Public Key Infrastructure in SCOSTA

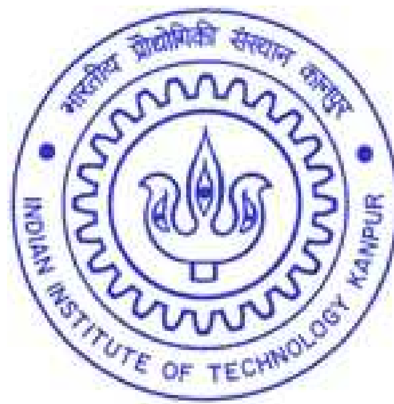
CS499 - Project Report

by

Venkata Rao Pedapati and Sri Simil Dutta
Y3395, Y3350

under the guidance of

Prof. Rajat Moona

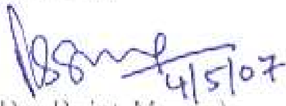


Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
May 2007

Certificate

This is to certify that the work contained in this report entitled "Public Key Infrastructure in SCOSTA", by *Venkata Rao Pedapati(Y3395)* and *Sri Sunil Dutta(Y3350)*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

May, 2007.



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

Abstract

Public Key Infrastructure(PKI) systems have gained popularity over the last few years because of their scalability, security and ease of maintainence. They are slowly turning into a necessity from being a luxury particularly in smart card industry. Smart cards fit seamlessly into Public Key Infrastructures. They are very natural key storage units. Today's extensive use of smart cards in variety of applications in conjuntion with the advent of advanced cryptographic smart cards that can do complex cryptographic calculations on board demand robust operating systems with PKIs built into them. Although some operating systems with these features exist, few implement open standards. In addition, many PKIs designed by other parties, very often outside India, do not address the native problems in appropriate ways. Since we understand the kind of applications that are going to be deployed in India and limitations of existing resources better, we felt the need of an efficient PKI system developed in-house. In this report, we present the design and implementation details of PKI in SCOSTA - a Smart Card Operating system that is already widely used by some Government agencies in India. We look at what different components or modules should constitute a PKI architecture and how we approached and implemented each of them. We shall also present details about how we tested our system to make it robust enough.

1 Introduction

Smart cards are intelligent cards equipped with a micro-controller. They are typically used to host security sensitive applications like Monetary applications and Identification systems where more conventional approaches using PCs are highly vulnerable to attacks. Smart cards need to be differentiated with Memory cards, which do not contain any micro-controller but only some internal memory which can be read or written to with substantial ease. In contrast, the data stored in the smart cards is under the control of the processor, and cannot be read or written to directly by an external entity. Todays smart cards typically contain an 8-bit microcontroller, a few kilobytes of RAM, few tens of kilobytes of ROM, and variable amounts of EEPROM to store temporary data, code and permanent data respectively. An anology can be drawn with conventional computing systems which also contain a processor, memory, and a hard disk to hold the file system.

Despite the feeble power of microcontrollers and very less memory when compared to regular PCs, there are several reasons why even smart cards need operating systems. To accomodate the growing markets, and ever growing list of potential applications, application development on smart cards needed to be made much faster. Hence, application developers needed to be exempted from details about hardware architecture, file system etc. Another reason is that security features are much easier to realise when a well defined OS is present at the foundations.

SCOSTA (Acronymn for Smart Card Operating System for Transport Applications) is one such operating system. When it was created in June 2001, it was intended to be used in Transport applications for identity and security. However, slowly it is evolving as a general purpose smart card operating system. The primary objectives of SCOSTA project are *Standardization of Information, Inter-operability, Multi vendor support/Non-Proprietary* and *Security and integrity of data*. The initial versions of SCOSTA partially realized these goals. We shall stress more on security features of SCOSTA from now. Starting from it's inital design, SCOSTA supported security operations involving symmetric key cryptography. In particular, it supported encryption and decryption of data, authentication procedures and even cryptographic checksum calculation and verification using DES algorithm in CBC mode. However, there was no architecture for using SCOSTA in conjunction with PKI. As the complexity and scope of applications increased, the security features provided by SCOSTA are no longer sufficient. So, for SCOSTA to become a usable operating system in large applications, an efficient implementation of PKI has become necessary.

The remaining part of this report is organized as follows: We give a precise problem statement in chapter 2, we shall list the goals to be achieved in order to bring up a

working PKI system. Then in chapter 3, we shall give further motivation for our work with some examples. In chapter 4, we shall give some background information a reader needs to be aware of, to understand the rest of the report. Chapter 5 describes a high level design of our system. We shall show the important components contained in the system and how they are connected. Chapter 6 gives details about the implementation of each of the components mentioned in Chapter 5. It also speaks about some challenges we encountered during implementation and how we solved them. Chapter 7 describes how we tested our system. Chapter 8 summarises our results and finally Chapter 9 points out some defects and areas of improvement after which we conclude the report.

