Securing Multi-Application Smart Cards by Security-by-Contract

Eduardo Lostal

Supervisor: Nicola Dragoni

Kongens Lyngby 2010
IMM-MSC-2010-55
Java card technology has progressed at the point of running Web servers and web clients on a smart card. Yet concrete deployment of multi-applications smart cards has remained extremely rare because the business model of the asynchronous download and update of applications by different parties requires the control of interactions among possible applications after the card has been fielded. Current security models and techniques do not support this type of evolution.

A promising approach to address this problem seems to be the Security-By-Contract (S×C) idea. S×C is a framework which makes mandatory that any post-issuance modification of the applications of the card comes bringing a specification of its security behavior, called contract. This must be compliant with the security policy of the multi-application smart card. Because of the constraint resources of these devices, the S×C approach can be applied at several level of abstractions, according to a defined hierarchy of models which provides benefits in terms of computational complexity or language expressivity. The most detailed one requires algorithms too complex to be executed on the card; hence, there is a need to send private data from the card to some Trusted Third Party which will be responsible for the comparison of the contract against the policy (process called Contract-Policy Matching) in order to decide whether the modification is compliant with the security policy or not.

The purpose of the project is to develop a system which addresses the problem of outsourcing the Contract-Policy Matching to an off-card entity for Java multi-application smart cards. It has to guarantee a secure communication between the card and some Trusted Third Party through an untrusted environment.
This communication has to be secure in terms of authentication, integrity and confidentiality. Achieving that goal will require to solve several problems, such as identities and keys management and use of cryptographic functions to make communication of the private data stored on the smart card secure.
This thesis was developed at DTU Informatics, Technical University of Denmark, in partial fulfillment of the requirements for acquiring the M.Sc. degree in engineering at Centro Politécnico Superior, University of Zaragoza (Spain).

This thesis deals with the problem of dynamic management post-issuance of multi-application smart cards. The main focus is on security and cryptographic smart card domains to get a secure exchange of data from the card to some server which plays as a trusted third party to perform operations required in the Security by Contract framework. Specifically, identities and keys management, encryption and decryption services are considered.

Preliminary results of the thesis have been accepted for publication at The Fourth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (Ubicomm) 2010:


Lyngby, August 2010

Eduardo Lostal
Structure of this Report

The report is organized as is shown in Figure 1. It consists of: background, problem to be solved, developed solution and conclusion.

- Background is a self-contained introduction about smart cards technology, its security and multi-application approach. It is divided into two chapters.

- The problem chapter is a description of the emerging problem with open multi-application smart cards. It also introduces a state-of-the-art and the Security-by Contract framework as a theoretical solution for the problem.

- The solution is explained in three chapters. The first one shows the design of the solution for the concrete problem to solve during the thesis, it provides the architecture of the system. The second chapter details the construction of the prototype which demonstrates the validity of the previous design. The last part is a lightweight manual of the prototype presented as a case study together with a memory analysis in order to introduce some result of the performance.

- The remaining chapters present conclusions and future work.
Figure 1: Structure of this report
Acknowledgements

The current thesis means (hopefully) to finish with the requirements to acquire my degree in Computer Science. It has been a long way with obstacles, study, work... but always full of happiness. Despite the million of hours in the library of "Filología" with its ceiling falling to pieces (literally), last years have been awesome. I cannot thank every body who I liked, because even with the whole pages of this report I would not have enough. So, let’s sum up.

In first time, I would like to thank my family. You always provide me the support that I need. Without you I would not be the person who I am. Both of us know our differences, but what you do not know is as thankful as I am to you. Pequeñaja, you are also included in the family. I cannot imagine how would have been these years without you. You are always supporting and getting a smile out of me.

Another person to whom I am deeply thankful are you Nicola. Thank you for the chance of working with you and for all the trust that you have put in me. This thesis has been very fruitful in a lot of ways and you are quite responsible for that. As my Spanish supervisor thanks Javi for your support. I know you are very busy, but you always get some time for me. Our Skype talks were very funny. Thanks also to Davide for your help, discussions and suggestions. Enjoy Firenze e molto grazie!

Thanks Lawrie for our discussions; I am sure if we had met later, they would have been more fruitful. Any way, it was nice to meet you. Also, thanks Olaf for your willingness, although parser’s structure was almost implemented when I received your reply. I will send you a copy of the report, I hope you find
it interesting! Also thanks to Java Card forum and the whole authors of the papers included in the references.

Thanks to Tingbjerg people (and adopted as that). I really have no words to describe as amazing as this year has been with you. I think we have shared something special and I am sure that I will miss you a lot. Thanks to colorines for bringing me light in moments of darkness... or it was at the reverse, I am not sure.

Thanks to my old friends, I miss you but I keep you in my mind yet. Thanks to the AMZ, Malabarí and juggling people in general; you do not know how much you have changed my point of view in these years.

If you are reading these lines means that either I know you or you are interested in my work, any way, thank you!

Smile, you and the world are better!
Contents

Summary i

Preface iii

Structure of this Report v

Acknowledgements vii

1 Smart Card Technology 1

1.1 Types 3
1.2 Different Types of Smart Card Communication Interfaces 4
1.3 Hardware 6
1.4 Smart Card’s Applications 8
1.5 Security in Smart Cards 9

2 Multi-Application Smart Cards 17

2.1 Main Standards 18
2.2 Open Multi-Application Smart Cards 24

3 Security-By-Contract for Multi-Application Smart Cards 27

3.1 Mobile Code Security 28
3.2 Security-by-Contract 29

4 Extending SxC for Off-card Contract-Policy Matching 37

4.1 Confidentiality and NONCE Issue 39
4.2 Architecture of the System 40
## List of Figures

1. Structure of this report .................................................. vi
   1.1 Contacts of a common contact smart card ..................... 4
   1.2 Contactless smart card ............................................. 5
   1.3 Physical view of a smart card chip ............................. 6
   1.4 Some applications of smart cards ............................... 10
   1.5 Physical view for two chips; both insecure (left) and secure (right) 12

2. Architecture for multi-application smart cards ................... 18
   2.1 Firewall containment ............................................. 20
   2.2 Split Java Card Virtual Machine ............................... 21
   2.3 Physical and logical layers of Java Card 3 Connected Edition 21
   2.4 Possible command APDU structures ............................. 23
   2.5 Response APDU ................................................... 23

3. 5×C Workflow .......................................................... 31
   3.1 Off-Card 5×C Contract-Policy Matching ....................... 34

4. Smart card’s communication by APDU ............................... 38
   4.1 Initialization Phase: CSR’s building diagram ................ 42
   4.2 Initialization Phase: Certificates issuing diagram .......... 44
   4.3 Initialization Phase ............................................. 45
   4.4 Contract Storage Phase .......................................... 46
   4.5 Contract-Policy Matching Phase ............................... 47

5. CSR’s building process on the prototype ......................... 63
   5.1 Signature verification process on the prototype ............ 71

6. New Card script running ............................................... 77
LIST OF FIGURES

6.2 Deploying applet.............................................. 77
6.3 Running Initialization Phase from Eclipse..................... 78
6.4 Initialization Phase Menu on Eclipse.......................... 78
6.5 Certificate generation with OpenSSL.......................... 79
6.6 OpenSSL\bin folder after certificates generation............. 80
6.7 Certificates storage on-card.................................. 80
6.8 Message when ApplicationContract has not been stored...... 81
6.9 Contract storage process....................................... 81
6.10 Message with the matching result............................ 82
6.11 Contract storage process...................................... 82
6.12 EEPROM and ROM content..................................... 84

7.1 Solution to validity period problem diagram.................. 89
7.2 Initialization phase with TR’s certificate...................... 90
7.3 Diagram for certificates renewal............................. 90

D.1 Analysis before and after of the deployment................ 104
D.2 Analysis before and after of the installation............... 104
D.3 Analysis before and after of the initialization............. 105
D.4 Analysis before and after of running the matching algorithm. 105
List of Tables

5.1 Equivalence among entities, projects and source codes . . . . . 60
6.1 Memory Analysis statics in bytes . . . . . . . . . . . . . . . . 85
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>OpenSSL commands to create a key pair usable by the system</td>
<td>66</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of X.509 certificate contents (Certificate used by smart card to encrypt)</td>
<td>68</td>
</tr>
<tr>
<td>5.3</td>
<td>Code to OID generation (sha1WithRSAEncryption example)</td>
<td>70</td>
</tr>
<tr>
<td>5.4</td>
<td>Code to Garbage collector over demand</td>
<td>74</td>
</tr>
<tr>
<td>6.1</td>
<td>Script to create, install and initialize a new card</td>
<td>76</td>
</tr>
<tr>
<td>6.2</td>
<td>Script to load the status of an already existent card</td>
<td>76</td>
</tr>
<tr>
<td>6.3</td>
<td>Command to create a CA certificate and private key</td>
<td>78</td>
</tr>
<tr>
<td>6.4</td>
<td>Commands to get certificates from CSRs</td>
<td>79</td>
</tr>
<tr>
<td>A.1</td>
<td>ASN.1 specification for Certification Signing Request</td>
<td>93</td>
</tr>
<tr>
<td>A.2</td>
<td>Alternative ASN.1 specification for Certification Signing Request</td>
<td>94</td>
</tr>
<tr>
<td>A.3</td>
<td>ASN.1 specification for Certification Request Info</td>
<td>94</td>
</tr>
<tr>
<td>A.4</td>
<td>ASN.1 specification for Certificate</td>
<td>94</td>
</tr>
<tr>
<td>A.5</td>
<td>ASN.1 specification for Certificate Info part</td>
<td>95</td>
</tr>
<tr>
<td>A.6</td>
<td>ASN.1 specification for digestInfo structure</td>
<td>96</td>
</tr>
</tbody>
</table>
Chapter 1

Smart Card Technology

The big explosion of the digital communication produced in the last years has done that the way to interact among the people had changed. That is why organizations are moving towards network communications ways because of the requirements for their information was available and secure at the same time. Many of them have realized of the advantages and benefits that smart cards are offering.

A smart card is a device able to store data, carry out functions on itself and interact in an intelligent way with an external reader by means of a microprocessor embedded in the plastic card. It is a widespread device thanks to its easiness of use, portability and cheap price [27]. Smart cards are tamper-resistant and provide security features what makes them to be considered as a secure and trusted device. These features allow smart card to be used in fields where high-security is needed and suitable to cryptographic systems and operations like authentication and authorization.

Its origins are dated back to plastic cards which were started to deploy in US in the early 1950s for payment applications. At the beginning, they were made of paper moving later to PVC to get a longer lifetime. VISA and Mastercard appeared at this moment. Magnetic stripe cards were introduced afterwards, mainly because of the fraud. They were able to store digital data. Nevertheless, magnetic stripes had an important weakness: everyone with the right equipment
could read, modify or delete the data, or even make a copy of the card bit by bit (technique called Card Skimming). Thus, sensitive data could not be stored on the card. To check them consisted in accessing online to a server or host with confidential data. This is as smart cards appeared. Two German inventors applied for its patent in 1968, a Japanese in 1970 and a French in 1974. In 1984 in France, telephone cards were distributed as a trial and the first bank card, which incorporated safety cryptographic keys and algorithms, was introduced [39]. Some authors argue that development of smart cards was slow at the beginning because of all information about smart cards was tried to keep withheld to avoid becoming an unsecure device [17].

At the beginning, development process was very long and complex. After choosing the chip (first step), code was written. This code was a mix between C and assembly in order to optimize. Developers had to manage for themselves memory pages. Once compiled, binary code was added to Operating System (OS) binary code and test on an emulator. In fact, OS was a few low level primitives used to managed the chip. Next step was to send it to Integrated Circuit Chip (ICC) manufacturer. After that, no change or update could be done, due to it was considered a possible weakness to attack. Masking process built Read-Only-Memory (ROM) on the wafer. Since process was very slow, manufacturers proposed generic soft masks which included a limited OS with some functionalities. Developer left OS issue aside and used Application Programming Interfaces (APIs). That made the process easier and faster: application was compiled and its binary was added to the soft mask to be sent to ICC manufacturer. Basic-Card apparition was the key point for the subsequent growth in the mid of 90s. It allows developer to program a smart card without knowing anything of low level and architecture. It was compiled in P-codes which were interpreted on the card in run-time. Tools and libraries (questioned about their security features) were available to developers and price per card was very cheap [44]. After that, Java Card birth opened a new perspective of possibilities for developing smart card applications, including multi-application smart card. From the last years to date, smart card technology has still not stopped of growing.

Smart card, to ensure their worldwide functionality, must be compliant with the international standards, particularly with ISO/IEC 7816 (International Organization for Standardization/International Electrotechnical Commission), indeed an extension from ISO 7810, which defines physical characteristics, dimensions and location of the contacts, electrical interface and transmission protocols, robustness of the card and functionality among other aspects for smart cards. This standard has been updated adding some other specifications like Cryptographic information application. More information could be found in [13].
1.1 Types

Several types of smart cards exist whose suitability depends on which was the application field. They are sorted by increasing cost. Features are better as higher as the cost is.

1.1.1 Memory Cards

These cards have some security features and logic to control the access to the memory (e.g., usually Personal Identification Number (PIN) code is required to access data). They are very common and cheap, but also less functional. The memory is made up of Electrically Erasable Programmable Read-Only-Memory (EEPROM), or Flash memory in some cases, and ROM. That allows card issuer to be able to implement more complex applications and modify and update data stored if necessary. Due to its simplicity, usage is suitable to simple task like pre paid cards (e.g., transport, phone).

Depending on the literature, cards described at this section are distinguished in two types: with logic, which have been detailed at the previous paragraph; and without it. Latter are the simplest and cheapest cards, and usually, useless after being used. For instance, pre-paid phone cards (not rechargeable).

1.1.2 Microprocessor Cards

The main distinct feature of these cards is that they have incorporated a microprocessor which allows them to carry out more complicated operations and functions. In contrast to memory cards, they have also included a Random Access Memory (RAM) which provides volatile memory. To take advantage of the microprocessor, these cards have their own OS. It has to handle file manipulation and data transmission and to manage the memory. Microprocessor card’s security features have been improved from memory cards and allow multi-application. These cards are used in high-security required applications; that is why they will be the study object of the current thesis.
1.2 Different Types of Smart Card Communication Interfaces

Different attacks might be tried against the card according to the communication interface. That is why the following classification is done according to this feature.

1.2.1 Contact Smart Cards

These cards need to be inserted into the Card Acceptance Device (CAD) in order to be powered on. CAD is the interface placed between the host (off-card side) and the card. Contact cards have several gold-plated pads in the contact area, which are in physical contact with the reader when the card is inserted in. At Figure 1.1 common smart card contacts show up.

![Contacts of a common contact smart card](image)

Figure 1.1: Contacts of a common contact smart card

The aim of every contact is the following [27], [39]:

- VCC: Supply voltage.
- RST: Reset input. Used itself or in combination with an internal reset.
- CLK: Clock signal input. It should be external, since the card is only powered on when it is put into a CAD.
- GND: Ground.
- VPP: External programming voltage input (depreciated, no longer used).
- I/O: Input/output for serial data communications.

Two remaining contacts are reserved to the manufacturer or card issuer; they are auxiliary contacts and will be used according future application’s needs.
1.2 Different Types of Smart Card Communication Interfaces

Thereby, some manufacturers build cards with the six useful contacts, instead of the eight.

VPP contact was used in 1960s to supply voltage to program and erase EEPROM, but since 1990s this voltage is supplied by a charge pump that exists on the chip. This contact cannot be used to a distinct goal because it would be not compliant with the ISO standard.

A drawback is that contacts might start to fail because of being worn out. That might be caused by electrostatic discharges, card tearing because user pulls out the card earlier, etc.

1.2.2 Contactless Smart Cards

In contrast to contact smart cards, communication is done over-the-air thanks to an antenna glued inside the plastic body of the card, as Figure 1.2 shows. The antenna communicates with the reader through a Radio Frequency interface. When the card is placed into a card reader’s electromagnetic field, both power supply and communication are carrying out [27].

They are used in applications which need to make quick and secure transactions.

Two types can be distinguished [22]; vicinity cards, which offer a maximum read distance between 1 and 1.5 meters, and proximity cards, with a range around ten centimeters. Both standards can be checked at [13].

Figure 1.2: Contactless smart card
Contactless smart cards are not inserted in any reader. Therefore, their contacts are not worn out with usage; i.e., they have a longer life. In addition, as the chip is within the plastic body, it is more protected. Thus, they are more reliable than contact smart cards, but more expensive.

An advantage of these cards from card issuer’s business point, it is that microprocessor is not visible (there is no contact area to insert into the reader) which allows card issuer to print whatever it want on the card \[39\], for instance the company logo.

### 1.2.3 Dual Interface Cards

They are cards whose communication can be established either contact or contactless mechanism what allows using the card in both ways. These cards are designed to work in more than one application.

### 1.3 Hardware

The appearance of a common smart card chip is as the shown in Figure \[1.3\]. The main parts (microprocessor, memories and coprocessors) can be distinguished easily.

![Physical view of a smart card chip](image)

Figure 1.3: Physical view of a smart card chip

### 1.3.1 Microprocessor

It is the most important part of the card. Usually they are not specially built with the aim of being used in a smart card. The main reasons are money and
security. Money because to develop a new processor is expensive. Security because smart cards need to be reliable; a well-known processor which has been in the market for a long time and correctly tested is more trustworthy than a new processor.

Traditional processors in smart card were based on 8-bit Complex Instruction Set Computer (CISC) architecture built on Motorola 6805 or Intel 8051 core, with an extension to the instruction set. As the smart card evolved, embedded processors started to be based on 32-bit Reduced Instruction Set Computer (RISC) architectures in order to be able to support new needs like Java Card requires, for instance [27].

### 1.3.2 Memory

Memory is divided in three parts: RAM, EEPROM and ROM.

RAM is used to hold temporary values, i.e., volatile memory, due to their content is erased every time that the card is powered off. EEPROM holds data which can be modified during the lifetime of the card, but which is stored across the card restart events: mainly cardholder data such as keys or passwords. ROM is used to hold the programs which run on the card. ROM’s contents cannot be modified and they are stored for the whole lifetime of the card. Both EEPROM and ROM are non-volatile (or persistent) memory.

Because of every type of memory uses a different amount of space per bit, they are added according to this constraint, trying to use ROM wherever was possible, later EEPROM and finally RAM.

### 1.3.3 Coprocessors

Processors used in smart cards are not very powerful; that is why some supplementary hardware is added to the chip to carry out particular tasks which cannot be satisfied easily by software. They are called coprocessors. This is true for 8-bit architectures, but in 32-bit architectures is no longer necessary, because as coprocessors are added as the price of the chip increases [27]. Consequently, manufacturers are who decide which coprocessors have to be added according to their needs. Particularly, two common and important coprocessors related to cryptography are added:
Coprocessor for cryptographic algorithms calculate each respective algorithm by hardware. For instance, since Data Encryption Standard (DES) and Advanced Encryption Standard (AES), which are algorithms which make bitwise and bitshift operations, are usually used in financial systems, both of them are available as coprocessors. In fact, DES processor is specially fast and small. In order to facilitate the running of Public Key Infrastructure (PKI), coprocessors working with large number exponentiation and modulo operations are also built. For instance, to Rivest, Shamir and Adleman (RSA) or elliptic curves algorithms.

Random-number generator is currently added in the most of smart cards to help PKI, since significant importance of the randomness in current cryptographic systems. A weak random key might cause a key was compromised, that is why their correct operation is critical. Although it can be enhanced by software (and it is usually done), coprocessor must generate enough randomness.

1.4 Smart Card’s Applications

Thanks to its security features a lot of companies have chosen to use a smart card as trusted part to give to their customer, that is the reason why smart card have been widespread in different fields. The most common fields can be looked at Figure 1.4 (including some of the most remarkable examples) and they are briefly summarized at the following; however, it may expect that they will be deployed in more kind of applications in a future.

Telecommunication The most known are the Subscriber Identity Model (SIM) cards which almost everybody has in their mobile phone. These cards are being improved with a lot of new features; even Hypertext Transfer Protocol (HTTP) requests are already allowed. It is also worth mentioning the almost extinct pre-charge cards for phoning.

