase or at any later stage during the card life cycle when the card owner decides to activate the logging policy. The $f_{K_{AC}}(DE_{i-1} \parallel DeBind)$ field links the previous data entry with the most recent one. Finally, 'E' denotes the end of this data entry.

In general, log files contain a timestamp. A smart card will not possess a real time clock and would need to rely on external devices for the value of the timestamp. Hence

in this environment a timestamp would be of limited value. Thus, it was decided to make the inclusion of a timestamp optional. We assume that the sequence number N_C is generated using a counter held internally to the card, and which cannot be reset or otherwise modified by any external events. In particular, downloading the log file contents to an Audit Log Storage Server (ALSS) will not involve resetting the sequence number counter. The sequence numbers of the data entries (DEs) can thus be used to order downloaded data entries, and gaps in numbers can be used to indicate lost or deleted log file data entries.

As previously stated, the purpose of including a message authentication code (MAC) of the previous generated data entry is to effectively link pairs of successively generated data entries. Eventually, this will create a chain of linking information [63] of all the generated entries, and along with the sequence number N_C, will act as a protection mechanism against unauthorised insertion or deletion of LEs.

5.5.3 The Size of the Log file Data Entry (DE) Field

The length in bytes of each field of the data entry (DE) is as follows. The end character (E), along with the field separating characters (;) will be one byte each. Please note that the (||) field, indicating the concatenation of two data items, is not taken into account when calculating the size of each DE. The AID field could be represented as a 16 byte string, as defined in [40] and N_C could be represented as a 4 byte number (i.e. '0001', the counter for the initial data entry). The MAC of the linking information field will contain 4 bytes.

Let us assume that the size of the unique event identification (EID) information of each event is around 5 bytes (e.g. 'ID056', indicates an Update operation). Furthermore, let us assume that applications contain a maximum number (e) of LEs to be stored in each data entry, where (e.g. $e \leq 15$). A simple calculation will reveal that the LE entries, i.e. $(LE_1; LE_2; \ldots; LE_e)$, of each data entry along with the separator characters between

the fields will have an expected length in bytes of $LeLength = (5e) + (e - 1)$.

If a number of events (e) is used then the formula which derives the total space occupied by each data entry (DE) is the following:

$$\text{DE SpaceA} = 6e + 25$$

where 25 is the number of bytes for the separator characters between the fields, the rest of the actual fields, and finally the ending character.

The proposed standard log file format maintains a high level of flexibility towards successfully adapting itself and logging information from a variety of different applications. Therefore, although it is suggested that the LE field of each data entry (DE) should contain a number e of events, allocation of this space can be left to the discretion of the application designer. In this case the formula which derives the total space occupied by each data entry is the following:

$$\text{DE SpaceB} = 26 + q$$

where q represents the total number of bytes in the (LE) entries and 26 is the number of bytes for the rest of the fixed fields, (separator and other characters). Following this approach the application developers are left with the task of how effectively they can use these LE entries, to best suit their needs. For example, instead of having multiple LEs the same space could be regarded as a single larger entry that the application developers can use as they want. The above observation also addresses the extensibility of the log files since they maintain a very flexible structure.

5.5.4 Number of Data Entries in Each Log File

There are two different options when calculating the number of data entries in each log file, depending on the number of events to be logged.

The first proposal assumes that the log file contains a fixed number of events in each data entry e.g. $e = 15$. The maximum space (in bytes) of each data entry will be calculated according to the above formula i.e. DE SpaceA $= (6 \times 15 + 25) = 115$. With this approach the number of data entries in a log file (of size LFS) is the following: $w = \lfloor (LFS/DE\ Space) \rfloor$. For example, when a log file of 1.5 Kbytes is used we will get $w = \lfloor (1536/115) \rfloor = 13$ entries, which is not very restrictive at all.

The second approach assumes that each data entry might not contain the maximum number of logged events. For example, if a data entry logs 5 out of its 15 events and another data entry logs 10 out of 15, and so on, there might be space for extra data entries. The number of the data entries to be claimed will be assigned dynamically by the smart card operating system. On average around 2–3 extra data entries will be added by the existence of such a mechanism.

5.5.5 Practical Issues of the Log File Standard Format

Our proposals indicate a small number of events which need logging in a truly multi-application smart card environment. We believe that logging such information in the smart card log files provides extra evidence on the successful or otherwise completion of certain events.

Currently, due to the limited storage capacity of smart cards, a real world implementation of smart card log file size (LFS) will only be around 1–1.5 Kbytes. With our proposed standard log file format, the maximum number of data entries (w) will vary within the range 13–16. This implies that the smart card log file will hold a reasonable

number of data entries before it fills up and needs to be downloaded (as described in chapter 6) or simply over-written.

Furthermore, if the second approach defined in §5.5.4 is to be followed when allocating the data entries, and all the unused space of each data entry along with the truncated space resulting from formula w is collected, then extra space for a small number of additional data entries might be created.

Finally, another issue which needs to be taken into account is the relation between the data entry size and the log file download frequency. For example, if the log files are regularly downloaded from the card, then the size of the data entry could increase in order to record any additional events.

5.6 Summary

In this chapter we examined different techniques that could be used for smart card application monitoring. Having examined the characteristics of the different proposals we concluded that the most appropriate method would be to provide a monitoring mechanism as part the SCOS. Today's smart card technology offers both the required processing power and the storage capacity in order to fulfill the requirements of the proposed dynamic logging mechanism.

Our mechanism can be used both for smart card application monitoring (identifying pay-as-you-use and licensing details), and for "filtering" receipt requests before they are added in the log files.

Finally, we explained the advantages of a standardised log file format. The log file format described in this chapter successfully meets the requirements of today's multi application smart cards, since it effectively utilises the relatively small amount of smart card memory.

Chapter 6

The Log File Download Manager

Contents

In this chapter we present various computationally practical methods for downloading the log files from a smart card to another device which does not suffer from immediate storage restrictions.

6.1 Introduction

The advanced processing power and the relatively unlimited storage capacities of today's computer systems simplify the use of log file handling mechanisms.

In a smart card environment, specific application prerequisites, different operating environments, or even slight variations of the trust relationships among the participants, often impose different log file uses. As we explained in the previous chapter, among the most important restrictions for an effective implementation of a log file handling mechanism is the limited storage space of the smart card.

Whatever space is allocated for the storage of a smart card log file will not be enough, as sooner or later it will fill up. Our goal here is to push the state-of-the-art a little further and deal with the complex and realistic issue of securely extracting the log files from the card and storing them in some other location.

In this chapter we discuss protocols and policies that need to be considered when the space available for log files in smart cards is filling up and the log files are about to be downloaded to some other device. We start by highlighting the general requirements for a secure log file transfer model. We then describe our approach to secure smart card log file transfer by presenting three different log file download protocols. Finally, we describe common implementation details and discuss several practical issues that highlight certain design restrictions.

6.2 Operational Requirements of the Log File Download Model

In this section we present some general requirements and observations for the Log File Download Model.

We assume that the Log File Download Manager (LFDM) has access to the cryptographic functions residing in the card, and is capable of assigning unique sequential numbers to the log files about to be downloaded. Secondly, we require that the log files will be linked with each other in order that it will become difficult to add, delete or modify any entries without such changes being evident. When the log files are linked together substantial auditing evidence is automatically created. In order to achieve appropriate log file chaining when the log files are downloaded we require the use of cryptography.

The main use of cryptography here is that the LFDM appends a cryptographic check value (MAC) on the downloaded log files. In our proposals we require that once a key is stored in the card it will never appear outside the card. For example, if the key is shared between an ALSS and the card it will only be known between these two entities. This key could be generated by a trusted entity and subsequently, stored in the card during the personalisation phase, along with publishing the key identification number or a corresponding public key certificate in a trusted public key directory.

In the log file download process we have to ensure that a log file stored in the card is securely transmitted and stored in an ALSS with the minimal number of messages exchanged. The steps involved in order to establish and accomplish communication with an ALSS is another responsibility of the LFDM. For example, the next time the card is used and the default storage space for the log file has reached a preset threshold, the LFDM proceeds with one of the download mechanisms described in §6.3. Upon successful completion of the download process the LFDM will permanently flush the space occupied by the log file only after it receives an acknowledgement by the ALSS indicating that the log file is successfully received.

While the log files are stored in the ALSS we assume that they are adequately protected against deletion. In one of our proposed solutions we favour the idea of assigning a more active role for the ALSS than just simply accepting any log files. In these cases

we require the ALSS to check if the log files have been tampered with. This feature will simplify the arbitration procedure, speed it up, and make it more efficient.

Finally, in all the protocols described below, we require the parties involved to have access to accurate directories that provide specific details on digital user IDs and key certificate IDs. Similarly, some of the protocols require the entities (C, A) to establish a secure connection [60] in order to exchange various messages.

6.3 Secure Log File Transfer

In this section we present our protocols for downloading log files into some external storage entity (i.e. the ALSS). We assume throughout that the ALSS has unlimited storage capabilities, a reasonable assumption given the current low price of high-speed disk storage. We then describe three different sets of protocols for downloading log files to an ALSS, appropriate for three different scenarios.

- The first scenario (§6.3.2) applies to the case where the ALSS is implemented on a cardholder-controlled device, e.g. their own PC.
- The second scenario (§6.3.3) covers the use of a remote ALSS, where confidentiality of the log file contents is not an issue.
- The third scenario (§6.3.4) again applies to a remote ALSS, this time providing confidentiality protection for the log file, i.e. the ALSS should not be able to read the log file contents.

We summarise our notation in Table 1. This notation is an extended version of the one defined in [46]. Descriptions of cryptographic algorithms appropriate for use in the protocols defined below can be found, for example, in [28, 57].

Table 6.1: Notation and terminology.

Notation	Description
$e_K(Z)$	The result of encipherment of data Z with a symmetric encipherment algorithm (e.g. DES) using key K.
$h(Z)$	The one way hash of data Z using an algorithm such as SHA-1.
$f_K(Z)$	The result of applying a MAC function f using as input a secret key K, and an arbitrary data string Z.
$z_{V_X}(Z)$	The result of encipherment of data string Z using a public key algorithm (e.g. RSA) with key V_X.
$s_{S_X}(Z)$	The signature resulting from applying the private signature transformation on data Z using the private signature key S_X. We assume that the signature scheme in use is a signature scheme with appendix [28], i.e. the data Z cannot be recovered from the signature.
K_{XY}	A secret key shared between entities X and Y; used in symmetric cryptographic techniques only.
I_K	A unique key identification information for key K.
T_X	A timestamp issued by entity X.
N_X	A sequence number issued by entity X.
S_X	A private signature key associated with entity X.
P_X	A public verification key associated with entity X.
B_X	A private decryption key associated with entity X.
V_X	A public encryption key associated with entity X.
E_X	The unique identification information for entity X.
$X \longrightarrow Y : W$	Entity X sends entity Y a message with contents W.
$Y \parallel Z$	Represents the concatenation of the data items Y, Z.
S	The ALSS.

One example of how the above signature scheme might be realised is as follows. The data Z is input to a hash-function, and the resulting hash-code is then formatted and padded prior to application of the RSA function using the signer's private RSA key.

6.3.1 Security Requirements for the Log File Download Manager

In the following paragraphs we present a set of general requirements for the behaviour of the LFDM.

1. No Log Files are Lost During Transmission. This implies that both the ALSS and the LFDM will have to make sure that they receive the appropriate acknowledgements before accepting or flushing any log files respectively.

2. Privacy and Integrity. The system must be secure in the sense that it should be very difficult for the participants (U, SP, A, ALSS) involved to deny the existence and origin of the log files. The privacy and integrity of the log files should be addressed by using encryption and Message Authentication Codes (MACs) or digital signatures.

3. Performance. Communication between the entities should take place with a minimal exchange of messages. Moreover the format of the messages exchanged between the participants should minimise the use of cryptography, given the relative limited computational capabilities of smart cards.

4. Auditable. The log file chain should be auditable. It will also be necessary to be able to identify changes and modifications in the log entries after the log files have been transmitted.

6.3.2 Using an ALSS on a Cardholder-Controlled Device

In our first scenario the log files are downloaded to an ALSS belonging to the cardholder (e.g. a PC or PD). This might be appropriate in situations where the cardholder does not trust a third party ALSS, or when the log files are simply kept for the user's reference.

The assumptions about the operational environment of this protocol are the following: It is not likely that there will be "external" attackers between the ALSS and the card. On the other hand, the cardholder may decide to attack the system, since it might have some interest in deleting certain information from the log files. Errors in the communication link between ALSS and the card are not likely to appear since they are almost directly connected. In other words, we simply have to protect the integrity of the log files after they are transmitted to the ALSS.

Additionally, we assume that the card shares a key (K_{AC}) with a trusted entity e.g. an arbitrator. This key can be stored in the card during the personalisation phase or securely transmitted to the card at any later stage of the card's lifecycle. In either case, once the key is securely stored in the card it will never leave the card. This will ensure that only a card, possessing such a key, can be involved in the log file download protocol.

We also assume that the card is maintaining a sequence number N_C which cannot be modified by any external events. The initial value of the message sequence number variable will be $N_C = 0$. The initial value of $CBind_{N_C}$ variable could be a standard pre-defined string of fixed length. Prior to downloading any log files, we require the cardholder to be authenticated by the card in order to ensure that the log file download procedure is authorised by the legitimate cardholder. An existing unilateral authentication scheme, e.g. verifying the cardholder's PIN, can be used for this purpose, the details of which are not within the scope of this thesis.

The protocol starts with the LFDM sending message (1) to the ALSS.

$$(1)\ LFDM \longrightarrow ALSS : M_{N_C} \parallel fK_{AC}(M_{N_C})$$

where

$$M_{N_C} = (F \parallel N_C \parallel I_{AC} \parallel h(h(M_{N_C-1}) \parallel h(F))$$

F is the log file data, N_C is a sequence number (current message sequence number stored internally in the card and incremented by one), I_{AC} is the key identifier for key K_{AC} (e.g. this identifier can be used as a reference point by the ALSS or the arbitrators in order to identify the key used in the MAC operation, or to obtain the expiry date of the key or any other key specific information).

The last part of M_{N_C} creates the hash chain. When adding linking information in the form of a hash chain [2, 63] of all the previously generated messages with the current

message, we create extra auditing evidence [1, 58]. Thus, this part of the message links the previously sent messages (M_{N_C-1}) with a hash of the current log file. At this stage the LFDM stores, in protected memory of the card, a hash of the current message sent (M_{N_C}) and a hash of the current log file (F). Note that the LFDM has not yet flushed the space occupied by the log file.

Depending on the system design, the ALSS could perform various checks on the validity of the previous message (1) received. For reliability purposes in this protocol we simply require the ALSS to acknowledge every message received. On receiving message (1) the ALSS extracts the log file (F) from message (M_{N_C}). The ALSS reply to the card consists of the following message (2).

$$(2)\ ALSS \longrightarrow LFDM : h(F) \parallel \text{“Log File Received”}$$

where, $h(F)$ is a hash of the log file received, along with a data string indicating that the message is successfully received.

On receiving the response message, the LFDM checks for the correct hash value of the log file. The LFDM has already computed and stored a copy of $h(F)$, in the construction of M_{N_C}, in order to speed up the verification of the ALSS's reply. Alternatively, in the construction of message (1) the LFDM should simply calculate $h(h(M_{N_C-1}) \parallel F)$ instead of $h(h(M_{N_C-1}) \parallel h(F)$. Subsequently, the LFDM should send the message and while waiting for the ALSS's reply it should compute $h(F)$. In either case, if a correct hash value of the log file is present, the LFDM flushes the memory space utilised by the log file and replaces the previous value of $h(M_{N_C-1})$ with the current value of $h(M_{N_C})$.

Given users have interest in the information stored in the log files, it is reasonable to expect them to ensure their adequate protection after they are downloaded and stored in the ALSS. On the other hand, it should be clear that this solution does not prevent the users from downloading the log files and subsequently deleting them, i.e. “Watergate

Attack" [58].

What can go wrong?

If the LFDM receives an invalid reply, it will request the ALSS to resend message (2) once more. If problems persist the LFDM will assume that there is a communication problem and will refrain from sending further messages. If that is the case, the LFDM simply ignores the previous operations and returns to the state before starting the log file download protocol. Another problem which might arise is the possibility of a malicious user (M) attacking the ALSS and somehow gaining access to the log files. Obviously, the attacker will gain access to the content of the log files, but since they are protected with a MAC, generated with a key known only internally in the card, any alterations will be detected when arbitration will take place.

6.3.3 Using an ALSS in a Remote Location

In our second scenario the LFDM securely transmit the log files to a physically secure ALSS located remotely in the network. The major difference from the previous proposal is that the ALSS is not within the immediate control of the user. Evidently, this approach prevents the "Watergate Attack". It also removes the user's concern on where exactly the log files or the log file backup copies will be stored.

The assumptions about the operational environment of the protocol are the following. First of all we assume that confidentiality is an issue and that communication between ALSS and C should be protected. This will prevent an attacker monitoring the network traffic from getting access to the content of the log files. Therefore, we require that all the participants have access to a number of cryptographic algorithms. We also assume that ALSS has the ability, processing and communication power, to deal with a large number of card requests for service at the same time. On the other hand, each card will

be dealing with a single ALSS. Furthermore, we assume that ALSSs are trusted entities in a sense that they will do their best not to lose or disclose any properly received log files.