2 Problem statement

Our goal for this project was to design and implement Public Key Infrastructure in SCOSTA. The goals of any public key infrastructure are best described by the functionalities or interface it provides to the users:

- A subsystem to store and retrieve keys and certificates easily.
- Encryption and Decryption of data using public keys and private keys respectively.
- External and Internal authentication procedures using asymmetric keys.
- Computation and Verification of digital signatures.
- Storage and verification of digital certificates.

Other than these, there are other non-functional goals like

- The system should be compliant to relevant international standards in order to be interoperable with different application vendors (ISO/IEC 7816-4[1], 8[2], 9[3] and 15[4], PKCS#1 v2.1[5]).
- The system should utilize native architecture specific facilities such as cyrpto coprocessors on some microcontrollers to produce fast executing code. In other words, the decrease in the response time of the system after incorporating new procedures, should be negligible.
- The implementation should be modular and layered in structure so that it can be easily extended later.
- The implementation should be backward compatible. That is, all the older procedures using symmetric keys shall be still working.

3 Motivation

SCOSTA is being used in DL(Driving license) and RC(Vehicle Registration Certificates) applications successfully. A surprisingly simple implementation added to the strong security features including encryption and decryption using 192-bit Triple DES enabled fast development of applications over it. Recently, an identification system was successfully developed and deployed at IIT Kanpur, based on SCOSTA OS. It further boosted building of applications that are going to change the way many administrative processes are performed at IIT Kanpur.

On a small scale like IIT Kanpur, this architecture seems fine. However, when escalated to national or international level, SCOSTA with symmetric key cryptography has some serious limitations. In near future, plans are on for using SCOSTA in applications like e-Passport and National ID. As more applications based on smart cards emerge, the security and ease of maintainence provided by a symmetric key system are no longer sufficient. At that level, where thousands or even millions of cards are distributed to individual users and multiple applications must be accomodated on a single card, it is necessary to have a stronger security system. Since applications typically deal with money transactions, security of user information is of utmost concern there. Since an architecture to support PKI systems already exists in India, implementation of PKI built in to it would make SCOSTA suitable for these kind of applications.

In most typical applications, there exist few authorities while number of users is usually much higher. Only these authorities should be able to modify certain parts of data stored in user's cards and no others. The obvious way to deal with this situation is to share a common secret key between the authorities and all the users' cards. But, since the whole security of the system depends on that single key, replicating it in thousands of cards clearly compromises the whole system. This problem is often dealt with by deriving user specific keys from a master key using some information which is specific to that user like password or chip serial number. Although this increases reliability a bit, it still does not compete against superior key management systems provided by asymmetric key cryptographic systems. Some requirements like Data integrity and Non-repudiation are impossible to achieve in symmetric key systems. The key management is one of the major problems in this architecture. Even using the derived keys, the security of individual users cannot be guaranteed.

Public key cryptographic (PKC) systems are much better in this respect, because of the fact that no separate key management is needed. With the availability of sophisticated certificate management systems already in place, which make key management seamless, adopting smart card operating systems towards PKI systems seems a necessary choice to

make.

The recent upsurge in production of microprocessors having extra capabilities to support costly calculations in public key systems further encourages us toward this goal. Highly secure chips with special coprocessors that run in parallel to the main processor to support mathematical operations of huge operands, makes more secure SCOSTA without compromising speed and efficiency, not just an idea for the far future anymore but a real life possibility. Modern processors can execute RSA (most popular among public key systems) algorithm in less than a second (when implemented using Chinese Remainder Theorem)[7].

There are very few smart card operating systems, that implement open international standards to support PKI, at least in India. PKI thus makes SCOSTA available to larger user base.

4 Background

In this chapter, we shall look at some basics of SCOSTA internals and existing security architecture. These concepts shall be used in further chapters without any further explanation. Readers who are already familiar with SCOSTA file system design and it's security architecture may skip this chapter.

4.1 SCOSTA file system and Security

The figure 1 illustrates the arrangement of files and directories in SCOSTA. It is very similar to unix file structure. There is a Master File(MF) at the top of the directory tree. There can be any number of Dedicated files (DF) and Elementary files (EF). Similarly, each DF can contain any number of DFs or EFs under itself. As identified in the figure, EF1, EF2 and EF3 are three special elementary files under MF (files with Short Identifiers 1, 2 and 3). They are called internal files and they are used by scosta for implementation of security features and storage of keys. EF1 stores passwords, EF2 stores keys and EF3 stores security environments. The contents of these three files controls the operations that can be done on other files under MF. This is called Global Security Environment. Likewise, each DF may contain it's own DEF1, DEF2 and DEF3 serving similar purposes. If such files are present and contain valid information, then the security environment defined by them shall override the security environment defined by the corresponding files under MF. EF4 in this example contains user data and it's contents are not interpreted by SCOSTA. Hence it is also called an external file.