Banking Nowadays, credit and debit card are widespread. Some well-known examples are Mastercard or Visa cards.

Health care The most well-known example is Sesam Vitale, a system deployed in France to pass from paper-based payments to reimburse money to electronic transactions. Some features were added afterwards like storing medical records and prescriptions. Also, Versichertenkarte in Germany is known.

Audiovisual industry Some pay per view channels and digital TV need a smart card to work as a decoder.
Transport A big amount of contactless smart card have been deployed in public transport industry used to pay (fast ticketing). Some examples are Oyster Card in London and T-money in South Korea.

Access control Smart cards are used to restrict physical access to only authorized staff in a lot of companies and buildings which need to limit the access inside them. For instance, in the university, it is used to control who is able to go to the library, departments, etc. It has not to be only for physical access, but also for some resources of a company.

Identification The best examples are electronic passports and national ID cards. Usage of smart cards in this field is very useful, if it is taken in account the possibility to keep safe either secret keys or certificates on the card. At this way, to forge these kind of documents is much harder and the authentication of the cardholder is ensured.

Authentication Since smart card is considered a safe token, it can be used in some applications as an authentication way on the web, for example.

Loyalty applications For instance, it is used in programs in which customer is earning points for using a service and receive a reward later (e.g. petrol station or mobile phone bill loyalty strategies).

Pocket-gamming An example is smart cards used in some casinos to be able to have cashless machines, with the according benefits for the casino [45].

E-services With the new smart cards which support TCP/IP, a new perspective is opened to smart card industry. Internet is suitable to be used in smart cards due to their capabilities to store and work with cryptographic keys. The most important e-services are e-commerce, remote banking and working.

Grid computing Some researchers have used smart cards as secure computing nodes.

1.5 Security in Smart Cards

One of the highlights which have made smart cards so widespread are their security features. Very often a smart card is used as a security solution with the aim of providing tamper-resistant functionality, in systems which provide some kind of security duties. At this point, it is worth mentioning that a smart card can be used as a secure token, but it does not guarantee the system where card was added was a secure system. One of the principles of the security says the
What, in this case, means that to use a smart card not ensures the security of the whole system, as it is exposed at [22].

Smart cards solve security problems like secure cryptographic keys storage and distribution, authentication implementation or data confidentiality. They seem to provide a portable and flexible computing platform due to the use of a limited interface which does not allow attackers to access protected information [5]. As far as PKI systems concern, smart cards are particularly suitable. This is why if private key never leaves the card, to compromise it from a card reader or a PC is very difficult. Consequently, some services which PKI provides like secure email access, encrypted files, or digitally signatures, fit very well to be used through smart cards systems. Also, private keys which the card stores are protected within a security perimeter of hardware token which avoid card from being used by someone who does not know the PIN [39]. In other words, they provide hardware non repudiation, due to it is unlikely that someone was able to access to the private key and pass for the cardholder.

This section gives a brief overview of the types of attacks against smart cards following a similar structure than [27]. As it has been pointed out in a previous section, this structure is done partially according to the interface which the card used.
1.5 Security in Smart Cards

1.5.1 Physical Attacks against Contact Smart Cards

These kinds of attacks are carrying out over the hardware either gaining physical access to the microprocessor or through observation of the token. Theoretically, both of them can reach the target of getting information about the card and their functionalities and compromising their security measures. This topic is widely explained at [5] by means of introducing some examples, and mainly at [23] and [52].

1.5.1.1 Invasive Attacks

Executing these attacks means to get physical access to the microprocessor, or in other words remove it from the smart card, in order to inspect and try to tamper it. The whole process requires expensive equipment to remove the microprocessor and work within it, a high ability and long time to reach results. That is the reason why they are not very common except among the manufacturers companies, laboratories and perhaps some researchers. These attacks are focused in attacks to bus lines, EEPROM, RAM and ROM and reverse engineering.

It is possible to place a probe on bus lines between blocks of a chip and by means of an oscilloscope seeing the values sent on aforesaid lines. To avoid buses being through probed, they are usually placed in the lower layers of the device and scrambled in a specific way.

Either scrambling non-volatile memory cells is a way to prevent similar attacks against EEPROM. By hardware, is easy, efficient in space and very hard to break by an attacker. By software, it is harder to program but it can be made chip-specific and dynamic [27].

In what RAM’s concerned, its content can be held in the memory if their cells are cooled to a temperature of -60 °C or if the content is not changed for a long time. Hence, on one hand, secret keys should not be kept in the memory for more than the elapsed time. On the other hand, RAM can be protected with a metal layer, which elimination leads to useless memory cells.

On one hand, EEPROM and RAM data are encrypted in real-time (even by using memory address, that is, cipher text is distinct for the same plaintext and different address) in order to be protected. This feature is available in newest microcontroller. On the other hand, smart card manufacturers use ion-implanted ROM to prevent this memory from being read bit by bit using an
optical microscope.

One of the most important invasive attacks is the reverse engineering. The goal is get to know how a chip works in order to be able to copy the design with the corresponding losses to the owner. Also weaknesses on the chip can be found. These attacks are usually carrying out by people inside to the smart card industry who can access the necessary equipment to make the attack and are the most interested in getting information about the competitors to have competitive advantage. Consequently, several countermeasures are usually implemented on the chips. The most important are glue logic, obfuscated logic and adding another metal layer. One known example of reverse engineering which got their goal was MIFARE. The proprietary cryptographic algorithm was broken by reverse engineering compromising the MIFARE chips which used this algorithm [24].

![Figure 1.5: Physical view for two chips; both insecure (left) and secure (right)](image)

As it can be seen at Figure 1.5 (got from [27]), a secure chip looks like the right one. It implements several of the countermeasures aforementioned to prevent the physical attacks against the chip. Some of them are glue logic and a new metal layer which makes access to the chip difficult.

### 1.5.1.2 Side-Channel Attacks

They work observing the changes in a smart card when it processes different information and deals with unexpected effects. The main techniques are timing, power and electromagnetic analysis and fault induction attacks.

Timing analysis, as its name suggests, consists in checking the time of operations to get conclusions and make inferences from it. To avoid attacks over:
1.5 Security in Smart Cards

- PIN code, the whole bytes are compared
- Cryptographic keys, current cards use noise-free cryptographic algorithms

Power analysis is the technique which observes the power consumption over the time (for instance adding a resistor in series with card and checking the voltage change) and gives information about which operation is carrying out the microprocessor. There are two ways to do it: Simple (SPA) and Differential (DPA). Simple looks for several patterns in the consumption. For example, the rounds in AES cipher or square and multiply algorithms in RSA. If power consumption can change due to the value of the keys used, an attacker might get these keys \[22\]. DPA analyzes statically the samples. It is one of the most important attacks due to carrying it out is relatively cheap and the consumption is yet dependent to the data and the microprocessor. Some countermeasures include either to add a voltage regulator or artificial noise generator, use random delays or an on-chip random generator which varies the frequency of the clock. It is also possible to use machine instructions with similar consumption or have prepared different algorithms to solve the same problem. Patented countermeasures against this kind of attacks can be read at Cryptography Research, Inc. \[42\].

An electromagnetic analysis can be carried out by means of the electromagnetic radiation of the chip. It is hard to get success with it because it is necessary the chip has to be close enough to get a properly signal. To prevent chips from this attack, a metal shields can be added or some layers stacked on top of it. Because of this attack is related to power analysis attack, their countermeasures help to protect against this one also.

Fault induction attacks try to produce some effect, which attackers can take advantage from, after injecting a fault. Some of them are focused in the movement of the silicon atoms which might provoke resetting data, data randomization or modification of operation codes. Shielding and scrambling wires on the chip help to make difficult to access the important parts.

1.5.2 Security of Contactless Smart Cards

Any possible attack to a contact smart card, it is possible to be carried out in a contactless smart card, and therefore, the attacks and their countermeasures explained in the previous sections are valid for these kind of cards too. One contactless smart card drawback is that as they communicate over the air, everyone who wants to carry out an attack, only needs a radio device (ready to
work at a short range) to try it against a card placed some meters far from the device. Some attacks are the following.

Due to communication is over the air, it is easy to try an eavesdropping attack without the cardholder realized that someone is accessing to its card. It is difficult to get private data, but all the communications should make encrypted to avoid problems. In fact, in order to avoid a man-in-the-middle attack which would be able to altering the data transmitted, data should not be only encrypted, but with certain randomness. That assures data integrity [27].

Denial of service is one of the hardest attacks to fight against because any countermeasure works properly for contactless smart card.

Finally, other important attack is radio frequency analysis, which is made up of a power and electromagnetic analysis. It is not necessary to have physical access because data is taken from radio field emitted. This attack is easier to carry out and harder to detect. It may prevent by means of a current stabilizer and message and exponents randomization.

1.5.3 Anomaly Monitors

Some parts of the smart card might be affected by changes in their environment when they do the conditions go too extreme [27]. That is the reason why some anomaly monitors are added. They expect the card works in a range of acceptable conditions. When these conditions change and go to an unacceptable value, they (monitors) notice it and the chip stops running up to conditions come back to be suitable. They need space on the chip, so usually not all of the following monitors are implemented on the same card; it depends on what is the task for.

Voltage monitor is used in the most of current cards because it can avoid some attacks to get information as secret keys by Differential Fault Analysis (to know more [9]). When it detects that the voltage is working over the upper or down the lower limit, it informs the smart card in order to stop running. To avoid monitor from being disabled by an attacker, they are protected in such a way that its manipulation stops the card. Another monitor always placed on the chip is a power on detector which leads the chip to an initial state when it detects a power on condition.

The clock signal is provided by an external clock (because of the lack of power supply). The frequency does not depend on the card then, but on the supplier.
Hence, a frequency monitoring may be added. It orders the card to stop when under or over frequency values appear.

As far as temperature concerns it is a controversial topic because; on one hand, it may change out of the specifications without being an attack. On the other hand, an attack modifying the temperature can lead to manipulate the random-number generator ([27]). That is why temperature sensors are discussed by the experts, and also, because of deactivating the chip when it is working in a temperature out of the specifications can increase failure rate.

### 1.5.4 Software Attacks

In addition to the normal bugs which can be found in any software development, it is worth mentioning the following aspects.

To do the verification of the bytecode is a task too expensive to run on the card, so it is not possible to perform it over the microprocessor of chip. Therefore, malicious applications built over illegal bytecode instructions is a problem to be solved. Otherwise, these kind of constructions can allow an attacker to get the total control of the card in the worst case.

It should be taken into account the parties which participate in the communication between the smart card and the host. Moreover the smart card, either the link layer or the host can be attacked. That is why the design of the system and a correct use of the cryptography are necessary to develop a trusted system. An example of how the communication may be compromised by a man-in-the-middle attack could be read at [6]. More information about software attacks in [12].
As its name suggest, a multi-application smart card is a smart card which can host several applications. After the huge development in 90’s, a lot of companies chose to use smart cards as a security solution and normal people used to go with several of these cards everyday. The idea is to be able to use the same card for the whole needs of the user instead of one card for each need; that is, to reduce the number of cards in the user’s wallet. Nevertheless, the most important reason to work in that topic is the needs of the issuer card companies of being able to offer more services in the same card. The best example is a SIM card which currently has been improved up to provide to end-user new services. It was necessary to improve the cards to get the flexibility and the security to make the market’s needs possible.

The most interesting issue in multi-application is the chance to allow the dynamic application load. That means to be able to add new, update or remove the already loaded applications on the card whenever the user wanted. For instance, if a customer wishes to change their bank account from one bank to other, the customer would be able to remove the application of the old bank and install application from the newer. All process may be done quickly, instead of canceling the old credit card and ordering a new one to the current bank. Another example about update operation is the following. Let’s suppose a worker who uses a smart card as an access control card in his job. Some day, he is
promoted, and then, has access to a new laboratory with a stronger security
measures. Application in the smart card needs to update itself to adapt to the
new algorithm to check the identity and permissions or update the previous
content.

The architecture of a multi-application smart card is shown at Figure 2.1. As it
can be seen, applications run over the OS which interprets the instructions to
do them irrespective from the hardware. This independence is achieved through
a virtual machine, for instance Java Card uses Java Card Virtual Machine (the
acronym JVCM will be used in the remain) which is similar to the Java Virtual
Machine.

![Architecture for multi-application smart cards](image)

Figure 2.1: Architecture for multi-application smart cards

### 2.1 Main Standards

The two main standards are MULTOS and Java Card which are explained below.
Some more are Smartcard.NET (to support .Net framework, freedom for the de-
velopers to choose the language to work), Multiapplication BasicCard (evolution
from BasicCard, file system similar to File Allocation Table (FAT) by means of
the data is sharing) or Windows for Smart Cards (WfSC, authentication ori-
ented, real FAT file system). A brief summary of all of them can be read at
[44]. In the following, Java Card will be described deeper than MULTOS for
two reasons: it is the most important standard for multi-application smart cards
and it is which will be used in the implementation. Also, Global Platform will
be slightly presented.
2.1 Main Standards

2.1.1 MULTOS

This operating system is considered the main competitor to Java Card. It was created emphasizing on interoperability and security. Its architecture is made up of a virtual machine and a security scheme from Secure Trusted Environment Provisioning (STEP) technology to protect smart card, application code and data [30]. It is possible to program for MULTOS in several high-level languages, thanks to the amount of available compilers, like Modula-2, Basic, etc. although the most common are Java and C. It is also possible to program in low-level assembly language. Applications are compiled into MEL bytecodes. MEL is an optimized language which is based on Pascal P-codes. Then they are executed by the virtual machine which checks every bytecode and the address accessed. Data sharing is not allowed among applications, thus any application can access to the data of the others [30]. The main drawback from MULTOS respect Java is that manufacturers of the first one are not allowed to add their own API, so to be able to use the functions everywhere they have to be replicated.

2.1.2 Java Card

Java Card is a technology which allows developer to build applications using Java Card programming language to run them in smart cards. It defines a sub-set of the Java Programming language and a Java Card API. As it happens with Java, it provides object oriented programming, a secure programming platform, interoperability and hardware independence (portability). Although its speed is limited because of the card’s hardware and the byte code run-time’s interpretation, Java Cards are not slower thanks to the usage of crypto-coprocessors which help them to make cryptographic operations much faster.

However, the key point of this technology is the chance to develop multi-application smart cards and manage (addition, removal and update) them dynamically, that is, post-issuance card application management. Since this dynamism may generate several security problems; a context isolation mechanism was added to the Java Card Runtime Environment (JCRE). It provides isolation among applications at context level, as explain [50], what means that applets (applications on-card) cannot access applets from other context. This granularity is appreciated at Figure 2.2 (from [17]), where ”Package” is used instead of ”context”.

Context isolation is enforced by an application firewall which checks every access; and if someone is not allowed, it throws an exception. But, some applets may need to call methods from others. This is possible by means of Shared
Interface Objects (SIO). It is an interface which stores the accessible methods. Thus, an applet accesses to data or methods from some distinct applet accessing SIO mechanisms through the firewall.

2.1.2.1 Previous Versions to Java Card 3.0

In previous versions to Java Card 3, the card always works as a server; it gets a request, processes it and sends response back to the client, but it never takes the initiative. Communications are made by Application Protocol Data Unit (APDU) commands, specified at ISO 7816 standard. Garbage collection works on demand, that is, developer is the responsible for the memory management. If it is neglected, execution might run out the memory resources of the card, and consequently, make it useless.

The most representative of these versions is the split virtual machine architecture, which it shows at Figure 2.3 (got from [17]). Owing to the constrained resources of traditional smart cards; it is not possible to do bytecode generation on-card. That is why it is carrying out on the PC. Thereby, loading, linking, optimization and bytecode verification are done off-card by a tool, the converter, whose output is a Converted Applet (CAP) file. This file is loaded on the card and ready to be interpreted by the JCVM [50]. To sum up, bytecode execution and security enforcement are done on-card and previous work off-card.

2.1.2.2 Java Card 3.0

Java Card 3.0 provides two editions, both of them backward compatible:
2.1 Main Standards

Classic is an evolution from the version 2.2.2, oriented to constrained resource cards, using the split-virtual machine and working as a server.

Connected is network oriented to high-end cards with improved connectivity; that is why HTTP(s) is added as communication protocol using high speed interfaces like USB [3]. Card can work as a client and a Web server is incorporated giving rise to a new kind of applets: servlets.

A relevant feature of this edition is the on-card bytecode verification which supposes the change in the architecture: JCVM is not split but it is completely integrated on the card.

Necessaries protocols and layers are shown in Figure 2.4. The protocols which are bordered by the black line are exclusive from Connected Edition, whilst the rest are available for Classic Edition too. New logical protocols were developed to replace APDU because it spent too much time with ISO (APDU) command. Connections were not enough fast and a high-speed protocol was needed for new applications [10]: e.g., for mobile applications.

Figure 2.3: Split Java Card Virtual Machine

Figure 2.4: Physical and logical layers of Java Card 3 Connected Edition
2.1.2.3 Application Protocol Data Unit

APDU is the protocol used by Java Card to establish communication. As it has been described previously, the card always (Classic Edition and previous versions) works as a server following a master-slave model between the card and the CAD, which is the responsible for supplying power to the card. Communication consists of two types of messages:

- Command APDU, which are the messages sent from the CAD to the card
- Response APDU, which are the messages sent from the card to the CAD

Every response is sent as a reply to a command and every command receives a response. Both types of messages are transmitted by Transmission Protocol Data Units [20]. The most common protocols are: byte-oriented, block-oriented and contactless.

A command APDU is merely made up of two parts: Header (mandatory) and Body (optional). Body consists of three optional parts which give rise to four possible command structures according to its presence (see Figure 2.5 from [38]). The whole parts are described at the following:

- CLA (1 byte): Identify an application class of instructions; in other words, identifies the applet among the stored on-card
- INS (1 byte): Specifies the instruction among the available in the CLA set
- P1, P2 (1 byte each one): Parameters 1 and 2, they are used to qualify the INS field or for input data
- Lc (variable): Length of the data included in the command
- Data (variable): Data included in the command
- Le (variable): Maximum length of the expected response

Response APDU is made up of the following fields. Its format can be seen at Figure 2.6 from [38].

- Data (variable): Data included in the response
- Status Word (2 bytes): SW specifies which was the result of the execution on-card (if no error 0x9000)
At [14] it is further described the standard for ISO 7816-4. This document is particularly interesting to start to develop Java Card applications because valid CLA, INS and SW values are described. CLA and the necessaries INS have to be defined by the developer.

When smart card starts to receive power, it sends a message to the host called Answer To Reset which contains some parameters for the transmission like the protocol supported. Once this message has been received by the host, the communication can start. First command to be sent should be a select in order to choose the applet with which will be worked. Since the card saves its state, it is also possible to work with the applet selected before the card was powered off. But the CLA in the command has to meet with the CLA of the applet already selected. Any command with a different CLA, whose instruction was different to select, will be ignored.

### 2.1.3 GlobalPlatform

They are a set of specifications whose goal was to create a standard for cards management irrespectively of hardware, manufacturers or applications. Spec-
ifications are related to cards, card terminals and global management of systems using smart cards. That means they can be implemented over any multi-application technology.

Some of the most interesting parts are Security Domains (SD), which ensuring a total separation (complete isolation) of cryptographic keys between Card Issuer and other Security domains. They support the most of the cryptographic services like encryption, key handling, etc. Every time one of the keys is required, application accesses to the corresponding SD through GlobalPlatform API which provides the service [44].

2.2 Open Multi-Application Smart Cards

Multi-Application technologies were strongly pushed few years ago because of the needs of service providers which required a flexible, open and secure platform which allows them to load dynamically applications on cards after their issuance, that is, dynamic load post-issuance. It led to the differentiation of multi-application smart-cards depending on their flexibility to allow dynamic application load. In other words, their content management. They are distinguished among [44]:

- Open policy which allows anybody to load, update or remove any application on the card
- Closed policy which limits load, update or remove applications to the card issuer and some trusted partners

There is an intermediate policy, called mid-open, which allows modifications to “less” trusted parties. That means a partner in contact with a partner of the card issuer could also load its application, for instance. The risk of every policy is obvious. Open policy is more prone to be attacked or infected by malware than closed policy, because anyone can load its application, whilst in closed policy the issuer is the only one who can. That is why the chance to a malicious application was load on the card is as bigger as opened is the policy. Nevertheless, it does not mean that the open policy is the only one which can be attacked.

