Implementing the Security Module of a Smart card Operating System *

Ankit Jalote, Marghoob Mohiyuddin
{ankit,marghoob}@cse.iitk.ac.in

Under the guidance of:
Dr. Deepak Gupta
Dr. Rajat Moona
{deepak,moona}@cse.iitk.ac.in

Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur
Kanpur,India-208016

Abstract

In this paper we describe the implementation of the security module of a smart card operating system. We follow the architecture as described in the ISO/IEC standards and the SCOSTA (Smart Card Operating System for Transport Applications) specifications.

The security module involves the implementation of the security related commands (as described in SCOSTA) and the security checks that need to be performed before any command is executed. The security checks determine whether the command can be allowed in the current context. A command (to be executed by the card) can be protected by password(s)/key(s). The card reader must prove its knowledge of the required password(s)/key(s) (if any) before the operation. At any point a security status describes the authentication status with respect to these two factors i.e. password verification and key authentication. Successful authentication changes the security status to specify that a particular password/key has been authenticated. The security attributes, which can be attached to any operation, complemented with the security environment describe the security status that should be effective at the time a particular operation is requested. A security environment also describes the keys to be used for operations like encipherment etc.

We start with the general background on smart cards, the basic security features in smart cards, related work done in this field. Then we describe the details of the security architecture. Finally we give the implementation details followed by conclusions and future work.

*This work was done as a B.Tech. project under the guidance of Dr. Deepak Gupta and Dr. Rajat Moona and was part of the Smart card Operating System which is currently being developed at IIT Kanpur. This Smart card Operating System Project is funded by Ministry of Information Technology of India.

1 Introduction

1.1 A Brief Overview of Smart Cards

A smart card is a card that is embedded with either, a microprocessor and a memory chip, or only a memory chip with non-programmable logic. The microprocessor card can add, delete, and otherwise manipulate information on the card, while a memory-chip card (for example, pre-paid phone cards) can only undertake a pre-defined operation. In this project we deal with microprocessor cards so any references to smart card will be to a microprocessor card. At every point we assume following 3 types of memory:

1. RAM: The the local variables and the function parameters are stored here.
2. ROM: The OS code and the constant variables are stored here.
3. EEPROM: The file system is stored here.

Depending on the amount of available RAM, the security environment and the security status (explained later) may be stored in the RAM or in the EEPROM.

To make a computer and a smart card communicate, the smart card is placed in a smart card reader. The smart card operating system is a passive entity. It waits for the commands that are given by the smart card reader. The communication between the card and a card reader follows a command-response pattern. The reader issues a command (depending on the application) to the card and the card responds depending on the fulfillment of certain security requirements (which again, are application dependent).The smart card reader and smart card follow a protocol to exchange messages in the form of command requests and responses. The messages are encapsulated in the APDU (Application Protocol Data Unit) [1] packet. The SCOSTA specification (derived from ISO/IEC-7816) describes in detail the structure of message APDU. Whenever a smart card receives a command it initiates the command handler specific to the respective command. Security conditions can be attached to any operation specific to the files or the commands itself. These security conditions can be specified with respect to any file or directory. In this paper we give references to these standards as appropriate.

The security architecture of a smart card is concerned with two system requirements: data security and data integrity. Data security means that a data value or a computational capability contained on the card can be accessed by those entities that are authorized to access it and not accessed by those entities that are not authorized to access it. Data integrity means that at all times the value of information stored on a card is defined; the value is not corrupted, even if power to the smart card is cut during a computation involving some piece of information stored on the card.

1.2 Standards (ISO-7816 and SCOSTA)

ISO-7816 is one of the major standards for smart card operating system and we have followed it throughout the implementation. SCOSTA is a variant of ISO-7816 and it basically describes the specifications for transport based (e.g driving license) applications. Thus apart from some slight modifications to ISO-7816, the specification is essentially the same as in ISO-7816. An overview of ISO 7816 follows. The standard specifies:

- The physical characteristics of the card.
- The allowed voltages, current and signal rates.

[1] An APDU contains either a command message or a response message, sent from the card reader to the card or conversely.

- The activation process of the card.
- The basic model of a file structure, file types and file and data reference methods.
- Selection of applications.
- Security architecture.
- Inter-industry commands, including commands for file selection, reading and updating of file content, card-holder verification and authentication commands.
- Secure messaging mechanisms to ensure data authentication and data confidentiality.

