# $(\mathsf{SC})^2$: Secure Communication over Smart Cards

## How to Secure Off-Card Matching in Security-by-Contract for Open Multi-Application Smart Cards

Nicola Dragoni[1], Eduardo Lostal[1], Davide Papini[1], and Javier Fabra[2]

[1] DTU Informatics, Technical University of Denmark, Denmark, ndra@imm.dtu.dk
[2] DIIS, University of Zaragoza, Spain, jfabra@unizar.es

**Keywords:** Security-by-Contract, Smart Cards, Secure Communication

**Abstract.** The Security-by-Contract ($\mathsf{S}\times\mathsf{C}$) framework has recently been proposed to support software evolution in open multi-application smart cards. The key idea lies in the notion of *contract*, a specification of the security behavior of an application that must be compliant with the *security policy* of the card hosting the application. In this paper we address a key issue to realize the $\mathsf{S}\times\mathsf{C}$ idea, namely the outsourcing of the contract-policy matching service to a Trusted Third Party (TTP). In particular, we present the design and implementation of $(\mathsf{SC})^2$ (Secure Communication over Smart Cards), a system securing the communication between a smart card and the TTP which provides the $\mathsf{S}\times\mathsf{C}$ matching service.

## 1 Introduction

The Security-by-Contract ($\mathsf{S}\times\mathsf{C}$) approach [8, 9] has recently been proposed as security framework for *open multi-application smart cards*. As its name suggests, a multi-application smart card is a smart card that can host several software applications, in contrast with the most widespread single application smart cards where each cards host only one application. The challenge is to make these cards "open" so that third-party applications can be dynamically loaded into and removed from the card at runtime (i.e., during the card's active life).

This openness requirement has a direct consequence in the security of such cards and this explains why concrete deployment of open multi-application cards has remained extremely rare. Indeed, although several standards for open multi-application smart cards have emerged (including Java Card [21], MULTOS [5] and GlobalPlatform (GP) [15]), openness introduces the still open problem of controlling applications' evolution, that is to control the interactions among possible applications *after* the card has been fielded.

To date, security models for smart cards (namely, permissions and firewall) do not support any type of applications' evolution. As a result, smart card developers have to prove that all changes suitable to be applied to the card are "security-invariant". More formally, they have to prove that proof of compliance with Common Criteria is still valid and a new certificate is not required.

The Security-by-Contract (S×C) approach has been proposed to address this applications' evolution challenge and, indirectly, to support the openness requirement, thus making possible to deploy new applications (owned and asynchronously controlled by different stakeholders) on a card once it is in the field. S×C builds upon the notion of Model Carrying Code (MCC) [26] and has successfully been developed for securing mobile code ([10, 7] to mention only a few papers). The overall 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 the hosting platform (i.e., the smart card). This compliance can be checked at load time, this way avoiding the need for costly run-time monitoring.

### 1.1 Contribution of this Paper

The effectiveness of S×C has been discussed in [8, 9], where the authors show how the approach could be used to prevent an 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 these papers the authors assume that the key S×C phase, namely *contract-policy matching*, is directly done on the card, which is a resource limited device. What remains open to bring the S×C idea to its full potential is the possibility of outsourcing the contract-matching phase to a Trusted Third Party. This need comes from the hierarchy of contract and policy models of the S×C framework [9], proposed to address the computational limitations of smart cards. The rationale is that each level of the hierarchy is used to specify contracts and policies at a certain degree of expressivity and, consequently, of computational cost. As a result, "light" contract and policy specifications allow the execution of the matching algorithm directly on the card, while richer specifications require an external (i.e., off-card) matching service. Thus, the communication between the card and the trusted service provider must be secured.