Some of the main attacks possible against open multi-application cards are the following [44]:
2.2 Open Multi-Application Smart Cards

- Collect information about services on-card in order to identify them
- Attack the OS
- Be familiarized with the card in order to select the best attack
- Denial of service

An important risk of open multi-application perspective is that the software to be installed might not be trustworthy; and that is an issue to solve. It should be taken in account that up to date, it was difficult to load new applications in a deployed card, and consequently, it was hard to exploit bugs in the software. But using open multi-application smart cards, new applications loaded in deployed cards might allow attackers to exploit software bugs which may be important (also because they might combine physical and logical attacks). In other words, a problem of the open multi-application cards is that it is possible to load a malicious application in order to carry out internal attacks. These attacks might identify available services on the card, collect information in order to try to deduce the possible behavior of an application or attack the JCVM and the firewall. It is longer explained at [43].

2.2.1 Security Challenge

The main problem with these cards is the interactions among the applications stored on it. In order to avoid them some tools have been developed and the security level of multi-application developments has been enhanced. For example, Java Card added its firewall and SIO. Firewall mechanism isolates applets from the rests within of its own space (context). But it is not enough, because applications can interact through the sharing methods. Let’s suppose that a class used to bank account management implements its interface for the SIO, due to it wishes that other application on the card related to bank issues could make use of its methods. The problem is that an application can make use of the shareable interfaces for its own purposes [34]. Any other application on the card will be also able to access its methods on the SIO carrying out malicious tasks. Of course, any application will not be able to access the context of the whole applications on-card (but only the presents on the SIO), but some unexpected behaviors may be carried out like illegal information exchange.

Some companies have already implemented some solutions but they are still only an improvement of Global Platform specifications. These improvements do not deal neither with the compliance of the new applications with the stored on the card nor with how the changes can interfere in the work of the applications already stored on the card.
The problem is that the semantic of the modification (either addition, removal or update) is not checked. It should be verified what the application intends to do on the card, its behavior; in order to make sure that it is not in conflict with the already stored on the card. In such a way, any application can do nothing unexpected or forbidden; because if the change is accepted is because its behavior is compliant with the expected on the card. That is the idea of Security-By-Contract.
Open smart cards allow consumers to dynamically load and remove applications during the card’s active life. The reason why concrete deployment of these cards is still rare is the lack of solutions to the control of interactions among applications, as the previous chapter has stated. Indeed, the business model of the asynchronous download and update of applications by different parties requires the control of interactions among possible applications after the card has been issued. The key point is to assure to stakeholders (regarding to the owners of the applications deployed onto the card) their applications cannot be accessed by other applications added after theirs, or at least that their applications will interact only with the ones of some business partners.

To date developers have to prove that all the changes that are possible to apply to the card are security-free, so that their formal proof of compliance with Common Criteria is still valid and they do not need to obtain a new certificate. The result is that there are essentially no multi-application smart cards, though the technology already supports them.[32]

This chapter gives a brief overview of the Security by Contract (S×C) approach
and examines some of the related work in what security of dynamically changes in open multi-application smart cards and mobile code concerns. Finally, the specific problem addressed in the thesis is introduced, namely securing off-card contract-policy matching in the $S\times C$ framework.

### 3.1 Mobile Code Security

The current State of the Art has its origin in the research done for Mobile Code (due to dynamic application load on the card is still only a kind of). That is why the first related work concerns it. The second part of the section will focus on the research on smart cards.

The three main approaches to Mobile Code security are Cryptographic Code-Signing, Proof-Carrying Code (PCC) and Model-Carrying Code (MCC).

Cryptographic Code-Signing is used to certify the origin of mobile code and its integrity by means of a signature. Usually, it is the software developer the responsible for signing the executable content. Using a public key infrastructure, the consumer can verify the content of the application which is loading by means of developer’s public key. This approach allows consumer to know that the software comes from a trusted developer, but it does not verify the mobile code behavior. Thereby, two wrong cases come up. First one the case of a right code from an unknown developer and the latter a bad code from a trusted developer. This approach does not allow unknown developers to install their applications, even if it is possible a safe execution of them. However, the code from a known developer can be installed either if it is faulty (case of trusted developer) or malicious (case of untrusted) code [33].

PCC is a technique which requires that the mobile code come bringing a proof regarding its safety. This proof assures the safe execution to the consumers who has to check that the proof is valid. Developer is the responsible for adding the proof to the code. The idea is that the proof should be automatically generated by means of an analysis done by a kind of compiler [36]. As with the previous technique, two problems arise. First one is related to the complexity of the automatic proof generation. The latter is that the technique expect that developer knew the security expectations of the whole potential consumers due to it has to send the suitable safety proof. That is impractical, since security needs change to each consumer [33].

The last approach is MCC, inspired in the previous one. Instead of a proof, MCC force the code to come with a model which stores the security-relevant
3.2 Security-by-Contract

Security-by-Contract is the approach which is proposed to prevent illegal information exchange among several applications, checking the interactions at load or run-time since the asynchronous loading and updating of applications by different parties business model. It is also expected to solve the problem of the behavior of the code to which is enclosed. When consumer receives the code and the model, it is also able to get the security needs of the code which is intended to be installed. Consumer has to check its security policy against the model received together the code in order to state whether the code is compliant with its policy or not. Models are expected to be simple; and consequently, the checking could be automated [41]. The main limitation was that MCC had not developed the whole lifecycle and had limited itself to a finite state automata. That one was too simple to describe a realistic case, but even for that, basic policy could not be addressed [34].

The idea of [15] is to use information flows to specify security policies of the smart card. Every application is certified when is loaded by means of the information flow signature assigned to each method. It is represented as relations established between two method variables together annotations indicating the type of flow. The problem of this approach is that its policies are too simple.

At [28] is proposed a framework and a tool set for compositional (thus allowing post-issuance loading of applications onto the card) verification of the interactions among applications on the card. A maximal applet with a behavioral safety property is built. The idea is that verifying the compliance of the local applications, the global correctness can be achieved. That is why the local properties are compared against the implementation of the new applications when they are loaded. To check that, model checking techniques are used. They assure the global property.

[16] presented an approach which suggests to associate security levels to attributes and methods by means of Bell/La Padula model which is a state machine model used for enforcing access control. The security policies in this model define which are the authorized flows among the levels [34].

Finally, S×C framework is proposed to deal with and address these problems. It works over the idea that the application has to come bringing a specification of its security behavior which will be check against the expected behavior of the target platform. The remainder of the chapter is focused on this framework.
post-issuance changes, that is dynamically loading, removal or update of applications, during card’s active life; in other words, dynamic evolution of smart card platform.

S×C approach was built upon the notion of MCC and successfully developed for mobile code ([33]), making more concrete the ideas behind it and allowing card to deal with more practical scenarios. Loosely speaking, it consists in the idea of contract which specifies the security behavior of an applet, policy which specifies the security policy of the card and the contract matching algorithm which compares the contract against to the policy. The goal is to make the checking at load time in order to avoid costly run-time monitoring. The aforementioned problems can be summarized as [35] does in the following way:

1. A new application should not interact with forbidden applications already stored on the card

2. A dynamic change should not affect the correct work of an already stored application. The changes can be:
   - Addition of a new application
   - Updating an existing application
   - Changes in the card’s policy
   - Removal of some existing application

A contract is a formal complete and correct specification of the behavior of an application for what concerns relevant security actions and particularly with problems of sensitive data exchange. It does not only store the behavior of the current application, but the desirable for other applications regarding to the direct and indirect communications with it [34]. The aim is that a new application could express its desire to interact with other application still not present on the card, for instance. Contract is provided by the application developer who is the responsible to enclose it to the application to be installed.

A policy is a formal complete specification of the acceptable behavior of applications to be executed on the platform for what concerns relevant security actions (both definitions from [33]). The policy has to be created in first time by the card issuer. It is the responsible for preparing the initial security requirements of the card.
3.2 Security-by-Contract

3.2.1 Security-by-Contract Workflow

Working on mobile code, the target platform (i.e., the smart card) follows a workflow similar to the one depicted in Figure 3.1 at load time. First, it checks that the evidence is correct. Such evidence can be a trusted signature. An alternative evidence can be a proof that the code satisfies the contract (i.e., [15] for smart cards).

Once we have evidence that the contract is trustworthy, the platform checks that the claimed policy is compliant with the policy that our platform wants to enforce. This is the Contract-Policy Matching. If it is, then the application can be run without further ado. At run-time, a firewall (such as the one provided by the JCRE) can just check that only the declared API in the contract can be called. The matching step guarantees that the resulting interactions are correct. This is a significant saving over full in-line reference monitors [32].

Coming back to what smart cards concerns, the matching algorithm should succeed only in case that every claim that the contract specifies is compliant with every security requirement that the policy demands. Particularly, they should match if contract and policy do not give rise to any illegal information leakage or exchange between the new application and anyone existing on the card.

Figure 3.1: SxC Workflow
3.2.2 A Hierarchy of Contract/Policy Models

It is considered more efficient and secure to make any checking or validation about the trustworthiness of any extern something on-card because it supposes not to rely on off-card applications. Thanks to be a tamper-resistant device, its data and applications are more trustworthy. However, the constraint resources of the card may lead to some operations cannot be carried out on-card, but off-card [25].

Precisely, because of the limitation of the computational resources of the smart cards, $S\times C$ proposes a hierarchical model of contracts and policies. Every level on the hierarchy has contracts and policies with several features and described with different deepness in function of the expected computational capacity for that level. In other words, every level works over different computational efforts and has different expressivity limitations. For instance, the lowest level gets the highest computational benefits, but their contracts and policies loose expressivity; that is, they are less concrete. On the other hand, the highest level spends a lot of resources of the microprocessor, but allows a very complete specification of the contracts and policies.

The levels proposed for $S\times C$ are [34]:

**L0: Application as Services.** At this level, applications are dealt as lists of available and required services, similar to set-up of the Global Platform specification. This is the level which needs less computational effort and it is expected to be run on the card.

**L1: Allowed Control Flow.** This level builds a call graph of the application where vertices are the states of the application and edges the invocations of different services. It allows doing a more fine grained information exchange control and even a bit of history-based access control. The more abstract levels are based on this, adding more features to the graph.

**L2: Allowed and Desired Control Flow.** It adds correct and error states to the previous graph. The goal of this addition is to provide a way to check that any change in the policy or the removal of some application do not go to avoid that the rest of applications worked properly; that is, any of the aforementioned change affect the correct work of the rest.

**L3: Full Information Flow.** At this level, the call graph is enhanced adding the information flow among variables. It requires the highest computational effort.
3.2 Security-by-Contract

3.2.2.1 Limitation of the Level 0

The problem of the Level 0 is that it only captures the possible information exchange among the applications instead of the actual exchange; what means that this level cannot specify the behavior of an application. As [35] discusses, it is not able to distinguish between the services that an application might need from the services which an application requires. For instance, let’s suppose an application A on-card, which is wished to be removed, provides a method which other application B on-card requires. What Level 0 is not able to extract is whether the application B, which requires the method, will keep working well or not after application A was removed. To do that, a further level of abstraction is needed. It can be solved by means of a call graph which was able to get the actual behavior of an application. It is worth mentioning that this level cannot capture the indirect communication between applications on the card neither.

That is why to get a higher security level is suitable to use some of the other levels. The problem with the limited computational power does not lie in the capacity to build the representation, but in the resources to run the contract-policy matching which is the key phase of the approach. Usually, it is done on-card, but if it requires a very expensive computational effort for a resource limited device, it is also possible to outsource this phase to a Trusted Third Party (TTP). That is called Off-Card Contract-Policy Matching.

3.2.3 Problem: Securing Off-Card Contract-Policy Matching

The Off-Card Contract-Policy Matching idea is depicted in Figure 3.2. It is carried out when Contract-Policy Matching Phase is too expensive (computationally talking) for doing on-card. Then it is necessary to make use of a TTP which provides its computational capabilities to run the matching algorithm. TTP supplies then a proof of contract-policy compliance to the smart card which should verify that. Smart Card’s (SC) policy is updated according to the results received by the TTP: if the compliance check was successful, then the SC’s policy is updated with the new contract and the application can be executed. Otherwise, the application is rejected or the policy enforced off-card (for example, by means of a service provided by TTP in addition to Contract-Policy Matching), as [32] details. In case TTP includes a proof of compliance in the reply, then a further check is needed to verify the proof, as shown in Figure 3.2.

Since to the communication between SC and TTP is carried out over an untrusted environment, it has to be secured. The system has to ensure authentication, integrity and confidentiality of the data sent during the whole communica-
tion. To guarantee integrity and authenticity, both parts (SC and TTP) have to encrypt and sign their messages by means of their key pairs. They should keep their private keys as secret and send their public keys to each other. Thereby, at the beginning of the communication, it is necessary a kind of key exchange or key set-up.

But it is not easy; some attacker can try to intercept their keys, like it may happen in a very common attack: Man-In-The-Middle. This attack is a kind of eavesdropping and consists in someone who is placed between the host and the card and launches independent connections to them, making them believe they are communicating with each other, while in fact they are only sending messages to the attacker. An attack of this type may be carried out in the aforementioned key exchange, making useless PKI architecture, because attacker can decrypt any message. It would be done like follows. TTP sends a public key to the card, but attacker intercepts the message replacing the key for a new one which only he knows. Card receives key sent by the attacker and reply with its key. Attacker intercepts the key again and replaces it from a new one which is sent to TTP. Therefore the attacker can intercept the rest of encoded messages and modify them as it wants without the rest of the parts of the system realizing of that.

This problem is solved using certificates which have been signed previously by a Certification Authority (CA) in which both parts rely on. Certificates identify
its owner unmistakably and they cannot be forged because they are signed with CA’s private key. But SC’s certificates cannot be sent through an insecure environment to be stored on the card. They must be stored through a secure and trusted environment and only once during the card initialization. That is why an initialization phase is necessary. It is carried out in a trusted environment by means of a trusted reader which works as an installer and is placed between the card and the CA. It is the responsible for sending the certification signing requests to the CA and gives them back signed to the card. The card cannot be used until initialization has properly finished with the storage. Otherwise, security requirements are not accomplished, security verifications cannot be carried out. Any change over any application on the card should be rejected until the end of the initialization. Once everything been stored, the card is ready to work as it is expected.

To summarize, the thesis is focused on solving the problem of securing the communication between SC and TTP during the Off-Card Contract-Policy Matching Phase. Specifically this is done by making the communication over an untrusted environment secure and trustworthy by satisfying the requirements of mutual authentication, integrity and confidentiality. To achieve this it makes use of a PKI scheme which needs certificates to handle identities.
Contract-Policy Matching is a key step in the $S\times C$ framework. When the limited resources of the device do not allow the card to run this algorithm, it has to be done off-card. A TTP which has enough computational capabilities will run the matching algorithm. To do that it is necessary to send the contract, policy and subsequently the results, over an untrusted environment. Therefore the communication has to be secured. The current chapter introduces the design of the system’s architecture designed to solve the problem of securing the communication for outsourcing the matching algorithm. The user will be able to securely rely on system decisions about matching-algorithm output to decide advisability or not of the change (addition, removal or update).

In order to fulfil the requirements of mutual authentication, integrity and confidentiality, the system is based on a PKI where keys and identities are handled through certificates that are exchanged between both parties at the beginning of each communication. For this reason, the card requires an initialization phase where certificates are stored in the SC along with the initial security policy of the card. Notice the difference between this phase and the installation which adds the system to the card and creates the instances necessary to its proper work.
Since TTP is the responsible for running the matching algorithm, it is expected to be a trusted and secure device. The security of the system relies on this assumption, therefore the communication is what needs to be secured. Each part of the PKI architecture involved in the matching algorithm (SC and TTP) has two key pairs. One of these key pairs is used for encryption, while the other one is used for signing. Although it is perfectly possible to use only one key pair for both encryption and signature, it is safer to use two different key pairs. Since both of them are distinct cryptographic functions, in the unlikely case of one of them being compromised the other will not. Hence, the common recommendation is to use two different key pairs. Therefore, every principal part of PKI architecture has two certificates, one for each key pair. To sum up, the use of two key pairs (and consequently two certificates) is optional, and it has been pointed out in the figures belonging to Initialization Phase; however it makes the system more secure.

It is worth mentioning that the card works always as a server. That means it never starts a communication, but it merely receives a command from off-card clients, processes it, sends the reply back and holds waiting next command as Figure 4.1 (modified from [38]) shows. This is important to understand the subsequent design. Throughout this chapter the term Command will be used to refer to the messages sent from any off-card entity to the card, whilst the term Response will refer the messages from the card to any entity.

Figure 4.1: Smart card’s communication by APDU
4.1 Confidentiality and NONCE Issue

In the previous section it has been stated that the requirements of the system are authentication, integrity and confidentiality, but someone might wonder why the last one is necessary in the system. Since during the communication what is sent are the contract and policy and the result of the matching algorithm, all of that might be public, it might seem that this is unnecessary. What is really important it is to keep the integrity and authenticity of the message safe because it guarantees that the message has been sent from the card to the TTP and vice versa without being modified. That is: the card is sure that the result which it receives is the one generated by the TTP by means of the contract and policy that it has sent.

However, let’s consider the following scenario. At the beginning, a trustworthy application wants to be installed on the card. Its contract is stored on the card, and those sends this contract and its policy to a TTP in order to run the contract-policy matching algorithm. The result is positive, and TTP gives back a message to the card telling it that the application is trustworthy. Now an attacker is able to intercept and save this message. The attacker wants to install its untrustworthy application on the card. Thereby, it sends the contract of its application to the card, which forwards the contract together the policy to a TTP to run the matching algorithm. The result is negative, the application should not be installed on the card. TTP sends this result to the card, but the message is intercepted again by the attacker, which substitutes it for the previous message containing a positive result. In such a way, the attacker gets to install the malicious application on the card (this kind of attack is known as Replay Attack).

The solution to that problem is to add a number used only once (NONCE, in this case a Cryptographic NONCE). It should be random in order to insure the freshness. This number is used as a kind of time-stamp which prevents the card to accept previous messages. The NONCE has to be sent encrypted in order to avoid malicious entities to know it, and consequently, forge some messages.

Concerning the confidentiality of the message (contract, policy and result), it is used to avoid some kind of spoofing attack against the content. Moreover, an attacker could get information about the behavior of applications on the card and use it in a bad way. That is why some application issuers, whose applications needed a high-security level like bank’s applications, may demand the use of confidentiality, refusing the card if it does not provide it. As a final comment, it might be considered to restructure the system for really constrained resource devices which cannot afford message encryption, but it is strongly advised because of the benefits are much bigger than the disadvantages.
4.2 Architecture of the System

The system has several parties involved during its different phases; Initialization, Contract Storage and Contract-Policy Matching. These phases can also be separated in other smaller parts. All of them are addressed in the remainder of this section according to the order in which they have to be run. Note that Installation Phase has to be performed first.

4.2.1 Installation Phase

Applications in any smart card have to be installed in order to be used. First of all, the application is deployed on the card, and the Installation is done afterwards. The keys are usually generated and stored at this phase. Keys generation can be done in the ways proposed at the following:

- TR produce the the keys and gets the certificate after sending the CSR to the CA
- CA is the responsible for generating the keys and the certificates
- The card creates the keys and by means of the TR which sends the CSR to the CA gets its certificates

The first two ways have a particularly significant drawback. They have to send the keys, or rather the private key, to the card in a secure manner. If this key was compromised the card would be useless. Also, they should remove physically all the information related to the generation and the own keys. Even physical security measures shall be used to ensure that the off-card entity and operations are free from tampering [1]. That is why is much more secure to generate the keys inside the own card. Indeed it is one of the security highlights of the system which makes it particularly secure. The reason is that the private keys do not leave never the card simply because there is no reason to do that. Therefore, no one apart from the card can either get or use the private keys.

4.2.2 Initialization Phase

This phase is the responsible for creating and storing SC certificates, CA certificate (needed to verify correctness of TTP certificates) and the policy of the card. The whole process is made up of three stages: Certificate Signing Requests
4.2 Architecture of the System

(CSR) building, certificates issuing, and finally certificates and policy storage. The off-card entity involved in this part is a Trusted Reader (TR).

