Development of an Operating System for Smart Cards

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
Ravinder Shankesi

to the
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

May, 2002

Certificate

This is to certify that the work contained in the thesis entitled " *Development of an Operating System for Smart Cards* ", by *Ravinder Shankesi*, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

May, 2002

(Dr. Deepak Gupta)	(Dr. Rajat Moona)
Department of Computer Science & Engineering,	Department of Computer Science & Engineering,
Indian Institute of Technology,	Indian Institute of Technology,
Kanpur.	Kanpur.

Abstract

Smart cards are increasingly getting used in various identification and verification applications. Traditionally, smart card operating systems were proprietary and application specific. This made it very difficult to use the same operating system for different applications or to use two cards with different operating systems for the same application.

In this thesis we describe the design and implementation of a standard (compliant to ISO 7816 standard and SCOSTA standard) application-independent smart card operating system, Card Operating System. We have developed a linux port for testing the operating system. We have ported the operating system to a few existing smart card chips.

Acknowledgements

I take this opportunity to express my sincere gratitude and thanks to both of my supervisors Dr. Deepak Gupta and Dr. Rajat Moona. But for their patience and guidance, I would never have completed the project. Working under them was a great learning experience. I would also like to thank Dr. Manindra Agarwal who has helped us immensly during various phases of our project.

I would like to thank MoST (Ministry of Surface Transport), members of SCAFI (Smart Card Association for India) and NIC (National Informatics Center) New Delhi, for their assistance at various phases of the project. In particular, I would like to thank Dr. B.K.Gairola of NIC for his support and guidance to the project.

I would like to thank Ankit Jalote and Marghoob Mohiyuddin who have implemented the security module of the Operating System. I would also like to thank Kapileshwar Rao Bolisetti who has ported the Operating System to Linux.

My batch here was really fun to be with. I would like to thank all of them for their help, for the irrelevant spam, for all the parties, for all the movies and for the Khajuraho trip. They had to put up with my laziness, absentmindedness and slight absurdities at various occassions and yet were very helpful and generous towards me. It was great being with all of you.

At last, I would like to thank members of my family who were there to support me at all times.

Contents

List of Figures

Chapter 1

Introduction

Smart cards are cards with electronic chips embedded inside them. They are used in different applications like identification, loyalty programs, SIM cards inside GSM phones etc., Most of the modern smart cards have a micro-controller, non-volatile memory for storing user data, ROM for storing the operating system code and RAM for use by the operating system. Some cards also have an extra crypto-co processor for improving the speed-up of cryptographic algorithms used by the operating system. Most smart cards have 8-bit micro controllers with a ROM size of 4K to 32K, RAM size of 256 bytes to 1K and an EEPROM size of 4K to 8K. However, newer cards with 16-bit and 32-bit micro-controllers with better configurations are beginning to appear in the market [1].

It is obvious that in such a resource-constrained environment, the operating system on the card must be developed to utilise these resources carefully. In this report we describe our implementation of a portable Card Operating System compliant to SCOSTA standard [2]. The ports of this card to a few architectures are also described.

1.1 History

Smart cards were intended as a solution to the short-comings of magnetic cards. These short-comings include, small memory (around 220 bytes in 3 rows of magnetic stripe), risk of tampering, lack of any processing logic. The earliest patent for smart cards was filed as early as 1968 by two German inventors, Jurgen Dethloff and Helmut Grotruppi [3], . Similar patents were filed by various other people later [4], [5]. Smart cards started getting used more prevalently in Europe in the 1980s. For instance, the French PTT(Postal and Telecommunications services) started circulating millions of smart cards by 1986 after a successful field trial in 1985. Currently smart cards are widely employed in fields other than identification like, payment cards, loyalty applications, health card applications etc., Certain innovative uses of smart cards include Gaming for smart cards[6], Smart Flash Card [7].

1.2 Introduction

1.2.1 Different classifications of smart cards

Smart cards are categorised in different ways depending on different criteria.

- **Access Mechanism**: Depending on the access mechanism smart cards can be classified as contact-based cards and contact-less cards. Contact-based cards are physically connected to the terminal for communication to take place. Contact less cards have communication with the terminal using radio frequencies. Some cards have both these access mechanisms embedded into them.

- **Functional grouping**: Depending on the functionality provided smart cards can be classified as memory cards or micro-processor cards. Memory cards have only memory present inside the card. This might be read-only or read-write. Some times they have certain extra logic for security like write-once protection for memory. Micro-processor cards have a micro-processor embedded into the card along with ROM, RAM and some non volatile memory.

- **Applications present**: Depending on the applications present inside the card they are classifed as single-application or multi-application cards. Single-application cards have only a single application present inside the card. Multi-application cards have more than one application supported inside the card.

- **Physical shape**: Normal smart cards are shaped like credit-cards. There are also SIM cards present inside GSM phones. There are also smart cards which are present inside a ring [9].