A further major difference from the previous proposal is that the ALSS is assigned a more active role. The ALSS has to verify the integrity of the transmitted log files prior to their acceptance. The verification will enable ALSS to reject any non legitimate log files, and at the same time help the arbitration procedure since in case of a dispute the arbitrators will be presented with already "filtered" information.

In this scenario we favour the use of public key cryptography, which has advantages and disadvantages. The main advantage is that it simplifies the verification procedure, as will be shown later. Furthermore, it eliminates the need to keep secure files of shared keys. On the other hand, public key cryptography is computationally expensive and might involve validating chains of key certificates. In this scenario the cards have to obtain a pair of cryptographic keys from a trusted entity. The card's private key will be available only internally in the card and the corresponding public key will become available to the ALSS. Communication with an ALSS involves two phases.

- The Card and ALSS Registration phase, where the entities are introduced to each other.
- The Secure Log File Transmission phase, where the log files are transmitted to the ALSS.

Card and ALSS Registration

The card and ALSS registration phase takes place once, in order to establish a fixed relationship between a card and an ALSS. This phase could be omitted from our proposals, if all the necessary information (ALSS related public keys and certificates) is

written into the card's memory during the personalisation phase. The real benefit of this phase is that it offers the users the freedom to select their favoured preferred ALSS or re-establish communication with an ALSS in case of key and certificate revocation problems [26]. As previously mentioned, confidentiality is an issue and therefore the communication between the ALSS and the card is encrypted.

We assume that the card verifies the ALSS's certificate. This process will verify that the ALSS's public encryption key (V_S) belong to the specific ALSS. The protocol starts with the LFDM sending the following message (1) to the ALSS:

$$(1)\ LFDM \longrightarrow ALSS : z_{V_S}(M_{N_C}) \parallel s_{S_C}(M_{N_C})$$

where

$$M_{N_C} = (\text{“Directory Initialisation”} \parallel N_C \parallel CBind_{N_C})$$

The “Directory Initialisation” field is a text entry used to indicate the card's intention for future communication, N_C is the incremented internally stored sequence number and

$$CBind_{N_C} = (E_S \parallel E_C \parallel I_C)$$

where E_S is the ALSS's unique identification information, E_C is the card's identification information, and I_C is the unique identification information of the card's public verification key. At this stage the LFDM stores $h(CBind_{N_C})$ internally.

On receiving the previous message the ALSS decrypts its first part. Subsequently it checks the following: whether it has already received a similar message before (i.e. a valid sequence number), and finally whether the message was intended for it. The latter is achieved by checking the presence of E_S in $CBind_{N_C}$. The next step requires the ALSS to obtain (by using the I_C value as an index), a copy of the card's public verification key. This key will be used in order to verify the signature in the second part of the message. Upon successful signature verification the ALSS sends the following message to the LFDM.

$$(2)\ ALSS \longrightarrow LFDM : z_{V_C}(L_{N_S}) \parallel s_{S_S}(L_{N_S})$$

where

$$L_{N_S} = (\text{“Initialisation Message Received”} \parallel N_S \parallel SBind_{N_S})$$

The “Initialisation Message Received” is a text entry to denote the successful receipt of the previous sent message, N_S is the server incremented sequence number which is unique for each card, and

$$SBind_{N_S} = h(E_C \parallel h(M_{N_C}))$$

The linking variable ($SBind_{N_S}$) contains a hash of the card's identity (E_C), along with the previous received message $h(M_{N_C})$.

On receiving the response message, the LFDM decrypts its first part by using its private decryption key B_C. Subsequently it checks for the unique identification string E_C and finally for a correct hash of the linking variable M_{N_C}. Using the ALSS's public signature key it verifies the digital signature in the second part of the message. Successful signature verification will verify that message (2) was sent by the ALSS. Finally, the LFDM copies the linking variable $SBind_{N_S}$ to the $CBind_{N_C}$ variable (i.e. $CBind_{N_C} = SBind_{N_S}$). Upon the successful completion of the registration protocol both entities have successfully established the required relationship and the required linking information has been created.

What can go wrong?

If the ALSS receives an invalid first message it will send back to the card a message with a data entry “Error Message” and a short description of the problem. When the LFDM receives such a message, it resends the original message.

If the LFDM does not receive a reply from the ALSS it assumes that either the message (1) never reached the ALSS, or message (2) never reached the LFDM. In either case it will resend the message once more. If the problems persist in the next message exchanged both the LFDM and the ALSS will avoid sending any other messages and the communication will terminate.

Secure Log File Transmission

The second phase involves the actual transmission of the log files to the ALSS. From the previous phase we assume that the ALSS has created a log file storage directory for the corresponding card. Similarly we assume that the ALSS possesses a copy of the card's information. The information exchanged between ALSS and C will be encrypted in order to achieve confidentiality. As previously described, we favour the use of public key cryptography.

The protocol starts with the LFDM sending the following message to the ALSS.

$$(1)\ LFDM \longrightarrow ALSS : z_{V_S}(M_{N_C}) \parallel s_{S_C}(M_{N_C})$$

where

$$M_{N_C} = (F \parallel N_C \parallel E_S \parallel E_C \parallel CBind_{N_C})$$

and F is the log file data, N_C is the current message sequence number, stored internally in the card, E_S and E_C are unique identification information of ALSS and C, and

$$CBind_{N_C} = h(h(CBind_{N_C-1}) \parallel N_C \parallel F).$$

Once more, the $CBind_{N_C}$ variable serves the role of the linking information. Thus, it contains a hash of the following: a hash of the previous sent message $h(Cbind_{N_C-1})$ stored internally, a message sequence number N_C and the current log file F. At this

stage the LFDM stores internally a copy of $h(CBind_{N_C})$. Alternatively, the LFDM can send the previous message and while waiting for the ALSS's reply it could compute $h(CBind_{N_C})$. This will reduce the time spent on the first part of the protocol.

On receiving message (1) the ALSS decrypts the first part of the message by using its private decryption key B_S. Subsequently, it verifies the details of the message M_{N_C} by checking the sequence number N_C along with the ALSS's identification information E_S and the card's identification information (E_C). If these details are present and valid, the ALSS assembles the message data M_{N_C} in order to verify, by using a copy of the card's public verification key P_C, the digital signature on the second half of the message. Upon successful signature verification the ALSS assumes that the message originated from a card which knows the secret signing key S_C and subsequently it sends the following message.

$$(2)\ ALSS \longrightarrow LFDM : z_{V_C}(L_{N_S}) \parallel s_{S_S}(L_{N_S})$$

where

$$L_{N_S} = (\text{“Log File Received”} \parallel N_S \parallel E_C \parallel SBind_{N_S})$$

As previously the field “Log File Received” is a text entry to denote the successful receipt of the log file, N_S is the ALSS's incremented sequence number for this card, E_C the identification information of the card, and

$$SBind_{N_S} = h(CBind_{N_C} \parallel N_S).$$

The $SBind_{N_S}$ variable serves the role of the linking information maintained by the ALSS. Thus, it links the previous information sent to the ALSS, with the current sequence number.

On receiving message (2) the LFDM decrypts (by using its private decryption key B_C) the first part of the message. Subsequently it makes sure that its name is correctly

present. It then assembles the linking variable L_{N_S} in order to verify the signature in the second half of the message. If everything appears to be correct, the LFDM is confident that message (2) is a reply to message (1). In that case it flushes the space occupied by the current log file, and overwrites the value of the old linking variable $CBind_{N_C}$ with $SBind_{N_S}$. In case of any problems in the messages exchanged, both entities proceed as described in the "What can wrong?" subsection of §6.3.3.

6.3.4 Using an ALSS in a Remote Location and Encrypted Log Files

In our previous example we addressed the problem of securely transmitting the log files to an ALSS. In that particular example we were not concerned about the confidentiality of the log files while stored in the ALSS. This means that while the log files are stored in the ALSS, their content is accessible to the ALSS.

In this section we slightly modify our requirements and propose a solution in which ALSSs do not have access to the content of the log files. The proposed solution is an extended combination of the previous two. Similarly to the previous example we assume that the information exchanged between ALSS and C is vulnerable and thus, it should be encrypted. The initialisation phase remains exactly the same as presented in §6.3.3. The protocol starts with the LFDM sending the following message.

$$(1)\ LFDM \longrightarrow ALSS : z_{V_S}(M_{N_C}) \parallel s_{S_C}(M_{N_C})$$

where

$$M_{N_C} = (N_C \parallel E_C \parallel E_S \parallel eK_{AC}(F) \parallel CBind_{N_C}).$$

N_C is the current message sequence number stored internally in C, E_C and E_S are the identity information of the card and ALSS respectively, $eK_{AC}(F)$ is the log file encrypted under the key shared between the card and a trusted entity (e.g. an arbitrator), and

$$CBind_{N_C} = h(h(CBind_{N_C-1}) \parallel eK_{AC}(F) \parallel N_C).$$

$CBind_{N_C}$ contains a hash of the card's previous linking variable $h(CBind_{N_C-1})$, a copy of the encrypted log file $eK_{AC}(F)$, along with the current message sequence number N_C. Note that the log file F is encrypted before it is actually included in the linking variable. At this stage the card stores internally a hashed copy of the encrypted log file $h(CBind_{N_C})$ or alternatively it sends message (1) and while waiting for the ALSS's reply it computes $h(CBind_{N_C})$.

On receiving the message, the ALSS decrypts the first half by using its private decryption key B_S. Subsequently, it looks for an appropriate sequence number N_C, along with the correct entity identifiers (i.e. E_C and E_S), and a valid hash value on the $CBind_{N_C}$ component. Then by using a copy of the public signature key of the card P_C it verifies the digital signature in the second part of the message. If everything is correct, i.e. the message decrypted correctly and there is valid digital signature in the second half of the message, the ALSS sends the following message.

$$(2)\ ALSS \longrightarrow LFDM : z_{V_C}(L_{N_S}) \parallel s_{S_S}(L_{N_S}).$$

where

$$L_{N_S} = (\text{“Log File Received”} \parallel N_S \parallel E_C \parallel SBind_{N_S}).$$

As previously, the “Log File Received” is a text entry to denote the successful receipt of the log file, N_S is the ALSS's incremented sequence number for this card, E_C is the identification information of the card (E_C), and

$$SBind_{N_S} = h(CBind_{N_C} \parallel N_S).$$

Once more the $SBind_{N_S}$ variable serves the role of the linking information maintained by the ALSS. Thus, it links the previous information sent by the ALSS with the current sequence number.

On receiving the message, the LFDM uses its private decryption key B_C and decrypts the first half. It then checks for a valid sequence number, a correct unique card identifier and for a valid hash of the linking variable. This will ensure that the ALSS has correctly received the transmitted log file. Subsequently, by using the ALSS'a public verification key P_S it verifies the signature in the second part of the message. If everything appears to be correct the LFDM flushes the space occupied by the log file, and overwrites the value of the previous linking variable $CBind_{N_C}$ with $SBind_{N_S}$. Similarly to the previous protocol, in case of any problems in the messages exchanged both entities proceed as described in the "What can wrong?" subsection of §6.3.3.

6.4 Common Implementation Details

In this section we present certain issues which could be common between the three different scenarios.

6.4.1 The Behaviour of ALSS

We assume that in a real world implementation of the log file download proposals the ALSSs will communicate with a large number of cards. Due to the possible large number of transactions involved we would like to ensure that ALSSs will be able to respond back to the cards within a reasonable time. On top of the ALSS processing overhead, we must also take into account the communications overhead, for example the time spent for the messages to travel from one entity to the other. The latter will depend on the traffic over the network, e.g. the Internet or the air link when using mobile phones.

6.4.2 Using Secret Key Cryptography

In case that public key cryptography is considered as inappropriate for encrypting the first half of each message, secret key cryptography could be used instead. In such a

case the two entities, the ALSS and the card, should share a secret key. This key could be generated in advance (i.e. during the personalisation phase of the card) or it could be securely distributed to the ALSS at a later stage. If the key is installed during the personalisation phase, then the registration phase could be considered as minimal, since the ALSS can create a log file storage directory as soon as it receives a copy of the shared key. Alternatively, the two entities can be involved in a key agreement protocol in a slightly modified registration phase. In either case, this shared key should be used in order to encrypt the first part of message (1).

Another issue is how the ALSS will identify the secret key used by each card and subsequently decrypt the first part of the message. In order the ALSS to uniquely identify each card it has to obtain the card's ATR. Subsequently, by using the card's unique serial number, it will search its database in order to obtain the card's secret key. The construction of the messages along with the digital signature at the second half of each message should remain the same.

6.4.3 Termination of the ALSS and Smart card Relationship

During the period of communication between the ALSS and the card, either of the two entities might decide to terminate their relationship.

We believe that in a real world implementation of the aforementioned protocols, the cards will not be permitted to switch over to a different ALSS. This could be a policy decision enforced by the card providers in order to satisfy their commercial interests, e.g. established relationships with certain ALSSs. On the other hand, offering the ability to terminate an existing ALSS and card relationship increases the confidence of both the cardholder and the ALSS. Although this is an implementation decision which depends on the systems security policy, for clarity purposes we briefly describe the main operations involved.

If the ALSS decides to terminate its relationship with a specific card, we propose the following procedure. The next time the ALSS is contacted by the specific card, it will decide whether to accept or reject the received log file depending on the security policy. Whether or not the log file is accepted, the ALSS forms a reply as described in the protocols but with a single difference. The text entry of the reply message will be like "Terminate Relationship". On receiving the message, the card verifies both the digital signature and the linking information, and subsequently switches into the registration mode described in §6.3.3 along with terminating any communication with the ALSS.

The card can also issue a similar message using the following procedure. The LFDM generates message M_{N_C} as described in the first step of the log file transmission phase. Subsequently, it includes an additional text entry "Terminate Relationship" and, depending on whether is permitted to proceed with submitting the log file, it proceeds accordingly. On receiving such a message the ALSS verifies its details and acknowledges with a reply similar to the one described above. As in the above example the card will switch into the registration phase mode and terminate any communication with the ALSS.

6.4.4 Dispute Resolution and the Arbitration Phase

In case a dispute arises and an arbitrator is involved, we have to consider two cases, depending on whether the log files are stored in a user controlled ALSS or in an ALSS somewhere on the network.

When the log files are stored in a user controlled ALSS the arbitrator will have to contact the user involved by using conventional methods (e.g. telephone or email), in order to obtain a copy of the log files. The user will have to establish a secure connection with the arbitrator and transmit his/her log files. The arbitrator, by using his/her copy of the shared key K_{AC}, will verify the linking information of the log files.

Let us suppose that the ALSS transmits the following sequence of log files presented in figure 6.1. In this example the valid sequence of log files from 45 to 47 is interrupted with two non legitimate log files with sequence numbers *XX*. When the arbitrator verifies the linking information, the two "fake" entries will be identified, firstly because they will not be encrypted with the valid key, and secondly, because their corresponding linking information will not make sense.

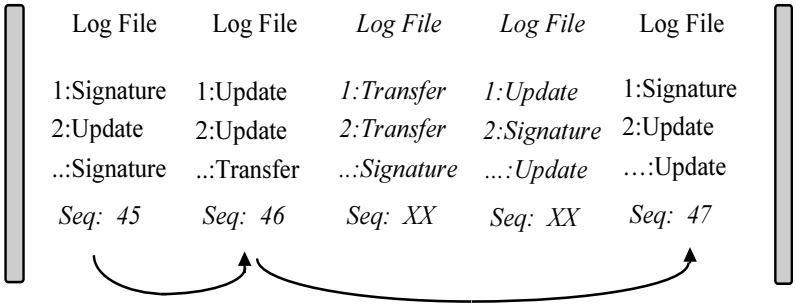

Figure 6.1: Sequence of log files stored in the ALSS.

When the log files are stored in an ALSS located somewhere in the network, the ALSS will be asked to submit a copy of the log files (i.e. the actual messages exchanged between ALSS and C) to the arbitrator or to the dispute resolution entity. Subsequently, the arbitrator will verify the authenticity of the log files by using the public encryption key P_C of the card.

6.4.5 Verification of the Log Files While Stored in the Smart card

If a dispute arises while the log files are still stored in the card, a simple solution requires the LFDM to offer the functionality of downloading the log files on the cardholders request, even when the log file space is not full.

In such a case the LFDM generates a text entry "Requested Log File Download" which is included in the current message (M_{N_C}) variable and in the linking variable ($CBind_{N_C}$). Subsequently, it follows one of the protocols described in the previous sections in order

to securely download the log files to an ALSS.

This functionality enables the card holder to empty the log file space before it actually becomes full, and therefore, participate more effectively in the log file space management process.

6.5 Summary

In this chapter we have presented several variations of a scheme that allows the log files to be securely transmitted from a card to another device which does not suffer from immediate storage restrictions. Our proposals successfully address the problem of securely transmitting the log files, by using only the absolutely necessary exchange of messages between the participants along with only the necessary computationally expensive cryptographic protection.

The first protocol simply extracts the log files from the card and transfers them to an ALSS, which is under the control of the card holder. The second and third protocols have a common registration phase in order to enable the two entities to be introduced to each other. Their main difference with the first protocol is that the ALSS is located somewhere in the network and is not under the control of the card holder. The main difference between the second and the third protocols is that in the latter the ALSS should not have access to the content of the log files. Their common issue is that, since the ALSS is located in the network, the log files need to be protected while they are in transient.