While deploying applications on to smart cards, usually all the data belonging to an

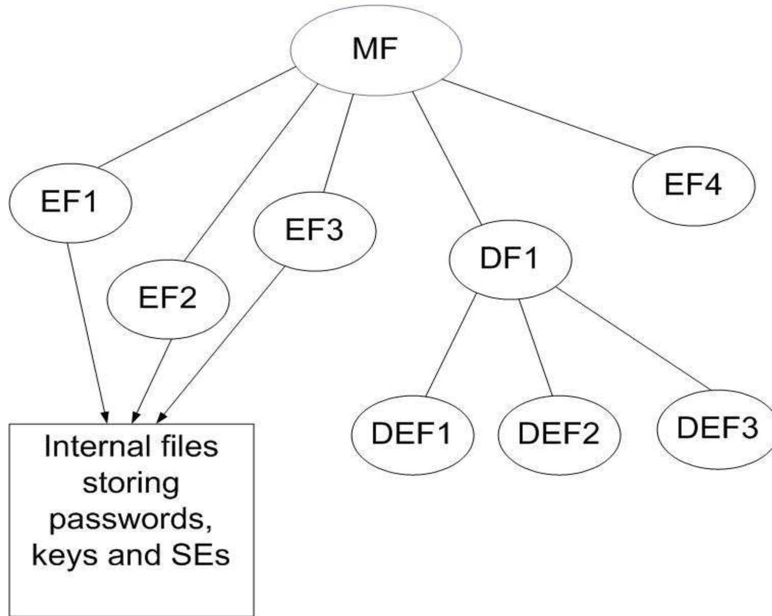


Figure 1: SCOSTA directory structure

application is put under a single DF. Then a security architecture customised for that application can be defined by the application provider and SCOSTA controls access to all the resources in that application as specified by the three internal files present in it's DF.

Cryptographic keys are present in EF2 in a DF as a set of records. The exact structure of each record and the metadata related to key is specified in SCOSTA Specifications 1.2b[11]. In essence, each key has a unique identifier. A command such as External Authentication shall specify the key identifier as a command parameter.

5 Design and Approach

The problem as a whole can be broadly divided into two components:

- A system of storing and retrieving keys when referred to, inside smart card. We call this component, Key Storage Subsystem.
- A collection of routines that implement the required encryption/decryption/sign procedures. We call this PKC subsystem.

5.1 Key Storage subsystem

In this subsection, we shall look at the first of the above two components - the Key Storage Subsystem.

Various keys need to be stored inside the smart card for it to be able to perform the operations it is supposed to. For instance, the card needs to have the private and public keys of the user holding the card. In addition, it may also contain public keys of several other users. These keys should be stored inside the card in appropriate place so that they can be retrieved easily when required. The commands refer to keys using their identifiers. Each key inside the card of a specific type(public or private) shall have a unique identifier. The earlier method of storing keys in special internal files has some limitations in this regard. The Key stores cannot be easily shared between applications. It also limits the total number of keys present. The International standard ISO/IEC 7816-15[4] or PKCS#15[6] describes a nice way to do this. Figure 2 shows an example. In this figure, under MF, there is a record EF called EF.DIR which contains list of mappings between an application (referring to a DF holding that application) and it's corresponding DF.CIA (CIA - Cryptographic Information Application). As mentioned earlier, different applications are usually present in different DFs under MF and each such DF shall have a DF name which acts like Application Identifier(AID). So, each record in EF.DIR links an AID with a DF.CIA. So, there can be multiple DF.CIAs in the same card. So different applications can use their own key stores. Also, using the same mechanism, two or more applications can share a key storage subsystem simply by mapping to same DF.CIA.

The DF.CIA contains a EF.CIAInfo - General Information file, and a EF.OD (Object Directory). The EF.OD contains links to EFs which then contain the actual keys themselves. The ASN.1 specifications of format and structure of this metadata about the keys are given in the standard ISO/IEC 7816-15[4]. So, while searching for a particular key, we have to parse these ASN.1 objects and extract out the required data. Before actually using the key, we have to cross check the intended operation against the attributes of the