In the rest of the report we limit ourselves to micro-processor based contact smart cards.

1.2.2 Physical layout

The physical interface of the contact-based smart card is given in the ISO 7816 standards [10, 11].

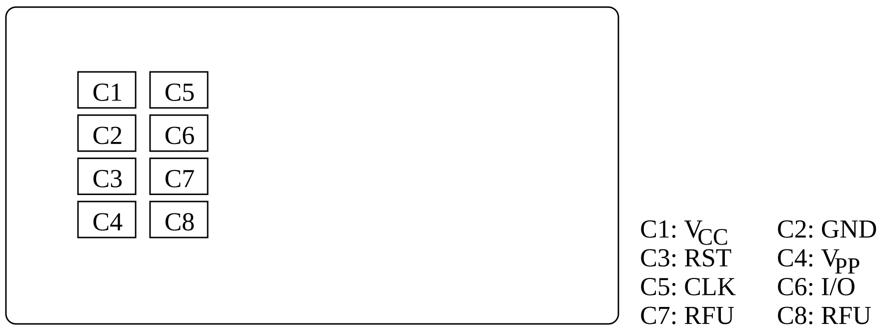

Figure 1.1: Physical interface of smart card

The Vcc pin is used to supply external voltage to the card. The Gnd pin is connected to the ground of the external terminal. The Vpp pin is used to supply programming voltage to the card. This was required because programming the EEPROM required higher voltage than Vcc. However, these days most cards ignore this pin and generate programming voltage internally. The I/O pin is used for I/O communication. This means that communication between the smart card and the terminal is always half-duplex. The RST pin is used by the terminal to send the Reset signal to the card.

1.2.3 Standards

There are many standards describing various aspects of smart cards. There are standards which are relevant to the behaviour of the card and the commands it must support. There are also standards for standardizing the access to the card. Apart from these we also have standards relevant for particular industries.

The most basic of these standards are the ISO 7816 group of stanadards. There are 10 of them ISO 7816-1 to ISO 7816-10. These describe various things from physical andcharachteristics of the card [10, 11] to the application commands [13, 18, 19] to be used by application developers. There are also many standards for the SIM cards used in GSM phones [20, 21].

Europay, Mastercard, Visa came up with a standard for smart cards for payment systems called EMV [32].

The standards for writing terminal side applications to transparently access any reader include PC/SC standard [29] and the Open Card Framework [30]. Currently implementations of PC/SC standard are mostly limited to the windows platform, although porting of the PC/SC standard for linux is also under way [31]. The current implementations of Open Card Framework are implemented in Java.

A part from this there are also standards for implementing the smart card operating system like Java Card [25] and MULTOS [26]

IIT Kanpur and a technical sub-committee of SCAFI (Smart Card Forum of India) together have come up with a standard for the development of smart card operating systems which is compliant with the ISO 7816 group of standards called, Smart Card Operating System for Transport Applications (SCOSTA) standard [2]. The Card Operating System described in this report is compliant to this standard.

1.2.4 Basic concepts

▮ *ATR*

Upon reset the the card returns a string to the terminal which indicates to the terminal the transmission protocols it supports and the protocol parameters. This string is called the Answer to Reset, ATR. The terminal can decide which protocol to use, if multiple protocols are offered, by doing a Protocol Parameter Selection PPS [12].

▮ *File System*

The file system in a Smart Card is organized as follows [13] The file are organized

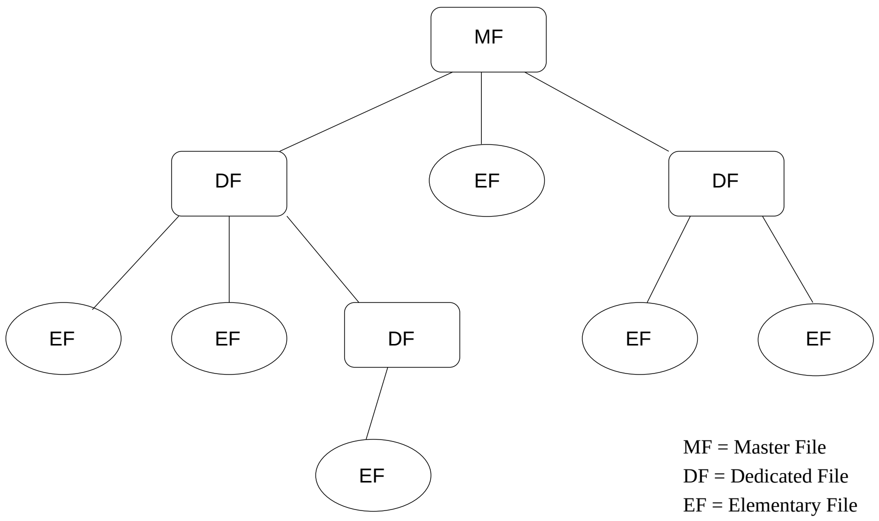

Figure 1.2: File System on the Card

into Dedicated Files (DFs) and Elementary Files(EFs). DFs are those files which contain other DFs or EFs. EFs contain the actual application data that the user want to store. Apart from Data Objects, at any DF we can also store Data Objects which can be accessed by its Tag.