2 Related Work

Many efforts have been made towards implementation of Smart card operating systems. Some of the companies working in this field are Schlumberger, Gemplus, Keycorp, Syprus, Data key etc. Gemplus and Schlumberger create standardised operating systems like Smart Card for Windows, Java Card etc. Java Card technology defines a platform on which applications written in a sub-set of Java language can run in Smart Cards and other memory-limited devices. Java Card allows a user to download an applet written by the user onto the card. Multiple applets which can handle multiple applications can be downloaded onto the card by the user. This offers a lot of flexibility to the card user at the cost of having a chip-independent Java Virtual Machine running on the card. Another similar model of a generic smart card operating system is the MULTOS operating system developed by the MAOSCO consortium. This is similar to the Java Card model and it allows the user to download his own code written in a chip independent form, called as the MEL (Multos Execution Language). The cost of such generality is the increase in the final production cost of the card. These cards are 3 to 4 times costlier than ordinary application-specific cards. Most commercial implementations of Smart cards, therefore, are application specifc and are made to handle individual applications such as credit/debit, loyalty, e-purse health, network access etc. These have limited functionality (only few commands, fixed file system etc.) but are cheap.

3 Security in a Smart Card

The security of the card is controlled by the card OS. The self-containment of smart card makes it resistant to attack, as it does not need to depend upon potentially vulnerable external resources. At no point does a smart card reveal sensitive information e.g. the secret key of the card-holder, to the outside world.

When a card connects to a card reader, the reader may need to prove its authenticity and identity to the card using a challenge and response method. The card also proves its identity in the same way.

As stated earlier, the behavior of the card follows a command- response pattern. The card reader might need to authenticate itself before it can issue any command(s). The card may send a response or may not send a response depending on whether security requirements are met. Security may be provided to the commands or to the files.

Smart card security is based on the following mechanisms:

- **Password Based Authentication:** The user/external agent must present the password to authenticate itself to the smart card. The smart card matches the password given by the external agent with the one stored inside the card and modifies its status e.g. password number 3 is verified.

- **Key based Authentication:** This is a two-way mechanism. Both the smart card and external agent prove to each other the knowledge of a key common to both of them. The external agent sends a challenge to the smart card and asks it to verify the knowledge of a certain key. The smart card then encrypts the challenge using an encryption algorithm and the key and returns back the result to the external agent. The external agent then authenticates the card by the matching the response that it gets from the smart card to the result it gets by executing the encryption algorithm itself with the key and the challenge. The smart card also authenticates external agent in the similar way by issuing a challenge to it.
- **Encryption/Decryption:** This allows for protection of data as well as command and response by encrypting them.
- **Data authentication:** This is done using cryptographic checksum, which can be used to verify that the data has not been modified by a third party.

4 Security Architecture

SCOSTA compliant OS supports the following security architecture as defined in ISO/IEC 7816-4 [1], ISO/IEC 7816-8 [3] and ISO/IEC 7816-9 [4].

4.1 Security status

The security status indicates which password(s) and key(s) have been successfully authenticated. The password(s) and key(s) could be global (card specific) or local (application specific). The SCOSTA compliant OS supports the following three security statuses:

- Global security status (related to the card authentication processes)
- File-specific security status (related to the application authentication processes)
- Command specific security status (related to the command authentication processes)

4.2 Security attributes

Security attributes define the security conditions to be fulfilled before a command can be executed. Security attributes consist of access rules which specify the security conditions (if any) to be fulfilled for a given command/a set of commands. Security attributes are stored in the meta-data of files (for file related commands and also for security commands). Both compact and expanded formats (as defined in ISO/IEC 7816 [4]) for encoding security attributes are supported.

4.2.1 Security to files

Access to data is made through the logical file structure of the card. Associated with each file, there are File Control Parameters, which also include the security attributes associated with the file. The security attributes specify:

- The security status of the card to be in force before access to data is allowed.
- Restrict access to data to certain functions if the card has a particular status.
- Define which security functions shall be performed to obtain a specific security status.

A Life Cycle Status (LCS) may also be associated with the files and with the card itself. Basically, this defines the primary states of the life cycle in the following order:

- Creation state: No security attributes in this case.
- Initialization state: Security attributes for this LCS apply.
- Operational state: Security attributes for this LCS apply.
- Termination state: No modification of data allowed in this case.

The LCS can be changed based on certain operations but cannot move backward. The LCS is used by the card, possibly in combination with security attributes to determine whether a requested operation with the file is in accordance with the specified security policy.