Chapter 7

Implementation Results

Contents

While there are a number of theoretical security protocols integrating smart cards in a variety of systems, there are very few test implementations. In this chapter we present details of an experimental implementation of both the secure log file download protocol and the proposed log file standard format. We also provide a discussion of the performance results along with our experiences from implementing the theoretical designs in a real multi-application smart card environment.

7.1 Introduction

This chapter describes results from test implementations [23, 27] of the ideas presented in chapters 5 and 6. The work in this chapter can also be considered as a reference point when designing more complex smart card applications.

For the test implementation we used two of the most popular Java Card API 2.0 [29, 30, 31] compliant smart card currently available in the market, namely the GemXpresso Java Card [12] from Gemplus and Cyberflex Open 16K Java Card [56] from Schlumberger. Notably, both these implementations allow dynamic application download/deletion and provide application isolation. Although Java Cards offer multi-application capabilities, there are still certain limitations, such as the limited size of EEPROM and RAM, the limited processing power, and the lack of cryptographic functions in general, that need to be taken into account when developing smart card applications. These factors forced certain design decisions which will be explained in the following subsections.

This chapter serves two purposes. Firstly, it provides performance measurements for the first log file download protocol specified in §6.3.3, and the audit log standard format given in §5.5, and thus proves that a real implementation of the concepts is feasible. Secondly, it highlights issues which are relevant when developing Java Card applications. This last point is of particular importance since there are very few Java Card applications available today and we expect a substantial increase in the future.

7.2 Common Design Details

Software solutions that use smart cards can be separated into the client-side application and the smart card application or applet. Thus, in order to test the performance of the log file download protocol or the standard log file format, two distinct entities have to be developed. The first one will reside in the card (e.g. the LFDM) and the

second will reside in the user's PC (e.g. the ALSS). The technology for implementing these two entities is already described in chapter 2. In the following subsections we present various limitations and architectural characteristics, in order to give the reader an understanding of the issues involved.

7.2.1 The Client Application

The interfaces that enable the client to interact with the Java Card are presented in section §2.4. In our implementation we used the PC/SC and the DMI interfaces.

7.2.2 Smart card Application Development Tools

Two widely known development tools that allow pre-processing, downloading and execution of Java Card applets are presented below. Both tools enable developers to write and test Java Card applets, and they are Java Card API 2.0 compliant.

- The Gemplus GemXpresso Rapid Applet Development (RAD) Kit [12], contains the first Java Card implementation on a 32-bit RISC smart card micro-processor. It is Java Card API 2.0 compatible but it also differentiates itself with a cut-down version of an RPC (Remote Procedure Call) [6] style protocol called DMI (Direct Method Invocation) [65]. Finally, it has 15 Kbytes of memory, of which 5 Kbytes are used for heap memory.

- The Cyberflex Open 16K [7], can accept applets with a total size of 16 Kbytes including the heap memory, and it has a stack size of 128 bytes. The heart of the smart card is an 8-bit micro-processor.

7.2.3 Limitations of the Smart cards and the Development Kits

During the development of the test applications we encountered the following limitations that had to do either with the smart card capabilities or the development kits that accompanied them.

Of the two Java Cards, only the GemXpresso supports garbage collection. However, garbage collection, by nature, does not take place immediately after the memory ceases to be used, but after some implementation specific delay. Furthermore, in many JVM implementations, garbage collection takes place when memory gets exhausted and such a procedure would adversely affect the potential speed of the given Java Card. Due to these issues, special attention had to be given when coding not to use local variables and not dynamically allocate memory in frequently called functions. In extreme cases, "frequently" is defined as twice or more. When a local variable is instantiated, the memory of the stack is used. When a dynamic allocation is requested, the memory of the heap is used. In both cases, after the exit of the function, no memory is claimed back, and we gradually become short of memory. The solution is to use global variables, and reuse them as much as possible within the applet. It is desirable to restrict memory allocation to inside the constructor of the applet, because this is a guaranteed location that is executed only once, and it is a location where the memory has not become fragmented. We must note that garbage collection is not a prerequisite for Java Card 2.0 API conformance.

Usually, when sending data (Application Protocol Data Units [42]) from the smart card to the client and vice versa, a limit of approximately 255 bytes exists. Actually, for the GemXpresso card the maximum packet of data that can be send from the client to the card is limited to 56 bytes. In order to solve the problem of sending larger data packets, special programming has to be used to send the data in blocks. This slows the applet execution significantly when communicating with the client, because switching from receiving to sending, and vice versa, is a slow procedure.

The (lack of) availability of cryptographic functions on the Java Cards is a two-fold problem. First, the export control restrictions imposed by many governments dissuade manufacturers from implementing them. Secondly, the JVM already takes up much of the resources of the Java Card, and manufacturers need to invest heavily to put the cryptographic core with the rest of the functionality. Both Java Cards used do not currently support real cryptographic functions.

The development environment usually offers a "simulator" that enables the programmer to easily test the applet without downloading it to the Java Card every time it is to be executed. In the case of the development kit of the Open 16K, the simulator was not reliable enough, and attempts to use it had to be abandoned. The GemXpresso Java Card simulator was significantly better and quite usable.

The time from the generation-compilation of the applet on the development environment until the applet is running on the Java Card was computed to determine the compile-to-run cycle. It was noticed that the resulting times were between 25-55 seconds. These times are very long and in some cases involve a series of repetitive steps. This often leads to errors such as failing to correctly update the applet on the Java Card and carrying out tests on the previous applet. This was the case with the development environment of the Open 16K card.

7.3 Implementing the Log File Download Protocol

In this section we describe the implementation details from the first secure log file download protocol as described in §6.3.3.

7.3.1 Design Goals for the Log File Download Protocol

The following are among the design principles followed while implementing the log file download protocol.

- Ensure that the log file download protocol will start only after the smart card holder's approval e.g. after the user presents the correct PIN to the card.
- Effectively identify restricted smart card resources in order to allow more adequate application design.
- Perform application code optimisation in order to achieve better application performance.
- Design dummy cryptographic primitives, since the available cards do not offer such functionality, and when offered it is not fully accessible, as explained later on.
- Evaluate the usability of the Java Card development tools.
- Examine the application execution performance for both platforms.

7.3.2 Common Implementation Details Between the two Java Card Platforms

Since cryptographic functionality was unavailable, dummy functions were implemented to cover the lack of hash and MAC functions. The input to the dummy MAC and hash functions was a buffer of arbitrary size and the output was a 16 byte string. The dummy hash was implemented using the modulus operation over the input and the dummy MAC was implemented using the dummy hash by adding an "exclusive or" with a key value.

Although log files of 1–1.5 Kbytes may be more desirable (as described in §5.5.5) we use smaller log files along with simple cryptographic functions for the reasons given below.

It is rather difficult to manage "large" files on the card, more than 248 bytes, and problems were encountered when reading such "large" files. It would also be time-consuming to perform cryptographic functions on "large" blocks of data. Additionally, a large log file space would further delay the protocol execution since a number of packets would need to be transmitted due to the 54 or 255 byte restriction on the APDU data buffer, depending on the smart card development platform.

Another issue that forced the common choice of log file size as 40 bytes, and such simple dummy cryptographic functions, was the poor performance of the card reader for the Cyberflex Java Card. As described in §7.3.4, this card reader does not support long processing in the Java Card due to the de-synchronisation of client/card communications.

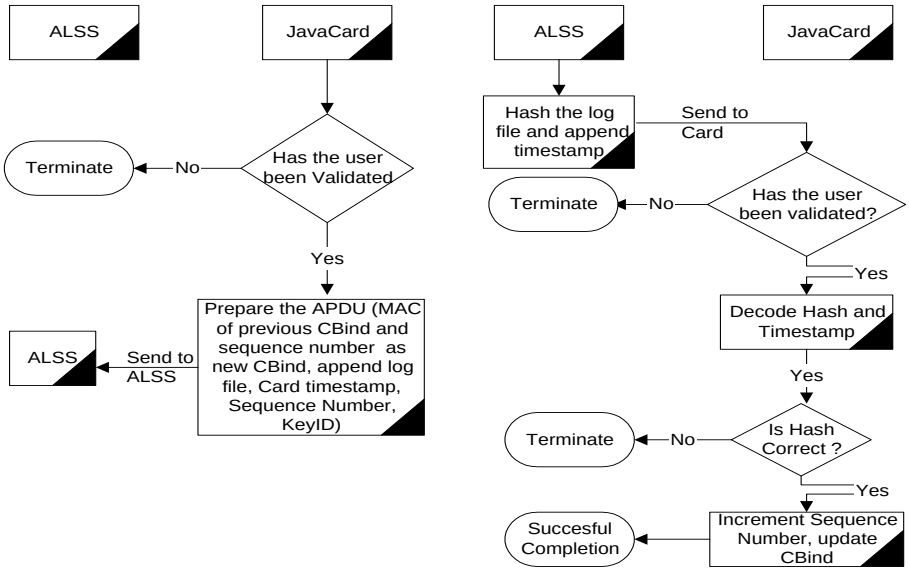

Figure 7.1: The three steps (Validation, SendData, VerifyReply) of the log file download protocol

The log file information could have been spread across more than one file to accommodate the limitations on the file system of the Java Cards. However, such a solution was

not adopted, as it would require changes to the log file download protocol and generally to the overall security design of the system. Therefore, it was decided that the log file should be relatively small, i.e. maximum of 248 bytes when running on the simulator and 40 bytes when running on the card. As previously mentioned, it was impossible to make the protocol work (on the card) when the log file was more than 40 bytes. The steps of the log file download protocol are presented in figure 7.1.

Note that for the protocol to be implemented, the following functionality needs to be available. The card has to implement a hash and a MAC function (§7.3.3 and §7.3.4). The card must also be capable of storing certain transient information (N_C, $CBind_{N_C}$) while waiting for the ALSS's reply. The ALSS has to implement the same cryptographic functions as the card.

The protocol functionality is encapsulated within three essential functions:

I. the `VerifyPIN`, accepts the PIN and compares it with a default one,

II. the `SendData`, implements the part of the protocol that sends, among other information, the log file data, and

III. the `VerifyReply`, implements the last step of the protocol that enables the LFDM to verify the ALSS's reply.

7.3.3 GemXpresso Implementation

In this section we present the implementation details when using the GemXpresso development kit.

GemXpresso Characteristics and Limitations

Note that, although the GemXpresso card contains certain dummy cryptographic functions (DES, 3DES, and hash), these were not used since the actual Java classes implementing the cryptographic functionality (e.g. DES) were not fully usable. Instead, the dummy functions specified in §7.3.2 were used in both implementations to facilitate the comparison between the two experimental platforms.

Implementation Architecture

In this section, we present a more detailed design for the GemXpresso Java Card application. The application is written according to the DMI specification in Java, refer to §2.4.5. In order to maintain compatibility with the Open 16K platform we did not use any 32-bit data types (specifically in the hash and MAC functions). We have shared the functionality of the log file download protocol between an applet and a library, as it can be seen in figure 7.2.

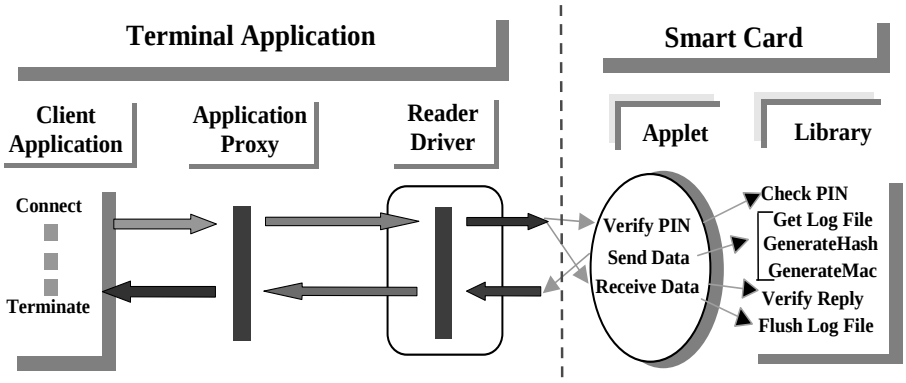

Figure 7.2: The architecture of the LFDM residing on the GemXpresso card and the Client application residing on the user's PC

The Java Card applet contains all the functions accessible to the outside world (i.e. the client). These functions provide an external interface for the functions, defined in the library, directly accessing the log file.

With this architecture, the sensitive log file information will only be accessed through the library classes (as the library is the sole owner of the log file). The library exposes, in the form of public procedures, only a limited number of functions and thus, a malicious client application can not directly call the procedures for accessing the log file. Since the library contains the basic log file access procedures (such as a log file API), these functions could be securely shared among other trusted Java Card applications.

To see the advantages of this architecture, suppose the log file download protocol is to be upgraded. Then, a new Java Card applet, implementing the new protocol and using the existing library, can be loaded into the smart card, without endangering the library functions. Also, a potential log file library owner, e.g. a Log File API, could easily provide controlled access from the library to other smart card applications.

Results and Performance Evaluation

Different results were generated depending on the actual size of the log files and whether the application ran on the card or on the simulator. The implementation and the testing of the client and LFDM applications used a 400Mhz PC with 128 MB of RAM under Windows NT with Service Pack (SP) 4. The Java code was written using Microsoft J++ and Microsoft Java SDK Ver 2.02. The GemXpresso card was communicating with the client application through the GCR410-X reader provided by Gemplus. When the client was communicating with the Java Card, the results were largely PC speed independent because the operations were mostly I/O intensive.

The results from the protocol execution on the simulator with a log file size of 248 bytes and after ten consecutive executions are presented in Table 7.11 and a more condensed version of this table is presented in Table 7.1. The `Connect` figure indicates the time spent by the client application connecting to the reader or the simulator. The next three values indicate the time spent in the following procedures: the `VerifyPIN` function verifies the user password, the `SendData` function forms the packets as defined

by the protocol and the `VerifyReply` verifies the ALSS's reply. The `Disconnect` figure represents the time spent closing the connection with the reader or the simulator.

Table 7.1: GemXpresso simulator results using a log file of 248 bytes.

Part name	*Time (ms)*	*Std.Deviation*
Connect	11246	34.62
Verify PIN	210	4.97
Send Data	200	0.42
Verify Reply	200	0.52
Disconnect	401	0.47
Total	12257	40.99

It is worth mentioning that the `Connect` figure, at least for the simulator, is not very accurate. This is because when the client is connecting to the simulator and tries to select the applet, in most cases it returns "false", i.e. the applet could not be selected. Surprisingly though, the applet execution continues as if the applet was properly selected.

When the protocol is executed in the simulator with a log file of 40 bytes we get the results presented in Table 7.12. A more condensed version is presented in Table 7.2.

Table 7.2: GemXpresso simulator results using a log file of 40 bytes.

Part name	*Time (ms)*	*Std.Deviation*
Connect	11246	24.18
Verify PIN	210	4.20
Send Data	140	0.32
Verify Reply	200	0.48
Disconnect	400.5	0.50
Total	12197	29.68

From the figures in table 7.1 and table 7.2 we observe that the `Connect` and `Disconnect` values are almost identical. Similarly, the `VerifyPIN` procedure consumes the same amount of time in both cases. When the log file is 40 bytes the `SendData` function takes less time since less data are involved in constructing the protocol packets. The `VerifyReply` procedure takes the same time in both implementations since the opera-

tions involved are independent of the log file length.

The main reason for providing execution results on the simulator, which does not successfully simulate the card's behaviour, is as follows. Firstly, it provides an indication of the differences in execution times. Secondly, it shows that the only reason that the log file download protocol does not operate on the card with relative large log files is the limited memory space.

Timing results when the protocol was executed in the card are provided in Table 7.13 and the condensed version is presented in Table 7.3. When the protocol is executed in the card, we tend to get slightly increased values, as was expected. This shows that the simulator does not correctly simulate the card processing time.

Table 7.3: GemXpresso Java Card results using log file of 40 bytes.

Part name	*Time (ms)*	*Std.Deviation*
Connect	2689	12,76
Verify PIN	371	6.13
Send Data	1091	5.27
Verify Reply	1862	9.81
Disconnect	130	4.59
Total	6143	15,57

The `Connect` time is very large, compared with some results for the Open 16K Java Card, as presented later. We could partially attribute this to the fact that, in this case, the client is implemented in Java, while in the second case it is implemented in Visual Basic. In other words, this implies that since Java is interpreted it could be slower when executed but mainly due to the different architectures when requesting access to the reader and the card. The above observation, along with the fact that the client application interfaces (i.e. PC/SC and DMI) are using different strategies or methodologies for connecting to the reader, contribute to the different results obtained.

To remove the influence of the timings for the "dummy" cryptographic functions, the execution times for the dummy hash and MAC over a 20 byte buffer were measured

(see Table 7.14), and the overall results are given in Table 7.4.

Table 7.4: Dummy hash and MAC execution times on the GemXpresso Card.

Java Card	*Hash (ms)*	*MAC (ms)*
GemXpresso	128	139

The results in Table 7.4 were used to estimate the time needed to execute the protocol on the GemXpresso Java Card, assuming that a hash takes x milliseconds and a MAC takes y milliseconds (see Table 7.5).

Table 7.5: GemXpresso Java Card with variable hash and MAC times.