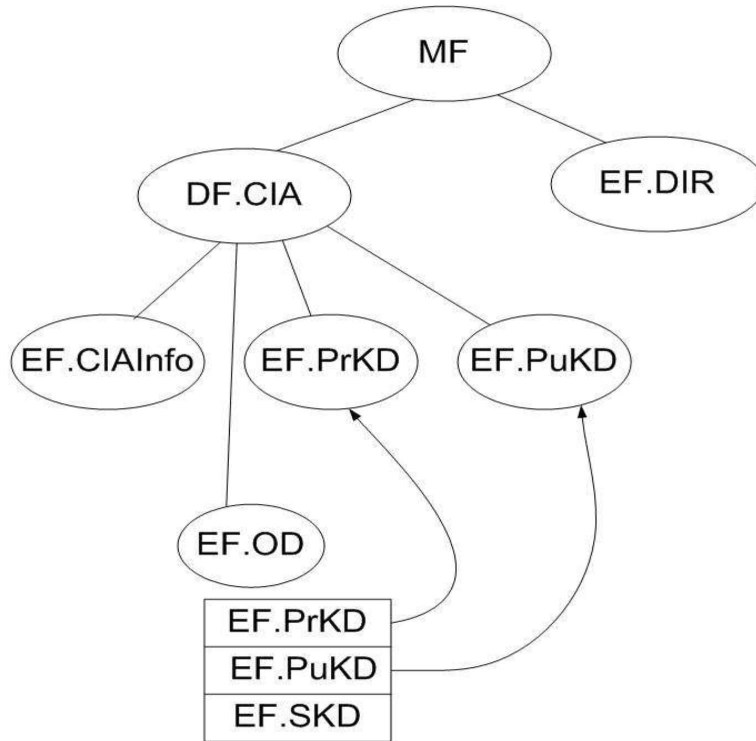


Figure 2: Key Storage subsystem

key. Some keys are allowed to be used only for specific purposes. For instance, a private key may not be used for encrypting data using ENCIPHER command. Similarly, a public key may not be used to compute a signature. Likewise, a key intended for External Authentication may not be used for Encrytion since it may lead to a reflection attack.

Given an identifier, to discover the location of the key, we use the following steps:

1. Get the Application Identifier (DF Name) of the current DF. If the current DF has no AID, get the AID of the parent of the current DF. Go upwards in the tree like this until we get a DF which has an AID or we reach MF. If MF also does not have AID, then use the older key stores (special internal files EF1, 2 and 3) otherwise proceed to step 2.

2. Once we got the AID, we parse the records in EF.DIR one by one to see if there is a mapping from this AID to a DF.CIA.

3. At this point, we know the path to DF.CIA to be used. We go inside this DF.CIA and search for the given key identifier. We don't have to search all the directories. Very often we know apriori whether we are looking for a public key or private key by

the operation intended. For instance, if the operation is to encrypt data or external authentication, then we can look only in Public Key Directory.

Once the location of key is discovered and all the checks are made, then we can use this key to perform the actual cryptographic operation.

5.2 PKC Subsystem

In this section, we shall see what are the different sub-components in PKI and how do we fit them in the existing SCOSTA architecture.

5.2.1 About P5CD036 microcontroller

Public key algorithms are computationally more intensive than algorithms that use symmetric keys. Given the limited computational capabilities of smart card processors, trying to implement them completely at the software layer takes us nowhere. They simply take too much time to complete. To facilitate the use of public key algorithms, modern day smart cards come with inbuilt coprocessors specialised in performing complex calculations faster. The P5CD036 microcontroller from NXP semiconductors is one such processor. This processor is equipped with 36 Kbytes of EEPROM, 160 Kbytes of ROM, and 4608 bytes of RAM which includes 1280 bytes of special purpose FXRAM usable by FameXE coprocessor[7]. The FameXE coprocessor can execute in parallel with the main processor and provides several instructions for doing all the basic mathematical operations required for public key algorithms.

5.2.2 High level design

Figure 3 describes the main sub-components of PKI in a layered structure. Layered structure is preferred to other designs because of it's portability. The arrows indicate the flow of command and response as observed from the user outside. As the figure indicates, there are three layers into which whole PKC subsystem can be divided into -

Command Interface Layer This layer takes in user commands and inputs and calls the appropriate routines from the Encoding/Decoding layer. This is the layer that interacts with the key storage subsystem to fetch the key that is referred to in the command.

Encoding/Decoding Layer This layer performs the actual encryption or decryption, signing or verification of the data. It performs necessary padding before passing the