It is carried out in an environment considered trusted, as well as the communication between both parties. Notice that TR is the responsible for storing the certificates and the policy; hence, the security of the card depends on its proper work. If the storage were compromised and what the TR stored on the card were infected, the whole system would be compromised. It is expected to be carried out by the card issuer who is trusted.

It is worth mentioning that it is only possible to run this phase only once, or rather, every storage can be done only once. Therefore it is avoided the case where everyone could store whatever (certificates or policy) they need on the card. Notice that the card is only a server which replies requests without being able to identify the requester. This issue is further discussed in the **Future Work** section.

4.2.2.1 Certificate Signing Requests Building

In this stage the CSRs are built and stored on the TR in order to be sent to the CA.

A certification request consists of a distinguished name, a public key, and optionally a set of attributes. Everything is signed by the entity which is requesting the certificate. A CA receives the CSR and if it is correct (information and signature), CA creates a public-key certificate containing the information of the CSR. The signature is used to prevent any entity from being able to request a certificate including the public key from another entity. Usually, certification authorities might require other non-electronic ways of request or reply in order to assure the information given by the requester; but that is beyond of the scope of the project.

TR has to create the two CSR, one for each key pair. CSR for encryption is created before, whilst CSR for signature is created later. In order to do that, it needs the public keys stored on the card. That is why to build the CSR, the first Command, which TR sends to the card, orders the public key for encryption (PuKSCEnc) and does not contain any data. SC receives the query and replies it with a Response whose data is PuKSCEnc.

TR by means of PuKSCEnc creates the CSR for encryption (SCEncCSR) adding the pertinent information. However the card has to sign the CSR since it is the owner of the private key. That is why at this moment TR has to sent
SCEncCSR to the card and be signed there with the private key for encryption (PrKSCEnc). In such a way, CA will be able to verify that the signature is right, and consequently, that the public key matches with the private key of the requester. In other words, signature is what CA needs to be sure that PuKSCEnc belongs to whom is sending SCEncCSR, because it is the only one who can sign with the private key corresponding to the public key in the CSR. Once it is signed, the card sends a Response containing the signed SCEncCSR (Sig_{PrKSCEnc}(SCEncCSR)) attached in the data. This process is shown at Figure 4.2.

As notation, it is worth mentioning that in the Figure 4.2 and subsequent, the messages typed in blue means that they do not contain data, whilst the black messages do.

To sum up the flow of messages for the first CSR’s building is:

1. Command 1: No data, it is ordering PuKSCEnc
2. Response 1: PuKSCEnc
3. Command 2: SCEncCSR
4. Response 2: Signed CSR, that is Sig_{PrKSCEnc}(SCEncCSR)
4.2 Architecture of the System

TR stores $\text{Sig}_{P_rK_{SCEnc}}(\text{SCEncCSR})$ and builds afterwards the CSR for signature (SCSigCSR) in the same way: orders the public key for signature to the card ($\text{PuKSCSig}$), builds SCSigCSR and sends it to the card in order to be signed with the private key for signature ($\text{PrKSCSig}$). Hence, the messages for the second CSR’s building will contain, in the following order:

1. **Command 1**: No data, it is ordering $\text{PuKSCSig}$
2. **Response 1**: $\text{PuKSCSig}$
3. **Command 2**: SCSigCSR
4. **Response 2**: Signed CSR, that is $\text{Sig}_{P_rK_{SCSig}}(\text{SCSigCSR})$

### 4.2.2.2 Certificates Issuing

Once both CSRs have been built and signed, TR has to send them to a CA in order to get the corresponding certificates. This process is shown in the Figure 4.3. TR is called TR-Certificates Manager in this figure because it depends on the actual implementation of the TR which is not within the purpose of this project.

The messages’ content exchanged between the TR-Certificates Manager and the CA are the following:

1. **Command 1**: $\text{Sig}_{P_rK_{SCEnc}}(\text{SCEncCSR})$
2. **Response 1**: SCCertEncr, that is the SC’s certificate to encrypt
3. **Command 2**: $\text{Sig}_{P_rK_{SCSig}}(\text{SCSigCSR})$
4. **Response 2**: SCCertSign, that is the SC’s certificate to sign

Each certificate contains (among other) information about the owner of the key, the public key and the signature. For instance, previous certificates may contain each one:

- Information which identifies the owner: $\text{ID}_{SC}$
- Public key which intends to be authenticated by the certificate: SCPuKEnc/SCPuK-Sig
- Certificate’s signature: $S_{P_rKCA}(\text{SCEncInfo/SCSigInfo})$
Once the certificates have been issued by the CA, sent and received, TR stores them in order to be available to be sent and stored to the card.

![Diagram](image)

**Figure 4.3: Initialization Phase: Certificates issuing diagram**

### 4.2.2.3 Certificates and Policy Storage

The last stage during the Initialization Phase is the storage of all the necessary on the card. TR has to send both SC’s and CA certificates and the security policy. On one hand, each one of this things is sent in a different *Command* by the TR. On the other hand, the card always replies with an acknowledgement without any data. The stage is detailed at the Figure [4.4](image). As it has been pointed out previously, these messages can only be sent once, that is, any object can be stored on the card once.

The data sent within the commands from TR are the following (the order is not relevant):

- **Command 1**: CA’s Certificate
- **Command 2**: SCCertEncr
- **Command 3**: SCCertSign
- **Command 4**: Card Security Policy

After the SC had been initialized, it is ready to be used to run securely any activity related to the contract-policy matching algorithm. At that time, the card
is able to verify the identity of the TTP (validate its certificates), authenticate and authorize its requests.

### 4.2.3 Contract Storage Phase

At this phase, the contract of the application wished to be installed on the card has to be sent and stored there. It has to be carried out previously to any Contract-Policy Matching Phase since it needs that the contract had already been stored on the card. After every phase which runs the matching algorithm, the contract is removed from the card, that is why other contract should be stored on the card in order to perform the matching algorithm again.

The process is very simple. An off-card entity, called Application Issuer (AI), sends a request to store the contract to the card, attaching aforementioned contract. The card stores the contract and replies the entity with an acknowledgment. The process is shown at Figure 4.5. The entity responsible for storing the contract is the application issuer because it should be the same entity that develops the same which enclose to it the contract with its behavior, as it is proposed in the S×C framework.
4.2.4 Contract-Policy Matching Phase

Contract-Policy Matching is the key phase of the $S\times C$ framework. It has to run the matching algorithm by means of the data provided through the secure communication between a TTP and the card. With more detail, contract and policy, stored on the card, are sent encrypted from the card to some TTP which runs the matching algorithm giving the result back to the card. While Initialization Phase is expected to be executed only once, this one is supposed to be executed commonly. In fact, always that a change was necessary in any application on the card; either addition, update or removal. As it has been pointed out in the previous section, Contract-Policy Matching is always performed afterwards Contract Storage Phase.

As far as this phase (contract-policy validation) is concerned, it has three stages:

1. Certificates’ exchange
2. Contract and policy sending
3. Matching result sending

The whole process can be observed at Figure 4.6.

4.2.4.1 Certificates’ Exchange

The exchange of the certificates between the card and the TTP is carried out by means of messages which encloses the corresponding certificates. Each party has to exchange two certificates to each other: one for encryption and one for signature. That means four messages are necessary to complete the exchange where two are from TTP and two from the card.
4.2 Architecture of the System

At the beginning, certificates for encryption are exchanged. TTP starts the communication sending its certificate to encrypt to the card enclosed within a Command. A parser is implemented by means of which the smart card is able to verify the certificate against CA certificate stored during Initialization Phase. Parser also takes the TTP’s public key from the certificate allowing the card to get it. If the verification is positive (the certificate is right), it replies the message with a Response attaching its certificate for encryption. TTP verifies the certificate got from the card and gets also smart card’s public key to encrypt. This process is repeated to exchange certificates to signature between both parties. It can be observed at Figure 4.6. The data which contain the messages sent during this stage are:

- **Command 1**: TTPCertEncr, that is TTP’s certificate to encrypt
• **Response**: SCCertEncr

• **Command 2**: TTPCertSign, that is TTP’s certificate to sign

• **Response**: SCCertSign

### 4.2.4.2 Contract and Policy Sending

When certificates’ exchange has finished, TTP has to order the contract and policy which are stored on the card. That is why it sends a **Command** without any data to make the query. Both contract and policy will be enclosed together within the same message in order to reduce the amount of messages to exchange, that is: SCContractPolicy (SC acronym has added in front to differentiate from the contract and policy which will be used during the verification of the message which will be TTPContractPolicy).

Contract and policy have to be encrypted and signed by the card before being sent in order to assure the requirements of the system (authenticity, confidentiality and integrity). The encryption will be done through block symmetric ciphering (for instance AES or DES). The reason to use symmetric cryptography instead of asymmetric (based on PKI) is that asymmetric encryption is very slow when it is carried out over a considerable amount of data, whilst symmetric encryption provides higher speed (decryption is the same process; hence, it is also very fast). In addition, symmetric cryptography provides high security due to their no linearity and it depends on the algorithm (i.e., DES can be broken by cryptanalysis [8]) but usually only brute force can work. That is why a Block cipher is suitable to used over these data which may be relatively long.

To use symmetric cryptography implies to both parties have to know the key. Thereby, the card that is which generates the key has to send the shared key to the TTP; but it cannot be sent in plain text because if someone could intercept the message, it could decrypt the content. That is why the key is encrypted and enclosed later to the message. The encryption at this moment is done by means of PKI because the key is not expected to be too long, but a few hundreds bits as maximum. It also guarantees that TTP is the only one which can decrypt the session key. The key for symmetric cryptography will change every session, that is, every time when it has to run a new Contract-Policy Matching Phase in order to assure freshness and randomness and get a higher security level.

Keeping in mind this kind of ciphering and taking in account also the NONCE previously explained, the sequence of actions, which the card has to follow aiming to secure the message containing the requested information (contract and policy) by the TTP, is:
4.2 Architecture of the System

1. Generate a session key that will be used for symmetric cryptography: $K_{sess}$

2. Generate a NONCE (Number used Once): $N_{sc}$

3. Encrypt the session key through TTP’s public key to encrypt: $\text{Enc}_{PuK_TTPEnc}(K_{sess})$

4. Encrypt the NONCE through TTP’s public key to encrypt: $\text{Enc}_{PuK_TTPEnc}(N_{sc})$

5. Encrypt the message with the session key: $\text{Enc}_{K_{sess}}(SCContractPolicy)$

6. Compute the HMAC (a hash mixed with a salt) of the content: $\text{HMAC}(SCContractPolicy, N_{sc})$

7. Sign the HMAC previously built through the key to sign: $\text{Sig}_{PrK_{SCSig}}(\text{HMAC}(SCContractPolicy, N_{sc}))$

As it has been detailed in the previous enumeration, a Hash-based Message Authentication Code (HMAC) function has been used to create the signature instead of a common hash function, like Secure Hash Algorithm (SHA). The reason is that HMAC adds a shared key (called salt) which assures the freshness of the digest: every HMAC built with different salt will result in different digests. In addition, it guarantees that only the partners in the communication which know the shared key can check the integrity and generate the proper digest. That also improves the security of the system. In order to build this kind of digest, it is necessary a random number which was renewed in every session. That is why NONCE has been used, it fits perfectly with the requirements.

Finally, the secure Response built to send back to the TTP containing the contract and policy is the following:

- **Response**: $[\text{Enc}_{PuK_TTPEnc}(K_{sess}), \text{Enc}_{PuK_TTPEnc}(N_{sc}), \text{Enc}_{K_{sess}}(SCContractPolicy), \text{Sig}_{PrK_{SCSig}}(\text{HMAC}(SCContractPolicy, N_{sc}))]$, which contains:
  - Encrypted session key: $\text{Enc}_{PuK_TTPEnc}(K_{sess})$
  - Encrypted NONCE: $\text{Enc}_{PuK_TTPEnc}(N_{sc})$
  - Encrypted contract and policy: $\text{Enc}_{K_{sess}}(SCContractPolicy)$
  - Signature: $\text{Sig}_{PrK_{SCSig}}(\text{HMAC}(SCContractPolicy, N_{sc}))$
TTP receives the *Response* and has to decrypt it and verify the content. To verify the integrity and authenticity, it builds a HMAC of the content, which it has just decrypted, and compares with the HMAC which it has obtained from decrypting the attached signature through the card’s public key to sign. If both HMAC match, authenticity has been proved because only the card can sign with its private key. Integrity would have been proved too, because the fact that both HMAC match means that the data, from the HMAC have been done, also match, and consequently anyone could modify the content.

The sequence of steps necessary to carry out the verification is the following:

1. Decrypt the session key: 
   \[
   \text{Dec}_{PrKTP_{Enc}}(\text{Enc}_{PuKTP_{Enc}}(K_{sess})) = K_{sess}
   \]

2. Decrypt the NONCE: 
   \[
   \text{Dec}_{PrKTP_{Enc}}(\text{Enc}_{PuKTP_{Enc}}(N_{sc})) = N_{sc}
   \]

3. Get the contract and policy decrypting with the session key:
   \[
   D_{K_{sess}}(\text{Enc}_{K_{sess}}(\text{TTPContractPolicy})) = \text{TTPContractPolicy}
   \]

4. Compute the HMAC of the decrypted contract and policy:
   \[
   \text{HMAC}(\text{TTPContractPolicy}, N_{sc})
   \]

5. Decrypt the signature through the public key to sign of the card:
   \[
   \text{Dec}_{PuKSC_{Sig}}(\text{Sig}_{PrKSC_{Sig}}(\text{HMAC}(\text{SCContractPolicy}, N_{sc}))) = \text{HMAC}(\text{SCContractPolicy}, N_{sc})
   \]

6. If the HMAC which has been computed on the TTP through the content decrypted from the message matches with the HMAC got from the signature attached to the message, both authenticity, confidentiality and integrity requirements have been accomplished; otherwise not. That is:
   \[
   \begin{align*}
   \text{If} & \quad \text{HMAC}(\text{SCContractPolicy}, N_{sc}) = \text{HMAC}(\text{TTPContractPolicy}, N_{sc}) \\
   \rightarrow & \quad \text{Message OK} \\
   \text{else} & \quad \text{Requirements not accomplished}
   \end{align*}
   \]

If the verification is correct, TTP have got the contract and the policy in a trustworthy way.

### 4.2.4.3 Matching Result Sending

Once contract and policy are in TTP’s possession, TTP can perform the matching algorithm against them getting the corresponding result. Then, it has to build a secure message containing the algorithm result to be sent to the smart
card in a similar way to how it has been done by the card previously. The key used for encryption is still the same session key with which the card had encrypted the contract and policy. The signature is done as before except with the difference that the HMAC uses as salt the value $N_{sc+1}$ instead of the previous one. This change is done in order to assure that the TTP has been able to read, store and modify the NONCE. It is also a countermeasure against attacks since it increases randomness of the key. Thanks to that, the HMAC function changes because the key used for its encryption operation is different; that increases the effort for an attacker who tries to forge the HMAC.

The sequence of actions which has to follow the TTP to build the secure message to send in a *Command* to the card, is the following:

1. Encrypt the result with the session key: $\text{Enc}_{K_{sess}}(\text{TTPResult})$
2. Compute the HMAC of the content: $\text{HMAC}(\text{TTPResult}, N_{sc+1})$
3. Sign the HMAC previously built through the key to sign: $\text{Sig}_{Pr_{K\text{TTPSig}}}(\text{HMAC}(\text{TTPResult}, N_{sc+1}))$

Finally, the secure *Command* built to send to the card containing the Result is:

- *Command*: $[\text{Enc}_{K_{sess}}(\text{TTPResult}), \text{Sig}_{Pr_{K\text{TTPSig}}}(\text{HMAC}(\text{TTPResult}, N_{sc+1}))]$, which contains:
  - Encrypted result: $\text{Enc}_{K_{sess}}(\text{TTPResult})$
  - Signature: $\text{Sig}_{Pr_{K\text{TTPSig}}}(\text{HMAC}(\text{TTPResult}, N_{sc+1}))$

Just in the same way that TTP has verified previously the message, the card has to do the same now. The sequence of actions to be carried out are the following:

1. Get the result by decrypting with the session key: $\text{D}_{K_{sess}}(\text{Enc}_{K_{sess}}(\text{SCResult})) = \text{SCResult}$
2. Compute the HMAC of the decrypted result: $\text{HMAC}(\text{SCResult}, N_{sc+1})$
3. Decrypt the signature through the public key to sign of the TTP: $\text{Dec}_{Pu_{K\text{TTPSig}}}(\text{Sig}_{Pr_{K\text{TTPSig}}}(\text{HMAC}(\text{TTPResult}, N_{sc+1}))) = \text{HMAC}(\text{TTPResult}, N_{sc+1})$
4. If the HMAC which has been computed on the SC through the content decrypted from the message match with the HMAC got from the signature attached to the message, both authenticity, confidentiality and integrity requirements have been accomplished; otherwise not. That is:

\[
\text{If } (\text{HMAC}(\text{TTPResult}, N_{sc+1}) == \text{HMAC}(\text{SCResult}, N_{sc+1})) \\
\rightarrow \text{Message OK} \\
\text{else } \rightarrow \text{Requirements not accomplished}
\]

If the message is correct, the card can get the result and sends an acknowledgement to TTP in a *Response*. With the result the card can also decide whether the application should be installed on the card or not.
Implementation of the Prototype

With the goal of validating the previous design of the system to solve the contract policy matching off-card, a prototype is built. This one should be interpreted as a proof of concept. That means the aim is to build a functional prototype which demonstrate the system can work, but without being deployed in a real card. That is why the previous design has been lightly modified in order to get it running in a simulator.

At this chapter, the tools and programming languages chosen to be used to develop as well to run the prototype are explained. Also, the main parts of the implementation are described together the problems, choices and solutions taken during this stage. Every kind of details are provided in order to help to understand how the prototype was built.

In what Global Platform Security Domains concerns, it is worth mentioning that they have not been considered because of time constraints. It is proposed as future work.
5.1 Programming Languages

As it was explained in the previous chapter, there are four separate parts in the system: AI, SC, TR and TTP.

On one hand, clients (AI, TR and TTP) should run in a reader device able to communicate with a smart card and perform complex matching algorithms like a PC. Any common programming language like C, Java, etc., could be used to build these parts. Java has been the language chosen. On the other hand, there are not so many languages for smart cards which support multi-application. Java Card has been chosen. Both of them and the reasons to have been decided for them, are described below.

5.1.1 Java

Java is a programming language object-oriented, robust and simple. To know more, check [19].

The main reason to choose this language was to be multi-platform and that its API provides a very complete set of methods to work with security issues. Multi-platform issue, if not fundamental, is suitable due to the different places where the TTP could be placed. Java virtual machine could be easily installed everywhere and thanks to be an interpreted language any code should run properly. Java API provides a wide variety of methods which have been verified by a huge amount of testers from Sun community. That makes the implementation secure and reliable. Developers do not have to implement new features already presents in the API. Particularly, it provides cryptographic methods to deal with security problems. For example, ciphers or randomness are easily implemented thanks to the API’s support. The version of the Java Development Kit (JDK) used has been the latest version to date’s project: JDK 1.6.0.18.

It was necessary to use some parts from sun.* package. These packages are not part of the public and supported Java API. What does it mean? They are included within Java Software Development Kit (JSDK) and they are usable, but Sun does not assure neither their portability nor their support in future versions [17]. Sun uses them to support its implementation of Java platform, that is why it wants to reserve the chance to modify them if necessary. Then, why to use that? They are used to generate CSRs because of the Java API does not provide any other way to do that. Other choice would be to build a new complete class to work with certificates generation. That implies to implement classes for Base64 and Distinguished Encoding Rules (DER) encoding or to
do hand-made. Anyway, a long and tedious task. Also, it should be taken in account that Java does not provide any other way to deal with certificate generation, that is why it may be expected that if any change is done in future versions, it should be to add these classes to the Java API; consequently the work to modify the classes used should not be too hard. Finally, it is worth underlining that the prototype, as it has been explained before, is only a proof of concept; so the prototype is assured to work at the specified version of the JDK. If it is wished a more reliable version in terms of software’s duration (out of the scope of the thesis), the use of these kind of packages should be avoided.