▮ *Types of Files*

EFs can be of many different types.

- Transparent
- Linear Fixed Length Record File
- Linear Variable Length Record File
- Cyclic Record File

A transparent file contains set of data units which can be accessed by their offset. The offset of a the first data unit in a file is 0.

A record-oriented file contains data stored as records. These records can be accessed by their record numbers (starting from 01 to 254).

A linear record-oriented file contains records stored in the order of their creation. Thus the most recently appended record is the last record of the file. Thus the first written record is record number 1.

A variable length record-oriented file allows records to be of variable length until the length is less than the max record length.

In a cyclic record file, the records are accessed in the opposite order of creation. Thus the most recently written record overwrites the last record of the file an becomes record number 1.

▮ *Security Architecture*

The security of the smart card can be guaranteed by various mechanisms. The user might be required to prove the knowledge of a key or a password or possible both for some commands to be executed. If the user proves the knowlege of a key or a password, that status (security status) is maintained until the user changes the directory or after reset. Any file may be protected against certain commands by giving some security attributes in the file at the time of creation of the file for that command. These attributes might specify, for instance, that the user has to perform External Authentication with Security Environment number 4.

A Security Environment (SE) is a set of templates specifying what conditions need to be satisfied for certain operations to take place. For instance, it might specify that we need to external authenticate should be done with key number 5 or user authenticate should be done with password number 6. At any point during the working of a program there is always a current SE.

1.3 Related Work

There are many implmentations of smart card operating systems. Many of the smart card operating systems are built for single applications and are customized for that application. These have a fixed files system and a few commands which are relevant to that application. Most of the time they have proprietary commands for their operation [22, 23]. The TCOS operating system is a ISO 7816 compliant operating system which supports multiple applications [24].

The above operating systems have a fixed command set which is burned into the ROM. As against these a number of implementations are present which allow the user to program the card according to his need. The BasicCard from Zeitcontrol [33] allows users to program applications in Basic and download them into the card. Keycord once marketed a smart card called OSSCA (Operating System for Smart Card Applications) which was programmable in Forth. A few vendors have implemented the Java Card standard which would be programmable using the Java language [27, 28]

The cost of such generality is the extra processing capability required on the card for implementing the Virtual Machines for those languages. These cards are typically 3 or more times costlier than the ordinary cards.

The Card Operating System described in this report is a fixed command set operating system which is not application specific. Thus it is similar to the TCOS operating system.

Chapter 2

System Design

In this chapter we describe the various issues involved in the design of the Card Operating System. The physical and logical organization of the files in EEPROM are explained. We also describe the different modules and their interaction with one another.

2.1 Design goals

The following were the design goals while designing the Card Operating System.

- **Portability:**The Card Operating System must be portable to different architectures. This means that the operating system must be designed to use a small interface for handling the hardware dependent support required from the procoessor.

- **Compactness:**The Card Operating System must fit into a small size. This means perferring simpler algorithms over efficient, but complicated algorithms.

- **Maintainability:**The Card Operating System must be easily maintainbale. This involves the use of modular programming techniques.

2.2 File System layout

The file system stored in the EEPROM has the following properties:

The representation of any file in the EEPROM has the following 2 components.

1. A header containing all the meta-data of the file.
2. An optional body containing the data stored in the file. The data portion is not present in the case of Dedicated Files.

The header will contain all the meta-data required for the file system structure that we are maintaining and the File Control Parameters.

2.2.1 Logical structure of the File system

The File System maintaining by the Card Operating System has the following logical layout.

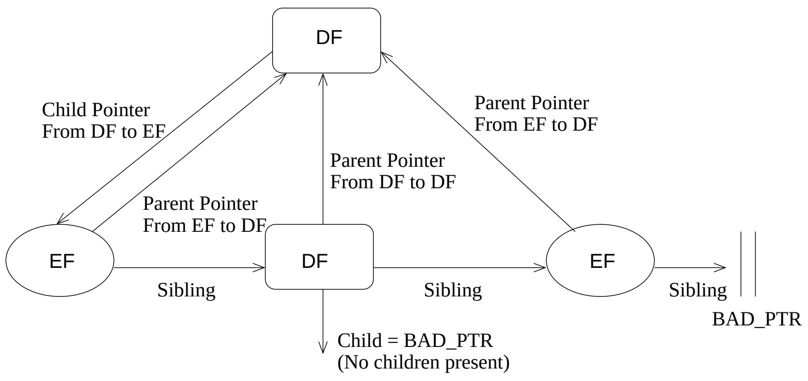
DF
Child Pointer From DF to EF
Parent Pointer From EF to DF
Parent Pointer From DF to DF
Parent Pointer From EF to DF
EF
DF
EF
Sibling
Sibling
Sibling
BAD_PTR
Child = BAD_PTR (No children present)

Figure 2.1: Logical layout of the file system in the EEPROM

Every file has a link to the next file (sibling) present in the same directory. Also each file has a link to the parent DF. The master file's parent file link points to itself.