4.2.2 Security to commands

The Security Environment (SE) is used to provide security to commands. It is basically a mechanism to specify to the card system, the security functions that are available to provide protection to commands for a specific application of card. The SE is a container for security-related data and mechanisms. A SE specifies:

- References to the cryptographic algorithm(s) to be executed
- The mode(s) of operation
- The key(s) to be used
- Any additional data needed by a security mechanism
- May also provide directions of handling the data resulting from the computation

4.3 Security Environments

At any time, a security environment is active for the card. The security environments (SEs) define the mechanisms to be used for security related commands. This includes the algorithms to be used, the key references and data for key derivation. The security environments (SEs) are stored in elementary files (EFs) or as meta-data of files.

4.4 Security mechanisms

The SCOSTA compliant OS supports the following security mechanisms as described in ISO/IEC 7816-4:

- Entity authentication with password (VERIFY command)
- Entity authentication with key (EXTERNAL AUTHENTICATE, INTERNAL AUTHENTICATE and MUTUAL AUTHENTICATE commands)
- Data authentication (computation/verification of cryptographic checksum).
- Data encipherment.
- Data decipherment.

Triple DES (3DES) algorithm in CBC (chaining block) mode is the algorithm used for data authentication, data encipherment and decipherment.

5 Security related commands in SCOSTA

The following commands are supported by the SCOSTA compliant operating system:

- **VERIFY:** Password based authentication.
- **INTERNAL AUTHENTICATE:** Responding to a challenge issued by the card reader.
- **EXTERNAL AUTHENTICATE:** Key based authentication.
- **MUTUAL AUTHENTICATE:** Key based authentication followed by response to challenge issued by the card reader.
- **GET CHALLENGE:** Card issues a challenge (for purpose of key based authentication) to the reader.
- **MANAGE SECURITY ENVIRONMENT (MSE):** Set the Security Environment (SE).
- **PERFORM SECURITY OPERATION (PSO):** Perform encipherment, decipherment and data authentication.
- **ENABLE VERIFICATION REQUIREMENT:** Enable password based authentication.
- **DISABLE VERIFICATION REQUIREMENT:** Disable password based authentication.
- **CHANGE REFERENCE DATA:** Change password.
- **RESET RETRY COUNTER:** Set the maximum number of retries allowed for authentication.

Details of the above commands can be found in ISO/IEC-7816 [1], [3]

6 Implementation Details

6.1 Limitations and Issues

Writing an implementation is plagued by the following limitations:

- The amount of memory available is severely limited. For example, a typical 8-bit micro-controller has about 128 bytes of internal (on- chip) RAM. The code memory is also small typically in the range of a few kilobytes. The card may also have an external (meaning off-chip) RAM which might be a few kilobytes. This means that the code being written has to be efficient in terms of runtime space required and also in terms of size of code.
- The size of data that can be handled is 8 bits. Typically, all the instructions deal with 8-bit data. Thus, to implement handling a larger size data, one needs to write additional instructions thus adding to the code size. For example, we might need to write a routine to add two 32-bit numbers. Therefore the code has to be '8 - bit friendly' which again is a factor which increases the code size.
- The maximum depth of function calls has to be small. This means integrating as much functionality into a module as possible as a greater depth of function calls means we eat up more of runtime stack and hence the less of available internal RAM.

6.2 Command handling

The main command handler loop executes in the following manner:

1. For every command read the command header. Check if the class and the instruction bytes are correct else return error.
2. Next check if the data field of the command can be accommodated in the memory else fail.
3. Check if the command requires some input data to be read. In this case send an acknowledgement to the card reader.
4. Call the appropriate command handler for the command.
5. Check if some output is to be sent to the card reader. If this is the case, send the output.
6. Return the appropriate status bytes to the card reader. The status bytes indicate whether the command succeeded or failed (in case of failure they might also specify the cause of failure).

6.3 Structure of a command handler function

Any command handler has to follow certain guidelines to work properly.

- 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 (length of the output) 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, 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, which is same as sendBuffer) for retrieval by a an immediate later GetResponse command. It should also indicate the length of data stored (in storeLength).
- Every command-handler should set the value of the status bytes it needs to send in 2 variables (bSw1, bSw2), before returning. (Except in the case of normal response i.e. 90, 00).
- Any command handler might call support routines for doing the file specific operations that it is supposed to do.