5.1.2 Java Card 2.2.2

Java Card programming language has already been described at multiapplication main standards section. In the following, some more of the main features for version 2.2.2 are exposed.

JCRE works as an operating system; it deals with communications, applet execution and security (it is done by the firewall). Both JCRE and JCVM never stop, since the moment when the card is initialized. This happens only once, at the beginning. JCVM together some static data structures are created and, from then onwards, virtual machine never exits [17]. Every time card is taken out, their content stop, and when is powered on again RAM content has been erased, but JCVM continues working. In other words, when the card is powered off, JCVM is in an infinite clock cycle [53]. For instance, if an application is selected and working in a smart card which, suddenly is taken out from the reader, when the card comes back to be powered on, the same application will be selected and their EEPROM’s content will be stored (unlike of RAM’s content). Due to card tear is possible to occur in every moment, persistent data writes must be guaranteed and this is reached by atomicity. Every write in non-volatile memory is atomic and if the write is unfinished it is rolled back.

5.1.2.1 Extended Length

APDU has a limitation which is the length of the data to send within a APDU message. Usually this length is restricted up to 255 bytes; a very small amount taking in account that the system has to send certificates (usually from 1 to 4 KB). To avoid the tedious task of implementing a way to send data bigger than 255 bytes, separating it in several messages, extended length functionality was used. It allows sending large data up to 32KB (32767 bytes) in a more efficient way. That is the reason why the length fields are variable; due to with extended
length they may be up to three bytes long, whilst in common APDU its length is one byte.

5.1.2.2 Structure of an Applet

Usually, Java Card classes have a particular structure. There are four methods (two mandatory and two optional), in addition to the constructor, which are commonly present in that classes. These methods, always called by the JCRE, are: deselect, install, process and select. Their use is described at the following.

- Deselect is a method called when the applet has been deselected. It is not mandatory unless any cleanup operation was necessary.
- Install is called only once during the installation. It is the responsible for creating an instance of the applet on the card. It must be overridden.
- Process receives the APDU command. As the card works as a server, this is the method which processes the command and decides what to do to reply it. Usually, it is considered a good practice to build the process as a case statement which decides what method from the class will reply the command according to the INS field received. This way to organize the code is useful to make it clearer. It also must be overridden.
- Select is similar to deselect. It has not to be overridden unless some initialization operation was necessary.

The code is completed with constants, objects declarations and other methods to help to reply the command.

5.1.2.3 Why Not to Use Java Card 3

Java Card 3 (JC3) is the latest version of Java Card. It has been more wide explained at multiapplication main standards section

When the implementation was starting, it was tried to use Netbeans as programming environment with its plugin for JC3. It worked when the entry was a script, but it was not possible to get a communication between an off-card part and the card. That was the basic feature of the system to be built, taking in account that its goal is to make the communication secure. There was not
found any tutorial or manual which could help to build the more basic communication between these two parts. This lack of information or examples for JC3 is caused because only a prototype can be built in this version. In other words, there is no any real implementation yet for JC3; there is no any real card which supports JC3. That is why the most of Java Card developers work on the previous version and JC3 is only used by some researchers. Hence, there is no many support and the information or manuals are scants.

Also, not many smart card experts talk positively of JC3. They complain because JC3 add a big overhead over the card response time owing to there are excessive abstraction levels in order to the bytecode was executed. Also, servlets increase transaction times. They claim that is not suitable to substitute APDU protocol due to although it is a bit more complicated than other protocols, it fits very well to smart card’s constraint resources. So, any change on that will suppose a loose of speed on-card. Some experts justify the change explaining that the new version is more suitable to target USB dongles and that is why the two editions (classic and connected) are created, maintaining backwards compliance by means of Classic edition which should be able to work with APDU.

Another point to be aware of is their cost. Although the memory and power of the card were improved in JC3; it supposes the cost of the card is higher too.

For all the exposed (mainly for the support), it was decided to move to Java Card 2.2.2 version. It is a reliable and common version used by most of Java Card developers. Anyway, JC3 it is supposed to be backward compatible; hence, in the event of moving to version 3 the changes should be small.

### 5.2 Limitations of the Prototype: Use of Eclipse, CREF and JCWDE

Eclipse is a multi-language software development environment which includes an Integrated Development Environment (IDE). It provides a comfortable way to develop software at the same time that it may be tested. For the goal of the current project, it exists a plugin to work over Java Card. Eclipse is a free and open source software. The version used has been Eclipse SDK 3.5.2. The main reason to use Eclipse instead of Netbeans (which was used at the beginning, as above mentioned), was the problems to get a communication between the card and an off-card part in the latter. In contrast, by means of the manual in [11], the basic communication between both parts was straightforward in the Eclipse environment. In addition, Netbeans only provided a plugin for the last version of Java Card, that is JC3; therefore Eclipse was the logical and natural choice.
Moreover this is a prototype, a proof of concept, a way to demonstrate the correctness of the system designed; therefore the choice of Eclipse and simulators to carry out the implementation is more than appropriate.

During the implementation phase, two simulators have been used, allowing both to communicate the card (emulated card) with a CAD via socket interface. They are:

- Java Card platform Workstation Development Environment tool (JCWDE) for the first part of this phase
- C-language Java Card Runtime Environment (CREF) for the second part

On one hand, JCWDE is a tool which allows running an applet simulating that applet is masked at a ROM of a Java Card; that is, the applet run as if it were stored on a card. To do that, it makes use of the Java virtual machine to emulate the JCRE. In other words, it uses a subset of the Java virtual machine to emulate a JCVM. That is why some of the features of the actual JCRE are not supported like package installation, persistent card state, firewall and transactions. Its main advantage is that as the applet has not to be stored on the card (because it does not save the card state), it is faster to test with it. That is the reason why it was used for the first part of the implementation stage where the structure of the system and communication were implemented and for the whole initialization phase. At this part, the card state was not stored (from one running to the next), only while the applet is running. To learn more about JCWDE, read chapter 4 from [2].

On the other hand, CREF is a similar simulator written in the C programming language which allows testing applets in a emulated environment in a very accurate way. The main difference is that it may be built with a ROM mask which allows it to simulate the persistent card memory (EEPROM), like a real card which saves its state. That means applets can be installed on the card. The main and obvious advantage over JCWDE is that it can save the card’s state; but also, it supports some features that JCWDE does not, like transactions. To learn more about CREF, read chapter 10 from [2]. This simulator was used for the second part of the implementation stage when the initialization phase was already finished. Then, it was necessary its use because contract-policy matching phase needed to work with a card which has already stored the certificates, contract, etc. and which was able to use its cryptographic keys.

Both simulators have a problem which limits seriously their performance. They are designed with the goal to be useful for the people who is starting to work
with Java Card. By means of them, they can create and test their own applets. That is why they do not support the whole of cryptographic classes and methods which Java Card 2.2.2 really provides. They only provide a basic security and cryptographic classes that \cite{2} details at chapter 13. It restricts key lengths, available cryptographic algorithms, etc. Indeed, the restrictions caused the system design was modified for the prototype. It may lead to an error: theoretical system design has to be separated from the prototype’s implementation. The system design presented at the previous chapter is the correct and which should be used in a real implementation. But, due to these simulators are the only available and what it was expected to build was a prototype as a proof of concept, it was decided to modify as less as possible the necessary parts to get the prototype working on that simulators. The main part which had to be modified was the CSR’s generation and the verification of the certificates’ signature, because of the lack of support for the RSA cipher implementation without padding. Also random number generation and key lengths have been affected. Every part which was changed will be properly explained at its corresponding section.

Consequently, two different source codes are provided in the appendixes. Source code in Appendix B belongs to the code for the prototype, whilst source code in Appendix C belongs to the code for a real implementation.

### 5.3 Communication and Involved Parts’ Structure

The communication among involved parts is the basis over the whole system is built; hence, their structure and the interaction among them was the first stage in the implementation.

Card works as a server, that is the reason why its process method is limited to assure that the request was replied calling the corresponding method. To select the correct method to be called, process method makes use of a case statement by means of received INS. To work with CLA, INS and SW, there are a lot of necessary constants at the beginning of the applet’s code. These constants are also duplicated within off-card applications which make use of them. After the constants, necessary attributes are placed. Among them, it is worth mentioning the bytes array to store contract and policy and the booleans which store whether the certificates, contract and policy have been stored or not. They are used to carry out verification to check if everything is properly stored to run the matching algorithm. Before process method, Constructor, Deselect, Install, and Select are included. Constructor creates instances of algorithms and
AES keys and generates RSA keys. Install calls constructor and registers the applet on the card. Deselect and select make cleaning operations. The rest of the code of the applet, is divided among some useful methods for utilities like converts between data types (e.g., short to byte array), cryptographic methods and APDU management for extended length.

The equivalence among entities of the system design and the implementation projects and source codes files is shown in the Table 5.1.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Project</th>
<th>Source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC</td>
<td>SmartCard</td>
<td>VerifyPolicyAndClaimApplet</td>
</tr>
<tr>
<td>TR</td>
<td>TrustedReader</td>
<td>InitializationClass</td>
</tr>
<tr>
<td>TTP</td>
<td>TrustedThirdParty</td>
<td>TerminalClass</td>
</tr>
<tr>
<td>AI</td>
<td>ApplicationIssuer</td>
<td>ContractDelivery</td>
</tr>
</tbody>
</table>

Table 5.1: Equivalence among entities, projects and source codes

The InitializationClass’s main method, responsible for the initialization phase, is presented as a menu. According to the option chosen, corresponding method is called. The APDU communication is carried out by means of methods which work on it. They try to abstract the details of APDU management and make it more general and easy to use. Thereby, to send an APDU command without data it is only necessary to call two methods: prepareAPDU and exchangeAPDUs, omitting low-level details about the values of the octets of CLA, INS, etc. Result can also be checked by means of checkAPDUDStatus method. These methods are also available at TerminalClass and ContractDelivery code. To complete InitializationClass’s code structure, cryptographic and some useful methods are included.

TerminalClass and ContractDelivery main methods are built in a sequential way. They run their operations and if any error occurs they finished properly. They rest of their structure is similar to InitializationClass.

### 5.4 Certificates

The main issues addressed in this section are the certification request and certificate generation, verification and storage, both on and off-card sides. In order to be able to understand that properly, it is necessary to be familiar with formats, Abstract Syntax Notation One (ASN.1) specification and encoding rules and formats of CSRs and X.509 certificates.
5.4 Certificates

ASN.1 is a notation used to describe both abstract types and values defined by Open Systems Interconnection (OSI). Every ASN.1 object has a tag which is made up of a class and a tag number. Only tag number identifies unequivocally a type. OSI also defined a set of rules which are used to convert the abstract objects in strings of ones and zeros. This set of rules is called Basic Encoding Rules (BER). BER provides an encoding for the object which consists on three parts: Identifier, length and content. Both ASN.1 and BER are explained wider at [18]. Finally, DER are a subset of BER which apply some restrictions getting a unique possible encoding for every object, in spite of BER which allows several encodings (this is not suitable for instance to check digital signatures on certificates). To know more about DER [21].

On one hand, CSR’s ASN.1 specification can be studied at Listing A.1 and A.3 and is described at [29]. On the other hand, ASN.1 specification for X.509 certificates can be observed at Listing A.4 and A.5. X.509 are detailed at [7] and [40]. X.509, although it can be large in size, is the most widely used certificate format.

The remainder of the section is organized in the following way. It first deals with the implementation of both certification request and certificates generation. Once the generation has been detailed, certificate’s storage and management are described on and off-card sides. Finally, the verification of the certificates on-card is explained through the parser implemented.

5.4.1 CSRs and Certificates Generation

This section examines the implementation of the CSR and Certificate generation. These tasks are carried out by means of the TR. As it has been described at previous chapter, CSR is supposed to be built in TR and signed on-card. After that, card issuer should send the CSR to a CA in order to get the expected certificate.

CSRs to encrypt and sign are built in the same way. First, TR orders public key to the card. Once it is received, CSR structure is built and prepared to be signed with the private key on-card. Main options to create the CSR are to make use of:

- Bouncy Castle
- Java sun.* packages
Bouncy Castle is, roughly speaking, a collection of lightweight cryptography APIs for Java. It provides a way to create CSR, but it is necessary to have the private key of the entity to be certified in order to sign it. That means it does not allow building a CSR structure to be sent to the card and signed there. Hence; the only way to create the CSR by means of Bouncy Castle is exporting the private key from the card, which is a significant security risk.

Java sun.* packages allows developer to create the handmade CSR thanks their classes to work with DER and Base64 Encoding. Once CSR structure has been created, DER encoded digest info structure has to be sent to the card in order to be signed. This one, called Encoded Message (EM), is built as follows ([26]):

\[
\text{EM} = 00 \ | | 01 \ | | \text{PS} \ | | 00 \ | | T,
\]

- \(T\) is the DER encoding of digestInfo structure (ASN.1 specification is at Listing A.6)
- \(PS\) is an octet string of length \(k-3-||T||\) with value FF. The length of \(PS\) must be at least 8 octets
- \(k\) is the RSA modulus size

EM is what has to be sent to the card. At this time a problem with the simulator appears. EM has been built according to PKCS#1 v1.5 (Public-Key Cryptography Standard) scheme, that means padding has been already added. That is why it should be encrypted on-card with the RSA implementation which does not add any padding; that is: ALG_NO_PAD. But the simulator does not provide this implementation (although JC 2.2.2 does), as it may be checked at chapter 13 of [2]. For this reason, the design of this part was modified to adapt to what simulator supports. Change consists in export the private key to the TR in order to sign by means of the needed algorithm off-card. It is shown at Figure 5.1. The source code which contains this implementation for the prototype can be observed at Appendix B.

It is worth mentioning that in the case of a real card implementation, Java Card 2.2.2 provides the necessary method; thus, it would not have to make this risky change. In other words, the change is only acceptable for the building of the prototype with the simulator; the private key never has to leave the card.

Once CSR is built and sent, CA authenticates the requesting entity and verifies its signature. If the request is valid, CA constructs a X.509 certificate. Certificate is built by means of the distinguished name and public key got from the CSR, and adding the issuer name, corresponding serial number, validity period, and signature algorithm from the CA. If any PKCS#9 attribute is included
Two are the main tools to carry out the creation of the certificate from the CSR, due to Java does not have any support for this operation: Keytool and OpenSSL. Both are tools which can be used to generate and display X.509 certificates. On one hand, Keytool, developed by Sun, is a tool to manage keys and certificates; usually used as a database to store keys and their certificates which authenticate them. On the other hand, OpenSSL is a wider goal tool which provides a toolkit to work with secure communication protocols and cryptographic library. The reason to choose OpenSSL was that it was closer to what it would be the real world. The behavior of this tool is exactly what it was necessary: it receives the CSR and by means of the CA’s key pair, it generates the certificate. After that, it finishes its work with certificate, CSR and keys. In contrast, with Keytool, certificate and keys are held in its database.

Generating the certificates with OpenSSL consists in moving the CSRs to the OpenSSL’s folder, run by command line the commands shown at Listing 6.4. When the command finishes, the certificate is ready. CA certificate used to do that is a self-signed certificate, because it was suitable for the developed prototype (free and quick to get). This is the task of the TR-Certificate Manager aforementioned above.

5.4.2 On-Card Certificates

They are stored as byte arrays because Java Card does not provide any other data structure to work with them. During the lifetime of the smart card, sev-
eral certificates are sent to the card for different purposes. On one hand, SC certificates (to encryption and signature) are sent and stored on the card for the whole smart card’s lifetime. Also, CA certificate. On the other hand, in every Contract-Policy Matching session two TTP certificates are sent to the card. These are not stored definitely, but temporary, during the necessary time to parse the certificate.

Difference is that while CA’s and SC’s certificates are stored as attributes because they will be used during the whole smart card lifetime, TTP’s certificates are stored in a ’temporary’ byte array (it is only one attribute because they are not sent at the same time, but sequentially). Temporary is between quotation marks because the variable where current TTP certificate is stored is a global, what means that it is alive during the whole smart card’s lifetime. However, certificate is stored only while it is being verified for the parser; because the variable is cleaned (the whole components of the variable are put to zero) once parser has finished. Verification is over, that is why it is not necessary to save the certificate longer. Auxiliary byte array which stores TTP certificates while they are parsed is currentCertificate. Reason, to be global (i.e., attribute) instead of a local (temporary) variable inside the method which receives the certificate, is to save space on the JCVM stack. JCVM has a stack-oriented architecture. That means every applet has its own stack \([51]\) which stores frames (i.e., pieces of each method called which stores environ, local variables, etc.). Every frame stores its parameter taking up space from the stack. If the certificate is stored as a local variable, this one has to be passed as a parameter to the methods which work with it. The usage of nested methods may get that the certificate was stored several times in the same stack wasting space and time what is not suitable for a constraint resources device. As bigger was the nesting level and the certificate size, bigger was the wasted. To avoid that, a global variable is created and every nested method calls it, saving the time of pass the certificate as a parameter (to copy and delete from the stack) and the space in the stack. Something similar is what concerns to parserOffset attribute which stores the current offset used to parse the current certificate.

5.4.2.1 Encoding for On-Card Certificates

Possible ideas for the format to store on-card certificates might be:

- As a certificate data structure sent from the off-card reader to the card
- As a string
- Base64 encoded (how the CA gives it back)
• DER encoded

First and second option are ruled out because of the lack of support in Java Card.

Base64 is not a good idea at all, mainly for two reasons. First one is that although the Base64 encoding were decoded, the certificate would not be usable yet, because it would follow DER encoded, that is why it would be better send it DER encoded directly. Let’s think the steps to send certificate Base64 encoded: Two tiresome steps are added if certificate is sent Base64 encoded.

1. TTP DER encodes certificate contained in its data structure
2. TTP Base64 encodes
3. Certificate is sent
4. Certificate is Base64 decoded
5. Certificate is parsed while is DER decoded

The second important reason in order not to send the certificate Base64 encoded is that there is not any support in Java Card to work with it; that is why a new Base64 class to decode should be implemented to work on-card. In contrast, Java Card provides a class to work with BER-TLV (Tag, Length, Value). As it has been explained previously, BER is more generic than DER, and consequently, any tool to work with BER, it will do with DER.

To sum up, certificates are stored DER encoded (as byte array) because there is class which provides support to work with it and it allows to save time avoiding some steps. Classes to work with BER are BER-TLV and BERTag. For instance, BERTag provides a useful method to get the tag number from the tag structure byte, whilst BER-TLV provides another to verify whether the certificate is properly encoded.

5.4.3 Off-Card Certificates

This section includes certificates on TR and on TTP. Also it addresses TTP’s private keys issue since they are stored in a similar way than certificates.

Certificates of TTP and TR are stored in files permanently. To retrieve certificates from their respective files, a Byte Array Input Stream is used. Thanks
to the classes provided by Java to generate it, a new instance of a X.509 is created containing the certificate, which is Base64 encoded within the file. This file is stored in the device where TTP or TR is running with Privacy-enhanced Electronic Mail (PEM) extension. PEM is a format which can store private and public keys (both RSA and DSA) and X.509 certificates. It stores the certificate Base64 encoded after applying DER format and wrapped between two headers (BEGIN and END CERTIFICATE). All this encoding is suitable to transmit it among systems.

But all the previous information about how to work in Java with certificates does not fit with TTP private keys. TTP keys are created by means of OpenSSL and stored in their respective files: TTPCertEnc, TTPCertEncPrivate, TTPCertSig and TTPCertSigPrivate, all with PEM extension. While the public keys’ files are certificates which contain them, private keys are Base64 encoded. But Java does not provide any class to work with Base64 encoding in its API (sun.* packages provide, but with the aforementioned problem). To solve that, by means of the last command of OpenSSL from Listing 5.1 Base64 encoding is removed allowing to work only with DER encoding, which is comprehensive by a PKCS#8 class provided by Java API. Therefore, private keys are stored with DER extension.