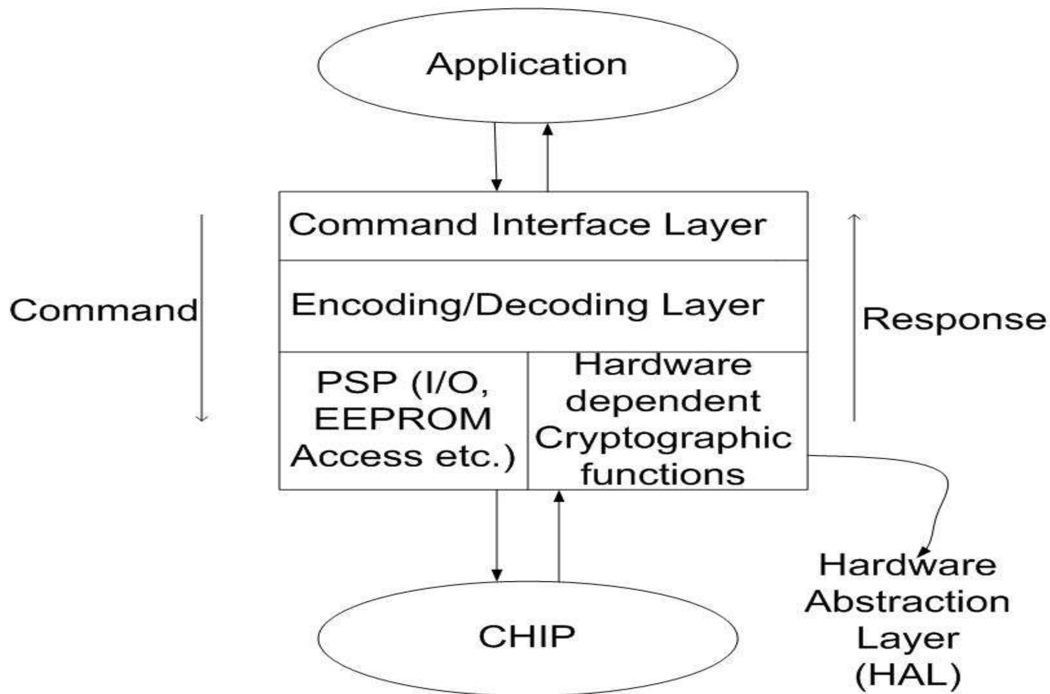


Figure 3: PKC subsystem - Design

padded message blocks to primitive mathematical functions at hardware abstraction layer. Also, it decodes a decrypted message to extract out the original message and returns it as output to the applications.

Hardware Abstraction Layer This constitutes a major part of the implementation. It consists of two parts - the Processor Specific Package(PSP) and Hardware dependant cryptographic functions(HDCF)(like modular exponentiation). The PSP is composed of functions for Input/Output using UART, functions for EEPROM access, random number generator, functions for getting chip serial number etc. While the HDCF consists of functions like Modular Exponentiation, Modular exponentiation using CRT, SHA1 hashing algorithm, etc. This is where processor specific code is written.

The following chapter explains in detail, the specifics of each of these layers and modules present in them.

6 Individual Modules

6.1 Hardware Abstraction Layer

HAL can be visualized as an interface between actual hardware and middle layer. It provides a set of interfaces that the middle layer functions can call and access the underlying hardware. The resources that are managed by HAL typically include EEPROM, I/O and Random Number Generator. HAL is also responsible for initializing the card when it is first reset. The set of functions mentioned until now, constitute a Processor Specific Package(PSP) in terminology used by SCOSTA. These are the minimum functionalities any HAL should provide. So, We need to implement a PSP for P5CD036 platform. That is not all, however. Since we want to use FameXE coprocessor to perform calculations, we need to implement a set of routines that help realizing these functionalities. For example, we want to have a routine for doing Modular Exponentiation which is common step in many public key calculations. The discussion about implementation of each of these Hardware dependant Cryptographic functions follow.

6.1.1 Modular Exponentiation

Modular Exponentiation is a key operation in many asymmetric key algorithms including RSA. The operation can be described by the so called Basic Equation (BEQ):

$$C = M^e \bmod N \tag{1}$$

N is called modulus, M the message, and e, the exponent. In case of Encryption, e is called public exponent while in decryption it is the secret exponent. Modular Exponentiation can be performed using what is called a "Straight forward method" or "Square and Multiply algorithm". The algorithm is given below.

SquareAndMultiply (M, e, N)

  initialize r = 1

  From lsb(e) to msb(e)

    if b == 1 then $r = r * M \bmod N$

    $M = M * M \bmod N$

  end

end

The FameXE coprocessor provides methods for basic calculations like multiplication with reduction, and squaring with reduction etc. So, we can use these methods to implement the modular exponentiation. However, there is a problem here. We need the following background to understand it.

Reduction method of J.J Quisquater:[9]

Reduction operation can be written as

$$X \bmod N = X - \left\lfloor \frac{X}{N} \right\rfloor .N \tag{2}$$

where z = $\left\lfloor \frac{X}{N} \right\rfloor$ is called the reduction factor. J.J. Quisquater proposed an efficient implementation of reduction operation. Assuming N to be very close to nth power of 2, an estimate of reduction factor can be obtained:
$z_1 = \left\lfloor \frac{X}{2^n} \right\rfloor$ where $N < 2^n$.
Now this z_1 can be calculated very efficiently since division by a power of 2 is nothing but a series of bit shift operations. Performing reduction with z_1 instead of z introduces an error in our calculation:

$$X \bmod N = X - z.N = X - z_1.N - (z - z_1).N \tag{3}$$

This error $(z - z_1).N$ can be subtracted from the obtained result $(X - z_1.N)$ to derive the desired result. But the main advantage of using quisquater method is that typically this error elimination is only done after large number of operations using N in a final one time post processing step. If N is chosen to be sufficiently close to a power of 2, then $z - z_1$ can be made to 1 and hence the post processing step of eliminating this error reduces to a single subtraction $X - N$ [8].