In addition every DF will have a pointer to the first child (if present). Thus given a DF, we can access all the children (other DFs/EFs) by first looking at its child pointer and then looking at all the siblings of the child (if present). In case no child is present, or no sibling is present we store a constant in its place, BAD_PTR

2.2.2 Physical layout of the file system

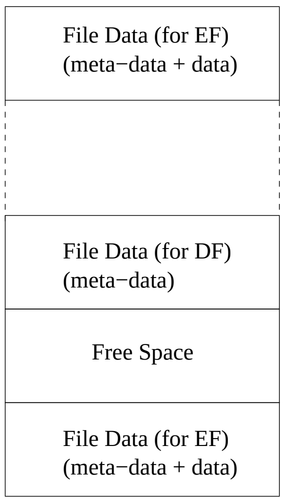

Figure 2.2: physical layout of the file system in the EEPROM

The File system comprises of individual files, stored as blocks (of variable length), laid out next to each other in the EEPROM. The files as part of their meta-data contain the total length of the file (including the header). Thus we can traverse the entire EEPROM to check out the files in the order they are stored in the EEPROM. This can be useful, for instance, when we are searching for a file with a given DF Name in the entire file system. It is also possible that we might have free space in between two files. We indicate that the current block is free (by the first byte) and the length of free space (2 bytes).

2.2.3 Meta Data and File Control Parameters

Each file contains the following meta-data stored in the header of the file.

1. The FDB is the first byte. Its value is equal to the file descriptor byte of the file. It also stores whether a the current block is free by storing a value which is an invalid FDB.

2. The length (the next 2 bytes). Its value is equal to the total length of the file (meta-data and data of the file).

3. Every file contains the link to its sibling and parent.

Apart from the meta-data for maintaining the logical structure, we also have File Control Parameters like File Identifier, Life Cycle Status Integer and Security-Attributes (variable length) at fixed offsets from the file starting. The security attributes are stored in their TLV format as given in the ISO 7816-9 documentation [19].

Depending on the file descriptor byte, the file specific meta-data follows.

- For a DF, we have a pointer to the first child of this DF, File Id of a file containing SE templates (Invalid File Id, if none is specified in the FCP) DF Name.

- For an EF, we have the Short File Identifier, Data Coding byte. Depending on the type of file we have further meta-data like, for a transparent file, the total no of bytes. No of records, Max length of records in case of a Record based file. If the file is write once, we also have a bit-vector indicating whether the given byte (in case of transparent file) or given record (in case of a record oriented file) is written to or not.

Following the meta-data for a DF, we have block of the next file. In case of an EF, we have the data for the given file. This would be equal to total number of bytes (incase of transparent file), Max.Record Length * Max Number of Record (incase of record based file) of data.

2.3 Basic Architecture

The following are the major components of the operating system.

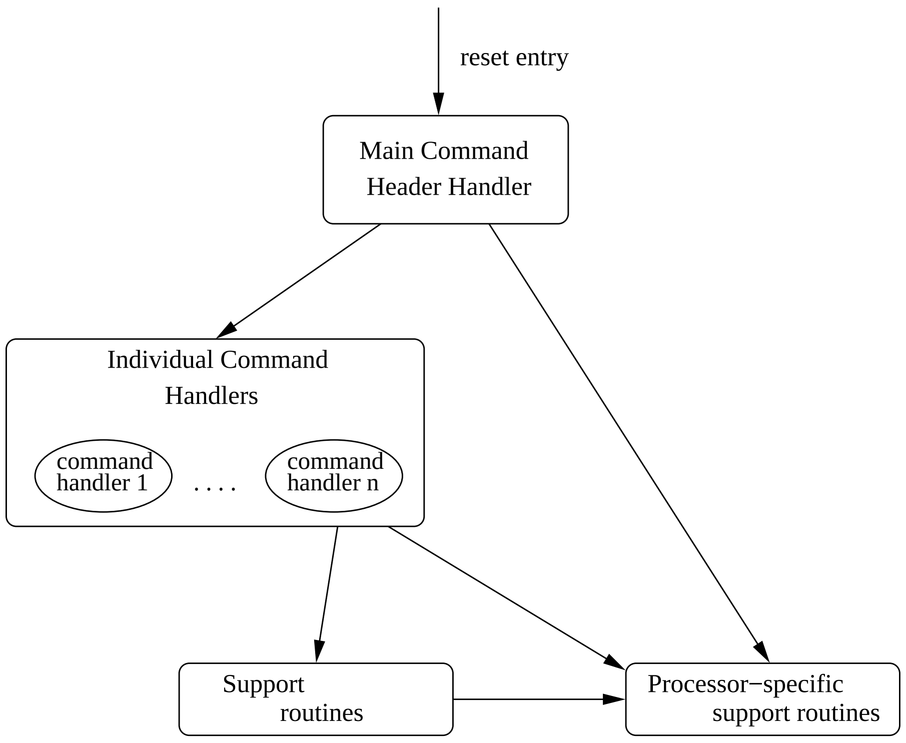
reset entry
Main Command Header Handler
Individual Command Handlers
command handler 1
. . . .
command handler n
Support routines
Processor−specific support routines