In this paper, we address this issue by means of the design and implementation of $(\mathsf{SC})^2$, a system specifically developed to secure the communication between a smart card and a Trusted Third Party responsible for the matching phase. In particular, the contributions of the paper can be listed as follows:

- extension of the S×C framework to deals with rich contract and policy specifications (i.e., belonging to a detailed level of the S×C hierarchy of contract/policy models)
- design of the $(\mathsf{SC})^2$ system to secure the communication between a smart card and the TTP providing the matching service
- a running Java Card based prototype implementing the proposed solution

*Outline of this Paper.* The remainder of this paper is organized as follows. Section 2 gives a concise introduction to the S×C framework and the specific problem we tackle within that framework. The design of the $(\mathsf{SC})^2$ system is sketched in Section 3, whereas the details about its implementation as well as its optimization are presented in Section 4. A security analysis and a discussion regarding another implementation of $(\mathsf{SC})^2$ follow in Sections 5 and 6, respectively. Finally, Section 7 concludes the paper.

## 2 S×C for Open Multi-Applications Smart Cards

The S×C framework [10, 7] was originally developed for securing mobile code [10], building on top of the notion of Model Carrying Code (MCC) [26]. The key idea behind the S×C approach lies in the notions of *contract* and *policy*: a mobile application carries with a claim on its security behavior (an *application's contract*) that has to be matched against a *platform's policy* before downloading the application. The main difference between these two concepts is that a contract is bounded to an application while a policy relates to a platform, such as a smart card, for instance. Thus, a contract can be seen as a security policy bounded to a piece of mobile code. This highlights one of the key features of the S×C foundational idea: by signing the code of an application, a developer binds it with the claims on its security-relevant behavior, i.e. its contract, providing a semantic to the digital signature (the digital signature does not certify only who did the application but also the behavior of the application in terms of security-relevant actions [10]).

### 2.1 S×C Framework... in a Nutshell

In its simplest form, the S×C approach follows a workflow similar to the one depicted in Fig. 1 [9]. The first step concerns the trustworthiness of the mobile application that one wants to load on the smart card. To do this one needs some evidence to be checked on the platform. Such evidence can be a trusted signature as in standard mobile applications [30] or a proof that the code satisfies the contract (by means of PCC techniques [23] or using specific techniques for smartcards [12]). Once there is evidence that the contract is trustworthy, the platform checks whether the contract is compliant with the policy that the platform wants to enforce. This is a key phase called *contract-policy matching* in the S×C jargon. If they do match, then the application can be run without further ado, because the application is compliant with both a trusted contract and the platform's security policy. On the contrary, if this match results in a failure, then we might reject the application or enforce the smart card's security policy. This can be done, for instance, through inlining techniques or monitoring the application at run-time. In both cases the application will run with some overhead.

In case an application comes without any evidence of its trustworthiness (trusted signature and/or proof of contract-code compliance), then the only S×C solution to run the application is to enforce the smart card's security policy.

### 2.2 Problem: Securing Off-Card S×C Contract-Policy Matching

A key issue in the S×C framework concerns where the contract-policy matching takes place and who is responsible for that key phase of the S×C workflow. Indeed, due to the computational limitations of a resource limited environment such as a smart card, 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 [8, 9]. As a matter of