The FameXE coprocessor uses this technique for efficiency. Hence, before giving instructions to FameXE to do multiplication or squaring, we need to make sure that N is sufficiently close to power of 2. This is called Normal-

ization operation. In Quisquater's method, the reduction is given by[10]:

$$X \bmod N = X - \left\lfloor \frac{X}{2^{n+64}} \right\rfloor . \left\lfloor \frac{2^{n+64}}{N} \right\rfloor .N \tag{4}$$

So the normalized modulus is given by

$$N' = \left\lfloor \frac{2^{n+64}}{N} \right\rfloor .N \tag{5}$$

But FameXE demands this normalized modulus in it's 2's complement form. So, the modulus we have to give to FameXE for calculations is

$$\overline{N'} = 2^{n+64} - N' = 2^{n+64} \bmod N \tag{6}$$

where n is the number of bits in N.

So we need to do this normalisation to modulus N before the beginning of square and multiply algorithm. Also giving normalized modulus intead of original modulus forces another step at the end of modular exponentiation - the Denormalization. This is achieved by the following two steps[8]:

$$R'' = d.R' \bmod N \tag{7}$$

and

$$R = \frac{R''}{d} \tag{8}$$

where d is the denormalization factor obtained at the beginning while calculating $\overline{N'}$ and is given by

$$d = \frac{2^{n+64}}{N} \tag{9}$$

Now the square and multiply algorithm itself is performed by using memory in FXRAM. The following subsection explains the memory layout we used:

6.1.2 Memory Layout of FXRAM for Modular Exponentiation

P5CD036 has 1280 bytes of dedicated memory space for FameXE calculations. We divide this space into 4 chunks each of size 320 bytes and use them repeatedly to save both time and memory from general purpose RAM. The following table illustrates the usage of chunks in FXRAM.

Chunk 4 Temporary space for calculations
Chunk 3 Intermediate result
Chunk 2 Message
Chunk 1 Normalized modulus

We place the intermediate result r in each iteration in chunk 3. The final result will also be in chunk 3 after all the iterations are complete. Since neither the multiplication and squaring cannot occur in-place, temporary space is needed to store the result of both the operations temporarily before shifting it to it's original place.

6.1.3 Modular Exponentiation using Chinese Remainder Theorem (CRT)

Modular Exponentiation can also be done using chinese remainder theorem if the key (private key) is provided in the required format. In fact, PKCS#1-v2.1[5] requires it to be done using CRT if the key is provided in this format. So, we implemented Modular exponentiation using CRT as well. The memory layout for this operation is similar to that of square and multiply algorithm. Hence we skip the details here. Please refer to PKCS#1-v2.1[5] for the algorithm that uses chinese remainder theorem to calculate modular exponentiation.

6.1.4 Hashing algorithm

Any standard PKI should have at least one hashing algorithm implemented in it's arsenal. We use hashing as part of Computation of digital signature and also as part of encoding in PKCS#1-v2.1[5] encryption and decryption. Hence we implemented SHA1 algorithm. It take arbitrarily large amount of data and gives out fixed length output (160 bit).

6.1.5 Processor Dependant Cryptographic functions for linux

From the beginning of the scosta project, there has been a HAL implementation for linux. That is, a virtual card program in linux acts as smart card and interacts with SCOSTA just as a real card does. It is called SCOSTA.Linux. It is useful for robust testing of upper layers which are processor independent since testing on linux is much faster and straight forward when compared to testing on real card emulators. So, to continue this, we also implemented the Modular exponentiation for linux. This is achieved by using GNU Multiprecision Library (GMP) (http://gmplib.org/)using which we can do multiplications of numbers of arbitrary size.

6.2 Encoding/Decoding Layer

The Encoding/Decoding Layer is independent of any specific hardware architecture. It consists of a set of routines that are used by the Command Interface (the first layer) and it uses the primitives we defined in the HAL. The following is the list of routines implemented at this layer:

- RSAES-OAEP-Encrypt
- RSAES-OAEP-Decrypt
- RSAES-PKCS1-v15-Encrypt
- RSAES-PKCS1-v15-Decrypt

- RSASSA-PSS-Sign
- RSASSA-PSS-Verify
- EMSA-PSS-Encode
- EMSA-PSS-Verify

RSAES-OAEP-Encrypt and RSAES-OAEP-Decrypt call modular exponentiation after properly padding the input message or removing the padding respectively. The padding mechanism also requires a Hash function and a Mask Generation function. A random seed is used in padding to prevent the chance of two encryptions of the same message being same.