The basic steps involved in a command handler are:

1. First check (by calling the function VerifySE) if this command is allowed in the current context. If this command is not allowed then fail with the appropriate status bytes.
2. Execute the command.
3. If the command is related to password verification/key authentication then update the security status to reflect the successful completion of the command (if it was successful).
4. Return.

6.4 The Password File

The reference data for the global data bank is stored in an EF(elementary file or normal file) immediately under the MF(master file or root directory). This EF is identified using a short EF identifier of 1 (EF1). Similarly, the local reference data for an application is stored in an EF immediately under a DF(dedicated file) for that application. The short EF identifier of 1 (EF1) is used to identify this EF. The OS may fix any 16-bit identifiers for such files. The changes in the EF1 may or may not be permitted depending upon the security attributes for that file. However this file will be referenced upon for validating the passwords. The EF1 will be a variable record file (up to a maximum of 32 records) with the following structure.

- Pin identifier: 1 Byte
- Retry counter: 4 Bits (see below)
- Max retry count: 4 Bits (see below)
- Pin: Variable length

Records in the EF1 will have one byte containing the Retry counter and Max retry count. The bits b8:b4 of this byte will provide the Retry counter while bits b4:b1 will provide the Max retry count. Bits b8 and b4 will be the MSB of their respective fields. A value F for the Max Retry count with non-zero Retry counter shall mean that there is no limit on the retries.

The Pin identifier will be coded as follows.

- Ref Data Number (b5:b1) represent the 5 bit password number.
- V bit (bit 8) represents if the corresponding entry is valid (1) or not (0).

6.5 The key file

The secrets are stored in EF2 immediately under the MF or a DF. The secrets stored in EF2 immediately under the MF are the global secrets. EF2 is a variable record structured file with the following structure.

- Key identifier: 1 Byte
- Key Type: 1 Byte
- Key Specific Information: Variable length
- Algorithm Reference: 1 Byte
- Key: Variable length

The key identifier is coded as follows.

- Secret Number(b5-b1): The secret number (by which the key is referred to in various security related operations) will be unique for all keys. Thus there can be only up to 32 keys in EF2. No two keys will have the same secret number even if they are used for two different purposes.
- V bit: is used to denote if the corresponding secret is valid or not. (0: invalid, 1: valid).

The key type field provides the operations for which the key can be used. The value is coded as follows. CC(b8), DS(b7), Enc-Sym(b6), Enc-Asym(b5), Hash(b4), Int Auth(b2) and Ext Auth(b1).

If the CC bit is set to 1, the key can be used for computation of the cryptographic checksum.

If the DS bit is set to 1, the key can be used for computation of digital signature. Since in the current version of the SCOSTA, public key cryptography is not supported, this field should be treated as RFU.

If the Enc-Sym bit is set to 1, the key can be used for symmetric encryption and decryption.

If the Enc-Asym bit is set to 1, the key can be used for asymmetric encryption and decryption. Since the public key cryptography is not supported in this version, this field should be treated as RFU.

If the Hash bit is set to 1, the key can be used for the hashing operation.

If the Int Auth bit is set to 1, the key can be used for the internal authentication.

If the Ext Auth bit is set to 1, the key can be used for the external authentication.

The type specific information is of variable length and is defined as per the key type field. The value is made available for each bit set to 1 in the key type field. The values are provided in the order of the bits in the key type field. Thus if the CC bit is set to 1, the type specific information will first contain the information regarding the usage of the key for the CC.

The following type specification information is used.

Operation	Information
CC	None (0 bytes)
DS	None (0 bytes)
Enc-Sym	Usage Counter (2 Bytes)
Enc-Asym	Usage Counter (2 Bytes)
Hash	None (0 bytes)
Int Auth	Usage Counter (2 Bytes)
Ext Auth	Retry Counter (4 bits)
	and Max Retry Count (4 bits)

The Usage counter for the Enc-Sym, Enc-Asym and Int Auth is a monotonically decreasing counter. The counter if set to FFFF, means that the counter is not used (and therefore is not changed by the usage of the key). Values other than FFFF refer to the number of times that the key can be used by the INTERNAL AUTHENTICATE command. The key can be used only if the Usage Counter is non-zero. Upon each usage (whether successful or unsuccessful), one is subtracted from the counter (only if the counter is non FFFF). The initial value of the counter is set at the time writing the record in the EF2. The value can be changed by UPDATE RECORD command if its execution is permitted by the security conditions.