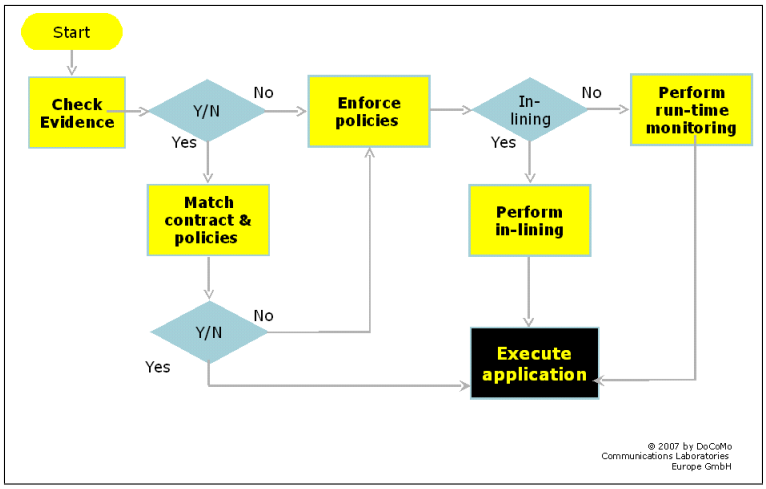



**Fig. 1.** S×C Workflow [taken from [9]]

fact, the framework is based on a hierarchy of contracts/policies models for smart cards, so that each model in the hierarchy can be used to specify contracts and policies at a certain level of expressivity. We do not recall such hierarchy here, because this would be out of the scope of the paper. What instead is important to stress is that more expressivity (that is, moving from one level to the other in the hierarchy) results in rich policies and contracts, but also in a complex matching algorithm requiring more computational efforts.

In this paper we do not consider how contract and policies are specified, that is, in which policy specification language. Without loss of generality, we assume that they are specified at a level of abstractions that require the outsourcing of the contract-policy matching phase, because too expensive to be performed on the card. This represents a key problem to support the S×C hierarchy of models, and more in general to bring the S×C approach to its full potential.

Fig. 2 depicts the main idea, where a Trusted Third Party (TTP), for instance the card issuer, provides its computational capabilities to perform the contract-policy compliance check. This TTP could supply a proof of contract-policy compliance to be checked on the smart card (SC). The SC's policy is then updated according to the results received by 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.

Since both SC and TTP act in an untrusted environment, the key challenge to develop the above scenario is to secure the communication between SC and TTP. In particular, both contract and policy must be encrypted and signed by SC before they are sent to the TTP to ensure authentication, integrity and confidentiality. Similarly, the results of the matching algorithm must be encrypted

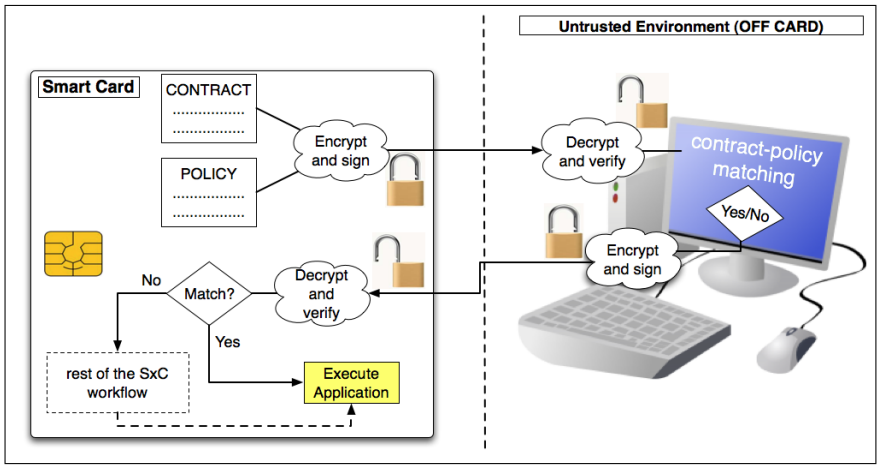



**Fig. 2.** Supporting Off-Card S×C Contract-Policy Matching

and signed by the TTP before they are sent back to the SC. In the remainder of this paper, our solution to support off-card S×C contract-policy matching by securing the communication between SC and TTP will be discussed. Before that, it is important to stress that our solution has been adapted to work on one of the most widespread technology for multi-application smart cards, namely Java Card. Java Card is based on the Application Protocol Data Unit (APDU) command-response paradigm, so well-known protocols such as Kerberos [24] or Online Certificate Status Protocol (OCSP) [22] cannot be used (or easily extended) for this purpose. This motivates the need for a new specific protocol.