Part name	*Time (ms)*
Connect	2689
Verify PIN	371
Send Data	$824 + x + y$
Verify Reply	$1584 + 2x$
Disconnect	130
Total	$5598 + 3x + y$

The size in bytes of each of the above three entities when implemented in Java and downloaded in the GemXpresso card are as follows: The smart card library (LFDM) is 1939 bytes and the smart card application calling the LFDM is 757 bytes. The client application is 8.342 bytes along with the proxy interface which is 3524 bytes. The proxy file is automatically generated by the GemXpresso development kit. The above file sizes are for the Java source code files.

7.3.4 Cyberflex Open 16K Implementation

In this section we present the implementation details when using the Cyberflex Open 16K Java Card.

Design and Limitations

The software that accompanies the Cyberflex Open 16K Java Card provides the developer with management utilities to download applets to the Java Card and offers a rather simple interface to test them.

As noted in §7.3.2, the same dummy cryptographic functions were used in both implementations. Because of constraints in the Java Cards, and to maintain consistency between implementations, the protocol was implemented using a 40 byte log file.

The Open 16K JVM only supports data types of one and two bytes because the Java Card has an 8-bit microprocessor. Also, an integer of 4 bytes as defined in GemXpresso, is not offered in Cyberflex. This means that all results of arithmetic operations should be "type casted" or converted to the one-byte or two-byte data types.

For the Open 16K Java Card, a relative "dumb" card reader (i.e. the Litronic 210 serial port reader) was available. Dumb card readers are relatively unsophisticated, and synchronisation problems often arise, causing frequent problems with the experimental implementation, e.g. requiring PC reboots, mainly during the application development phase.

The simulator supplied for the Open 16K Java Card did not operate correctly and a direct implementation to the Java Card had to be carried out. However, despite the difficulties, a stable implementation of the protocol was eventually produced.

Implementation Architecture

The functionality of the `VerifyPIN`, `SendData` and `VerifyReply` functions is described in §7.3.2 and depicted in figure 7.3. The PIN is a 4-digit number and, once the user has been validated, the rest of the functions will be invoked.

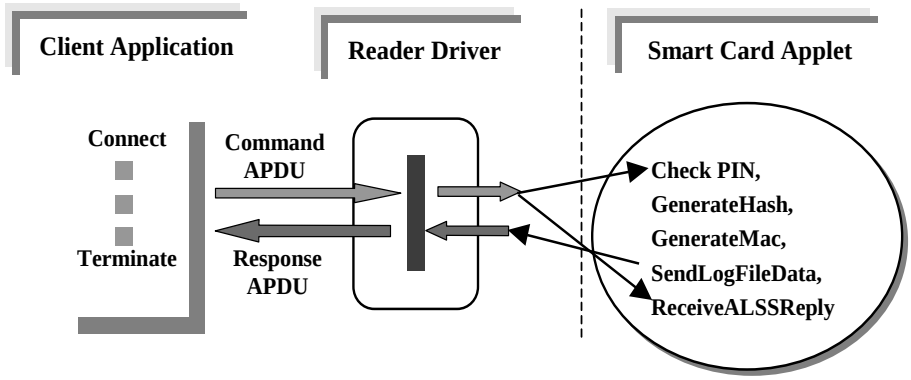

Figure 7.3: The architecture of the LFDM residing on the Cyberflex card the Client application residing on the user's PC

The client side of the application was written in Visual Basic Ver. 5 using a COM component provided with the Cyberflex Development kit that enables communication with the smart card application through the PC/SC framework.

Results and Performance Evaluation

The results of the 10 consecutive execution timings are presented in Table 7.15 and a more condensed version appears in Table 7.6.

Table 7.6: Open 16K Java Card using a log file of 40 bytes.

Part name	*Time (ms)*	*Std.Deviation*
Connect	200	4.90
Verify PIN	190.5	5.22
Send Data	2473	5.14
Verify Reply	2173	4.28
Disconnect	951	0.42
Total	5987.5	19.97

The `Connect` time also includes the initialisation time that is needed to access the COM component, to configure the connection with the card reader and select the applet. Once this is done, validation can take place. Afterwards, the client sends a request to the Java Card to receive the Log File. Finally, the client parses the response and answers to the card providing a hash that the latter has to verify.

The `Connect` phase includes initialisation procedures carried out on the client, such as the Java Card's selection of which applet to run. The relatively small time is explained by the fact that much of the processing is done on the host PC and as previously mentioned the specific implementation details of PC/SC.

The `VerifyPIN` process is very short, and involves comparing the given PIN with the correct one and informing the user whether it was correct.

The `SendData` function is the most time-consuming function. It makes use of array copies as described in §7.3.2 to create the message sent to the client. Also, it computes a dummy hash and a dummy MAC. The execution time of this function is higher than for the GemXpresso card. The only explanation would appear to be either that reading and updating files in the Cyberflex card is slower or that generally the Cyberflex Open 16K micro-processor is slower than the GemXpresso.

The `VerifyReply` function accepts the client's reply, and following the necessary checks it finally notifies the client whether the hash was properly verified.

The `Disconnect` function takes a relatively long time, compared with the `Connect` time. It can be assumed that this is PC/SC specific, or more precisely, it is related to the drivers that implement the support for the Open 16K. Note that the `Connect` time is very short compared with the GemXpresso, although exactly the opposite is true with respect to the `Disconnect` time. Again the explanation is that there are implementation differences on how to connect or disconnect with a card.

Table 7.7: Dummy hash and MAC execution times on the Open 16K Card.

Java Card	*Hash (ms)*	*MAC (ms)*
Open 16K	177.75	187.75

The time to execute the dummy hash and MAC over a 20 byte buffer were measured (see Table 7.16) and the results are given in Table 7.7. The MAC and hash performance measurements on the Cyberflex Open 16K card appear to be slower compared with the

ones obtained by the GemXpresso card, for reasons already analysed in the previous sections.

Using the figures of Table 7.7, we can estimate the protocol performance assuming that a hash takes x milliseconds and a MAC y milliseconds, see Table 7.8.

Table 7.8: Execution times on the Open 16K Card with variable hash and MAC times.

Part name	*Time (ms)*
Connect	200
Verify PIN	190.5
Send Data	$2107 + x + y$
Verify Reply	$1817.5 + 2x$
Disconnect	951
Total	$5266.5 + 3x + y$

The size in bytes of each of the above entities when implemented in Java and downloaded in the Cyberflex card are as follows: The smart card application calling the LFDM is 5867 bytes. The client application is 17 Kbytes. All figures indicate the compiled versions of the corresponding files and not of the source code files.

7.4 Implementing the Standard Log File Format

In this section we describe the implementation details from the log file standard format as described in §5.5.

7.4.1 Implementation Analysis

In a practical implementation of the log file standard format the data entries (DEs) would be generated either by the SCOS or by applications. As the standard requires that information from either source will have to be transformed into the specified format, both the SCOS and the applications will have to be provided with this functionality.

The required functionality will be provided with the `GenerateDataEntry` function that will perform the following tasks:

- take as input the LEs information,
- generate the format required by the standard (DE), and
- append the new entry to the log file.

Ideally this function should be implemented as a SCOS primitive-service for the following reasons:

- it will be more adequately protected if masked in ROM,
- it will run faster since it could be written in machine native code, and
- it will probably eliminate the possibility of having applications passing information outside their own application space.

The last observation is of particular importance since in both Java Card API and particularly in Multos, applications are restricted within their own application space. Thus, it would be more efficient to call a SCOS primitive that will securely receive the logging information and perform the above steps, rather than having applications passing information to other applications, outside their own application space. Additionally, it will be more secure for the log file, which holds information from multiple applications, to be accessed through a SCOS call rather directly by each smart card application.

Another approach, which is more convenient to implement with our currently available Java Card tools [12], suggests that the required functionality will be provided in the form of a shared library. The specific functions can be made publicly available through a shared library that will be written in Java Card code and will be accessible by the

smart card applications. Actually, this provides an indirect method of extending the functionality of the smart card operating system.

Certain issues that need to be taken into account when implementing the log file standard format are the following: the `(GenerateDataEntry)` should be called only once during the execution period of an application, and preferably after all changes to variables are committed to their final values. This will prevent a single application filling up the space of the log file by repeatedly calling the `GenerateDataEntry` command. A further protection mechanism would be to maintain a database (Application Logging Table, described in §5.3) of all the trusted applications and subsequently grant permission to use the command only to these specific applications.

However, certain information need to be stored in the card in order to be able to construct the messages in the format specified by the standard. For example, the `GenerateDataEntry` must be provided with the means to get the unique application identifier (AID) of the currently running application. Additionally, a unique sequence counter is required in order to serialise the data entries, as described in §5.5.2.

The standard requires that a hash of the previous data entry DE_{i-1} will always be available in order to link the previous generated messages with the current one. If the log file is overwritten in a cyclic mode, then the previously generated hash will be always available. If the log file is to be downloaded as described in §6.3.3 then a hashed copy of the last data entry should be securely stored in the card.

7.4.2 Further Implementation Details

For implementing the standardised log file format functionality we used the GemXpresso development kit from Gemplus. As in the previous example this implementation design consist of three different components: the shared library, the smart card application, and the client application.

The "shared" library contains the actual class, i.e. `GenerateDataEntry`, that implements the standard log file format functionality. This function accepts the LE entries i.e. $(LE_1; LE_1; \cdots; LE_e)$ and constructs the data entry DE (115 bytes) as required by the standard. Additionally, prior to generating a data entry it also verifies that the currently running application has not previously called the `GenerateDataEntry` function.

The log file used for the implementation of the standard log file format is relatively small consisting of 235 bytes when running on the card and 800 bytes when running on the simulator. As already mentioned, in the implementation of the LFDM we could not read files larger that 248 bytes. In case the DEs were distributed in more that one log file (in order to reach the required 1.0–1.5 Kbytes log file size) then, the security architecture of the system changes and therefore such a proposal had to be abandoned.

Although with a log file of 235 bytes there is space only for two DEs the concept has been proved since at later stages when the card file system is improved it will be possible to keep all the entries under a single log file.

The smart card application simply calls the `GenerateDataEntry` function of the shared library and returns the results to the client. It does not perform any further operations since in that case the timing measurements involved will not be accurate.

The client application provides the means to call the smart card application and also measures the time spent while each data entry (DE) is created.

7.4.3 Results and Performance Evaluation

We have generated a set of results depending on the actual size of the log files and whether the application runs on the card or on the simulator. The implementation and the testing of the client and smart card application used the same setup as the one used for the implementation and testing of LFDM.

The `Connect` figure indicates the time spent by the client application in order to connect with the reader or the simulator. The next value indicates the time spent in the `GenerateDataEntry` procedure. The `Disconnect` figure represents the time spent to close the connection with the reader or the simulator. The results when the `GenerateDataEntry` is executed on the simulator with a log file of 800 bytes, after ten consecutive executions, appear in Table 7.17. A more condensed version is presented in Table 7.9.

Table 7.9: GemXpresso simulator results using a log file of 800 bytes.

Part name	*Time (ms)*	*Std.Deviation*
Connect	11266	46.81
CreateDataEntry	200	3.34
Disconnect	400	5.84
Total	11866	55.99

For example, the figure for the `CreateDataEntry` function also includes the aforementioned communication overhead when calling the function from the client application and sending the results back to the client. Obviously, when the function will be called within the smart card application it will not have to send any information back to the "client". At the time of implementation we did not had access to the tools that will accurately measure the time spent when a function is executed in the card (e.g. by measuring the actual smart card clock cycles) without been invoked within a client application.

Table 7.10: GemXpresso card results using a log file of 235 bytes.

Part name	*Time (ms)*	*Std.Deviation*
Connect	2704	39.96
CreateDataEntry	2293	62.81
Disconnect	120.5	4.86
Total	5117.5	107.63

When the `GenerateDataEntry` is executed in the card with a log file of 235 bytes we get the results given in Table 7.10, for an extended version with the actual measurements

please refer in Table 7.18.

As we can see from Tables 7.9 and 7.10 the `Disconnect` figure presents smaller variation compared to the `Connect` figure. This could be explained by the fact that when the card terminates any communications there is a small number of fixed steps to be followed. On the other hand, when establishing connection with the reader there are more processing steps involved. The results from the simulator are provided in order to demonstrate that our implementation actually works with large files as proposed in the standard.

The size in bytes of each of the above three entities (client, applet and library) when implemented in Java and downloaded in the Java Card are as follows: The smart card library, offering standard log file format functionality, is 1287 bytes and the smart card application calling the `GenerateDataEntry` function is 253 bytes. The client application is 5500 bytes. The client is using the proxy interface which is 2691 bytes. Similarly to the implementation of the LFDM the above file sizes are for the Java Card source files.

7.5 Observations for Both Implementations

The availability of a working simulator is very important in the development phases of the Java applet. It is not practical to have to download the applet to the Java Card in order to test it. Also, it would be very useful for the simulator to be able to simulate aspects of the Java Card, such as memory restrictions and smart card processor speed.

From our testing implementation we actually discovered that the more times the smart card application is downloaded to the card the slower the application execution becomes. It seems somehow that the card is affected after a number of application downloads. This was certainly the case for the GemXpresso card.

From the overall execution results of the protocols, it is shown that a real-world implementation of the proposals should be considered viable in the near future. Once

the cryptographic functions become available and the Java Card performance is further enhanced (towards better handling of larger data structures, faster execution times), it is expected that more complex applications will become a reality. In order to get a better understanding of the issues involved, more realistic implementations that will provide the proposed functionality as part of the operating system are required.

Unfortunately, when implementing the proposals it was very difficult to get hold of any development tools that will enable us to provide the log file standard format functionality as part of the SCOS. When we contacted Gemplus, they suggested that their testing environments and development tools are relatively expensive and contain proprietary designs that make them very difficult to be used outside the Gemplus laboratories.

Furthermore, in order to be able to provide accurate measurements on the actual time spent by the smart card applications, specialized tools are required. If these tools are provided as part of the development kits they will help the application developers to obtain more accurate performance measurements.

Finally, from the performance timings of the standard log file format we observe that the figures do not add substantial delay on the overall performance of a smart card application. Similarly, the size of all the Java Card byte code applications, for both implementations, is relatively small. The smart card application sizes will become smaller since at this stage they also contain the functionality for generating and handling the actual log files.

7.6 Summary

Although certain compromises had to be made with regard to the small log file size and the use of dummy cryptographic functions, we believe that we demonstrated that the concepts of secure log file download and standard log file format can be implemented.

The conclusion to be drawn from the speed of the hash and MAC functions [15] is that these dummy functions do not contribute substantially to the speed of the LFDM protocol and the standard log file format. Other factors, such as file system access, internal array copying and communicating to/from the smart card contribute to the speed slowdown.

7.7 Overall Performance Tables

In this section we provide the tables containing the actual measurements from the test implementations of the log file download protocol and the log file standard format.

Table 7.11: Ten consecutive measurements on the GemXpresso Java Card simulator using log file size of 248 bytes.

GemXpresso Java Card Simulator with a log file of 248 bytes						
	Connect	*Validate*	*StepOne*	*Verify*	Disconnect	Total
1	11260	210	201	200	401	12548
2	11296	210	200	200	401	12668
3	11196	210	200	200	401	12468
4	11226	210	200	200	401	12498
5	11246	210	200	200	401	12518
6	11196	200	201	200	401	12468
7	11276	221	200	200	400	12478
8	11206	211	200	201	400	12478
9	11246	211	200	201	400	12518
10	11266	211	200	201	401	12538
Median	11246	210	200	200	401	12508
StDev	34,62	4,97	0,42	0,52	0,47	60

Table 7.12: Ten consecutive measurements on the GemXpresso Java Card simulator using a log file of 40 bytes.

GemXpresso Java Card Simulator with log file of 40 bytes						
	Connect	*Validate*	*StepOne*	*Verify*	Disconnect	Total
1	11271	210	140	200	401	12248
2	11236	210	140	201	400	12207
3	11256	201	140	200	401	12228
4	11256	211	140	200	401	12225
5	11256	211	140	200	401	12228
6	11236	210	140	201	400	12207
7	11186	210	140	201	400	12157
8	11236	200	140	200	400	12207
9	11236	210	141	200	401	12208
10	11266	211	140	200	401	12238
Median	11246	210	140	200	401	12216
StDev	24,18	4,20	0,32	0,48	0,50	25,08

Table 7.13: Ten consecutive measurements on the GemXpresso Java Card using log file size 40 bytes.

GemXpresso Card with log file of 40 bytes						
	Connect	*Validate*	*StepOne*	*Verify*	Disconnect	Total
1	2683	381	1092	1872	120	6168
2	2694	361	1091	1863	130	6169
3	2704	371	1091	1873	130	6189
4	2714	380	1082	1862	130	6188
5	2703	371	1081	1872	130	6189
6	2684	371	1082	1892	120	6208
7	2684	370	1092	1882	131	6179
8	2684	380	1092	1882	121	6179
9	2684	370	1092	1862	130	6158
10	2674	371	1081	1873	130	6159
Median	2689	371	1091	1862	130	6179
StDev	12,76	6,13	5,27	9,80	4,59	15,57

Table 7.14: GemXpresso Java Card measurements for four Hashs and four MACs.

GemXpresso Java Card measurements		
	Four Hashst	*Four MACs*
1	621	521
2	551	521
3	551	521
4	551	511
5	551	510
6	561	510
7	561	511
8	560	511
9	560	511
10	550	521
Median	555,5	511
StDev	21,39	5,35

Table 7.15: Ten consecutive measurements on the Open 16K Java Card using a log file of 40 bytes.