The Retry Counter and Max Retry Count are coded as per the coding given for the EF1. Bits b8:b5 provide the Retry Counter value while bits b4:b1 provide the Max Retry Count. These values are used only upon the use of the EXTERNAL AUTHENTICATE command. If the value of the Retry Counter is 0, the EXTERNAL AUTHENTICATE command results in a failure. If the Retry Counter is other than 0, it is decremented by 1 (only if the Max Retry Count is not F) upon each unsuccessful authentication.

The algorithm reference codes the algorithm for which the key usage is valid. A value of 00 for the algorithm reference implies that the key is valid for all the algorithms available in the card. The changes in the EF2 may or may not be permitted depending upon the security attributes for that file. However this file will be referenced for validating the keys internally by the operating system.

6.6 Global Data Structures

In this section we give the description of some global variables used in our implementation.

6.6.1 CurrentStatus

Description: The current security status is used by VerifySE function to determine if a command can be executed under the current security status.

Every directory has its respective password and key file. A maximum of 32 passwords/keys are possible for each depth. 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.

6.6.2 Last Challenge

Description: The last challenge maintains the last challenge issued by the card as a result of a GET CHALLENGE command. The last challenge is required for external authentication (where the external agent sends the response to the challenge) or for deriving a session key (if MSE command specifies so). This response is decrypted and compared with the last challenge to verify the card reader's knowledge of a certain key. The length of the last challenge is used as a flag to indicate the validity of the last challenge. If the last challenge is invalid then it cannot be used. The last challenge ceases to be valid after the next command following the GET CHALLENGE command.

6.6.3 Derived Key

Description: The derived key (same as a session key) is used when the MSE 'set' command states that a session key be computed. Derived Key is generated by the MSE set command if the CRT[2] in the current SE being set specifies that the key reference be used directly (key reference with tag 83) or for deriving a session key (key reference with tag 84). If the key is to be used for deriving the session key then the CRT specifies how to derive the session key. The data for deriving the key may be given in the CRT (DO with tag 94) or the last challenge (which should be valid) issued by the card can be used. The length of the challenge or the data should be equal to the length of a 3DES key. The session key is then encrypted with the key specified to generate the session key.

In External/Internal Authenticate if the key reference received matches with the derived key reference and the derived key is valid for External/Internal Authenticate then the derived key is used otherwise the key is the read from the Key file as usual.

In Encipher/Decipher/Verify-CCT/Compute-CCT if the Current SE in the corresponding CRT contains key reference with 84 tag (i.e. compute the session key) then the derived key is used. If the CRT contains key reference with 83 tag (i.e. use the key directly) then use the key from the key file as usual.

The derived key is invalid in the beginning and once derived remains valid till :

1. Another key is derived (in which case the new key becomes valid).
2. The currentDF is changed by select file command (in which case the key is made invalid).

6.6.4 Security Environment

Introduction: The security environment (SE) defines the security mechanisms that are available for reference in security related commands and in secure messaging. An SE shall specify references to the cryptographic algorithm(s) to be used, the mode(s) of operation, the key(s) to be used and any additional data needed by a security mechanism. An SE might also contain mechanism to perform initialization of non-persistent data e.g. session key.

At any time during operation of the card a current SE is active (by default or as a result of commands from the interface device). The current SE can be set or replaced with the MANAGE SECURITY ENVIRONMENT command.

[2]A CRT is a component of an SE. CRT specifies the security mechanisms to be used. The tag of the CRT specifies the function (e.g. encryption) for which this CRT can be used. A CRT itself contains DOs. A DO (Data Object) encapsulates an object. The object is identified by the tag of the DO. The length field of the DO specifies the length of the object. The object might be a simple plain value or a combination of DOs.

The default SE is active if the current SE has not been set by an MSE command.

Components: Control Reference Templates (CRT) are used to describe the various components of a SE. Five templates have been defined in ISO/IEC 7816:

- Cryptographic Checksum
- Digital Signature
- Confidentiality
- Hash
- Authentication

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

The SEs are 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 and whether the key is to be used directly or for generating a session key, the algorithm reference (3DES is used by default), data for computing the session key. 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 and whether the key is to be used directly or for generating a session key, the algorithm reference (3DES is used by default), the mode of operation and data for computing the session key. 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.