## 3 $(SC)^2$: Secure Communication over Smart Cards

A *Public Key Infrastructure* (PKI) is used to secure the system, where keys and identities are handled through certificates exchanged between the communicating parties. For this reason, SC must engage an *initialization phase*, where its certificate is created and then stored in the SC along with the initial security policy and the *Certification Authority* (CA) certificate. The security of the system relies on the assumption that the environment in this phase is completely trusted and secure. If this is not true, certificates stored at this time are not trustworthy. All messages exchanged between SC and TTP will be encrypted and signed in order to accomplish the aforementioned requirements for mutual authentication, integrity and confidentiality.

In this Section we describe the design of the $(SC)^2$ system, distinguishing between an *initialization phase* and a *contract-policy matching* phase. The system is based on Java Card: the SC acts as a server which responds only to *Application Protocol Data Unit* (APDU) commands by means of APDU-response messages.

### 3.1 Initialization Phase

This phase is divided into three different steps: *Certificate Signing Request* (CSR) building [25], certificates issuing, and finally certificates and policy storage. Before this phase, the system has to be deployed on the card. SC's key pair is generated on the card during deployment, the private key never leaves the card: this is one of the security highlights of the system which makes more difficult to break the PKI.

As shown in Fig. 3, the first step consists in building the CSR for the certificate to be sent to the CA. In `message #1` the Trusted Reader (TR) queries the SC for its public key which is sent in `message #2`. TR then builds the CSR and in `message #3` sends it to SC who signs it and send it back to TR in `message #4`.

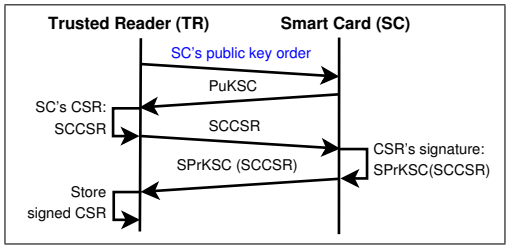


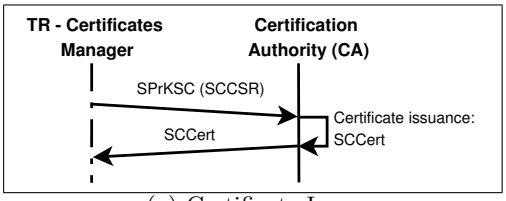

**Fig. 3.** $(SC)^2$: CSRs Building

Finally, the TR stores the signed certificate. `Message #4: SPrKSC(SCCSR)` means that the CSR from SC (`SCCSR`) is signed (`S`) with the private key (`PrK`) of `SC`.

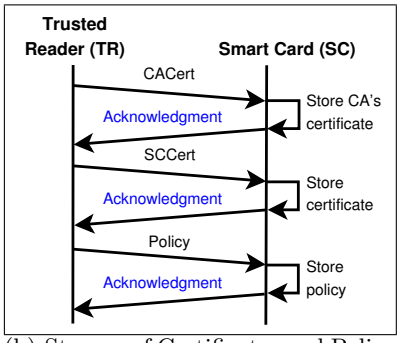
In the second step, depicted on Fig. 4(a), TR - Certificates Manager (TRCM) sends to CA the CSR. CA issues the certificate and sends it back to TRCM.

The last step, shown in Fig. 4(b), completes the *initialization phase* by storing in the SC its certificate, the initial security policy and the CA certificate, which is needed by the SC to verify certificates of TTPs.

Once the SC has been initialized, it is ready to securely engage in any activity that involves the *contract-policy matching*. Specifically, the card will be able to verify the identity of the TTP, to authenticate and to authorize its requests.




(a) Certificate Issue




(b) Storage of Certificates and Policy on Card

**Fig. 4.** $(SC)^2$: Certificate Issue and Storage of Certificates and Policy

### 3.2 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. The goal is to make the communication between SC and TTP secure. The proposed solution is shown in Fig. 5. It is divided into three parts: *certificates exchange*, *contract and policy sending* and *matching result sending*.