Open 16K Card with log file of 40 bytes						
	Connect	*Validate*	*StepOne*	*Verify*	Disconnect	Total
1	200	200	2464	2173	951	5988
2	201	190	2474	2173	951	5989
3	190	190	2474	2173	951	5978
4	200	191	2473	2163	952	5979
5	200	190	2464	2173	951	5978
6	201	190	2474	2163	951	5979
7	201	190	2474	2173	951	5989
8	190	200	2464	2173	951	5978
9	200	201	2473	2173	952	5999
10	191	200	2463	2174	951	5979
Median	200	190.5	2473	2173	951	5979
StDev	4,90	5,22	5,14	4,28	0,42	7,24

Table 7.16: Open 16K Java Card measurements for four Hashs and four MACs.

GemXpresso Java Card measurements		
	Four Hashst	*Four MACs*
1	701	751
2	711	751
3	711	751
4	701	761
5	701	761
6	711	751
7	711	761
8	711	751
9	701	761
10	711	751
Median	711	751
StDev	5,16	5,16

Table 7.17: GemXpresso Java Card simulator measurements for a standard log file format of 800 bytes.

GemXpresso Card Simulator standard log file format				
	Connect	*Create Entry*	*Disconnect*	Total
1	11348	190	381	13189
2	11316	201	400	12287
3	11277	200	401	12248
4	11207	200	401	12178
5	11236	201	400	12207
6	11227	200	401	12198
7	11256	201	400	12227
8	11286	200	400	12257
9	11206	201	400	12177
10	11286	200	401	12258
Median	11266,5	200	400	12237,5
StDev	46,81	3,84	5,87	306,61

Table 7.18: GemXpresso Java Card Measurements for a standard log file format of 235 bytes.

GemXpresso Card standard log file format				
	Connect	*Create Entry*	Disconnect	Total
1	2800	2360	131	9875
2	2734	2353	130	8913
3	2694	2353	121	8573
4	2774	2183	120	8512
5	2674	2283	120	8702
6	2694	2303	120	8602
7	2694	2213	131	8553
8	2694	2343	120	8602
9	2714	2253	121	8402
10	2716	2260	121	8380
Median	2704	2293	120,5	8587,5
StDev	39,96	62,81	4,86	107,63

Chapter 8

Conclusions

Contents

In this chapter we provide some concluding remarks along with suggestions for future work.

8.1 Summary and Conclusions

The main goal of this thesis was to explore how smart card log files can be used in the light of recent smart card software and hardware developments, and also to provide an alternative to the currently favoured solution of overwriting smart card log files as soon as they fill up.

We have explained that smart card log files should be considered as a vital part of the overall concept of smart card security. We have also provided what appears to be the missing link for most of the earlier theoretical smart card work, namely smart card application performance measurements. Throughout our proposals, we aimed to provide schemes that can be applied in current smart cards without sacrificing security and without restricting the cardholder's behaviour. In the following paragraphs we highlight the essential contribution of this thesis, in order to identify which problems have been addressed and how they have been solved.

Initially we presented a detailed coverage of smart card technology in order to help the reader to understand its characteristics. Subsequently, when examining the related work in this area, it became clear that until recently smart card log files have only been used for auditing or recovery purposes in a single application platform. In the new multi-application smart card environment, the number of entities involved has increased and the relationships among the entities are becoming more complex. Therefore, we have proposed an event logging model for smart cards and we have identified the characteristics of each of the participants. In this logging model there is a whole new class of events that require logging. For example, it is not good security practice to rely on the smart card application programmer in order to log certain events. Thus, we identified those events that require logging and we also proposed a centralised entity (i.e. the LFM) to be responsible for enforcing a logging policy instead of relying on the good will of the applications to comply.

In order to provide a more detailed coverage of the logging concept in multi-application smart cards, we divided it into three phases: generating the evidence in the log files, providing controlled access to the log files and extracting the log files to another entity as soon as they become full. This categorisation helps to understand the problem and also makes implementation easier since the behaviour of each entity is carefully defined.

The fact that the LFUM is a trusted SCOS entity responsible for performing independent and unbiased logging, makes it possible for smart card applications and dispute resolution entities to rely on this service in order to obtain the required information. As we explained in chapter 4, this information can be used for pay-as-you-use purposes by other applications or for auditing purposes.

Since the log files contain sensitive information, it is obvious that access to these files (when stored in the card) should be controlled; this is achieved by the LFBM. We also pointed out that in an environment where resources are limited, it is likely that without proper log file space management, this space might be misused. As a result we proposed a standard log file format for smart cards. This format provides an upper limit to the information that each smart card application can write to the log files. We also proposed that the mechanism enforcing the standard log file format can also collect any truncated space which is left unused by applications, and subsequently, reallocate the space to be used by other applications. The standard log file format also aims to speed up the dispute resolution phase since the context of the log files can easily be interpreted.

Moving on to the central theme of the thesis, we provided a solution to the full smart card log file problem. We suggested that the log files should be downloaded to another entity (i.e. an ALSS), which does not suffer from immediate storage restrictions. We provided some theoretical contributions in the form of three log file downloading protocols the choice of which depends on the characteristics of the participants. The log file download protocols take into account the characteristics of smart card technology,

and by using appropriate cryptographic protection, they successfully protect the log files both in transit and when they are stored in an ALSS. These protocols are useful primitives for more complex systems and therefore, we believe that their functionality should be extended.

Finally, in order to prove the applicability of our proposals to current smart card technology, we provided their implementation details for two of the most widely available multi-application smart cards. We implemented the standard log file format and the first of the log file download protocols in order to get an idea of their size and performance in real smart cards. Both results indicated that a real world implementation is perfectly feasible since their execution is not substantially delayed by the performance of the smart card processor. Furthermore, their implementation can fit within the limited storage space of the current smart cards.

Since the protocols require cryptography and the cards did not have any cryptographic primitives, we designed our own dummy cryptographic functions. For the LFDM the hash and MAC functions were 16 bytes long, whereas for the standard log file format the MAC function was 4 bytes. The main reason for choosing a smaller MAC size for the standard log file format is that in this case for each DE a MAC function have to be stored in the log file. Thus, by using a MAC of a smaller but still adequate size we gain extra log file space. The second and the third log file download protocols require additional cryptographic primitives, in order to protect the log file while in transit or when stored in the ALSS. Thus, we decided not to implement them, since such work would mainly involve smart card cryptographic function implementation rather than actual protocol implementation. When cryptographic primitives become available with newer versions of the development kits, it will become straight forward to implement the remaining protocols.

The major conclusions that can be drawn from the work presented in the thesis are the following. It is assumed in many cryptography papers that smart cards are “anaemic

devices" that should do as little computation as possible. At the same time, these authors tend to assume that communications come for free. For example, they tend to provide improvements in the performance of cryptographic algorithms assuming that it is the major delaying factor in smart card performance. Actually, our measurements show the direct opposite. Therefore, we can claim with a high degree of confidence that special attention should be paid to improving smart card communication protocols, rather than only improving the performance of cryptographic functions.

The size of APDUs is another critical factor affecting the performance of smart card applications. For example, when sending/receiving large amounts of data, i.e. more than 250 bytes to/from the card, then special programming is required in order to disassemble and reassemble a large packet. When this process is taking place at the client side, "speed" is not an issue since the processing power is available. When the process is taking place in the card, this operation becomes very time consuming. In the Java Card API 2.1 there is an initial attempt to make the problem more abstract by providing the functionality for handling large packets of data, within the underlying API.

Another issue that appears to increase the overall application execution time is file management in the card and generally the speed at which data are read/written, from/to the EEPROM memory of the card. For applications that require a large number of file accesses, such as the moving of data between arrays, etc. the execution time is also increased.

Smart card logging mechanisms can improve the overall level of security found within smart cards. We demonstrated that multi-application smart card technology introduces new events that require logging. We also pointed out that, since smart cards can be involved in a large number of transactions the possibility of disputes is increased, therefore it is important that log file information should be preserved for the longest possible time. We also derived the essential requirements for a smart card logging

mechanism. The implementation results from both our standard log file format and the log file downloading protocols indicate that a real implementation of the proposals is feasible. Finally, we believe that with the issues raised from our work the smart card logging mechanisms will receive the attention they deserve.

8.2 Suggestions for Future Work

Our intention in this work was to highlight the issues involved in smart card log file logging. We have achieved this goal but there are a number of suggestions for further improvements and directions for future research.

For example, in order to get performance figures for the log file download protocol and the standard log file format we used the PC's clock within the client application. This implies that the resulting figures also contain the communication overhead when data travel to/from the smart card. Thus, it would be very convenient if the development kits provided the functionality for measuring the time spent executing a function in the card. Although GemXpresso does offer the ability to measure the time spent when an APDU is executed in the card, this feature does not work when the APDUs are replaced by DMI calls, as in our implementation.

Smart card development kits need further refinement and improvement. For example the compile-to-run cycle must be reduced by improving the development kit user interfaces or by merging certain tasks, e.g. the smart card byte code verification and application download procedure. Similarly, simulators should become more reliable (since they crash very often) and they should also simulate the exact performance of the card (e.g. smart card processor speed, EEPROM and RAM restrictions, etc.) and not just the application execution behaviour.

There are two further observations about smart card development kits: Firstly, we encountered problems when declaring and handling smart card files larger than 248 bytes.

Certainly, this is an issue that must be addressed by the smart card manufacturers, since other types of applications (e.g. health care) will inevitably require large files. Secondly, the more frequently a smart card application is executed the slower it becomes. Since this was certainly the case for the GemXpresso card we have reported these findings to Gemplus France.

It would also be helpful to examine the two client application proposals (i.e. PC/SC and OCF) and comment on their performance. This will reveal which of the proposals might perform better in terms of speed. The above work will also indicate how much time is spent when a smart card application is executed and how much time is spent on communications.

To get more accurate results on the performance of the proposed architectures and protocols they should be implemented as part of the SCOS. For example, the LFUM could not be implemented at the application level since in order to achieve the required functionality it requires detailed low level knowledge of the SCOS. When the functionality is within the SCOS and stored in ROM (i.e. it could also use certain SCOS primitives which are not available at the application level) it may be that the overall performance might improve and the application size might be reduced.

As a concluding remark we have to mention that a real world implementation of the proposals presented in the thesis is inextricably linked to the following: the type of the card (merchant card, user card, high value transaction card, etc.), its technological characteristics and limitations (e.g. size of EEPROM memory, SCOS functionality, etc.) along with the whole surrounding infrastructure (e.g. the number of ALSSs). And finally, one must again stress the importance of defining appropriate security policies that clearly address the relationships of the participants along with the exact legal framework for the arbitration procedure.

Appendix A

Glossary and the Java Source Code Program Listings

Contents

In this section, we provide a glossary of the abbreviations used in the dissertation along with the code listings from the experimental implementations mentioned in the previous chapters.

A.1 Glossary

A collection of selected acronyms and abbreviations used in the dissertation.

A	an arbitrator responsible for resolving any disputes
AET	auditable event table
AID	application identifier
ALSS	audit log storage server
ALT	application logging table
AP	the application running in the smart card
APDU	application protocol data unit
API	application programming interface
ATM	automated telling machine
ATR	smart card answer to reset
C	a smart card
CPU	central processing unit
DE	data entry to be included the log file
DLL	dynamic link libraries
DMI	direct method invocation
EEPROM	electrically erasable programmable read only memory
EID	unique event identification information, of the event to be logged
EMV	Europay, Mastercard, Visa Specification
GSM	global system for mobile communications
GPOS	general purpose operating system
JCVM	a Java Card virtual machine
IFD	smart card interface device
KIP	kilo instruction per second
LE	the actual logged event
LF	the smart card log file
LFM	the smart card log file manager
LFS	log file space
LFBM	log file browse manager
LFDM	log file download manager
LFUM	log file update manager
M	a malicious user
Mask	the smart card operating system and other data stored in the card
MAOS	multi application operating system
MCOS	multi-application chip operating system
MEL	multos executable language
MPCOS	multi-application payment chip operating system
NC	network computer
OCF	opencard framework specification
PC/SC	personal computer smart card specifications
PD	personal device
RAM	random access memory
ROM	read only memory
SCOS	the smart card operating system

SCQL	structured card query language
SIM	secure identity module
SSL	secure sockets layer
SP	the smart card application service provider
TCB	trusted computing base
U	the smart card holder
VM	virtual machine

A.2 The GemXpresso Log File Download Manager Code Listings

In this section we provide the code listings from the Log File Donwload Manager implementation on the GemXpresso Smart Card.

A.2.1 The LFDM Interface

```
//————————————————————————————————————————————————————————————————————————-
//      File Details    : ilfdm.java, 1999/7/16, 16:47, File Size: 1243 bytes
//      Description     : The interface for the Log File Download Manager using DMI.
//
//      Author  :       Constantinos Markantonakis
//      Date            : 28/10/1998
//
//————————————————————————————————————————————————————————————————————————-

package PhD.FirstDMI ;

import javacard.framework.* ;
import javacardx.framework.* ;
import com.gemplus.gemxpresso.library.util.IApplet ;
import com.gemplus.gemxpresso.library.iso.ISOFileSystem ;
import com.gemplus.gemxpresso.library.util.SimpleOwnerPIN ;
import com.gemplus.gemxpresso.library.mylibrary.Constants;
import com.gemplus.gemxpresso.library.mylibrary.Core;

public interface ILFDM extends IApplet
{
  /**
  * Initialise the MF and the Log file.
  */
  public byte InitializeMFandLogFile() throws UserException;

  /**
  * Present the Cardholder or the Authority PIN.
  public byte Present(byte[] TypedCode, byte CodeNumber)
        throws UserException;

  /**
  * Sends message to ALSS
  */
  public byte[] Send_LogFile123() throws UserException;

  /**
  * Verify the ALSS reply.
  */
  public byte ALSS_Reply123(byte[] DataIn) throws UserException;

}
```

A.2.2 The Log File Download Manager Application

```
//-----------------------------------------------------------------------
//      File Details    : lfdm.java, 1999/7/16, 16:47, File Size: 3049 bytes
//      Description     : The actual Log File Download Manager (LFDM) using DMI.
//
//      Author  :       Constantinos Markantonakis
//      Date            : 10/9/1998
//
//-----------------------------------------------------------------------

package PhD.FirstDMI ;

import javacard.framework.* ;
import javacardx.framework.* ;
import com.gemplus.gemxpresso.library.iso.ISOFileSystem ;
import com.gemplus.gemxpresso.library.util.SimpleOwnerPIN ;
import com.gemplus.gemxpresso.library.mylibrary.Constants;
import com.gemplus.gemxpresso.library.mylibrary.Core;

public class LFDM extends Applet
  implements ILFDM
{
  /**
  * The Aythorization for enabling the LFDM to do download the log file
  */
  private SimpleOwnerPIN thePIN ;

  /**
  *       Access to the Core package
  */
  private Core theCore;

  // This variable is used in the ALSS_Reply function
  int Incoming_Packet_No=0;
  // This variable controls the packet number sent by the Card
  int Outgoing_Packet_No=0;
  // Use in Initialize and Present_PIN procedures
  byte Result;

  /**
  * Construct the LFDM.
  */
  public LFDM(byte[] code, byte tryLimit)
        throws UserException
  {
        thePIN = new SimpleOwnerPIN(code, tryLimit) ;
        Core theCore = new Core();
  }

  /*************************
  * Selects the LFDM applet.
  *************************/
  public boolean select()
  {
        thePIN.reset() ;
        return true ;
  }
```

```
/************************************
* Initialise the MF and the Log file.
************************************/
public byte InitializeMFandLogFile() throws UserException
{
      try
      {
        if (theCore.Initialize_File_System() &&
              theCore.Initialize_LogFile())
        {
              Result = 1    ;
        }
        else
        {
              Result = 0    ;
        }
      }
      catch (Exception e)
      {
        throw new
        UserException(Constants.PROBLEM_INITIALIZING_FILE_SYSTEM);
      }

      return Result;
}

/*******************************************
* Present the Cardholder or the Authority PIN.
*******************************************/
public byte Present(byte[] TypedCode, byte CodeNumber)
      throws UserException
{
      try
      {
        Result = theCore.PresentCode(thePIN, TypedCode, CodeNumber);
      }

      catch (Exception e)
      {
        throw new UserException(Constants.PIN_IS_BLOCKED);
      }

      return Result;
}

/************************
* Sent the First packet.
************************/
public byte[] Send_LogFile123() throws UserException
{
      // This function simply calls the send first packet procedure
      return theCore.LFDM_Step_One();
}

/************************
* Verify the ALSS reply.
************************/
public byte ALSS_Reply123(byte[] DataIn) throws UserException
```

```
  {
        boolean Value=false;
        try
        {
          if (theCore.Verify_ALSS_Reply(DataIn))
          {
                return (byte)1;
          }
          else
          {
                return (byte)0;
          }

        }
        catch (Exception e)
        {
          throw new
          UserException(Constants.PIN_NOT_VERIFIED_or_FILE_NOT_FULL);
        }
  }
}
```