Listing 5.1 shows the commands necessary to build a key pair which can be used by the system. At this example the key pair and public certificate for TTP to encrypt are created. The first command create the key pair, while the second command by means of the CA key pair gets a certificate signed by a CA. Finally, the last command makes the conversion from PEM to DER.

```
openssl req -newkey rsa:512 -subj "/OU=Cryptography/CN=TTPEncrypt" -keyout TTPCertEncPrivate.pem -out TTPCertEncAux.pem

openssl x509 -CA CA_publicKey.pem -CAkey CA_privateKey.pem -req -in TTPCertEncAux.pem -days 3650 -sha1 -CAcreateserial -out TTPCertEnc.pem -extfile config.txt

openssl pkcs8 -topk8 -inform PEM -in TTPCertEncPrivate.pem -outform DER -nocrypt -out TTPCertEncPrivate.der
```

Listing 5.1: OpenSSL commands to create a key pair usable by the system

Certificates on TR. Its use is simple because it is limited to verify CA and SC certificates before sending them to the card. TR only has to get the certificates from their respective files, creating a certificate data structure. By means of that structures, certificates can be verified easily and converted in a DER encoded byte array to send to the card.
Certificates on TTP  In the same way that TTP certificates on-card, SC certificates are not stored permanently in TTP. They are only present while TTP verifies them and get their public keys. Once, it finishes with them, local variable which stores them disappears. Only SC public keys are stored during the rest of TTP’s lifetime. The rest of the content of the certificate is ignored, after verified. It is worth mentioning that TTP gets from its certificates its public keys.

5.4.4 Parsing on-Card

Parser on-card is the responsible for verifying the correctness of the certificates which are received. In order to carry out this verification, it has to check:

- Certificate is compliant with DER encoding
- Certificate is compliant with ASN.1
- Key algorithm is the expected
- Signature algorithm is the expected
- Key length is the expected
- Signature is valid

If this verification could not be done on-card, SC would not be able to trust in any TTP since it would not be sure about its identity. In other words, what parser allows is to make in a trusted way the initial key exchange. Card knows then whether the certificate is trustworthy.

It is worth mentioning that it was considered it is not necessary to parse SC’s certificates when they are stored because they come from TR (which is trusted environment).

5.4.4.1 What to Parse on-Card

A common certificate comes with quite information about the owner of the public key and issuer CA. All this information which is useful in Web developments, it is not in smart cards because of the limitations of these devices to check information’s truthfulness.

What parser needs from the certificate to check or store is:
• Signature algorithm: Both from TBSCertificate and Certificate (according to ASN.1 specification of X.509 certificate) structure. It has been checked that both algorithms match.

• Issuer

• Public Key Algorithm

• Public key

• Signature

Certificate:

Data:

Version: 3 (0x2)
Serial Number: 94:1b:92:b0:e1:10:a9:9d
Signature Algorithm: sha1WithRSAEncryption
Issuer: C=DK, ST=Lingby, L=Lingby, O=DTU, OU=IMM.DTU, CN=Secbycontract, emailAddress=s091011@student.dtu.dk
Validity
Not Before: Jun 2 11:56:15 2010 GMT
Not After: May 30 11:56:15 2020 GMT
Subject: ST=Copenhagen, C=DK, O=SCissuer, OU=Cryptography, CN=SCEncrypt
Subject Public Key Info:
Public Key Algorithm: rsaEncryption
RSA Public Key: (512 bit)
Modulus (512 bit):
00:b4:05:c4:0e:...
Exponent: 65537 (0x10001)
X509v3 extensions:
X509v3 Basic Constraints: critical
CA:FALSE
Signature Algorithm: sha1WithRSAEncryption
5f:89:b7:3e:aa:...

Listing 5.2: Example of X.509 certificate contents (Certificate used by smart card to encrypt)

As it may be observed in Listing 5.2 (one of the SC certificates), certificate provides some fields with information useless to the smart card for not being interesting or for the lack of tools to verify them. These fields are Version, Serial Number, Subject (unless the certificate was the CA certificate, case when the offset of this field is stored to be checked against the Issuer field from the rest of certificates) or the most of the extensions (if any, except CA extension).

About extensions, anyone of them is checked on-card. That is why only CA extension would be interesting: It assures that a certificate expected to be a CA certificate, it is really a CA certificate. It may be observed in Listing 5.2 that in the extensions part shows up a field CA marked as false because certificate belongs to the SC. This verification is done in the initialization phase in the InitializationClass. It checks the certificate before sending it to the card.
A function gets the value of *BasicConstraints* extension which stores whether certificate’s subject is a CA or not. As [48] indicates, only returns minus one when the latter option.

At this point, it is worth mentioning that two important issues related to certificates were left aside: Validity and Revocation. [37] deals with both of them proposing some solutions, but it has not them into account in its prototype. A solution is proposed in the [future work section] but they are ignored during the implementation.

### 5.4.4.2 Parser’s Implementation Details

This section will focus on providing some details of the parser’s implementation. Verification process of a certificate, which the parser carries out, can be separated in two parts: the actual parser of the certificate and the verification of the signature. First one checks whether the format and the encoding of the certificate is correct and gets the data elements to be verified. The second one, when the first part has finished and has given back the signature, checks its correctness.

Parser takes some ideas from [37], but adapts them to its needs. The certificate is scanned once, but it is not necessary to save the offset of the fields because their contents will be got during the parser process and the only one which is necessary to store is the public key, but it is stored directly instead of saving its offset and be stored later. Therefore, in contrast to the parser implemented in [37], the content’s offsets are not stored, because they will be checked only once.

The only exception is CA certificate which is stored permanently on-card. In order to verify that TTP’s certificate has been issued by CA, issuer field has to be checked against CA’s name field. Thereby, the offset of this field from CA’s certificate is stored in a global variable. In such a way, it is easy and fast to take the CA’s name to make the comparison.

Every certificate BER format is verified by means of a method provided by *BERTLV* Java Card class. It is checked that every expected field is present. As it has been described previously, only a few fields’ contents are really parsed. Issuer field has already detailed. The public key (not from CA) is got according to the expected BER encoding and stored in its corresponding variable: *TTPPublicKeyRSAEnc* or *TTPPublicKeyRSASig*. It is worth mentioning that when the key is retrieved, its length may vary. The reason is that to avoid modulus and exponent from being understood as negative because of its first byte, a zero
is added before. Java Card classes does not work in the same way, thus it is necessary to remove the first component in that cases or the system will fail on-card.

The fields which are left out are signature and public key algorithm, both are algorithm identifiers according to ASN.1 specification. Both contains wrapped in a SEQUENCE statement an Object Identifier (OID) which identifies theirs respective methods. How to DER encode an OID is explained in a revision of the document [21]. But to make it easier, it was implemented several lines of Java code (at Listing 5.3) by means of which the OID is DER encoded. That encoding was wrapped in a byte array which is only compared against the OID got from the certificate, both to check signature and public key algorithms.

```java
Oid aoid = null;
aoid = new Oid("1.2.840.113549.1.1.5");
byte[] oid = null;
oid = aoid.getDER();
System.out.println(byteArrayToHexString(oid));
```

Listing 5.3: Code to OID generation (sha1WithRSAEncryption example)

The verification of the signature can be made in two ways. After decrypting with the public key of the corresponding CA, an EM is got. Either hash code is retrieved from that Digest Info structure and compared against the hash code built or a Digest Info structure is built through the hash code of the TBSCertificate part (ASN.1 specification) of the certificate and compared against to the Digest Info structure got decrypting the signature. The latter was chosen for the current implementation. While parser is scanning the certificate gets the offset and length of TBSCertificate part and make a hash code which is stored in order to be used to build the Digest Info structure. That one is checked against to the decrypted one.

The problem is coming when the signature has to be decrypted, like it happened with CSRs and certificates generation. To decrypt the signature is necessary to make use of an RSA algorithm implementation with no padding which is not available on the simulators used. It can be observed at Appendix C which corresponds to the real implementation source code where this algorithm is available, that it is enough with decrypting and following the aforementioned process to compare Digest Info structures. Since the algorithm is not available, it is necessary to modify the system design again for the prototype.

There are three methods which have to parse distinct certificates:

- Method which receives TTP certificate to encrypt
5.5 Cryptographic Functions

- Method which receives TTP certificate to sign
- Method which receives CA certificate

Owing to the previous limitation, this methods were split in two methods each one. Thereby, for instance in the source code of the prototype there are the methods: `processStoreCACertificate` and `processStoreCACertificate2`. The structure of these two methods for the rest of certificates (TTP certificates) is the same. The first method receives the certificate, parses it, gets the public key (if TTP certificate) and gets the signature encrypted. It gives that signature back to the off-card side which decrypts the signature by means of the CA’s public key and an RSA algorithm implementation with no padding. Once decrypted, it sends the `Digest Info` structure to the card, being processed by the second method. It checks if the `Digest Info` received matches with the built. If it was not a CA certificate, the card replies with its corresponding certificate. The whole process is shown at Figure 5.2. The result is that is necessary to send one APDU command and response more which make the system more complex. The source code which contains this implementation for the prototype can be observed at Appendix B.

![Signature verification process on the prototype](image)

5.5 Cryptographic Functions

This section is related to the implementation of the cryptographic functions in the second part of the Contract-Policy Matching process, after the exchange of
Implementation of the Prototype

certificates. It can be distinguished between the part of the Symmetric Block and the RSA Ciphering, although it is by means of both when the encryption system makes sense.

Symmetric block cipher chosen was AES instead of DES (both are the provided by the simulator). The reason to use the first one is that it was proved DES is not suitable to transmission of sensitive information. It is vulnerable to differential and linear cryptanalysis because of the usage of weak substitution tables; and its length is too short taking into account the bits used for odd parity \[8\]. In contrast, AES shortest key length is more than twice the DES’; and AES has been built with the aim of being strong against differential, truncated differential, linear, interpolation and square attacks.

Length used for the AES key is 128 bits because it is the only length provided by the simulator. Since it is a block cipher, it works with blocks of a fixed number of bytes in size, 16 for the length used. That means the length of the input has to be 16 multiple; hence a padding is needed when the length is not. Padding bits are randomly added. In order to recover the encrypted message properly, the length is also added at the beginning.

This block cipher has several modes to work which depends on the implementation chosen to encrypt. The most common are Electronic Codebook (ECB) and Cipher Block Chaining (CBC). The latter is the suitable because it avoids cipher from leaking information about the structure of the plaintext \[31\]. In contrast to ECB which does not operate among different blocks of the encryption, CBC makes a \textit{XOR} of the data with the encrypted bytes of the previous block, avoiding any information leak. Fortunately, this mode was supported by the simulator and it could be possible to make use of it. The problem is that this mode needs an Initialization Vector (IV) which has to be used in both sides of the communication in order to be able to complete the encryption and decryption process properly. Several approaches can be set out:

- Create at the beginning of every session a new IV
- Use every time the same IV
- Use NONCE as IV

The first approach implies higher traffic, slowness and complexity. Second one is easy to implement and faster, but insecure (it is used the same IV in every communication of the card’s lifetime). Finally, NONCE fits perfectly to the needs of the IV. It is random and created newly every session. In addition, it is already necessary to send it; thereby, it is not suppose to increase neither
complexity nor traffic. That is why the first part (16 bytes, i.e., AES block size) of the NONCE is used as IV.

On the other hand, RSA encryption has been carried out by means a key length of 512 bits, because again, it is the only key length supported by the simulator. At Contract-Policy Matching process, in contrast to the certificates’ issue, it does not matter which algorithm to use, if it is the same in both sides. That is why, the implementation provided by the simulator was useful at this time: it was used a RSA cipher which pads input data according to the PKCS#1 (v1.5) scheme.

A significant security issue is the use of Pseudo-Random Number Generator (PRNG) algorithm in the prototype. This is done again for a limitation of the simulator which only provides this implementation. However, the use of PRNG is a serious security threat. Therefore Secure Random Number Generator (SRNG) should be used. As a matter of fact SRNG has been specifically designed for being used in cryptographic requirements as [49] specifies. The problem with PRNG is the predictability of the linear congruential algorithms used which follow a deterministic sequence according to the values of its parameters. An attacker which could get a small part of the sequence will be able to determine the parameters of the algorithm, and consequently the subsequent numbers [46].

As a final comment for this section, it is worth mentioning that for a real implementation it is highly recommended to increase the length of the keys, both AES and RSA (strongly the latter), used for the prototype’s implementation. They are too short (128 bits for AES and 512 for RSA) because of the simulator’s limitations. For AES it is not a really significant problem since it is not too small and because it is used only once and the time to break it is much bigger than the time while it is used. Nevertheless it is relevant to RSA whose keys are expected to be used for a long time and its length is very short. Therefore it is strongly suggested to increase the length up to 2048 bits for a real implementation. Further details for changes for real implementation at Appendix B.

### 5.6 Some Other Implementation Details

At this section, some important implementation details which do not fit in the previous sections are described.

There is some possible attack which can be carried out when an error occurs. For
instance, the parser reads the TTP’s certificate only once checking and getting the fields which needs, avoiding storing the offset to be accessed later. That is why it stores TTP’s public key when it arrives to the corresponding field. If the signature is not valid afterwards, an error appears but the public key has been already stored; hence, an attacker may make use of that. To avoid this and similar situations, clean method is called every time an exception is launched on-card. This method clears or initializes to zero the content of several attributes contained in EEPROM, making them useless after the cleaning operation. It is also called after the end of any phase.

One of the Java Card 2.2.2 features is that there is no automatic garbage collector, but over demand. That means if it is not called every instance created within a method is permanently stored on-card, even a new called to the same method will create a new instance which will be stored on-card. The consequence is logic: after an undetermined number of uses of the applet on-card, the card will be without memory resources. That is why Java Card manuals advise to reuse variables and use global attributes, instead of creating new instances in the methods. Therefore, garbage collector is called every time a request arrives to the card in order to avoid it getting without memory resources. The code is shown at Listing 5.4

```java
if(JCSystem.isObjectDeletionSupported())
    JCSystem.requestObjectDeletion();
```

Listing 5.4: Code to Garbage collector over demand

The issue of the instances created within methods different to constructor led a problem. If an instance for one algorithm (a cipher for RSA algorithm instance for example) is created out of the constructor, the second time that the method is called and consequently the instance created again, an error occurs and execution is stopped. That is the reason why all this algorithms have to be instanced within the constructor. That the case of Cipher, Message Digest and Random Data classes.
Throughout this section it will be detailed a Case Study whose aim is to present the usage of the prototype developed together the needed tools to run it. Last part of the section assesses the performance of the application in terms of memory.

In order to work with the card saving its state CREF is used, whilst the clients run in Eclipse. That is why it is necessary both tools were installed to be able to work with them properly. [11] provides an excellent guide to install and get them work properly. For the goal of the section, this part is considered as a previous step which is not be longer detailed with the exception of the next paragraph.

It is worth mentioning that depending on the Java version in which it was working, a problem might appear at the moment when the applet is trying to be deployed on the file which stores an image of the EEPROM. This error could say something similar to: "unsupported class file format of version 50.0". It is related to the existence of some conflicts between the format which is expected for a JC 2.2.2 file and the file which the current JDK creates (notice that it is the JDK which simulates the JCVM, and consequently, the JC 2.2.2 file is generated by it). To solve that is necessary to change the compliance level of the JDK compiler. In Eclipse’s tool bar→Window→Preferences: Java→Compiler: JDK compliance: Compiler compliance Level from 1.6 to 1.4. Immediately after se-
lecting 1.4; it appears in the bottom the following message: "When selecting 1.4 compliance, make sure to have a compatible Java Runtime Environment (JRE) installed and activated (currently 1.6). Configure...". If Configure is clicked, it carries user out to Installed JREs where clicking to Execution Environments, it is possible to choose the compatible JREs for the Execution Environments which will be used. Select jre6 in every case.

On one hand, let’s define ApplicationContract as the contract which will be compared against the card’s policy. It will be stored on a file within ApplicationIssuer folder with the name: "Contract". This contract specifies the behavior of the application which is wished to be installed on the card. On the other hand, the card’s policy which contains the security policy of the card, is stored in a file called "Policy" in the TrustedReader folder. Let’s call it CardPolicy. CA certificate is stored both in TrustedReader and TrustedThirdParty folder. TrustedThirdParty also stores its certificates and private keys to encrypt and sign.

Let’s call the developed application (which runs Contract-Policy Matching) VerifyPolicyAndClaimApplet. First of all, application has to be installed and initialized on the card. To do that it is necessary to create a file which stores the EEPROM of the card in order to be able to save its status. This is done by means of a script called NewCard.bat which can be found within SmartCard folder. The script’s content is shown at Listing 6.1. First two lines create the card and install the applet. Last one calls other script (OldCard.bat) which is the responsible for running something on the card saving the status afterwards. At this moment it is used to initialize the card (creation and storage of certificates and policy). The content of this script is detailed at Listing 6.2. As it is shown at the scripts, CREF is used as the simulator in order to be able to save the status. To run the scripts is necessary to open a command line and to go up to the folder which contains the script (SmartCard→bin).

```
cref -o VerifyAndClaimApplet.eeprom
cref -i VerifyAndClaimApplet.eeprom -o VerifyAndClaimApplet.eeprom
OldCard.bat
```

Listing 6.1: Script to create, install and initialize a new card

```
cref -i VerifyAndClaimApplet.eeprom -o VerifyAndClaimApplet.eeprom
```

Listing 6.2: Script to load the status of an already existent card

When NewCard.bat script is called, it keeps waiting for the sending of APDUs which ordered the installation of the applet on the card, as it can be observed at Figure 6.1. Then, card issuer should right click over the package security on the application in Eclipse, go to Java Card Tools and select Deploy as it shown
at Figure 6.2.

Figure 6.1: New Card script running

Figure 6.2: Deploying applet

After that, the command line has changed, but still will be waiting more APDUs. Similarly to the previous order to deploy the applet, Card Issuer should right click on create-VerifyPolicyAndClaimApplet script, Java Card Tools and choose Run Script, which will install the applet on the card making the application ready for its use. But it is still left out the initialization, and if it is observed again the command line it is waiting for more APDUs yet because of the last line of the script has been executed. InitializationClass should be called now. To do that, card issuer should go to TrustedReader project in Eclipse, open InitializationClass source code and over it, right click, Run as, Java Application as it is shown at Figure 6.3. The menu of the Figure 6.4 should appear.
First, it should be chosen option 1. It creates two CSRs for the keys on the card: one to encrypt and the other to sign. Both files are stored within TrustedReader folder with the names: EncCertReq.pem and SigCertReq.pem. These CSRs have to be sent to the CA to get the certificates. Since OpenSSL does that, it is necessary that files which contain the CSRs were dragged to bin folder within OpenSSL folder. Previously, a CA should have been created and stored there. To do that, use Listing 6.3 establishing a password. A private key and a self-signed certificate will be created.

```
opasswd req -x509 -newkey rsa:512 -keyout CA_privateKey.pem -out CA_public.pem -days 3650
```
Listing 6.3: Command to create a CA certificate and private key

Once CSRs are located at bin folder, open a new command line, go to this folder and run the commands of the Listing 6.4 in order to get the certificates. It will
be necessary to write the password inserted previously during the CA generation. At Figure 6.5 can be observed the certificate generation by command line.

```
openssl x509 -CA CA_publicKey.pem -CAkey CA_privateKey.pem -req -in EncCertReq.pem -days 3650 -sha1 -CAcreateserial -out SCEncCert.pem -extfile config.txt
openssl x509 -CA CA_publicKey.pem -CAkey CA_privateKey.pem -req -in SigCertReq.pem -days 3650 -sha1 -CAcreateserial -out SCSigCert.pem -extfile config.txt
```

Listing 6.4: Commands to get certificates from CSRs

As a result of the previous generation, two certificates should have been created in the folder: `SCEncCert.pem` and `SCSigCert.pem`. The content of the folder is expected to be similar to the shown at Figure 6.6. To sum up, it should contain at least both CSRs and certificates, CA certificate and its private key. When both certificates have been generated, card issuer must to drag them to `TrustedReader` folder, and then, option 2 of the Initialization Phase menu can be launched. It will store CA and SC certificates on the card. A screenshot of its storage can be observed at Figure 6.7. At this moment both CSRs and certificates should be removed from OpenSSL and `TrustedReader` respective folders.