Figure 2.3: Major components of the Card Operating System

- **Main Command Header Handler:** The main command handler receives the control when the Reset is sent to the card. It sends the ATR to the terminal and starts receiving the individual commands. After receiving a command , it calls the appropriate command handler in the module Command Handlers. After returning from the individual Command Handler, the response, if any, of the Command Handler along with the status bytes is sent to the terminal.

- **Individual Command Handlers:** This module contains the individual command handlers for handling the individual commands. Some command handlers handle multiple commands.

- **Support Module:** This module contains the support routines used to access the data of the file system. These routines include functions to access the File Control Parameters of any file and routines to access the individual data of the files. These routines make the internal layout of the file system transparent to other entities, like, the command handlers. Thus we can later change the internal layout of the file system with out changing these routines.

- **Processor specific support module:** This module contains processor specific support for handling the architecture specific routines. These routines include terminal I/O, EEPROM reading and writing, timing, random number generation and accessing the chip serial number.

Chapter 3

Implementation

In this chapter we give a brief description of the software developed to implement the design proposed in the previous chapter. The software implements a SCOSTA compliant smart card operating system which is largely architecture independent. We describe the various modules present and give details about their implementation. We also describe the existing ports of the operating system for different architectures.

3.1 Main Command Header Handler

After reset, the first thing performed by the Card OS is to initialize the variables that it requires. This consists of both hardware dependent and hardware independent initialization. The hardware independent initialization consists of setting the currentFile, currentDF to point to Master File and clearing the security status. Then a function, initparam() is called which does the hardware dependent initialization. These include, setting the appropriate values in I/O control registers, setting the timer register with appropriate values and any other processor specific initialization required.

After initialization is done, the card must return the ATR. We assume that the card has Master File and ATR file pre-present in the EEPROM. We read the ATR file and return the ATR string present in it. Currently the Card Operating System returns an ATR which is non-negotiable. This means that it does not support any protocol parameter selection as defined in ISO 7816-3 [12].

After sending the ATR, it checks the life cycle status of the Master File. If the Master File is terminated, it goes into an infinite loop and become unresponsive. Otherwise, it enters into the command header handling loop.

▮ *Command Header handling loop*

For every command, it first reads the command header of 5 bytes. If the class byte is wrong, or Instruction byte is not found in the command table then it returns the appropriate error. The command table contains Instruction byte, the function pointer of the function which handles that instruction and a field which tells whether the command needs input data and/or sends output data. If the length of input/output is greater than the maximum buffer length (which is compile time configurable) then the error Wrong Length is returned and we go back to beginning of the command-handling loop.

If the command, requires some input data to be read then an ACK (which is the same as INS byte) is sent back and all the input data is read and stored in a buffer (inputBuffer).

The values of sendLength (length of the output to be returned by the command) and the status bytes are initialized to their most common values (0 in case of sendLength, 90, 00 for status bytes).

The appropriate function, which handles this command, is called. Upon returning from the function it commits the changes made by the command handler to the EEPROM. The output, if any, to be sent is stored by the command handler function in a global buffer (sendBuffer, which is the same as inputBuffer). This output is sent to the terminal.

The status bytes are set by command handler in the global variables bSw1, bSw2. These are sent to the terminal and the control goes back to the starting of the command header handling loop.

3.2 Individual Command Handlers

All the command handlers written follow a certain protocol.

If the command handler function is handling an output command and it needs to send some output, it stores the output in a buffer (sendBuffer). Then it sets the value of sendLength to indicate the total number of bytes available for sending (excluding the status bytes).

If the command handler function is an input command then it already has the command data given in inputBuffer.

If the command handler function requires both input and output (Case 4 command), then it already has the input in inputBuffer when it is called. If it needs to send some output, it stores the output to be sent in a global buffer (storeBuffer for retrieval by a an immediately followed GetResponse command. It also indicates the length of data stored (in storeLength).