A.2.3 The Constant Definition for the Core Library (Constants)

```
//-------------------------------------------------------------------------
//      File Details    : constants.java, 1999/7/16, 16:47, File Size: 975 bytes
//      Description     : This interface simply defines the constants that are shared
//                        by all the applications that may use the core package.
//
//      Author   :      Constantinos Markantonakis
//      Date            : 16/10/1998
//
//-------------------------------------------------------------------------

package com.gemplus.gemxpresso.library.mylibrary;

/**
* This interface simply defines the constants that are shared by all the
* applications that may use the core package.
*/
public interface Constants
{
    //
    // Status codes used in exceptions
    //
    public static final byte PROBLEM_INITIALIZING_FILE_SYSTEM = 0x30 ;
  public static final byte PROBLEM_INITIALIZING_FILES = 0x31 ;
  public static final byte PIN_IS_BLOCKED = 0x32 ;
  public static final byte PIN_NOT_VERIFIED_or_FILE_NOT_FULL = 0x33 ;
}
```

A.2.4 The Core Library Implementing the LFDM

```
//————————————————————————————————————————————————————————————————————————-
//      File Details    : core.java, 1999/7/16, 16:47, File Size: 11074 bytes
//      Description     : The core library containing all the functions
//                        accessing the log file.
//
//      Author   :      Constantinos Markantonakis
//      Date            : 10/11/1998
//
//————————————————————————————————————————————————————————————————————————-

package com.gemplus.gemxpresso.library.mylibrary;
import javacard.framework.* ;
import javacardx.framework.* ;
import com.gemplus.gemxpresso.library.iso.ISOFileSystem ;
import com.gemplus.gemxpresso.library.util.*;

public final class Core implements Constants
{
  /**
  *The reference to the Files.
  */
  private static FileSystem MF ;
  private static DedicatedFile PIN_DF ;
  private static DedicatedFile LogFile_DF ;
  private static TransparentFile LogFile ;
  private static TransparentFile CBindFile ;
  private static TransparentFile KeyID_File ;
  private static TransparentFile SequenceNumber_File ;

  /**
  * The initialization status variables.
  */
  private static boolean MFInitialized = false;
  private static boolean LogFileInitialized = false;
  private static boolean LogFile_is_Full= false;

  /**
  * This section holds the global variables
  */
  //This holds the Log file
  static byte [] buffer = new byte[40];
  //An array index variable
  static int Index;

  //These variables are used in the initilaize log files variable
  static byte [] Temp_CBind = {0,0,0,10,0,0,0,8,0,0,0,5,0,0,0,11};
  static byte[] ID =          {0,1,2,3,4,5,6,7,8,9};
  static byte[] Seq =         {0,0,0,1};

  //These variables are used in LFDM_Step_One()
  static byte[] Tc = {2,5,0,5,1,9,7,3};
  static byte[] Message = new byte[20];
  private static byte[] Mn= new byte[94]; //was 78

  //These variables are used in Verify_ALSS_Reply
  static boolean Verified_Reply = false;
```

```
static byte[] write={0,0,0,0} ;
static int NewNc = 0;

/**
* This section holds the global variables
*/
private static int Execution_Step = 0;

/**
* This is the basic constructor. The initializations are in fact taking place
* later on in the following functions.
*/
public Core ()
{
}

/**
*     The following section is added by me I do not know if it is working
*     properly.
*/
public boolean select()
{
      MF.reset() ;
      return true ;
}

/*****************************************************************
* Create the file system structure.
*****************************************************************/

public static boolean Initialize_File_System () throws UserException
{
      try
      {
        MF =
        new ISOFileSystem((short)0x3F00, null, (byte)2) ;
        PIN_DF =
              new DedicatedFile((short)0x4000, null, (byte)2) ;
        LogFile_DF =
              new DedicatedFile((short)0x5000, null, (byte)4);
        LogFile =
              new TransparentFile((short)0x5001, (short)40);
        CBindFile =
              new TransparentFile((short)0x5002, (short)16);
        KeyID_File =
              new TransparentFile((short)0x5003, (short)10);
        SequenceNumber_File =
              new TransparentFile((short)0x5004, (short)4);

        // The READ access control for the files
        MF.setSecurity(File.ACCESS_READ, File.ALLOW_ANY) ;
        PIN_DF.setSecurity(File.ACCESS_READ, File.ALLOW_AUTH2) ;
        LogFile_DF.setSecurity(File.ACCESS_READ, File.ALLOW_AUTH1) ;
        LogFile.setSecurity(File.ACCESS_READ, File.ALLOW_AUTH1) ;
        CBindFile.setSecurity(File.ACCESS_READ, File.ALLOW_AUTH1) ;
        KeyID_File.setSecurity(File.ACCESS_READ, File.ALLOW_AUTH1) ;
        SequenceNumber_File.setSecurity(File.ACCESS_READ, File.ALLOW_AUTH1) ;

        // The WRITE access control for the files
```

```
        MF.setSecurity(File.ACCESS_WRITE, File.ALLOW_AUTH1) ;
        PIN_DF.setSecurity(File.ACCESS_WRITE, File.ALLOW_AUTH1) ;
        LogFile_DF.setSecurity(File.ACCESS_WRITE, File.ALLOW_AUTH1) ;
        LogFile.setSecurity(File.ACCESS_WRITE, File.ALLOW_AUTH1) ;
        CBindFile.setSecurity(File.ACCESS_WRITE, File.ALLOW_AUTH1) ;
        KeyID_File.setSecurity(File.ACCESS_WRITE, File.ALLOW_AUTH1) ;
        SequenceNumber_File.setSecurity(File.ACCESS_WRITE, File.ALLOW_AUTH1) ;

        // Now, create the hierarchy
        MF.addChildFile(PIN_DF) ;
        MF.addChildFile(LogFile_DF) ;
        LogFile_DF.addChildFile(LogFile) ;
        LogFile_DF.addChildFile(CBindFile) ;
        LogFile_DF.addChildFile(KeyID_File) ;
        LogFile_DF.addChildFile(SequenceNumber_File) ;

        MFInitialized = true;
      }
      catch (Exception e)
        {
            throw new
            UserException(PROBLEM_INITIALIZING_FILE_SYSTEM);
        }

      return MFInitialized;
}

/*************************************************************
* Initializes the file system and the files.
*************************************************************/
public static boolean Initialize_LogFile () throws UserException
{
      // The Buffer which holds the initialization for the files
      //
      //      ATTENTION: IF the value is > 50, we get memory full messages
      // ( When the applet is running in the card)
      try
      {
        //Create the log file entries
        for (Index =0; Index<buffer.length; Index++)
              buffer[Index] = (byte)Index;

              // Initialize the contents of the files
        Util.arrayCopy( buffer, (short)(0),LogFile.getData(), (short)0,
                        (short)40) ;
        // For the Cbind VAriable or the Mn
        Util.arrayCopy( Temp_CBind, (short)(0), CBindFile.getData(), (short)0,
                        (short)16) ;
        // For the KeyID
        Util.arrayCopy(ID, (short)(0),KeyID_File.getData(), (short)0,(short)10);
        // For the Sequence Number
        Util.arrayCopy(Seq, (short)(0), SequenceNumber_File.getData(), (short)0,
              (short)4) ;

        LogFileInitialized = true;
        LogFile_is_Full= true;
      }
      catch (Exception e)
      {
```

```
        throw new UserException(PROBLEM_INITIALIZING_FILES);
      }
      return LogFileInitialized;
}


/*********************************************************
* Presents a code or a PIN number.
*********************************************************/
public static byte PresentCode(SimpleOwnerPIN thePIN, byte[] code, byte which)
      throws UserException
{
      //Present the the user code or PIN
      MF.setAuthFlag((byte)1 ,false) ;
      thePIN.check(code) ;

      if (thePIN.getTriesRemaining()==0)
      {
        throw new UserException(PIN_IS_BLOCKED) ;
      }
      else
      //if PIN is validated
      if (thePIN.isValidated()==true)
      {
        MF.setAuthFlag((byte)1 ,true) ;
      }
      return (byte)thePIN.getTriesRemaining();
}


/*********************************************************
* Reads the LogFile.
*********************************************************/
public static byte[] ReadLogFile()
{
      if (MF.getAuthFlag((byte)1) == true )
      {
        return  LogFile.getData() ;
      }
      else
      {
        byte[] Failure={0,0,0} ;
        return Failure ;
      }
}

/*********************************************************
* This function flushes the log file space
*********************************************************/
private static void Flush()
{
      byte [] buffer = new byte[40];
      int I;

      for (I =0; I<buffer.length; I++)
            buffer[I] = (byte)I;

        // Re-write the contents of the log files
      Util.arrayCopy(buffer, (short)(0),LogFile.getData(), (short)0,(short)40) ;
```

```
}

/*******************************************************
* This function computes a MAC or Hash
*******************************************************/

public static byte[] ComputeMAC_or_Hash( byte[] inBuffer, int wantMACorHash )
{
      byte[] outBuffer = new byte[16];
      outBuffer[ 0 ] = inBuffer[ 0 ];
      outBuffer[ 1 ] = inBuffer[ 1 ];
      outBuffer[ 2 ] = inBuffer[ 2 ];
      outBuffer[ 3 ] = inBuffer[ 3 ];

      int counterB = 0;

      // The following hiccups on the Open16K :(
      byte [] CA_Key = {2,8,9,5};
      if ( wantMACorHash == 1 )
      {
        outBuffer[0] = (byte )((byte )outBuffer[0] ^ (byte )CA_Key[0]);
        outBuffer[1] = (byte )((byte )outBuffer[1] ^ (byte )CA_Key[1]);
        outBuffer[2] = (byte )((byte )outBuffer[2] ^ (byte )CA_Key[2]);
        outBuffer[3] = (byte )((byte )outBuffer[3] ^ (byte )CA_Key[3]);
      }
      // Some major optimization. :)
      counterB = 0;
      int counter =0;

      for ( counter = 0; counter < 4 - 1; counter++ )
      {
        counterB = (short )(counterB + 4);
        Util.arrayCopy( outBuffer, (short )0, outBuffer, (short)counterB, (short )4 );
      }

      // outBuffer has the result.
      return outBuffer;
}

/*******************************************************
* Sends the log file information from the card to ALSS
*******************************************************/
public static byte[] LFDM_Step_One() throws UserException
{

if ( (MF.getAuthFlag((byte)1) == true) && (LogFile_is_Full=true) )
      {

      //Get Timestamp, in this case we have fixed timestamp since there is no way
      // currently to get a valid timestamp from the card.

      //Compute a Hash for the CBind variable
      Util.arrayCopy( CBindFile.getData(), (short)(0), Message, (short)0,
                      (short)16);
      Util.arrayCopy( SequenceNumber_File.getData(), (short)(0), Message,
                      (short)16, (short)4);

      Util.arrayCopy( ReadLogFile(), (short)(0), Mn, (short)0,
        (short)40);
```

```
        Util.arrayCopy( Tc,   (short)(0), Mn, (short)40, (short)8);
        Util.arrayCopy( SequenceNumber_File.getData(),(short)(0),
          Mn, (short)48,  (short)4);
        Util.arrayCopy( KeyID_File.getData(),(short)(0), Mn,
          (short)52, (short)10);

        //Compute a Hash, 0 indicates a Hash.
        Util.arrayCopy( ComputeMAC_or_Hash(Message, 0), (short)(0),
          Mn, (short)62, (short)16);

        //Compute a MAC, 1 indicates a MAC.
        Util.arrayCopy(ComputeMAC_or_Hash(Mn,1), (short)(0), Mn,
          (short)78, (short)16);

        //Indicate the next step i.e. Verify_ALSS_Reply.
        Execution_Step = 1;

        return Mn;
        }
else
        {
          throw new
          UserException(Constants.PIN_NOT_VERIFIED_or_FILE_NOT_FULL);
        }
}

/*****************************************************
* Verify information sent by the ALSS.
*****************************************************/
public static boolean Verify_ALSS_Reply(byte[] Reply) throws UserException
{
        //Check if the PIN is validated and if the data are already sent to the ALSS
        if ( (MF.getAuthFlag((byte)1) == true) && (Execution_Step == 1) )
        {
          //Check the hash of the log ifle just received.
          if (  Util.arrayCompare(ComputeMAC_or_Hash(ReadLogFile(), 0),
                  (short)0, Reply, (short)0, (short)16) == 0)
          {
                Verified_Reply = true;
          }
          /**
          *     This is the Commit Value Section where all the values
          *     (Nc, CBind, ) are updated in the corresponding files.
          */
          if (Verified_Reply==true)
          {
                //This section updates the Nc Value

                NewNc = Util32.getInt(SequenceNumber_File.getData(), (short)0);
                NewNc = NewNc + 1;
                short k=Util32.setInt(write, (short)0, (int)NewNc);

                Util.arrayCopy( write, (short)(0), SequenceNumber_File.getData(),
                                (short)0, (short)4) ;
                //1) This section updates the Cbind Variable in the file
                Util.arrayCopy( ComputeMAC_or_Hash(Mn, 0), (short)(0),
                                CBindFile.getData(), (short)0, (short)16) ;

                //2) Call the Flush() function in order to Flush the Log File space
```

```
                Flush();
            }
            Execution_Step = 0;
            return Verified_Reply ;
        }
        else
        {
            return false;
        }
    }

}
```