Finally, Card Issuer must store `CardPolicy` choosing option 3. Once everything aforementioned has been done, what supposes CA and SC certificates and Policy have been stored, Card Issuer can finish with the initialization phase choosing option 4. Notice that the `TrustedReader` can be called only once in the sense that, since the moment when certificates and policy are stored on the card, they cannot be stored again. Certificate’s renewal will be discuss in the future work section.
Figure 6.6: OpenSSL\bin folder after certificates generation

Figure 6.7: Certificates storage on-card
At the end of the initialization phase, the card can be used as many times as required to check the compliance of any change of the applications on the card. To do that, before running the Contract-Policy Matching algorithm, it is necessary to load ApplicationContract. That is carried out by Application Issuer who is the responsible for sending the contract to the card. Hence, ContractDelivery class has to be run on Eclipse before TerminalClass. Otherwise, the error shown at Figure 6.8 appears. This error also appears if some of the elements expected to be on-card when the algorithm is run (like certificates) are not already stored. Every error which can be captured on-card, it is shown with the same error modal window than the previous one.

![Figure 6.8: Message when ApplicationContract has not been stored](image)

To run ContractDelivery class, it has to be run OldCard.bat script. After that, right clicking over the corresponding class and choosing Run as, Java Application, like to run the TrustedReader, ApplicationContract is stored on the card. The APDUs involved at this process can be observed at Figure 6.9.

![Figure 6.9: Contract storage process](image)

The last part is the Contract-Policy Matching. In order to avoid a message like the shown at Figure 6.8, certificates, contract and policy have to be already stored on the card. Notice that when a matching algorithm finishes, contract is removed; hence, any contract has to be stored again before running the algorithm again. Like the other clients, to run TrustedThirdParty, it is necessary to
run the *OldCard.bat* script and from Eclipse, right click on the source code, Run as, Java Application. The communication will start and at the end a modal info window communicate which has been the result of the algorithm, as it can be seen at Figure 6.10.

![Figure 6.10: Message with the matching result](image)

Together the window with the result, the APDUs which have been sent during the communication between TTP and SC can be observed. They are shown separated according to the parts of the communication, as Figure 6.11 illustrates.

![Figure 6.11: Contract storage process](image)
6.1 Memory Analysis

Smart Cards are devices with constraint resources particularly in terms of memory. There lies the importance of this analysis, because if the theoretical idea needed to take advantage of more than the half of the available memory, it would not be the suitable solution.

The analysis has been done by means of the data provided by a command line option of CREF. This option prints resource consumption statistics of memory usage at the startup and the shutdown of the card, giving an idea of the needed resources for installing and executing the applet as it is explained at Chapter 10 of [2]. Every time that CREF is launched with this option the statics related to the following resources appear (once at startup and once at shutdown):

- EEPROM
- Transaction buffer
- Stack usage
- Clear-on-reset RAM
- Clear-on-deselect RAM

Also some statics about the ROM are shown at the beginning of the statics. At Appendix D screenshots corresponding to the memory statics of each stage of the applet lifetime can be observed. To sum up, the results of these statics are shown at Table 6.1. There only results related to EEPROM are included. The reason why only these statics are shown is because are the most interesting for a smart card developer. On one hand, ROM is the memory which stores binary codes of the OS and the JCVM among others. This memory is created and initialized by the smart card manufacturer and it is not able to be modified later. That is why it is lacking in interest for a developer who cannot alter it. On the other hand, RAM is the memory which stores the whole application which is running at every moment and its data. This is very important due to if an applet needs for its running more memory than RAM provides and uses it up, an error appear because of RAM memory resources of the card are exhausted. However, that is a problem which every developer has to keep in mind when it is working on smart cards; but it is not interesting from the point of view of the project. That is because the RAM is always the same and the developer should know with which size it can work and to adapt its development to that. Also RAM is cleared at every shutdown and cleaned by the garbage collector over demand; hence, it changes every card-tearing. That is why the statics are
referred to EEPROM memory which stores the applications and data which are
dynamically loaded to the card; load which is tried to be properly managed by
the S×C framework. The key point of checking the memory statics is to know
whether it is worth adding the system developed to the card or in change, it
takes too many memory resources reducing excessively the space on-card and
making multi-application framework useless. A structure of the non-volatile
memory can be observed at Figure 6.12 from [38], which shows the common
content of the ROM at the bottom (OS, JCVM, etc.) and of the EEPROM (the
applets) at the top.

The statics of the Table 6.1 are shown in bytes. They have been taken from the
prototype developed by means of CREF. This simulator provides an EEPROM
memory with a size of 64 KB (i.e., 65536 bytes). The common size for Java
Card 2.2.2 real implementations ranges between 32 (old and constrained) and
128 KB, although it is starting to use greater.
The stages chosen to appear on the table are compliant with the highlights in
the applet lifetime. They cause the main changes over the EEPROM. These
stages are the following:

- Deployment: It consists in download the applet to the card; store the
  bytecode there
- Installation: It is done calling the applet’s static install method which
  install the applet on the card invoking somewhere register method
6.1 Memory Analysis

<table>
<thead>
<tr>
<th>Stage</th>
<th>Figure</th>
<th>Consumed before</th>
<th>Consumed after</th>
<th>Available before</th>
<th>Available after</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployment</td>
<td>D.1</td>
<td>6994</td>
<td>12837</td>
<td>58510</td>
<td>52667</td>
</tr>
<tr>
<td>Installation</td>
<td>D.2</td>
<td>12837</td>
<td>14322</td>
<td>52667</td>
<td>51182</td>
</tr>
<tr>
<td>Initialization</td>
<td>D.3</td>
<td>14322</td>
<td>17919</td>
<td>51182</td>
<td>47585</td>
</tr>
<tr>
<td>Running</td>
<td>D.4</td>
<td>18298</td>
<td>18135</td>
<td>47206</td>
<td>47369</td>
</tr>
</tbody>
</table>

Table 6.1: Memory Analysis statistics in bytes

- Initialization: This stage corresponds with the Initialization Phase detailed in previous sections
- Running: That is the Contract-Policy Matching Phase

The last static (Running) is taken after several runnings in order to see the statics when the memory is established. About the tags used for the columns they are referred to the moment previous and next to the stage was run. For example, ”Consumed before” refers to the bytes which were used from the EEPROM before the corresponding stage started.

As it can be seen at the table, the card reserves almost 7 KB for itself before storing something what seems to be some space for the OS or some card’s needs. Downloading the applet to the card takes almost 6 KB, whilst its installation more than 1 KB. It is worth mentioning that during the installation is when the most of necessary data create their instance, reserving space on the card. That is what happened with keys and algorithms, for instance. The initialization decreases the available memory in 3 KB. At this stage certificates (both SC and CA) and Policy have been stored. As an example of initial Policy for the card, a file of 518 bytes was used. Obviously, this value will change according to security needs of the card, installed applications, etc. Finally, the EEPROM’s consumption of the matching algorithm only makes memory vary a few hundred bytes. To sum up, the system developed needs a rough memory space in the EEPROM of 11 KB.

Previous result has to be taken as an upper-limit for the necessary space on the memory of the developed system since an optimal implementation was not a goal during its implementation. That is why the result of the analysis is positive. There are also several points to take in account:

- If the statics are looked through, what takes more bytes is the applet’s download because of the extensive source code. That is because the applet has to deal with several cryptographic problems, even including a parser
on-card. The common applets are not as large as it what means that a high number of applets are still possible to be stored. For instance, let’s suppose that every application takes between 4 and 5 KB of memory space as average; still more than eleven applets could be stored on-card. If the size is smaller (commonly) the approach is better since much more applets could be stored.

- The heaviest issue is the bytecode download. Notice that the analysis has been done over the prototype which contains some methods necessaries because of the limitations of the simulator, but which there were not in a real implementation. Therefore this performance is expected to be better there.

- It should keep in mind than the smart card like the rest of the current hardware is continually evolving what carries out to think that the available memory will be greater in short time, whilst the necessary space for the system developed will be the same.

- As it is pointed out in the Future work section, the priority while the implementation stage was to get a clear and easy to understand source code. Thereby, an optimization is able (and recommended) to be made over the code, reducing its weight.

Although an optimization should be done over the code, it is considered that the system is suitable and could fit properly in a multi-application smart card. It takes more space than a usual applet since it has to carry out more operations than them, but it does not reduce the available memory considerably allowing to store a large number of applications in a secure way.
This section lays out possible directions of future work. It proposes improvements and solutions to issues left open.

During the implementation stage, the limitation resources (in terms of memory) of the card were taken into account, but not as a priority. The actual priority was to get a clear code easy to understand, since the work with APDU and the cryptographic issues are not trivial for someone who is not familiar with it. Thereby, if in addition to that, the code is complex and with reused variables which name is not representative (as an example of optimization); to start to work with that code may be considerably hard. It is precisely that reason (code’s clarity) why an exhaustive study of the code, would allow developer to get an optimized code in terms of memory. This study should cover the use of the garbage collector, creation of new instances, reuse of variables, extend the use of global variables and attributes, etc. The reason to decide to state clarity of the code as a priority against to optimization is that it is expected that this project is only a beginning and it will be continued in the future.

Other aspect about implementation is to improve CSR generation with the aim of using only the Java API, avoiding classes from sun.* package, in order to get a more stable version of the system and independently of Sun’s code updates. The way to implement that, it is to build the classes working with DER encoding and which allows developer to build a CSR handmade. It should be taken
into account that all the features for a usual certificate are not necessary to be implemented (e.g., extensions can be omitted), due to it will not be used on-card. The goal is to get a CSR which format was compliant with the standard. Since to implement these classes to generate handmade CSRs does not make use of any class not provided in Java Card (due to every encoding of DER should be implemented), the CSR generation could be carried out on-card. In other words, the same work to develop the necessary classes to build a CSR off-card may be done on-card. In such a way, the initialization phase would be less complicated and faster due to two communications less would be necessaries, but it will take more space on the memory.

A regard left aside while the project’s development was to deal with several CAs. To simplify the task it was decided to make use of only one CA. But in real world, the card should store some CA’s certificates in order to be able to verify certificates signed from distinct CAs. To do that implies to think about the way to store the CA’s certificates; taking in account the constrained resources in terms of memory. E.g., it may be interesting not to store the whole certificate, but the necessary fields. Regarding to execution time, also it would be very inefficient if the certificate has to be parsed every time that some part is necessary; that is why it should be though in a way to access quickly to the needed parts. Notice that Java Card does not allow developer to create any new data structure. Some of these issues had been already discussed on [37].

Another line of future work may be oriented to improve the parser implemented. There are some issues which were left aside because of the limitation of the card and considered out of the scope of the thesis. Nevertheless, there are some interesting points to work in. They are mainly related with addressing the verification of the validity period and certificate’s revocation. The problem to check if the certificate is valid according to the validity period is that the card does not have any clock (because of the lack of a uninterrupted supply of power); hence, it does not know the current time. Since it needs the time to check if TTP is trustworthy, the card cannot trust in it yet. That is why the proposed solution is the use of some kind of server, known and trusted by the card and accessible from TTP, which was able to provide the current time to the card. In this communication, TTP behaves as a gateway between the card and the server. This scheme is depicted at Figure 7.1. This scheme would start as a reply to TTP’s certificate sending, due to the card cannot start a new communication. SC sends its certificate to TTP. It should forward the message to Server (S). It gets the data and replies with the date and incremented nonce, both encrypted and signed. The reason to use nonce, encryption and signature are the same that for the system built in the thesis. TTP forwards the message again from S to SC which gets the date and can carry out the verification of the validity period.
The proposed solution for certification problem (whether revoked or not) is very close. The difference is the information which the messages contain. Exactly this problem is addressed at [25]. To point out an idea in contrast to [25] which is too worried about how server works; since the idea is to trust in the server to solve the problem, it does not matter how it did it: either certificate validation protocol or Certificate Revocation List (CRL). The work would consist in designing the communication among SC, TTP and S regarding to the information which has to be sent.

As regards to certificates, there is an interesting issue left aside. That is the renewal of certificates on-card. It was considered out of the scope of the project because it is not directly related to $S \times C$'s topic, but it is something to keep in mind. e.g., if the validity period of some certificate on-card is expired, the card is completely useless. The problem consists in, since SC works as a server, card cannot know every time who is receiving the request from. That means an Application Issuer could store a new Policy before running the matching algorithm getting its non-valid application was accepted without being a TR, for example. The solution in the prototype was to allow TR to store the necessary things on the card only once; that is, SC and CA certificates and Policy are only stored once by the TR. Therefore, any attack which tried to substitute components already stored on-card would fail. The underlying problem is that even the TR could not replace certificates when someone had expired. The idea proposed to solve that is TR also stored its certificate on the card. Therefore, it is able to renew anything of the applet thanks to the digital signature (SC can verify its identity every request). The sending of a certificate renewal would come signed what would identify unequivocally the sender: if the sender is the TR (what can be checked by means of its certificate) the renewal is accepted, otherwise rejected. This solution supposes a change in the design and to add a new stage to the certificates renewal (Figure 7.2 and 7.3), which can be:
• SC certificate to encrypt or sign
• CA certificate
• TR’s certificate (if validity period issue has been addressed)

Instead of being the same TR which has initialized the card the responsible for renewing the certificates, if it is wished that other TR was able to do, it should be necessary to store its certificate also; similarly to multi-CA issue already discussed.

An aspect did not take in account during the development for time constraint reasons are the Global Platform SDs. They are used to enable applications to share space on a card without compromising security of their providers. What it is missing is to extend the project with SD concerning to the installation of applications.
Conclusion

Smart cards technology has evolved and spread very quickly during the last decade. The reason for this development has been the security and tamper-resistant features of their chips which allows user to keep its card’s contents physically safe. Smart cards have progressed to the point of allowing several applications to run on a smart card and to dynamically load, remove and update applications during the card active life; they are open multi-application smart cards. However, its deployment has not been widely spread because of security issues related to post-issuance modification possibility. This problem concerns mainly to the lack of control of the interactions among applications (it is possible that the owner of an application wished that its application was not accessed by unexpected partners). The asynchronous download and update of applications by different parties requires the control of these interactions after the card has been issued.

Security-by-Contract (SxC) framework is proposed as a measure to address this open-issue. It consists in the idea of contract. A contract specifies the behavior of an application in terms of security. Any modification of the card must come with a contract attached in order to be compared against the policy of the card, which defines the security behavior expected, or rather required, by the card. The process of checking whether the contract is compliant with the policy or not is called Contract-Policy Matching and it is the key-phase of the framework. Since the constraint resources of the smart cards a hierarchical model
of contracts and policies is developed, where the models depend on the level of detail required. Hierarchy of models allow to have some benefits in terms of computational complexity or language expressivity. The highest expressivity level requires a too expensive computational effort for the card making necessary to carry out the matching phase off-card. It implies the need for a secure communication of the data sent between the card and the entity responsible for performing the matching phase.

Developed project addresses the issue of outsourcing the $S \times C$ Contract-Policy Matching Phase to a Trusted Third Party. Contributions of the project are threefold. It provides a design to make the communication secure, the implementation of a prototype as a proof-of-concept and some test results which show the design is viable, suitable and define an upper-limit for the memory needs. The solution provides confidentiality, integrity and mutual authentication altogether. They are achieved by means of a Public Key Infrastructure (PKI) where the identities are handled by certificates. Also, a parser has had to be implemented in order to the card was able to verify certificates by itself.

8.1 Limitations

During the development of the project some limitations have had to be overcome. Main limitation was the simulator used to run the prototype which did not support several of the cryptographic algorithms needed to implement a totally secure system. Since the prototype to be built was expected to be a proof-of-concept which demonstrated the feasibility and suitability of the $S \times C$ approach, two implementations were prepared. First one was the prototype implementation which necessarily had to modify the design by means of change which make the prototype secureless for a real card (e.g. it exports the private key). The latter belongs to a real implementation and contains the code ready (but without having been tested) for a real card. It makes use of the algorithms which the simulator do not provide.
Appendix A

ASN.1 Specifications

The current appendix contains the ASN.1 specifications of the Certification Signing Request and X.509 certificates. These specifications could be found also in [29] for Certification Signing Request’s and in [1], [7] and [40] for X.509 certificate’s specification.

A.1 Certification Signing Request

A.1.1 Certification Request

```
CertificationRequest ::= SEQUENCE {
  certificationRequestInfo CertificationRequestInfo,
  signatureAlgorithm AlgorithmIdentifier{{ SignatureAlgorithms }},
  signature BIT STRING
}

AlgorithmIdentifier ::= SEQUENCE {
  algorithm OBJECT IDENTIFIER,
  parameters ANY DEFINED BY algorithm OPTIONAL
}
```

Listing A.1: ASN.1 specification for Certification Signing Request
A.1.2 Alternative Certification Request

```asn1
CertificationRequest ::= SIGNED { EncodedCertificationRequestInfo }
                             (CONSTRAINED BY { -- Verify or sign encoded --
                               CertificationRequestInfo -- })
```

```asn1
EncodedCertificationRequestInfo ::= TYPE-IDENTIFIER.& Type(
                                       CertificationRequestInfo)
```

```asn1
SIGNED { ToBeSigned } ::= SEQUENCE {
                          ToBeSigned ToBeSigned,
                          algorithm AlgorithmIdentifier { {SignatureAlgorithms} },
                          signature BIT STRING
}
```

Listing A.2: Alternative ASN.1 specification for Certification Signing Request

A.1.3 Certification Request Info

```asn1
CertificationRequestInfo ::= SEQUENCE {
                       version INTEGER { v1(0) } (v1,...),
                       subject Name,
                       subjectPKInfo SubjectPublicKeyInfo{{ PKInfoAlgorithms }},
                       attributes [0] Attributes{{ CRIAttributes }}
}
```

```asn1
SubjectPublicKeyInfo { ALGORITHM : ISet} ::= SEQUENCE {
                        algorithm AlgorithmIdentifier {{ ISet}},
                        subjectPublicKey BIT STRING
}
```

```asn1
PKInfoAlgorithms ALGORITHM ::= {
    ... -- add any locally defined algorithms here --
}
```

```asn1
Attributes { ATTRIBUTE:IOSet } ::= SET OF Attribute{{ ISet }}
```

```asn1
CRIAttributes ATTRIBUTE ::= {
    ... -- add any locally defined attributes here --
}
```

```asn1
Attribute { ATTRIBUTE:IOSet } ::= SEQUENCE {
                      type ATTRIBUTE.& id{{ ISet}},
                      values SET SIZE(1..MAX) OF ATTRIBUTE.& Type{{ ISet}{@type})
}
```

Listing A.3: ASN.1 specification for Certification Request Info

A.2 X.509 Certificate

A.2.1 Certificate

```asn1
Certificate ::= SEQUENCE {
                 tbsCertificate TBSCertificate,
                 signatureAlgorithm AlgorithmIdentifier,
                 signatureValue BIT STRING
}
```
A.2 X.509 Certificate

AlgorithmIdentifier ::= SEQUENCE {
  algorithm OBJECT IDENTIFIER,
  parameters ANY DEFINED BY algorithm OPTIONAL
}

Listing A.4: ASN.1 specification for Certificate

A.2.2 TBS Certificate

TBSCertificate ::= SEQUENCE {
  version [0] EXPLICIT Version DEFAULT v1(0),
  serialNumber CertificateSerialNumber,
  signature AlgorithmIdentifier,
  issuer Name,
  validity Validity,
  subject Name,
  subjectPublicKeyInfo SubjectPublicKeyInfo,
  issuerUniqueID [1] IMPLICIT UniqueIdentifier OPTIONAL,
    -- If present, version MUST be v2 or v3
  subjectUniqueID [2] IMPLICIT UniqueIdentifier OPTIONAL,
    -- If present, version MUST be v2 or v3
  extensions [3] EXPLICIT Extensions OPTIONAL
    -- If are present, version MUST be v3
}

Version ::= INTEGER { v1(0), v2(1), v3(2) }

CertificateSerialNumber ::= INTEGER

Name ::= CHOICE { RDNSequence }

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::= SET OF AttributeTypeAndValue

AttributeTypeAndValue ::= SEQUENCE {
  type AttributeType,
  value AttributeValue
}

AttributeType ::= OBJECT IDENTIFIER

AttributeValue ::= ANY DEFINED BY AttributeType

Validity ::= SEQUENCE {
  notBefore Time,
  notAfter Time
}

Time ::= CHOICE {
  utcTime UTCTime,
  generalTime GeneralizedTime
}

UniqueIdentifier ::= BIT STRING

SubjectPublicKeyInfo ::= SEQUENCE {
  algorithm AlgorithmIdentifier,
  subjectPublicKey BIT STRING
}
Extensions ::= SEQUENCE SIZE (1..MAX) OF Extension

Extension ::= SEQUENCE {
    extnId OBJECT IDENTIFIER,
    critical BOOLEAN DEFAULT FALSE,
    extnValue OCTET STRING
}

Listing A.5: ASN.1 specification for Certificate Info part

A.3 Signature

digestInfo ::= SEQUENCE {
    digestAlgorithm AlgorithmIdentifier,
    digest OCTET STRING
}

Listing A.6: ASN.1 specification for digestInfo structure
Appendix B

Source Code of the Prototype

The current appendix contains the source code of the prototype implemented. As it was explained as the document progressed, this code is not the suitable for a real card, but it was built according to the limitations of the simulator.