Every command-handler should set the value of the status bytes it needs to send in the global status variables (bSw1, bSw2), before returning. (Except, when the response is normal ending, ie.,when the status bytes are 0x90, 0x00). The command handler use the support routines for doing the file specific operations that it requires.

3.3 Support routines

The command-handlers use these support routines to access the file system data transparently to the command handler. This makes the command handler function immune to changes in the file system layout.

These include routines that can access the meta-data (required for maintaining the file system), file control parameters and the data of the file. For instance we have routines to access and modify the meta-data like the Total length of the file (header + data), the Sibling of the file, the parent of the file etc.

We also have routines to access/modify the file control parameters like File Descriptor byte, Data Coding Bytes, Short File Identifier, Security Attributes, DF Name, Max Record Length, Maximum Number of records File Id of the file containing SE template files.

We also have routines to access/modify the actual data like read/write given range of bytes (for a transparent file), read or write given record (for a record-oriented file), read or write the length of a given record (for variable length record-oriented file).

3.4 Security Architecture

The security architecture of the operating system was implemented by Ankit Jalote and Marghoob Mohiyuddin [34]. It is included here for completeness' sake.

The card at, any point, maintains the security status for every file in the path from the Master File to the currently selected file. The security status of the Master File is always present.

Every DF has a respective password and key file [2]. A maximum of 32 passwords/keys are possible for each depth. Thus each bit in the 4 bytes for password/key status represents a unique password/key. If Verify/External Authenticate succeeds, the corresponding password/key status bit is set indicating that the particular password/key has been authenticated.

When a directory is changed the Current Security Status is cleared on the path starting from the lowest common ancestor of the current and previous directory till the previous directory.

The current security status is used by VerifySE() function to tell which commands can be executed under the current security status. Thus before preforming the operating, the command handler calls this function to check if the security conditions corresponding to this command are satisfied.

In our implementation, we are handling only the Cryptographic Checksum Template (CCT), Confidentiality Template (CT) and the Authentication Template (AT).

The SEs can be stored as records (and accessed by their number) in the SE Template files in DFs or in the FCP of the current DF. SE is a concatenation of all the components (CRTs) present in the SE Template. The current SE (encoded in the variable currentSE) contains the SE as a concatenation of CRTs.

An SE is modified explicitly through the MANAGE SECURITY ENVIRONMENT (MSE) command (set, restore, erase, store SE). In case of 'set' in the MSE command, all the components (DOs) in the new value of the CRT specified in the data field, should already be present in the current SE. Furthermore, the lengths of the DOs in the data field should also match with the lengths of the corresponding DOs in the current SE. Only when these conditions are satisfied, the current SE will be changed. In the implementation of the MSE 'restore' command, we load the record with the matching SE number from the SE Template file in the current DF. MSE 'store' is similarly implemented by copying the current SE into a record in the SE Template file. MSE 'erase' results in the deletion of the record for the SE number being deleted from the SE Template file.

Whenever, the current SE changes or a component of the current SE changes, we look at the SE to generate the session key (if required). The data required to generate the session key (also known as the derived key) is given as part of a component of the SE. The session key mechanism is specified in the SE which is used to generate it and keep it in the RAM as long as it is valid.

Only 3DES is being used in all the cryptographic algorithms. The current SE is accessed when security operations like encipher, decipher, cryptographic checksum, authentication are performed.

The use of the SE in different contexts is described below:

- **Authentication:** The AT in the SE specifies the key reference (tags 83 and 84) and whether the key is to be used directly or for generating a session key, the algorithm reference (tag 80) (3DES is used by default), data for computing the session key (tag 94). The key reference is mandatory while the rest are optional. The CRT usage qualifier DO in the AT gives further information about the applicability of the CRT (whether it can be used for external authentication, internal authentication). If the key is to be used directly then it is directly used to authenticate. If the use is for computing a session key, then all references to this key implicitly mean that the session key is to be used.

- **Confidentiality:** The CT in the SE specifies the key reference (tag 83 and 84) and whether the key is to be used directly or for generating a session key, the algorithm reference (tag 80) (3DES is used by default), the mode of operation and data for computing the session key (tag 94). The key reference is mandatory while the rest are optional. 3DES in chained block mode is used for encryption/decryption. As in AT, the CRT usage qualifier DO in the CT gives information about the applicability of the CRT (whether it can be used for encryption, decryption). The use of the session key is same as mentioned in authentication. Furthermore, only CT-sym is being supported.

- **Cryptographic Checksum:** The CCT in this case gives the required information which is the same as in Confidentiality case.

3.5 Anti-tearing protection

Anti-tearing protection refers to the mechanisms used by the card operating system to ensure that the data stored inside the files of the card is not inconsistent because of abnormal interruptions in the functioning of the card like power-off, forcible removal of the card from the terminal.