RSAES-PKCS1-v15-Encrypt and RSAES-PKCS1-v15-Decrypt are the older counterparts of above mentioned routines. They also use Modular exponentiation but the padding mechanism is much simpler.

RSASSS-PSS-Sign and RSASSA-PSS-Verify perform computation and verification of signatures. Again, at a basic level, these routines too use modular exponentiation. The encoding and decoding mechanisms of the message before signing or verifying are implemented in EMSA-PSS-Encode and EMSA-PSS-Verify respectively.

6.3 Command Interface layer

This is the upper most layer and is directly interacting with outside entity (a reader). The following table lists all the commands related to PKI we implemented, their inputs, outputs and the middle layer functions they will be using.

Operation	Inputs	Outputs	Middle layer function
ENCIPHER	plain text	cipher text	RSAES-OAEP-Encrypt or RSAES-PKCS1-v15-Encrypt
DECIPHER	cipher text	plain text	RSAES-OAEP-Decrypt or RSAES-PKCS1-v15-Decrypt
SIGN	Hash value to sign	signature	RSASSA-PSS-Sign
VERIFY	Hash value and its signature	YES or NO	RSASSA-PSS-Verify
EXT-AUTH	encrypted challenge	YES or NO	RSAES-OAEP-Decrypt or RSAES-PKCS1-v15-Decrypt
INT-AUTH	challenge	Encrypted challenge	RSAES-OAEP-Encrypt or RSAES-PKCS1-v15-Encrypt

In addition to just calling the middle layer functions, these routines have a lot of error checking to do. They have to check that input size is not beyond acceptable bounds, they have to contact the key storage subsystem to fetch the location of the keys required for the operation, they have to check that the security attributes mentioned in the key are satisfied by the operation. That is they have to make sure that this key can be used to do this operation.

7 Testing SCOSTA

A smart card operating system can never crash - the stakes are too high. If the system crashes while a money transaction is in progress, there can be unexpected results. Hence SCOSTA must be robust, and bug free. To facilitate this, we needed to speed up testing. Conventional methods of testing by giving individual commands is not enough because there are simply too many things to check for. Hence we used the following two strategies for testing:

- Test Processor independent parts by using SCOSTA.Linux

- Use STTool in conjuntion with Card reader interface board provided by philips.

7.1 Testing with SCOSTA.Linux

This kind of testing is not so efficient in the sense that it takes lot of time to recover bugs because we have to manually type all the commands for ourselves. But this is useful in quickly testing processor independent part of any specific command in the initial stages of debugging. Implementation of Processor specific implementations of complex operations like Modular exponentiation is difficult and time taking. So, once we implement the processor independent part(PIP), we don't have to wait for PSP to get completed before testing the PIP. Writing processor specific code for linux is trivial.

7.2 Testing with STTool

STTool is a testing software developed especially for SCOSTA. Figure 4 illustrates how STTool works:

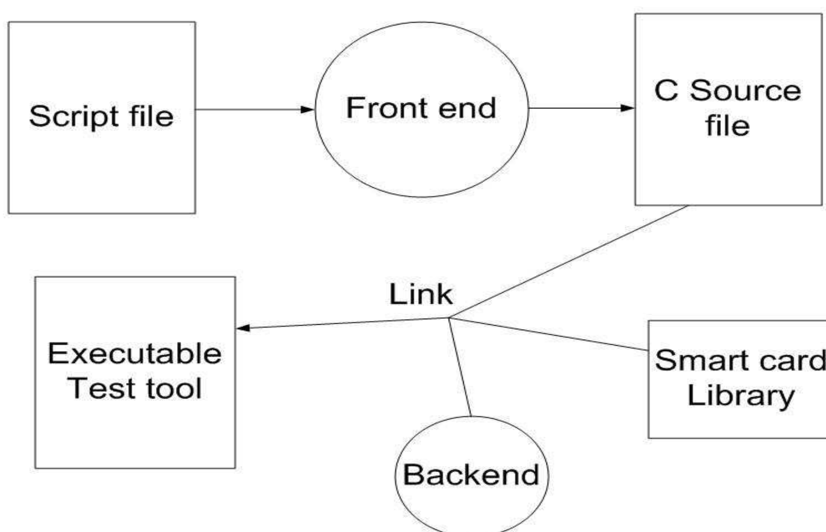


Figure 4: STTool components

The input file to STTool is a script file containing commands written in easily understandable language. The frontend is basically a compiler that converts the commands written in the scrip file into function calls with appropriate arguments. So, the output of frontend is a c source file. This file is then compiled and linked against the backend which contains actual implementation of these functions - translate these function calls into array of bytes that can be sent to the card and actually sends them to the card through smart card library (libpcsclite in linux and winscard.lib in windows). The tool also indicates the status of each command executed and properly logs all the commands and their components into an output file. It indicates if there are any failures. A failure in the log file means there is something wrong with SCOSTA that needs to be corrected.