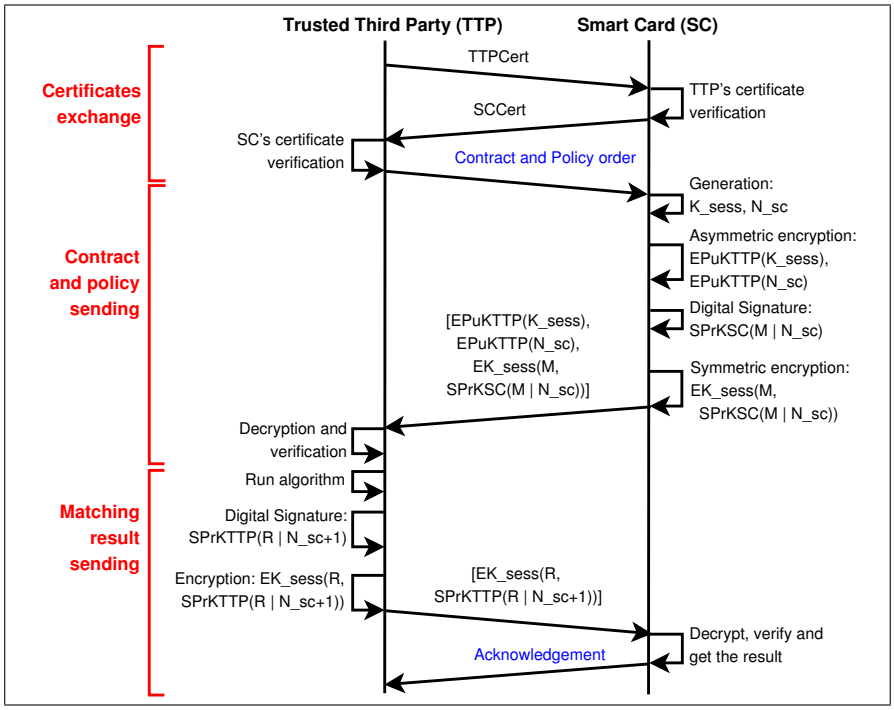



**Fig. 5.** Off-Card Contract-Policy Matching

In the first part, both parties TTP and SC exchange their own certificates and then respectively check their validity. In particular, SC checks the certificate received against CA certificate which was stored during the *initialization phase*. If some certificate is not valid, the communication terminates. Otherwise 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 the requested information.

Firstly, a *session key* and a *NONCE* (Number used Once) that will be used for this communication are randomly generated. *NONCE* ($N_{sc}$) is used to avoid replay attacks. The encryption of the message M is done by means of symmetric cryptography, mainly because it provides higher speed than the asymmetric one (based on PKI). Security is also improved by the lack of linearity since the *session*

*key* changes for each session. After that, SC encrypts the *session key* and $N_{sc}$ with TTP's public key ensuring that only TTP will be able to get them. Once that is done, the message M concatenated with the $N_{sc}$ is hashed and then signed. This way, the system provides to the signature with freshness. As $N_{sc}$ changes, signature does as well for the same contract and policy. Finally, confidentiality is ensured through the encryption of message M and signature by means of the *session key*. The message to be sent to TTP contains the *session key* and the $N_{sc}$ encrypted by TTP's public key, and the message and signature encrypted by the *session key*, this is, `Message #4:` $\mathtt{E}_{PuKTTP}(K_{sess})$, $\mathtt{E}_{PuKTTP}(N_{sc})$, $\mathtt{E}_{K_{sess}}(M, \mathtt{S}_{PrKSC}(M|N_{sc}))$. The message is then sent to TTP, which verifies it and extracts the needed information.

In the last step, the TTP runs the matching algorithm using the information received in the previous message. When the algorithm finishes, it builds a secure message containing the result R to be sent to SC. The session key, which has been previously generated on SC and sent encrypted to TTP, is used again to encrypt the result R along with the signature, which is done over the result R concatenated with $N_{sc}+1$. The change in the value of $N_{sc}$ introduces variability in the hash, making it more unlikely to forge. Thus, the message to be sent is `Message #5:` $\mathtt{E}_{K_{sess}}(R, \mathtt{S}_{PrKTTP}(R|N_{sc}+1))$.