When ever a command has to write into a file or in the EEPROM the data is written in a temporary cache instead of the EEPROM. A flag is associated with the cache data, which is invalid before the command starts execution. When the command completes execution, we set the flag in the cache to valid. The data in the cache is written into the corresponding EEPROM location by the main command loop. The cache is checked for the availability of the required data during a read from EEPROM and if found then data is read from the cache instead of the EEPROM. Thus when a command doesn't complete and the Operating System resets due to some possible error (card taken away from the reader, power lost etc.,), the data written by the command is not updated in the EEPROM. However, if the command completes and then as the valid bit is left set, upon the next power up the EEPROM is updated. This preserves the consistency of the data in the Smart card.

3.6 Current implementations of the Card Operating System

The Card Operating System is currently ported for three different architectures. The first implementation is on a linux platform, where the card operating system works as a program communicating with other programs through the standard input and standard output.

3.6.1 Linux port of the Card Operating System

The Card Operating System is ported on Linux by implementing the processor dependent part of the OS in Linux. The initializations corresponding to Linux are done in initparam() function. External reset is handled as a signal to the OS process. The signal handler restarts the working of OS once it receives a signal.

The hardware specific routines that are required by the Card Operating System are handled as follows.

- **EEPROM:** EEPROM is implemented as a memory mapped file in Linux implementation. The file name and the size of EEPROM is known by looking at a configruration file whose name is stored in the environment variable SCOSTACONF. The configuration file contains two variables EEPROMFILE and EEPROMSIZE. Memory is mapped to the EEPROMFILE with EEPROMSIZE of memory. Any update to the EEPROM is done by assigning the values to memory locations that is reflected in corresponding byte in the file to which memory is mapped as EEPROM.

- **Random No:**In Linux implementation random number is generated by reading a byte from /dev/random which gives random bytes. The function getRandomByte() is implemented by reading a single byte from /dev/random and returning the byte.

- **Input/Output:** Input and output between the external world Card OS is done by terminal I/O. The Card Operating System reads the input bytes from the standard input and writes the output to standard output.

- **Chip Serial No:** The chip serial number in case of the linux implementation is the concatenation of its IP address (4 bytes) and the inode number of the file which is mapped to the EEPROM.

3.7 Limitations

The following are some of the limitations of our code.

Currently only the T=0 protocol with the default parameters is supported.

There is a limit on the depth of the file system supported. This is stored in the compile-time configurable constant MAX_DEPTH

The maximum size of a file, which we can allow is 64K. The total EEPROM size is also assumed to fit in this size.

Chapter 4

Conclusion and Future Work

In this report, we have described the design and implementation of a SCOSTA-compliant operating system for smart cards. We have also described the implementation of our operating system on the linux platform. We have observed that very few changes needed to be made to the original code to port it to another architecture.

Future work will be in the direction of porting the operating system for different architectures as well as adding to the basic functionality of the operating system by implementing more functionality to support payment applications etc.,

Appendix A

Terminology

This appendix describes the terminology used in the report. Further descriptions of these terms are present in the ISO standards [12, 13, 18, 19]

- **ATR:** Answer to Reset
- **PPS:** Protocol Parameters Selection
- **CRT:** Control Reference Template
- **AT:** Authentication Template
- **CT-sym:** Cryptographic Template, symmetric
- **CCT:** Cryptgraphic Checksum Template
- **DF:** Dedicated File
- **EF:** Elementary File
- **SE:** Security Environment
- **SIM:** Subscriber Identity Module
- **CLA:** Class byte
- **INS:** Instruction byte

- **FDB:** File Descriptor byte
- **LCSI:** Life Cycle Status Integer
- **SFI:** Short File Identifier

Appendix B

Support Routines

The following are the important support routines present in the support module.

B.1 Routines for accessing meta data

- GetLength: Gets the total length of the given file
- SetLength: Sets the total length of the given file
- GetParent: Gets the parent DF of the given file
- SetParent: Sets the parent DF of the given file
- GetSibling: Gets the sibling of the current file
- SetSibling: Sets the sibling of the current file
- GetChild: Gets the child of a DF, if present.
- SetChild: Sets the child of a DF to the given file

B.2 Routines for accessing the File Control Parameters of a file

- GetFDB: Gets the file descriptor byte of a file

- SetFDB: Sets the file descriptor byte of a file
- GetLCSI: Gets the Life Cycle Status Integer of a file
- SetLCSI: Sets the Life Cycle Status Integer of a file
- GetSETemplateId: Gets the File Identifier of the file which contains the SE template for a DF
- SetSETemplateId: Sets the SE Template Identifier of a DF to the given value
- GetDFNameLength: Gets the length of the DF Name for a DF
- SetDFNameLength: Sets the length of the DF Name for a DF
- GetSecurityAttrLength: Gets the length of the security attributes of a file
- SetSecurityAttrLength: Sets the length of the security attributes of a file to the given value
- GetSecurityAttrAddr: Gets the address of the beginning of the Security Attributes
- GetSFI: Gets the Short File Identifier of the given EF
- SetSFI: Sets the SFI of the EF to the given value
- GetDCB: Gets the Data Coding byte of the given EF
- SetDCB: Sets the Data Coding byte of the given EF
- GetDataLength: Gets the length of the data of a Transparent EF
- SetDataLength: Sets the length of the data of a Transparent EF
- GetMNR: Gets the Maximum Number of Records of a record-oriented EF
- SetMNR: Sets the Maximum Number of Records of a record-oriented EF