6.7 Implementation of Major Command Handlers

We have merged the command handlers of similar commands to reduce the code size and to avoid the overhead of passing parameters to functions since RAM size is limited. We have merged Verify, External Authenticate, Internal Authenticate and Mutual Authenticate commands into one function. Similarly, we have merged Encipher, Decipher, Calculate Cryptographic Checksum and Verify Cryptographic Checksum into one function. We have also merged Enable and Disable Verification requirement commands into one function. Following we describe the flow of some commands.

6.7.1 Implementation of Verify and External Authenticate Commands

First VerifySE is called to check whether the command is allowed under the current Security Status. The password/key reference (specified in the command header) indicates the password/key (which may be stored in local/global data bank) to be used. In case of any error appropriate Status Bytes are set and the command returns. In case of VERIFY, the input data is compared against the stored password for verification. In case of EXTERNAL AUTHENTICATE, the last challenge is encrypted with the specified key and then the result is compared against the last challenge (which should be valid) for key authentication. On failing, appropriate error conditions are set. The retry counters (which specify the number of further allow retries). On success, the security status is updated to reflect the successful verification/authentication.

6.7.2 Implementation of Manage Security Environment

Through this command we can:

- Set: Modify a component of the current SE. Since only modification is allowed, the lengths and tags of the new DOs should match with those in the currentSE. replace the current security environment with another SE (stored either in a SE template file or in the File Control Parameters for the current directory). This command may result in the computation of a derived key as described previously.
- Store: Store the current SE. Again, the DOs in the current SE should already be present in the SE (in which the current Se is being stored).
- Restore: Load an SE into the current SE.
- Erase: Erase an SE.

6.8 Implementation of Perform Security Operation

This command may specify one of the following operations:

- Encipher:Encrypt the data given as input.
- Decipher:Decrypt the data given as input.
- Compute Cryptographic Checksum:Compute the cryptographic checksum.
- Verify Cryptographic Checksum:Verify whether cryptographic checksum of the plain data matches the cryptographic checksum value given in the input.

The keys required for the above operations are obtained from the current SE.

6.8.1 Implementation of VerifySE

VerifySE is called by every command handler to determine of the current command is allowed. This is determined by checking the current security status against the security status as required by the security attributes. The file where the security attributes are present is supplied as a parameter. The basic implementation of the VerifySE function follows:

- First check the life cycle status (LCS) of the file to determine whether the security attributes apply. For the creation state, the security attributes do not apply.
- If the security attributes apply, then they are read from the FCP of the file.
- Depending on the format of the security attributes (the format may be compact or expanded), the attributes are scanned to check whether the current command is allowed. The security attributes might refer to a Security Environment (which basically will specify the key/password references which need to have been verified/authenticated). The key references might also be specified in the security attributes. Furthermore, the conditions to be satisfied might be OR/AND of sub-conditions.
- If there is an access rule which is satisfied for the command, then VerifySE succeeds else fails.

6.9 Algorithms Implemented

We implemented the following algorithms:

- 3DES for encryption/decryption/cryptographic checksum.
- MD5 [6] for hashing.

7 Conclusions and Future Work

We were successful in implementing the security module of the Smart Card OS as specified by SCOSTA. The security module is a very important part of a Smart Card OS since it provides complete protection to the smart card from outside world. Things get more complicated due to the fact that we have only limited memory RAM and ROM available on a Smart Card. For this reason an efficient, clean and compact implementation of this module is a necessity. The major complications while coding arose due to the fact that we had to look at a number of error conditions and to set appropriate status bytes.

Future Work can include implementing the Secure Messaging Extensions as defined in the ISO/IEC-7816 standards (secure messaging has not been covered in SCOSTA). Secure messaging involves the encryption/decryption/authentication of commands.

8 Acknowledgements

We would like to express our deep gratitude to our supervisors Dr. Deepak Gupta and Dr. Rajat Moona, for their guidance and invaluable suggestions at all stages of this project. We would like to thank them for their enthusiasm and motivation throughout the discussions we had with them during our meetings which saw us through some rough patches during the coding stage of the project.

We are also thankful to Dr. Manindra Agarwal for giving valuable tips and help related to efficient implementation of 3DES algorithm and helping us in understanding the standards.

We are also thankful to Mr. S. Ravinder who provided us with the support functions to access the records in the file etc. and also for helping us in integrating the security module with the rest of the OS. Finally, we would also like to thank Mr. Kapileshwar Rao Bolisetti and Mr. S. Ravinder for providing us the Linux based testing tool which made testing very fast and clean.

References

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

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

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

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

[5] SCOSTA document version 1.2b

[6] RFC 1321 MD5 Message-Digest Algorithm