Finally, SC decrypts and verifies the message getting the result of the matching algorithm. Due to the fact that every APDU communication is made up of a command and a response, the protocol finishes with the sending of an acknowledgement to the TTP.

## 4 (SC)² Implementation

The implementation of the (SC)² system is depicted in this Section. Due to the constrained nature of smart cards, we will particularly focus on (SC)² optimization in terms of memory usage and also performance.

### 4.1 Implementation

Different programming languages have been used according to whether the entity was an off-card one or not. Java version 1.6 has been used to implement the TTP and the TR, while SC has been implemented with Java Card 2.2.2. On one hand, Java was chosen because of its multi-platform feature since TTP should be run over different devices. On the other hand, Java Card 2.2.2 has been chosen due to the lack of maturity of Java Card 3 (actually, there are no cards supporting its implementation). It is worth mentioning that an APDU extended length capability has been implemented in order to send up to 32 KB data messages instead of the by-default maximum 255 bytes size. Moreover, the garbage collection is done on-demand. Concerning the execution over a SC simulator, two different environments have been used: at the earliest stages we adopted Java Card Platform Workstation Development Environment (Java Card WDE) tool and then we moved to C-language Java Card Runtime Environment (CREF) as soon as development needed to save the status of the card.

**Certificates** Authentication is ensured by means of X.509 certificates [17]. Certificates used in $(\mathsf{SC})^2$ are generated by means of *OpenSSL 0.9.8n*. In order to test the system, certificates for CA and TTP have to be created. CA root certificate is generated as a self-signed certificate.

During the initialization phase, the CSR's signature is done through SHA as hash algorithm whose digest is encrypted using RSA with a padding according to the PKCS#1 (v1.5) scheme [19]. Once the CSR is ready, it is sent to the CA. OpenSSL verifies it and, by means of the CA root certificate, issues a new certificate corresponding to the CSR received. The certificates are stored on-card as byte arrays and DER-encoded [16].

**Parser** An on-card parser has been developed to verify the validity of certificates received by the SC. Specifically, what is checked is the compliance with DER and ASN.1 encoding, the signature, the key algorithm and length and the corresponding issuer. The offsets of a couple of parts of CA's certificate are stored on SC in order to ease its access during TTP certificate's verification [14]. In contrast, SC only needs to temporary store the offset of the TBSCertificate part [17] from TTP's certificate. TTP's certificates are stored temporary because each certificate is analyzed only once when it is parsed. Certificates are parsed from the beginning following the DER encoding which guides the parser through their TLV-structure (Tag-Length-Value). The encoding is checked and every part is reached, extracting it if needed (i.e., public key). At the end, the signature is verified against the CA's public key.

**Cryptography** Both the aforementioned simulation environments suffer from a problem that sets limitations on the prototype: not every algorithm from the API is implemented for these environments [29]. Namely, the main problems are related to the length in RSA keys and the secure random algorithm. In the following, they will be detailed more thoroughly.

Concerning asymmetric cryptography, RSA is used with a padding according to the PKCS#1 (v1.5) schema. The length of RSA keys used in the prototype is 512 bits because it was the only one provided by the environment as it has been previously pointed out. However, without this limitation a 2048-bit key size would be used. The other limitation is related to the random number generator. In the prototype a Pseudo-Random Generator has been used. Nevertheless, it is recommended to use a Secure Random Number Generator to avoid the predictability of the linear congruential algorithms [28].

The chosen symmetric block cipher is AES with 128 bits key length and block size in *Cipher Block Chaining* (CBC) mode. This mode makes necessary the use of an *Initialization Vector* (IV), known by both sides of the communication in order to properly finish the encryption/decryption process. $N_{sc}$ fits perfectly, since it is random and fresh in every new session. Therefore, IV takes the first 128 bits from $N_{sc}$.