A.3 The Cyberflex Log File Download Manager Code Listings

In this section we provide the code listing from the Log File Donwload Manager implementation on the Cyberflex Open 16K Smart Card.

```
////////////////////////////////////////////////////////////////////////
//File Details        : logmanager.java, 1999/7/16, 17:18, File Size: 12462 bytes
//Secure logfile download for Java Cards.
// Implementation for Open16K Javacard by Schlumberger.
//
// Date: 26Apr1999
////////////////////////////////////////////////////////////////////////

import javacard.framework.*;
  import javacardx.framework.*;

  public final class LogManager extends javacard.framework.Applet
{
      static final short MACTEMPSIZE = 4;
      static final short MACSIZE = 16;
      static final short REPLYSIZE = 34;
      // Must change following if change above for MACTEMPSIZE.
      static final byte  WANT_MAC  = (byte )0x0;
      static final byte  WANT_HASH = (byte )0x1;

      // Holds the value of Mn.
      static final byte MNSIZE = (byte ) 94;
      // Common filenames on smartcard
      static final short RootDirectory = (short ) 0x3F00;
      static final short LogDirectory = (short ) 0x1065;
      static final short LogFile = (short ) 0x1060;
      static final short KeyIDFile = (short ) 0x1062;
      static final short CBindFile = (short ) 0x1068;
      static final short SeqNumFile = (short ) 0x1069;
      // Constants declaration
      // Code of CLA byte in the command APDU header
      static final byte LogManager_CLA   = (byte)0x00;
      // Codes of INS byte in the command APDU header
      static final byte INS_Validate = (byte ) 0x10;
      static final byte INS_TestHASH = (byte ) 0x20;
      static final byte INS_TestMAC = (byte ) 0x30;
      static final byte INS_SendLogFileData = (byte ) 0x40;
      static final byte INS_ReceiveALSSReply = (byte ) 0x50;
      // maximum number of incorrect tries before the PIN is blocked (0xf for tests)
      static final byte PinTryLimit = (byte)0x0f;
      // maximum size PIN
      static final byte MaxPinSize = (byte)0x04;
      // status words (SW1-SW2)
      static final short SW_NOT_VERIFIED = (short ) 0x5AAA;
      static final short SW_WRONG_PIN = (short ) 0x5BBB;
      static final short SW_COULD_NOT_STATE_FILE = (short ) 0x5CCC;
      static final short SW_FILE_SIZE_DISCREPANCIES = (short ) 0x5DDD;
      static final short SW_INVALID_SIZE = (short ) 0x5EEE;
```

```
static final short SW_FIRST_STEP_ONE_THEN_VERIFY= (short ) 0x5FFF;

// Value for timestamp. We do not have a time mechanism to provide timestamps
// so we assume the following one.
byte Tc[]; // = { 2, 5, 0, 5, 1, 9, 7, 3 };

// We allocate memory for this one. It will be recieved.
byte Ts[]; // = { 0, 0, 0, 0, 0, 0, 0, 0 };

// File variables. The files are read in here.
byte[] CBind_FILE;
byte[] PreviousCBind_FILE;
byte[] SequenceNumber_FILE;
byte[] ReadLogFile_FILE;
byte[] KeyIDFile_FILE;
byte[] machash;
byte[] CA_Key;

// We save memory here also. We set Mn to be the buffer of APDU class :)
// We use it as a pointer, that we will point to "buffer" when we use it.
byte Mn[];

byte[] LogFileRec_STR;
byte[] Message;

byte[] CBind;
byte[] fKca;

/* instance variables declaration */
OwnerPIN pin;

byte buffer[]; // APDU buffer
// A variable for the "for" loops, whoever needs it.
byte var;
short counter;
short counterB;

short FileSize = -1;                    // File size in bytes.
boolean StepOneTookPlace = false;
boolean theResult;

private LogManager()
{
  // It is good programming practice to allocate all the memory that an
  // applet needs during its lifetime inside the constructor.

  pin = new OwnerPIN(PinTryLimit, MaxPinSize);
  PreviousCBind_FILE = new byte[20];
  SequenceNumber_FILE = new byte[4];
  ReadLogFile_FILE = new byte[40];
  KeyIDFile_FILE = new byte[10];
  CBind_FILE = new byte[16];
  machash = new byte[16];
  Message = new byte[20];
  LogFileRec_STR = new byte[10];

  LogFileRec_STR[0] = (byte )0x4c;
  LogFileRec_STR[1] = (byte )0x6f;
  LogFileRec_STR[2] = (byte )0x67;
```

```
  LogFileRec_STR[3] = (byte )0x46;
  LogFileRec_STR[4] = (byte )0x69;

  LogFileRec_STR[5] = (byte )0x6c;
  LogFileRec_STR[6] = (byte )0x65;
  LogFileRec_STR[7] = (byte )0x52;
  LogFileRec_STR[8] = (byte )0x65;
  LogFileRec_STR[9] = (byte )0x63;
  Tc = new byte[8];
  Tc[0] = (byte ) 0x2;
  Tc[1] = (byte ) 0x5;
  Tc[2] = (byte ) 0x0;
  Tc[3] = (byte ) 0x5;
  Tc[4] = (byte ) 0x1;
  Tc[5] = (byte ) 0x9;
  Tc[6] = (byte ) 0x7;
  Tc[7] = (byte ) 0x3;

  Ts = new byte[8];
  Mn = new byte[MNSIZE];
  CA_Key = new byte[4];
  CA_Key[0] = (byte )0x4;
  CA_Key[1] = (byte )0x6;
  CA_Key[2] = (byte )0x8;
  CA_Key[3] = (byte )0x2;

  // PreviousCBind has to be initialised so as to be used.
  Util.arrayFillNonAtomic( PreviousCBind_FILE, (byte ) 0xDD );
  Util.arrayFillNonAtomic( machash, (byte ) 0xDD );

  register();
} // end of the constructor

public static void install(APDU apdu)
{
  // create a LogManager applet instance (card)
    new LogManager();

}

public boolean select()
{
  // reset validation flag in the PIN object to false
  pin.reset();
  StepOneTookPlace = false;
  return true;
} // end of select method

public void process(APDU apdu)
{
  buffer = apdu.getBuffer();
  if (buffer[ISO.OFFSET_CLA] != LogManager_CLA)
        ISOException.throwIt(ISO.SW_CLA_NOT_SUPPORTED);

  switch (buffer[ISO.OFFSET_INS])
  {
        case INS_Validate : validate(apdu);          return;
        case INS_TestHASH : testHASH(apdu);          return;
        case INS_TestMAC : testMAC(apdu);            return;
```

```
            case INS_SendLogFileData : sendLogFileData(apdu); return;
            case INS_ReceiveALSSReply : receiveALSSReply(apdu); return;
            default: ISOException.throwIt( ISO.SW_INS_NOT_SUPPORTED );
        }
    }

    private void validate(APDU apdu)
    {
        byte byteRead = (byte)(apdu.setIncomingAndReceive());
        if (!pin.check(buffer, ISO.OFFSET_CDATA, byteRead))
            ISOException.throwIt(SW_WRONG_PIN);
    }

private final void ComputeMAC_or_Hash( byte[] inBuffer, byte[] outBuffer,
    byte wantMACorHash )
{
    outBuffer[ 0 ] = inBuffer[ 0 ];
    outBuffer[ 1 ] = inBuffer[ 1 ];
    outBuffer[ 2 ] = inBuffer[ 2 ];
    outBuffer[ 3 ] = inBuffer[ 3 ];
    counterB = 0;
    // The following hicups on the Open16K :(
    if ( wantMACorHash == WANT_MAC )
    {
        outBuffer[0] = (byte )((byte )outBuffer[0] ^ (byte )CA_Key[0]);
        outBuffer[1] = (byte )((byte )outBuffer[1] ^ (byte )CA_Key[1]);
        outBuffer[2] = (byte )((byte )outBuffer[2] ^ (byte )CA_Key[2]);
        outBuffer[3] = (byte )((byte )outBuffer[3] ^ (byte )CA_Key[3]);
    }
    // Some major optimization. :)
    counterB = 0;
    for ( counter = 0; counter < MACTEMPSIZE - 1; counter++ )
    {
        counterB = (short )(counterB + 4);
        Util.arrayCopy( outBuffer, (short )0, outBuffer, counterB, (short )4 );
    }
}

// The value returned is Mn, we do not use return to avoid excessive
// memory usage. (Well, it does not work otherwise).
private final void sendLogFileData(APDU apdu)
{
    // access authentication
    if ( ! pin.isValidated() )
        ISOException.throwIt(ISO.SW_PIN_REQUIRED);
    // we want to send data out.
    apdu.setOutgoing();
    // indicate the number of bytes in the data field
    apdu.setOutgoingLength((byte)MNSIZE);
    // Read the Logfile.
    readLogFile( ReadLogFile_FILE, LogFile );
    // Read the saved (previous) value of CBind.
    readLogFile( PreviousCBind_FILE, CBindFile );
    // Read the sequence number file.
    readLogFile( SequenceNumber_FILE, SeqNumFile );
    // Read the unique key identifier for the corresponding CA Key.
    readLogFile( KeyIDFile_FILE, KeyIDFile );
    // Construct the message Mn
    Util.arrayCopy( PreviousCBind_FILE, (short )0, Message, (short )0, (short )16 );
```

```
        Util.arrayCopy( SequenceNumber_FILE, (short )0, Message, (short )16, (short )4 );
        // Compute the hash of Message
        // OPTI: CBind was initialised dynamicaly, waste of memory.
        CBind = machash;
        ComputeMAC_or_Hash( Message, CBind, WANT_HASH );

        Util.arrayCopy( ReadLogFile_FILE, (short )0, Mn, (short )0, (short )40 );
        Util.arrayCopy( Tc, (short )0, Mn, (short )40, (short )8 );
        Util.arrayCopy( SequenceNumber_FILE, (short )0, Mn, (short )48, (short )4 );
        Util.arrayCopy( KeyIDFile_FILE, (short )0, Mn, (short )52, (short )10 );
        Util.arrayCopy( CBind, (short )0, Mn, (short )62, (short )16 );

        // Compute the MAC of the Message.
        // OPTI: fKca was initialised dynamicaly, waste of memory.
        fKca = machash;
        ComputeMAC_or_Hash( Mn, fKca, WANT_MAC );
        // Copy fKca (size:16) to the end of Mn.
        Util.arrayCopy( fKca, (short )0, Mn, (short )78, (short )16);
        // move the data into the APDU buffer starting at offset 0
        Util.arrayCopy( Mn, (short )0, buffer, (short )0, (short )MNSIZE);
        StepOneTookPlace = true;
        // send MNSIZE bytes of data at offset 0 in the APDU buffer to the client.
        apdu.sendBytes((short)0, (short)MNSIZE);
}

// Receives data from the client and checks for validity.
private final void receiveALSSReply( APDU apdu)
{
        // access authentication
        if ( ! pin.isValidated() )
          ISOException.throwIt(ISO.SW_PIN_REQUIRED);
        if ( StepOneTookPlace == false)
          ISOException.throwIt(SW_FIRST_STEP_ONE_THEN_VERIFY);
        // We need to read from the terminal the reply to LFDM_StepOne
        var = (byte)(apdu.setIncomingAndReceive());
        if ( var != (byte )REPLYSIZE )
          ISOException.throwIt(SW_INVALID_SIZE);

        ComputeMAC_or_Hash( ReadLogFile_FILE, machash, WANT_HASH );

        // Verify hash send by the ALSS
        // The first 16 bytes of the information send, are the mac to check.
        if ( Util.arrayCompare( machash, (short )0, buffer, (short )ISO.OFFSET_CDATA,
          (short )16 ) != 0
          ||
          Util.arrayCompare( buffer, (short )((short )ISO.OFFSET_CDATA + (short )16),
          LogFileRec_STR, (short )0, (short )1 ) != 0 )
        {
          // This is the message for unsuccesful verification.
          ISOException.throwIt(SW_NOT_VERIFIED);
        }
        else
        {
          // We need more time on this!
          apdu.waitExtension();
          // This is the Commit Value Section where all the values
          // (Nc, CBind, ) are updated in the corresponding files.
          // This may be ugly...
          if ( SequenceNumber_FILE[ 0 ] < (byte )0xff )
```

```
            SequenceNumber_FILE[ 0 ]++;
        else
            if ( SequenceNumber_FILE[ 1 ] < (byte )0xff )
            {
              SequenceNumber_FILE[ 0 ] = 0;
              SequenceNumber_FILE[ 1 ]++;
            }
            else
              if ( SequenceNumber_FILE[ 3 ] < (byte )0xff )
              {
                    SequenceNumber_FILE[ 0 ] = 0;
                    SequenceNumber_FILE[ 1 ] = 0;
                    SequenceNumber_FILE[ 2 ]++;
              }
              else
              {
                    SequenceNumber_FILE[ 0 ] = 0;
                    SequenceNumber_FILE[ 1 ] = 0;
                    SequenceNumber_FILE[ 2 ] = 0;
                    SequenceNumber_FILE[ 3 ]++;
              }
        // Write next sequence number to file.
        writeLogFile( SequenceNumber_FILE, SeqNumFile );
        // Compute the MAC of Mn.
        ComputeMAC_or_Hash( Mn, machash, WANT_HASH );
        // Flush, that is, write file to disk!
        Util.arrayCopy( machash, (short )0, CBind_FILE, (short )0, (short )16 );
        // Write CBind to the CBind file.
        writeLogFile( CBind_FILE, CBindFile );
        // We retain a copy of Ts for possible future processing.
        Util.arrayCopy( buffer, (short )((short )ISO.OFFSET_CDATA +
              (short )16), Ts, (short )0, (short )8 );
        Util.arrayFillNonAtomic( ReadLogFile_FILE, (byte )0);
        writeLogFile( ReadLogFile_FILE, LogFile );
      }
}

private final boolean readLogFile( byte[] bufferFile, short filename )
{
      FileSystem.selectRoot();

      if ( FileSystem.selectFile( LogDirectory ) != ST.SUCCESS )
        ISOException.throwIt(SW_COULD_NOT_STATE_FILE);
      if ( FileSystem.selectFile( filename ) != ST.SUCCESS )
        ISOException.throwIt(SW_COULD_NOT_STATE_FILE);

      if ( bufferFile.length > (short )FileSystem.getFileSize() )
        ISOException.throwIt(SW_FILE_SIZE_DISCREPANCIES);
      if ( CyberflexOS.readBinaryFile(bufferFile, (short )0, (short )0,
        (short )bufferFile.length) != ST.SUCCESS )
        ISOException.throwIt(SW_COULD_NOT_STATE_FILE);
      return true;
}

private final boolean writeLogFile( byte[] bufferFile, short filename )
{
      FileSystem.selectRoot();
      if ( FileSystem.selectFile( LogDirectory ) != ST.SUCCESS )
        ISOException.throwIt(SW_COULD_NOT_STATE_FILE);
```

```
        if ( FileSystem.selectFile( filename ) != ST.SUCCESS )
          ISOException.throwIt(SW_COULD_NOT_STATE_FILE);

        if ( bufferFile.length > (short )FileSystem.getFileSize() )
          ISOException.throwIt(SW_FILE_SIZE_DISCREPANCIES);
        if ( CyberflexOS.writeBinaryFile(bufferFile, (short )0, (short )0,
          (short )bufferFile.length) != ST.SUCCESS )
          ISOException.throwIt(SW_COULD_NOT_STATE_FILE);
        return true;
    }
}
```

A.4 The GemXpresso Standard Log File Format Code Listings

In this section we provide the code listing from the Standard Log File Format implementation on the GemXpresso Smart Card.

A.4.1 The Interface of the Log File Standard Format

```
//-----------------------------------------------------------------------
//      File Details    : ilfdm.java, 1999/7/16, 13:25, File Size: 987 bytes
//      Description     : The interface for the Standard Log File Format Smart card
//                        application.
//
//      Author   :      Constantinos Markantonakis
//      Date            : 16/3/1999
//
//-----------------------------------------------------------------------

package PhD.Standard.StandardFormat ;

import javacard.framework.* ;
import javacardx.framework.* ;
import com.gemplus.gemxpresso.library.util.IApplet ;
import com.gemplus.gemxpresso.library.iso.ISOFileSystem ;
import com.gemplus.gemxpresso.library.util.SimpleOwnerPIN ;
import com.gemplus.gemxpresso.library.standard.Constants1;
import com.gemplus.gemxpresso.library.standard.Core1;

public interface ILFDM extends IApplet
{
  /**
   * This function will create a Data Entry entry in the log file
   */
  public byte Get_Information(byte[] Information, byte How_Many);
}
```

A.4.2 The Log File Standard Format Application

```
//-----------------------------------------------------------------------
//      File Details    : lfdm.java, 1999/7/16, 16:47, File Size: 1797 bytes
//      Description     : This smart card appletion calls Log File Standard
//                        format from the core library.
//
//      Author  :       Constantinos Markantonakis
//      Date            : 16/3/1999
//
//-----------------------------------------------------------------------

package PhD.Standard.StandardFormat ;

import javacard.framework.* ;
import javacardx.framework.* ;
import com.gemplus.gemxpresso.library.iso.ISOFileSystem ;
import com.gemplus.gemxpresso.library.util.SimpleOwnerPIN ;
import com.gemplus.gemxpresso.library.standard.Constants1;
import com.gemplus.gemxpresso.library.standard.Core1;

public class LFDM extends Applet
  implements ILFDM
{
  /**
  * The Aythorization for enabling the create the DE
  */
  private SimpleOwnerPIN thePIN ;

  /**
  *       Access to the Core package
  */
  private Core1 theCore;

  int Incoming_Packet_No=0;
  // This variable controls the packet number sent by the Card
  int Outgoing_Packet_No=0;
  // Use in Initialize and Present_PIN procedures
  byte Result;

  /**
  * Constructs the Log File Standard Format (LFSF).
  */
  public LFDM(byte[] code, byte tryLimit)
        throws UserException
  {
        thePIN = new SimpleOwnerPIN(code, tryLimit) ;
        Core1 theCore = new Core1();
  }

  /**************************
  * Select the LFSF applet.
  **************************/
  public boolean select()
  {
        thePIN.reset() ;
        return true ;
  }
```

```
  /************************************************
  * This function create's the Data Entry.
  ************************************************/
  public byte Get_Information(byte[] Information, byte How_Many)
  {
        return theCore.GenerateDataEntry(Information, How_Many);
  }
}
```

A.4.3 The Constant Definitions for the Core Library

```
//————————————————————————————————————————————————————————————————————-
//      File Details    : constants1.java, 1999/7/16, 16:47, File Size: 976 bytes
//      Description     : This package simply define the shared constants that
//                        may be used by the core package.
//
//      Author   :      Constantinos Markantonakis
//      Date            : 16/3/1999
//
//————————————————————————————————————————————————————————————————————-

package com.gemplus.gemxpresso.library.standard;

/**
* This interface simply defines the constants that are shared by all the
* applications that may use the core package.
*/
public interface Constants1
{
    //
    // Status codes used in exceptions
    //
    public static final byte PROBLEM_INITIALIZING_FILE_SYSTEM = 0x30 ;
  public static final byte PROBLEM_INITIALIZING_FILES = 0x31 ;
  public static final byte PIN_IS_BLOCKED = 0x32 ;
  public static final byte PIN_NOT_VERIFIED_or_FILE_NOT_FULL = 0x33 ;
}
```