Since it is very large to include in the appendices, the code of the implementation of the prototype can be found in the CD attached to the report. In the following sections, which correspond to the entities of the system, it is detailed where the project and the source code of each one of these entities is placed inside the CD.

B.1 SmartCard

Project can be found at: Implementation\Projects\Prototype\SmartCard

Source code is in two different places in order to make easier to access it:

- Implementation\Projects\Prototype\SmartCard\src\security\ VerifyPolicyAndClaimApplet.java
- Implementation\Source Codes\Prototype\ VerifyPolicyAndClaimApplet.java
B.2 TrustedReader

Project: Implementation\Projects\Prototype\TrustedReader

Source code:

- Implementation\Projects\Prototype\TrustedReader\security\InitializationClass.java
- Implementation\Source Codes\Prototype\InitializationClass.java

B.3 TrustedThirdParty

Project: Implementation\Projects\Prototype\TrustedThirdParty

Source code:

- Implementation\Projects\Prototype\TrustedThirdParty\security\TerminalClass.java
- Implementation\Source Codes\Prototype\TerminalClass.java

B.4 ApplicationIssuer

Project: Implementation\Projects\Prototype\ApplicationIssuer

Source code:

- Implementation\Projects\Prototype\ApplicationIssuer\appIssuerPackage\ContractDelivery.java
- Implementation\Source Codes\Prototype\ContractDelivery.java
The current appendix contains the source code for a real implementation of the prototype which has been built. In order to try to deploy this applet in a real card, it should be necessary to check that the card supports the security services that the implementation makes use. All of them are contained in the Java Card API, therefore it is expected not to be difficult to find a suitable card. Main changes from the implementation of the prototype are the following:

- Not to export the private key. Sign the data sent by TR by means of ALG_NO_PAD RSA algorithm (not available at the simulator used)
- Use Secure random (suitable to cryptographic operations) method to generate random numbers, instead of Pseudo-Random algorithm
- Increase RSA keys’ length at least to 2048 bits
- Not to send the signature from CA and TTP certificates to TR and TTP respectively, in order to they decrypt and give it back to the card. Use ALG_NO_PAD RSA algorithm (not available at the simulator used)

The structure of the section is similar to the previous appendix (Appendix B).
C.1 SmartCard

Project can be found at: Implementation\Projects\RealImplementation\SmartCardRI

Source code is in two different places in order to make easier to access it:

- Implementation\Projects\RealImplementation\SmartCardRI\src\security\VerifyPolicyAndClaimApplet.java
- Implementation\Source Codes\RealImplementation\VerifyPolicyAndClaimApplet.java

C.2 TrustedReader

Project: Implementation\Projects\RealImplementation\TrustedReaderRI

Source code:

- Implementation\Projects\RealImplementation\TrustedReaderRI\security\InitializationClass.java
- Implementation\Source Codes\RealImplementation\InitializationClass.java

C.3 TrustedThirdParty

Project: Implementation\Projects\RealImplementation\TrustedThirdPartyRI

Source code:

- Implementation\Projects\RealImplementation\TrustedThirdPartyRI\security\TerminalClass.java
- Implementation\Source Codes\RealImplementation\TerminalClass.java
C.4 ApplicationIssuer

Project: Implementation\Projects\RealImplementation\ApplicationIssuer

Source code:

- Implementation\Projects\RealImplementation\ApplicationIssuer\appIssuerPackage\ContractDelivery.java
- Implementation\Source Codes\RealImplementation\ContractDelivery.java
Appendix D

Memory Analysis Figures

The current appendix contains the figures corresponding to the memory analysis’ screenshots. In each one of the four figures it can be observed the statics at the startup and the shutdown of the following resources:

- EEPROM
- Transaction buffer
- Stack usage
- clear-on-reset RAM
- clear-on-deselect RAM

The four stages of the system which have been considered (and they appear in that order along the appendix) have been deployment of the applet on the card, its installation, its initialization and the matching algorithm after several running.
Figure D.1: Analysis before and after of the deployment

Figure D.2: Analysis before and after of the installation
Figure D.3: Analysis before and after of the initialization

Figure D.4: Analysis before and after of running the matching algorithm
Appendix E

Securing Off-Card Contract-Policy Matching in Security-By-Contract for Multi-Application Smart Cards

Securing Off-Card Contract-Policy Matching in Security-By-Contract for Multi-Application Smart Cards

Nicola Dragoni, Eduardo Lostal, Davide Papini  
DTU Informatics  
Technical University of Denmark  
{ndra,dpap}@imm.dtu.dk  
eduardolostal@gmail.com

Javier Fabra  
Department of Computer Science and Systems Engineering  
University of Zaragoza  
jfabra@unizar.es

Abstract—The Security-by-Contract (S×C) framework has recently been proposed to support applications evolution in multi-application smart cards. The key idea is based on the notion of contract, a specification of the security behavior of an application that must be compliant with the security policy of a smart card. In this paper we address one of the key features needed to apply the S×C idea to a resource limited device such as a smart card, namely the outsourcing of the contract-policy matching to a Trusted Third Party. The design of the overall system as well as a first implemented prototype are presented.

Keywords—Multi-Application Smart Cards; Security; Contract-Matching.

I. INTRODUCTION

Java card technology has progressed at the point of allowing several Web applications to run on a smart card and to dynamically load and remove applications during the card’s active life¹. With the advent of these new Web enabled multi-application smart cards the industry potential is huge. However, concrete deployment of multi-application smart cards have remained extremely rare. One reason is the lack of solutions to an old problem: the control of interactions among applications. Indeed, the business model of the asynchronous download and update of applications by different parties requires the control of interactions among possible applications after the card has been fielded. In other words, what is missing is a quick way to deploy new applications on the smart card once it is in the field, so that applications are owned and asynchronously controlled by different stakeholders. In particular, owners of different applications (banks, airline companies, etc.) would like to make sure their applications cannot be accessed by new (bad) applications added after theirs, or that their applications will interact only with the ones of some business partners.

To date, current security models and techniques for smart cards (namely, permissions and firewall) do not support any type of applications’ evolution. Smart card developers have to prove that all the changes that are possible to apply to the card are security-free, so that their formal proof of compliance with Common Criteria is still valid and they do not need to obtain a new certificate. The result is that there are essentially no multi-application smart cards, though the technology already supports them (Java Card and Global Platform specifications).

The Security-by-Contract (S×C) framework has recently been proposed to address this challenge [1]. The approach was built upon the notion of Model Carrying Code (MCC) [2] and successfully developed for mobile code ([3], [4] to mention only a few). The overall idea is based on the notion of contract, that is a specification of the security behavior of an application that must be compliant with the security policy of the hosting platform (i.e., the smart card). This compliance can be checked at load time and in this way avoid the need for costly run-time monitoring.

The effectiveness of S×C has been discussed in [1], [5], where the authors show how the approach can be used to prevent illegal information exchange among several applications on a single smart card, and how to deal with dynamic changes in both contracts and platform policy. However, in those papers the authors assume that the key S×C phase, namely contract-policy matching, is done on the card, which is a resource limited device. What they leave open is the issue of outsourcing the contract-matching phase to a Trusted Third Party, in case this phase requires a too expensive computational effort for the card. In this paper we explicitly address this issue, discussing the design and a first prototype of this key functionality of the S×C framework.

The paper is organized as follows. In Section II we introduce the S×C framework and the problem we tackle. Then the discussion of the design and implementation details of the proposed system are depicted in Section III and IV, respectively. Section V concludes the paper summarizing its contribution.

II. SECURITY-BY-CONTRACT (S×C)... IN A NUTSHELL

In the S×C approach, mobile code carries with a claim on its security behavior (an application’s contract) that could be matched against a mobile platform’s policy before downloading the code. In this setting, a digital signature does not only certify the origin of the code but also binds together

¹http://java.sun.com/javacard/specs.html
the code with a contract with the main goal to provide a semantics for digital signatures on mobile code.

At load time, the target platform follows a workflow similar to the one depicted in Fig. 1 (see also [6]). First, it checks that the evidence is correct. Such evidence can be a trusted signature as in standard mobile applications [7]. An alternative evidence can be a proof that the code satisfies the contract (and then one can use PCC techniques to check it [8] or specific techniques for smart-cards such as [9]).

Once we have evidence that the contract is trustworthy, the platform checks that the claimed policy is compliant with the policy that our platform wants to enforce. This is a key phase called contract-policy matching in the S×C jargon. If it is, then the application can be run without further ado. At run-time, a firewall (such as the one provided by the Java Card Runtime Environment) can just check that only the declared API in the contract can be called. The matching step guarantees that the resulting interactions are correct. This is a significant saving over full in-line reference monitors.

**A. Off-Card Contract-Policy Matching**

A key issue in the S×C framework concerns who is responsible for executing the contract-policy matching. Due to the computational limitations of a resource limited environment such as a smart card (SC), running a full matching process on the card might be too expensive. In the S×C setting, the choice between “on-card” and “off-card” matching relies on the level of contract/policy abstraction [1], [5]. Indeed, the framework is based on a hierarchy of contracts/policies models for smart cards, so that each level of the hierarchy can be used to specify contracts/policies with different computational efforts and expressivity limitations.

This paper focuses on the situation where contract-policy matching is too expensive to be performed on the card. The idea, depicted in Fig. 2, is that a Trusted Third Party (TTP), for instance the card issuer, provides its computational capabilities to perform the contract-policy compliance check. The TTP could supply a proof of contract-policy compliance to be checked on the smart card. The SC’s policy is then updated according to the results received by the TTP: if the compliance check was successful, then the SC’s policy is updated with the new contract and the application can be executed. Otherwise, the application is rejected or the policy enforced off-card (for example, by means of a service provided by the TTP in addition to contract-policy matching). In case the TTP includes a proof of compliance in the reply, then a further check is needed to verify the proof, as shown in Fig. 2.

In this scenario, the communication between SC and TTP must be secured in order to deal with an untrusted environment. Both contract and policy must be encrypted and signed by SC before they are sent to the TTP to ensure authentication, integrity and confidentiality. Analogously, the results of the compliance check should be encrypted and signed by TTP before they are sent back to SC.

**III. SECURING OFF-CARD MATCHING**

To secure the system we use Public Key Infrastructure (PKI), where keys and identities are handled through certificates (namely, X.509 certificates [10]) that are exchanged between parties during communication. For this reason, the SC must engage an initialization phase, where certificates are stored in the SC along with security policies. The security of the system relies on the assumption that the environment in this phase is completely trusted and secure. As above mentioned all messages between SC-TTP will be signed and encrypted. We have decided to use two certificates (i.e. two different key-pairs), one for the signature and one for the encryption, so that in the unlikely event of one being compromised the other is not. The use of two certificates is optional, but it makes the system more secure.

In this Section we first show the design of the initialization phase and then pass over the contract-policy matching one. Since the system is based on Java card 2.2.2, the SC acts as a server which responds only to Application Protocol Data Unit (APDU) commands by
means of APDU-response messages.

**Initialization Phase.** This phase is divided into three different steps: Certificate Signing Request (CSR) building [11], certicates issuing, and finally certicates and policy storage. As shown in Fig. 3 the first step consists in building the CSR for the two certicates to be sent to the Certification Authority (CA). The Trusted Reader (TR) queries the SC for its public key, then TR builds the CSR and sends it back to SC that signs it. Message #4 $\text{SPrKSCEnc(SECencCSR)}$ means that the CSR for encryption is signed (S) with private key (PrK) of SC for encryption (Enc). Messages throughout all figures are likewise.

![Figure 3. CSRs Building](image1)

In the second step (Fig. 4) the TR - Certificates Manager (TRCM) sends to CA the CSRs previously built, CA issues the certicates and then sends them back to the TRCM.

![Figure 4. Certificates Issuing](image2)

The final step, shown in Fig. 5, completes the initialization phase by storing in the SC the two certicates, the security policy and the CA digital certicate (this is needed by the SC to verify certicates of TTP).

![Figure 5. Storage of Keys and Certificates on Smart Card](image3)

**Contract-Policy Matching Phase.** During this phase the contract and the security policy, stored in the card, are sent from SC to some TTP which runs the matching algorithm and then sends the result back to SC. Our goal is to secure communication between TTP and SC in terms of mutual authentication, integrity and conﬁdentiality. The solution we propose is shown in Fig. 6. It is divided into three parts: certificates exchange, contract and policy sending, matching result sending.

![Figure 6. Protocol for Off-Card Contract-Policy Matching](image4)

In the first part TTP and SC exchange their own pair of certicates (one for encryption and one for the signature) and then respectively check the validity of those. Particularly, the SC checks them against CA certicate stored during Initialization phase. If the certicates are valid then the TTP asks SC for the contract and policy. At this point the SC engages in a sequence of actions aiming to secure the message $M$ containing requested information that needs to
be sent back to TTP. Specifically:

1) It generates a session key and a NONCE (Number used Once) that will be used for this communication.

2) It encrypts the session key and the NONCE with TTP Public Key, and then message M with the session key.

3) It computes the HMAC (a hash mixed with a salt, i.e. the NONCE ($N_{sc}$)) and it signs it.

Then the message is sent to TTP which verifies the message and extracts the needed information.

In the last part TTP runs the matching algorithm against contract and policy, and builds a secure message containing the algorithm result R to be sent to SC. The key used for encryption is still the session key generated previously by SC. The signature is done as before except that the HMAC uses as salt the value $N_{sc} + 1$. At this point SC decrypts and verifies the result and sends an acknowledgement to TTP.

IV. PROTOTYPE IMPLEMENTATION

A first prototype of the proposed framework has been implemented, representing almost a fully-functional implementation. Java version 1.6 has been used to implement the TTP and the TR, and Java Card 2.2.2 was used for the SC. This version was used instead of Java Card 3 due to the lack of mature in version 3 (actually, there are no cards supporting its real implementation). An APDU extended length capability has been implemented in order to allow sending up to 32KB data messages instead of the by-default maximum 255 bytes size.

All message exchange protocols have been implemented and authentication, integrity and confidentiality are ensured by means of X.509 certificates in communications between the TTP and the card. These certificates are managed by means of the CA, which generates self-signed certificates using OpenSSL 0.9.8n.

The implementation of the initialization phase is almost finished. All required data is stored and sent to the installer and also sent back to the card. On the other hand, some work must be done in the contract and policy matching phase. Certificate exchange is working properly, but verification is only carried out in the TTP and not on the card yet. RSA keys are used to achieve PKI encryption, but digital signatures and block ciphering must be developed too.

To test the prototype, two different simulation environments have been used. At first stages, the Java Card platform Workstation Development Environment tool (Java Card WDE) was used. However, saving the status of the card and all the session data is currently being addressed, so the environment has been changed to the C-language Java Card RE (CREF), which eases this feature.

V. CONCLUSION

In this paper we have addressed the issue of outsourcing the S×C contract-policy matching service to a Trusted Third Party. The design of the overall system as well as a first implemented prototype have been presented. The solution provides confidentiality, integrity and mutual authentication altogether. In particular, the following mechanisms have been implemented to strengthen the security of the system:

(i) The use of two different certificates for signature and encryption. (ii) A NONCE created for each session to ensure freshness of the messages. (iii) Both the session key and the NONCE are generated within the SC, and then sent encrypted to TTP. The fact that TTP uses them to correctly compose the message R is a proof that TTP is the one that decrypted the message in the same session (due to the freshness of NONCE) and no one else did (the only way would be to get the Private Key of TTP but Public Key Cryptosystems are considered secure and unbreakable). (iv) The HMAC sent within the response is salted with $N_{sc} + 1$. The change in the value of the salt introduces variability in the hash making it more unlikely to forge.

REFERENCES


Glossary of Acronyms

**AES**  Advanced Encryption Standard

**AI**  Application Issuer

**APDU**  Application Protocol Data Unit

**API**  Application Programming Interface

**ASN.1**  Abstract Syntax Notation One

**BER**  Basic Encoding Rules

**CA**  Certification Authority

**CACert**  Certification Authority Certificate

**CAD**  Card Acceptance Device

**CAP**  Converted Applet

**CBC**  Cipher Block Chaining

**CISC**  Complex Instruction Set Computer

**CLA**  Field from an APDU message which identifies an application class of instructions
CREF  C-language Java Card Runtime Environment
CRL  Certificate Revocation List
CSR  Certificate Signing Request
DER  Distinguished Encoding Rules
DES  Data Encryption Standard
DPA  Differential Power Analysis
ECB  Electronic Codebook
EEPROM  Electrically Erasable Programmable Read-Only-Memory
EM  Encoded Message
Enc$_A$(B)  Encryption of B by means of A (in figures: EA(B))
FAT  File Allocation Table
HMAC  Hash-based Message Authentication Code
HTTP  Hypertext Transfer Protocol
ICC  Integrated Circuit Chip
ID$_A$  Identity of A
IDE  Integrated Development Kit
IEC  International Electrotechnical Commission
INS  Field from an APDU message which specifies the instruction
ISO  International Organization for Standardization
IV  Initialization Vector
JCRE  Java Card Runtime Environment
JCRE  Java Card Virtual Machine
JCWDE  Java Card platform Workstation Development Environment
JC3  Java Card 3
JDK  Java Development Kit
JRE  Java Runtime Environment
JSDK  Java Software Development Kit
KB Kilobyte
MCC Model-Carrying Code
NONCE Number used only once
OID Object Identifier
OS Operating System
OSI Open Systems Interconnection
PCC Proof-Carrying Code
PEM Privacy-enhanced Electronic Mail
PIN Personal Identification Number
PKCS Public-Key Cryptography Standard
PKI Public Key Infrastructure
PRNG Pseudo-Random Number Generator
PrKSC Smart Card Private Key
PrKSCEnc Smart Card Private Key for Encryption
PrKSCSig Smart Card Private Key for Signature
PuKSC Smart Card Public Key
PuKSCEnc Smart Card Public Key for Encryption
PuKSCSig Smart Card Public Key for Signature
RAM Random Access Memory
RISC Reduced Instruction Set Computer
ROM Read-Only-Memory
RSA Rivest, Shamir and Adleman
SC Smart Card
SCCertEnc Smart Card Certificate for Encryption
SCCertSig Smart Card Certificate for Signature
SCContractPolicy Smart Card data used to build the message containing Contract and Policy
SCCSR  Smart Card Certificate Signing Request
SCEncCSR  Smart Card Certificate Signing Request for Encryption
SCEncInfo  Smart Card Certificate for Encryption Information Part (public key, distinguished name, etc.)
SCSigCSR  Smart Card Certificate Signing Request for Signature
SCSigInfo  Smart Card Certificate for Signature Information Part (public key, distinguished name, etc.)
SD  Security Domain
SHA  Secure Hash Algorithm
Sig_A(B)  Signature of B by means of A (in figures: SA(B))
SIM  Subscriber Identity Model
SIO  Shared Interface Objects
SPA  Simple Power Analysis
SRNG  Secure Random Number Generator
STEP  Secure Trusted Environment Provisioning
SxC  Security-By-Contract
TLV  Tag Length Value
TR  Trusted Reader
TTP  Trusted Third Party
TTPCertEncr  Trusted Third Party Certificate for Encryption
TTPCertSign  Trusted Third Party Certificate for Signature
TTPContractPolicy  Data recovered by TTP from the message containing Contract and Policy and whose HMAC is checked against to HMAC in the signature of the message
WfSC  Windows for Smart Cards


[42] Cryptography Research. DPA countermeasures.