### 4.2 Optimization

Smart cards are limited in terms of resources, mainly in memory aspects. Some issues must be considered by developers and also by card suppliers, especially that the card applet will not provoke a memory overflow due to dynamic class instantiations and also that memory limits are not reached. The smart card industry has provided developers with some specific programming guidelines and methodologies in order to keep memory usage under control. Optimization of Java Card applications normally concerns adapting and formalizing traditional methods of optimization to Java Card programs as well as developing new techniques, with the main aim of minimizing execution time and memory consumption.

There are some approaches which have faced the problem of memory overflow at Java Card. In [13], for example, a constraint-based algorithm able to determine potential loops and mutually recursive methods is proposed. This algorithm processes the applet bytecodes by means of a set of processing rules, designed in such a way that a certified analyzer could be extracted from their proof of correctness. A similar approach was previously depicted in [3], where a constraint-based algorithm was built and then formally extracted from the proof of correctness of the algorithm in the proof assistant Coq [1]. However, the approach presented in [13] improved the one presented in [3] with respect to memory usage and also with respect to its scope (the first one also covered subroutines and exceptions, which were not addressed in the second one).

Both the problem of instantiating classes inside loops and the incorrect usage of recursive functions in Java-based smart cards are still open challenges. An interesting approach is [20], where the author describes an on-card bytecode verifier, but it does not address properties related to memory usage. Previous work [3] presented a certified analyzer to determine loops and mutually recursive methods but its memory footprint prevents it from being deployed on-card.

Other works have faced the problem of Java Card optimization from an analytical point of view. In [6], authors propose to optimize Java Card applications by adding several new instructions to the Java Card Virtual Machine (JCVM). These instructions allow to transmit a result inside the bytecodes, thus improving the resolution of the virtual machine, reducing the quantity of the code and shortening the runtime overall performance. Other approaches propose to optimize only the bytecodes generated by a subset of operations of the virtual machine, such as in [18, 11]. However, the application of these solutions to a common development cycle is quite complex, since it requires modifying the JCVM. Finally, a very interesting review to two basic techniques to optimize the Java Card bytecode set is provided in the IBM WOK [3]. These techniques can be used in the conversion step executed before downloading applets to the card, namely i) instruction set optimization, and ii) overall data layout optimization.

In this work, some of the main guidelines introduced in [4] have been followed. Mainly, neither persistent nor transient objects have been created unless strictly required. Also, nesting method invocations has been avoided, since this practice

[3] http://www.zurich.ibm.com/pdf/javacard.pdf

usually leads to stack overflows. The code has been reviewed to ensure that objects are instantiated only once, especially when creating new instances inside loops and functions. Other well known best practices have been followed: i) merging methods to get a lighter code, avoiding the duplication of code and the addition of new signatures (which requires extra bytes); ii) a survey of useless variables has been carried out and then removed. Also, the use of declared final static variables has been studied in order to find the useless ones; iii) certain variables have been moved to the method where are used instead of being an attribute; iv) native functions for the management of arrays have been used to improve the execution time and also the memory usage of the resulting code, instead of generating a different, more sophisticated code which does not get benefit from the native execution; v) finally, complex code constructions have been avoided, replacing them by more simple ones.

All these code programming optimizations have allowed reducing the applet memory usage before it is deployed in the card up to a 32% in code size with respect to the first implementation of the prototype. Also, it is still possible to apply some of the optimization techniques presented previously, although this last step is out of the scope of this work.

### 4.3 Memory Usage Analysis

The memory usage of the resulting implementation has been evaluated. The importance of this analysis lies in showing that the theoretical idea is suitable to be implemented and fits in the constrained and limited smart card resources.