A.4.4 The Core Library Implementing the Log File Standard Format

```
//————————————————————————————————————————————————————————————————————-
//      File Details    : core1.java, 1999/7/16, 16:47, File Size: 8261 bytes
//      Description     : The core library containing all the functions that
//                        offer the standard log file format functionality.
//
//      Author   :      Constantinos Markantonakis
//      Date            : 25/4/1999
//
//————————————————————————————————————————————————————————————————————-

package com.gemplus.gemxpresso.library.standard;

import javacard.framework.* ;
import javacardx.framework.* ;
import com.gemplus.gemxpresso.library.iso.ISOFileSystem ;
import com.gemplus.gemxpresso.library.util.*;

public final class Core1 implements Constants1
{
  private static byte[] Data_Entry = new byte[141];
  private static boolean Already_Called = false;

  /**
  *The reference to the Files.
  */
  private static FileSystem MF ;
  private static DedicatedFile LogFile_DF ;
  private static TransparentFile LogFile ;
  private static TransparentFile SequenceNumber_File ;

  /**
  * This section holds the global variables
  */
  //This holds the Log file
  static byte [] buffer = new byte[1024];
  //An array index variable
  static int Index;

  //These variables are used in the initilaize log files variable
  static byte [] Temp_CBind = {0,0,0,10,0,0,0,8,0,0,0,5,0,0,0,11};
  static byte[] ID =          {0,1,2,3,4,5,6,7,8,9};
  static byte[] Seq =         {0,0,0,0};

  /**
  * This section holds the global variables
  */
  private static int Execution_Step = 0;

  /**
  * This is the basic constructor. The initializations are in fact taking place
  * later on in the following functions.
  */
  public Core1 ()
  {
  }
```

```
/**
*     The following section is added by me I do not know if it is working
*     properly.
*/
public boolean select()
{
      MF.reset() ;
      return true ;
}

/*********************************************************
* Compute a MAC or a HASH
*********************************************************/
public static byte[] ComputeMAC_or_Hash( byte[] inBuffer, int wantMACorHash )
{
      byte[] outBuffer = new byte[16];
      outBuffer[ 0 ] = inBuffer[ 0 ];
      outBuffer[ 1 ] = inBuffer[ 1 ];
      outBuffer[ 2 ] = inBuffer[ 2 ];
      outBuffer[ 3 ] = inBuffer[ 3 ];

      int counterB = 0;

      byte [] CA_Key = {2,8,9,5};
      if ( wantMACorHash == 1 )
      {
        outBuffer[0] = (byte )((byte )outBuffer[0] ^ (byte )CA_Key[0]);
        outBuffer[1] = (byte )((byte )outBuffer[1] ^ (byte )CA_Key[1]);
        outBuffer[2] = (byte )((byte )outBuffer[2] ^ (byte )CA_Key[2]);
        outBuffer[3] = (byte )((byte )outBuffer[3] ^ (byte )CA_Key[3]);
      }

      // Some major optimization. :)
      counterB = 0;
      int counter =0;

      for ( counter = 0; counter < 4 - 1; counter++ )
      {
        counterB = (short )(counterB + 4);
        Util.arrayCopy( outBuffer, (short )0, outBuffer,
              (short)counterB, (short )4 );
      }

      // outBuffer has the result.
      return outBuffer;
}

/*******************************
* Get the sequence number
*******************************/
private static byte[] Get_Nc()
{
      return SequenceNumber_File.getData();
}

/*******************************
* Get a Timestamp
*******************************/
private static byte[] Get_Tc()
```

```
{
      byte[] Timestamp = {2,5,0,5,1,9,7,3};
      return Timestamp;
}

/*******************************
* Retrieve the Application ID
*******************************/
private static byte[] Get_AID()
{
      byte[] Application_AID = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
      return Application_AID;
}

/*****************************************************************************
* Check if the calling application ahs already called the GENERATE_DE function
*****************************************************************************/
private static boolean Check_ifalready_Called()
{
      boolean Called = false ;

      if (Already_Called == true)
      {
        //Indicate that the command has already been called once from within this
        // application.
        Called = true;
      }

      return Called;
}

/*********************************************************
* Update the Data_Entry in the log file
*********************************************************/
private static byte Write_DataEntry(byte[] Data_In)
{
      short Copy_in_Position =0;
      byte Write_Status = 0;
      int Sequence_in_Int = Util32.getInt(SequenceNumber_File.getData(), (short)0);
      int Sequence_Number_Value;

      Sequence_Number_Value = Sequence_in_Int;
      Sequence_in_Int = (Sequence_in_Int % 7);

      Copy_in_Position = (short)(Sequence_in_Int *141);

      Util.arrayCopy( Data_In, (short)0, LogFile.getData(), (short)Copy_in_Position,
                      (short)141);

      //Check in order to raise the flag that the log file is full
      if (Sequence_in_Int==6)
      {
        LogFile_is_Full= true;
      }

      //Increment the seqeunce numebr and write the new value to the file
      Sequence_Number_Value = Sequence_Number_Value + 1;
      short Value_To_Write = Util32.setInt( SequenceNumber_File.getData(), (short)0,
                                            (int)Sequence_Number_Value);
```

```
        /// A T T E N T I O N:
        // FOR MULTIPLE CECKS UNCOMMENT THE FOLLOWING LINE
        //      Already_Called = true;

        return (byte)Sequence_in_Int;
}

/*********************************************************************
* This function will create an entry in format required by the standard
*********************************************************************/
// Convert to private
public static byte FormatDataEntry(byte[] LE, byte Number_of_Events)
{
        byte Creating_Status = 0;

        //Temporary Value to check the status of an application
        boolean Get_Application_Priviledges = true;

        byte[] DeLink = new byte[16];

        short Counter=0;
        int Inner_Counter = 5 ;
        int Semicolumns_Found = 0;

        if (LE.length>89)             //Make sure that some entries are present
        {
          Creating_Status = 2;
        }
        else if (Get_Application_Priviledges == false) //if application is authorised
                                                       //to add an entry in the log file
        {
          Creating_Status = 3;
        }
        else
        {
          if (Number_of_Events!=0)
          {
                while (((LE.length/5)-1)>Counter)
                {
                  if (LE[Inner_Counter]==0x3B)
                  {
                        Semicolumns_Found = Semicolumns_Found +1;
                  }
                  Inner_Counter=Inner_Counter+6;
                  Counter++;
                }
          }
          //Make sure that the application provided the number of entries
          if (Number_of_Events == Semicolumns_Found)
          {
                //Construct the first half of the DE
                Data_Entry[0]=83;
                Data_Entry[1]=59;
                Util.arrayCopy( Get_AID(), (short)(0), Data_Entry, (short)2,(short)16);
                Data_Entry[18]=59;
                Util.arrayCopy( Get_Tc(), (short)(0), Data_Entry, (short)19,(short)8);
                Data_Entry[27]=59;
                Util.arrayCopy( Get_Nc(), (short)(0), Data_Entry, (short)28,(short)4);
```

```
            Data_Entry[32]=59;

            //Add the DATA entry of the LEs
            Util.arrayCopy( LE, (short)(0), Data_Entry, (short)33,(short)LE.length);

            int Array_Index=33+LE.length;
            Data_Entry[Array_Index]=59;
            //Compute the DeLink Variable
            Util.arrayCopy(ComputeMAC_or_Hash(LE, 1), (short)(0), Data_Entry,
                        (short)(Array_Index+1),(short)16);
            Data_Entry[Array_Index+16+1]=59;
            Data_Entry[Array_Index+16+2]=69;

            //Indicate that everything is OK
            Creating_Status = 1;
        }
        else
            //The number of semicolumns found, doesn't match with the number provided
            Creating_Status=8;
      }
      // Return the value for the command status
      return Creating_Status;
  }

  /*****************************************************************************
  * This function is the interface to outside world, appedning the DE in the LF
  *****************************************************************************/
  public static byte GenerateDataEntry(byte[] LE, byte Number_of_Events)
  {
      byte Status_Value=0;

      //Check if function has already been called
      if (Check_ifalready_Called() == true)
      {
        //Indicate that the command has already been called once from within this
        // application.
        Status_Value = 5;
      }
      else
      {
        //Create the format required by the standard
        byte Format_Status = FormatDataEntry(LE, Number_of_Events);
        if (Format_Status==1)
        {
            //*** UPDATE THE ENTRY IN THE LOG FILE
            //Status_Value = (byte)100;
            Status_Value = Write_DataEntry(Data_Entry);
        }
        else
            Status_Value = Format_Status;
      }
      return Status_Value ;
  }
}
```

Bibliography

[1] Ross Anderson. UEPS — A Second Generation Electronic Wallet. In *Springer Verlag Lecture Notes in Computer Science*, volume 648, pages 411–418. ESORICS 92, 1992.

[2] Ross Anderson and Roger Needham. Robustness Principles for Public Key Protocols. In *Springer Verlag Lecture Notes in Computer Science*, pages 236–247. Advances in Cryptology - CRYPTO 95, 1995.

[3] Mihir Bellare and Bennet Yee. Forward Integrity for Secure Audit Logs. Technical report, Computer Science and Engineering Department, University of California at San Diego, November 1997.

[4] Matt Bishop. A Standard Audit Log Format. In *Proceedings of the 19th National Information Systems Security Conference*, pages 136–145, 1995. Also available in `http://seclab.cs.ucdavis.edu/~bishop/scriv/index.html`.

[5] Ruth Cherneff, John Griffin, Dave Outcalt, Dr. Carmen Pufialito, Rhonda Kaplan Singer, and Michelle Stapleton. Smart cards 97. `http://www1.shore.net/bauster/cap/s-card/index.html`, November 1996.

[6] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company Inc., 1994.

[7] Cyberflex. *Cyberflex Open16K Reference Manual.* Schlumberger, 1998.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Pattern — Elements of Reusable Object — Oriented Software.* Addison-Wesley Publishing Company Reading MA, 1994.

[9] Gemplus. *Multi-Application Chip Operating System (MCOS) Reference Manual Ver 2.2.* Gemplus, 1990.

[10] Gemplus. *Multi-application Payment Chip Operating System (MPCOS) 16K EEPROM DES Reference Manual.* Gemplus, 1994.

[11] Gemplus. The First 32-bit RISC Processor Ever Embedded in a Smart Card. `http://www.gemplus.fr/presse/cascade2uk.htm`, 1996.

[12] Gemplus. *GemXpresso Reference Manual.* Gemplus, 1998.

[13] General Information Systems Ltd. (GIS). Specification of a Smart Card Filling System Incorporating Data Security and Message Authentication. `http://www.gis.co.uk/oscman1.htm`, 1997.

[14] UCL Crypto Group. CASCADE, a Smarter Chip for Smart Cards. `http://www.dice.ucl.ac.be.crypto/cascade`, 1996.

[15] Helena Handschuch and Pascal Paillier. Smart Card Cryptoprocessors for Public Key Cryptography. In *Springer Verlag.* Third Smart Card Research and Advanced Application Conference - CARDIS'98, September 1998. to be published.

[16] Pieter Hartel and E. de Jong Frz. Smart Cards and Card Operating Systems. In *Conference Proceedings*, pages 725–730. Int. Conf. UNIFORUM'96, San-Francisco, California, February 1996.

[17] Hitachi. Hitachi 8bit Micro-controlers for Smart Cards. `http://www.halsp.hitachi.com/smartcard/index.html`, 1997.

[18] Mary Kirtland. The COM Programming Model Makes it Easy to Write Components in Any Language. Technical report, Microsoft Systems Journal, December 1997. `http://www.microsoft.com/msj/1297/complus2/complus2.htm`.

[19] MAOSCO. *MULTOS Reference Manual Ver 1.2.* MAOSCO, 1998.

[20] Constantinos Markantonakis. The Case for a Secure Multi-Application Smart Card Operating System. In *Lecture Notes in Computer Science*, volume 1396, pages 188–197. First International Information Security Workshop, ISW'97, Japan., September 1997.

[21] Constantinos Markantonakis. Secure Log File Download Mechanisms for Smart Cards. In *Lecture Notes in Computer Science*, volume to be published. Third Smart Card Research and Advanced Application Conference CARDIS'98, Louvain-la-Neuve, Belgium, September 1998.

[22] Constantinos Markantonakis. An architecture of Audit Logging in a Multi-application Smart card Environment. In *EICAR' 99 Conference Proceedings, ISBN: 87-98727-0-9.* EICAR'99 E-Commerce and New Media Managing Safety and Malware Challenges Effectively, Aalborg, Denmark, March 1999.

[23] Constantinos Markantonakis. Boundary Conditions that Influence Decisions about Log File Formats in Multi-application Smart Cards. In *Lecture Notes in Computer Science*, volume 1726, pages 230–243. The Second International Conference on Information and Communication Security (ICICS '99), Sydney, Australia, November 1999.

[24] Constantinos Markantonakis. Interfacing with Smart Card Applications (The PC/SC and OpenCard Framework). *Elsevier Information Security Technical Report*, 3(2):82–89, 1999.

[25] Constantinos Markantonakis. Java Card Technology and Security. *Elsevier Information Security Technical Report*, 4(2):69–77, 1999.

[26] Constantinos Markantonakis and Konstantinos Rantos. On the Life Cycle of the Certification Authority key pairs in EMV'96. In *Society for Computer Simulation International (SCS).* 4th EUROMEDIA Conference 1999, Munich, Germany, April 1999. ISBN: 1-56555-169-9.

[27] Constantinos Markantonakis and Simeon Xenitellis. Implementating a Secure Log File Download Manager for the Java Card. In *Secure Information Networks, Communication and Multimedia Security, Kluwer Academic Publishers*, volume 23, pages 143–159. CMS'99 Communications and Multimedia Security, Katholieke Universiteit Leuven, Belgium, September 1999.

[28] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, 1997.

[29] Sun Microsystems. Java Card 2.0 Language Subset and Virtual Machnine Specification. `http://www.javasoft.com/products/javacard/`, 1998.

[30] Sun Microsystems. Java Card 2.0 Programming Concepts. `http://www.javasoft.com/products/javacard/`, 1998.

[31] Sun Microsystems. The Java Card API Ver 2.0 Specification. `http://www.javasoft.com/products/javacard/`, 1998.

[32] Sun Microsystems. The Java Card API Ver 2.1 Specification. `http://www.javasoft.com/products/javacard/javacard21.html`, 1999.

[33] Motorola. M68HC05SC Family - At a Glance. `http://design-net.com/csic/SMARTCRD/sctable.htm`, 1997.

[34] M.U.S.C.L.E. Movement for the Use of Smart Cards in the Linux Environment. `http://www.linuxnet.com/index.html`, 1998.

[35] Ajitkumar Natarajan and Cjin Pheow Lee. An ARIES Log Manager for Minirel CS 764. `http://www.cs.ndsu.nodak.edu/~tat.minibase/logMgr/report/main.html`, 1994.

[36] National Computer Security Center (NCSC). A Guide to Understanding Audit in Trusted Systems. Technical report, Department of Defense (DoD), NCSC-TG-001, Library no. S-228-470, July 1987.

[37] U.S Department of Defence. *Trusted Computer System Evaluation Criteria.* Computer Security Centre, 1985.

[38] OpenCard Framework Specification (OCF). Opencard framework consortium. `www.opencard.org`, 1997.

[39] International Standard Organisation. *ISO 10202-1, Financial transaction cards — Security architecture of financial transaction systems using integrated circuit cards — Part 1: Card life cycle.* International Organization for Standardization, 1991.

[40] International Standard Organisation. *ISO/IEC 7816-5, Identification cards — Integrated Circuit(s) Cards with Contacts, Part 5, Numbering System and Registration Procedure for Application Identifiers.* International Organization for Standardization, 1994.

[41] International Standard Organisation. *ISO/IEC 7810, Identification cards — Physical characteristics.* International Organization for Standardization, 1995.

[42] International Standard Organisation. *ISO/IEC 7816-4, Information technology — Identification cards — Integrated circuits(s) cards with contacts — Inderindustry Commands for Interchange.* International Organization for Standardization, 1995.

[43] International Standard Organisation. *ISO/IEC 7816-2:1999, Information technology — Identification cards — Integrated circuit(s) cards with contacts — Part 2: Dimensions and location of the contacts.* International Organization for Standardization, 1996.

[44] International Standard Organisation. *ISO/IEC 7816-6, Identification cards — Integrated Circuit(s) Cards with Contacts, Part 6, Inter-industry data elements.* International Organization for Standardization, 1996.

[45] International Standard Organisation. *ISO/IEC 7816-3, Information technology — Identification cards — Integrated circuit(s) cards with contacts — Part 3: Electronic signals and transmission protocols.* International Organization for Standardization, 1997.

[46] International Standard Organisation. *ISO/IEC 9798-1, Information technology — Security Techniques — Entity Authentication — Part 1: General.* International Organization for Standardization, 1997.

[47] International Standard Organisation. *ISO/IEC 7816-1, Identification cards — Integrated circuit(s) cards with contacts — Part 1: Physical characteristics.* International Organization for Standardization, 1998.

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

[49] International Standard Organisation. *ISO/IEC DIS 7816-8, Identification cards — Integrated circuit(s) cards with contacts — Part 8: Security related interindustry commands.* International Organization for Standardization, 1999.

[50] Java Card Forum Participants. Java Card Forum. `http://www.javacardforum.org/`, April 1997.

[51] Vikram Pesati, Thomas Keefe, and Shankar Pal. The Design and Implementation of a Multilevel Secure Log Manager. In *IEEE Conference Proceedings*, pages 55–64. 1997 IEEE Symposium on Security and Privacy, Oakland, California, May 1997.

[52] Patrice Peyret. Application-Enabling Card Systems with Plug-and-Play Applets. In *Smart Card'96 Conference Proceedings.* 9th Annual International Advanced Card Exhibition and Convention, February 1996.

[53] Pierre Paradinas and Jean-Jacques Vandewalle. New Directions for Integrated Circuit Cards Operating Systems. *Operating Systems Review*, 29(1):56–61, 1995.

[54] Jean-Marie Place, Thierry Peltier, and Patrick Trane. Secured Co-operation of Partners and Applications in the Blank Card. In *GDM-Darmstadt 95 - Struif Editors*, July 1995.

[55] Schlumberger. *Cyberflex Smart card Series Developers manual.* Schlumberger, 1997.

[56] Schlumberger. Cyberflex Open 16K. `http://www.cyberflex.austin.et.slb.com`, October 1998.

[57] Bruce Schneier. *Applied Cryptography.* John Wiley and Sons, Second edition, 1996.

[58] Bruce Schneier and John Kelsey. Automatic Event - Stream Notarization Using Digital Signatures. In *Springer Verlag*, pages 155–169. Security Protocols, International Workshop, April 1996.

[59] Bruce Schneier and John Kelsey. Cryptographic Support for Secure Logs on Untrusted Machines. In *The Seventh USENIX Security Symposium Proceedings*, pages 53–62. Usenix Press, January 1998.

[60] Adam Shostack. SSL 3.0 SPECIFICATION. `http://www.homeport.org/adam/ssl.html`, May 1995.

[61] SIEMENS. STARCOS. `http://www.gdm.de/index.htm`, 1996.

[62] SIEMENS. CardOS. `http://www.ad.siemens.de/cardos/index76.htm`, September 1997.

[63] Stuart Haber and W. Scott Stornetta. How To Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2):99–111, 1996.

[64] Patrick Trane and Sylvain Lecomte. Failure Recovery Using Action Logs for Smart Cards Transactions Based Systems. Presented at the Third IEEE International On-Line Testing Workshop, July 1997.

[65] Jean-Jacques Vandewalle and Eric Vetillard. Developing Smart Card Based Applications Using Java Card. In *Springer Verlag.* Third Smart Card Research and Advanced Application Conference - CARDIS'98, September 1998. to be published.

[66] PC/SC Workgroup. Specifications for PC-ICC Interoperability. `www.smartcardsys.com`, 1996.