B.3 Routines for Accessing the data of the Elementary Files

- GetFileBytes:Gets given number of bytes from the Transparent File, starting from an offset.
- WriteFileBytes: Writes the given number of bytes in the transparent file, starting from an offset. The type of write behaviour is determined by the DCB of the file.
- UpdateFileBytes: Updates the given number of bytes in the transparent file,starting from an offset.
- EraseFileBytes: Erases the given number of bytes in the transparent file, starting from an offset.
- GetIthRecord: Gets the record number given from a Record Oriented File
- WriteIthRecord: Writes the record number given to a Record Oriented File. The type of write behaviour is determined by the DCB of the file.
- UpdateIthRecord: Updates the record number given to a Record Oriented File.

Bibliography

[1] Gemplus R&D Topics page *http://www.gemplus.com/smart/enews/st3/32bit.html*

[2] The SCOSTA standards page *http://www.cse.iitk.ac.in/ moona/scosta/*

[3] Jurgen Dethloff, Helmut Grottrup "Identifikanden/Identifikationsschalter", German Patent, DE 19 45 777 C2, February 1969.

[4] Ellinboe Jules, "Active Element Card", US Patent, US 3,637,944, January 1972.

[5] Paul Castruci, "Information Card", US Patent, US 3,702,464, November 1972.

[6] Gaming for smart cards, home page of Kaosc. *http://www.kaosc.com/*

[7] The SmartFlash Cards page *http://www.britneyspears.com/smartflashcard/index.php*

[8] The respironics home page *http://www.respironics.com/*

[9] The IButton introduction page *http://www.ibutton.com/ibuttons/*

[10] ISO/IEC 7816-1:1998 Identification cards – Integrated circuit(s) cards with contacts – Part 1: Physical characteristics

[11] ISO/IEC 7816-2:1999 Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part 2: Dimensions and location of the contacts

[12] ISO/IEC 7816-3:1997 Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part 3: Electronic signals and transmission protocols

[13] ISO/IEC 7816-4:1995 Information technology – Identification cards – Integrated circuit(s) cards with contacts – Part 4: Interindustry commands for interchange

[14] ISO/IEC 7816-4:1995/Amd 1:1997 secure messaging on the structures of APDU messages

[15] ISO/IEC 7816-6:1996 Identification cards – Integrated circuit(s) cards with contacts – Part 6: Interindustry data elements

[16] ISO/IEC 7816-6:1996/Cor 1:1998

[17] ISO/IEC 7816-6:1996/Amd 1:2000 IC manufacturer registration

[18] ISO/IEC 7816-8:1999 Identification cards – Integrated circuit(s) cards with contacts – Part 8: Security related interindustry commands

[19] ISO/IEC 7816-9:2000 Identification cards – Integrated circuit(s) cards with contacts – Part 9: Additional interindustry commands and security attributes.

[20] Digital cellular telecommunications system (Phase 2+); Subscriber Identity Module Application Programming Interface (SIM API); Service description; Stage 1

[21] Digital cellular telecommunications system (Phase 2+); Security mechanisms for the SIM application toolkit; Stage 1

[22] Schlumbeger MicroPayflex card *http://www.cardstore.slb.com/*

[23] Gemplus GemSafe cards home page *http://www.gemsafe.com/*

[24] Deutsche Telekom Multifunction Card TCOS Cryptographic Card *http://www.telesec.de/*

[25] The Java Card Management Specifications Version 1.0b *http://www.javacardforum.org/Documents/Jcms10.PDF*

[26] The MULTOS home page *http://www.multos.com/*

[27] The Schlumberger's Cyberflex card home page *http://www.cardstore.slb.com*

[28] The Gemplus GemXpresso RAD 211 *http://www.gemplus.com/*

[29] The pcsc work group home page *http://www.pcscworkgroup.com/*

[30] The open card group home page *http://www.opencard.org/*

[31] The M.U.S.C.L.E home page *http://www.linuxnet.com/*

[32] The EMV home page *http://www.emvco.org/*

[33] The BasicCard from ZeitControl home page *http://www.basiccard.com/*

[34] Ankit Jalote and Marghoob Mohiyuddin "Implementing the Security Module of a Smart Card Operating System", BTP 2002, Department of CSE, IIT Kanpur. *http://www.cse.iitk.ac.in/research/btp2002/scosta.ps.gz*