The original code for STTool was developed many years ago. But we added many significant changes. Most significant of all, is the ability to provide expected data along with the commands. For those commands, which return some data, STTool shall now compare the actual output with the expected output and indicates if there is a mismatch. Also, earlier there were two different versions each for windows and linux. Now, we refined and modularised the code a bit in linux, and ported the same code to windows. So now the tool is made portable.

8 Results

At the end of this project, in summary, we have the following results:

- A working implementation of Key Storage subsystem.
- A working implementation of Modular exponentiation both for linux and P5CD036 platforms.
- A working implementation of SHA1 hashing algorithm for P5CD036.

- A working implementation of Middle layer - a set of routines as described in PKCS#1-v2.1 standard[5].
- A working implementation of command interface to the following commands:
  - ENCIPHER
  - DECIPHER
  - EXTERNAL AUTHENTICATION
  - INTERNAL AUTHENTICATION
  - COMPUTE DIGITAL SIGNATURE
  - VERIFY DIGITAL SIGNATURE
- A working implementation of improved STTool both for windows and linux

The following may be cited as advantages of our implementation over other existing ones:

- Completely modular and layered code. Easy maintainence.
- Custom implementations of primitives without using any external libraries gives extreme control over minute details like memory usage and speed.
- Since all the development is done in the context of SCOSTA, it merges extremely well with the existing architecture of SCOSTA and preserves all the good attributes of SCOSTA.
- SCOSTA is about to be deployed in most challenging applications. Our PKI implementation gives SCOSTA the power to deal with these upcoming applications.

9 Future work

The following items are either incomplete or needs improvement in future when compared with our original goals:

- The key storage subsystem is partially incompliant to ISO/IEC 7816-15 standard[4]. The algorithm identifiers and some small details are not standardised yet.
- We could not complete the implementation for storage and verification of digital certificates.
- The implementation of Modular Exponentiation on P5CD036 have been constrained keeping less memory usage in mind and hence, takes little more time than it should have taken. With the whole PKI, compiled into the code, maximum RAM usage at any time is only 1/3rd of the total available RAM. So, this trade off can be utilized to obtain a better balance between speed and memory usage.

10 Conclusion

We started out to implement Public Key Infrastructure in SCOSTA. Our major source of motivation was to make SCOSTA usable for large variety of applications that require PKIs built into OS. We divided the task into components and sub-components and identified individual modules to be implemented. We identified which primitive algorithms we need to implement and we have taken into consideration, the constraints we have - memory and speed limitations - in implementing those primitives. We were able to achieve almost all of our goals. Ultimately, we have a working PKI implementation in SCOSTA. We hope this work shall be embedded in future releases of SCOSTA and thus shall be put to use in many thousands of smart cards.

References

[1] *ISO/IEC 7816-4:1995(E), Information Technology – Identification cards – Integrated circuit(s) cards with contacts – Part 4: Interindustry commands for interchange*. First Edition. 1995-09-01.

[2] *ISO/IEC 7816-8:1999(E), Identification cards – Integrated circuit(s) cards with contacts – Part 8: Security related industry commands*. First Edition. 1999-10-01.

[3] *ISO/IEC 7816-9:2000(E), Identification cards – Integrated circuit(s) cards with contacts – Part 9: Additional interindustry commands and security attributes*. First Edition. 2000-09-01.

[4] *ISO/IEC 7816-15:2004(E), Identification cards – Integrated circuit(s) cards with contacts – Part 15: Cryptographic information applicaion.*

[5] RSA Laboratories, *PKCS#1 v2.1: RSA Cryptography Standard*. June 14, 2002.

[6] RSA Laboratories, *PKCS#15 v1.1: Cryptographic Token Information Syntax Standard*. June 6, 2000.

[7] NXP Semiconductors, *P5CD036 Secure Dual Interface PKI smart card microcontroller - Product Datasheet*. Edition 3.0.

[8] NXP Semiconductors, *User Manual, HSIS/UM0001 Secured Crypto Library for P8WE50XX Smart Card Controller Family*. Edition 1.0.

[9] J. J. Quisquater and C. Couvreur, *Fast Decipherment algorithm for RSA public key cryptosystem*. Electronics Letters. 18(21):905-907. October 1982.

[10] Jean-Francois DHEM, *Design of an efficient public key cryptography library for RISC-based smart cards*. PhD thesis. Université Catholique de Louvain. 1998.

[11] *Specifications for smart card operating system for transport applications (SCOSTA)*. 2002.