Several measures have been taken through the output of the CREF commands. Although the execution of the CREF commands gives us the chance to retrieve statistics and information related to the EEPROM, the transaction buffer, the stack usage, the clear-on-reset RAM, the clear-on-deselect RAM and also the ROM, only the EEPROM data are shown and analyzed. This is due to the fact that, on the 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 lacks of 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 the fact that if an applet needs for more memory to be executed than RAM provided, this would end in an error because RAM memory resources are exhausted. However, albeit this is a problem which every developer has to keep in mind when working on smart cards, in this work it is not representative since the RAM amount remains the same and the developer should know its working size. Also, RAM is cleared at every shutdown and might be cleaned over demand; hence, it changes every card-tearing. In the case of EEPROM, this memory 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 if it

| **Stage** | **Consumed before** | **Consumed after** | **Available before** | **Available after** |
|---|---|---|---|---|
| Deployment | 6994 | 11476 | 58510 | 54028 |
| Installation | 11476 | 12572 | 54028 | 52932 |
| Initialization | 12572 | 15279 | 52932 | 50225 |
| Running | 15118 | 14572 | 50386 | 50932 |

**Table 1.** Memory Usage in Bytes

takes too many memory resources otherwise, thus excessively reducing the space on-card and making the multi-application framework non suitable.

Table 1 depicts the memory usage in bytes for the prototype implementation. The CREF simulator provides a 64KB EEPROM memory (this is, 65536 bytes). The common size for Java Card 2.2.2 real implementations ranges between 32KB (old and constrained) and 128KB, although it is starting to use greater values. The stages shown in the left column represent the applet lifecycle, which causes the main changes over the EEPROM. First of all, the deployment stage consists of downloading the applet to the card and then storing the bytecode there. The installation stage is done by means of the static install method, which installs the applet on the card invoking a registration method. The initialization stage corresponds to the *initialization phase* detailed in Secton 3. Finally, the running stage is the contract-policy matching phase.

Rows in Table 1 depict the memory usage before and after the corresponding stage was executed. As shown, the card requires almost 7 KB which are normally reserved for some OS initialization. Downloading the applet to the card takes almost 5 KB, whilst its installation more than 1 KB. It is worth mentioning that the initialization stage is the most memory consuming, since all instances are created and most of the space is reserved in the card at that moment (keys and algorithms, for instance). However, the optimizations carried out in the code have allowed to decreases the available memory in less than 3 KB at initialization. After this stage has been performed, both SC and CA certificates as well as 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 and installed applications. Finally, the EEPROM consumption of the matching algorithm only makes memory vary a few hundred bytes. To sum up, the developed system needs a rough memory space in the EEPROM of 7.5 KB.

Let us now focus on several points related to the obtained values. If the stats are looked through, downloading the applet is the most consuming stage because of the extensive source code. That is because the applet has to deal with several cryptographic problems, even including an on-card parser. Usually, common applets are not as large, what means that is still possible to store a high number of them. On the other hand, the heaviest issue is bytecode downloading, as it was expected. It must be kept in mind than the smart card and also the rest of current hardware is continuously evolving, so available memory will be greater

in a short time whilst the necessary space for the system developed will remain. As shown, the application takes more space than a usual applet because of its higher complexity, but it does not reduce the available memory considerably, thus allowing to store a large number of applications in a secure way.

## 5 Security Analysis

In the S×C framework the goal is to secure a smart card against installation of malicious software, therefore our focus is to guarantee the security of data and applications stored on the card. Ultimately this means that origin of the result of *contract-policy matching* (i.e. the TTP) must be authenticated. To make it simpler, if an attacker would manage to forge the last message of *off-card contract-policy matching* phase, the security of the system would be invalidated. In order to prove the system secure we have to analyze it from two different perspectives: the designed protocols and the cryptographic primitives involved. The latter is usually assumed secure as long as proper key-sizes and crypto-algorithms are used. On the other hand the first is the most critical: the simplest flaw in the protocol design can render a secure algorithm with the longest key practically useless. This Section is divided into two parts: the first one focuses on protocol security and the second on cryptographic algorithms and key-sizes.

### 5.1 Protocol Analysis

In Section 3 we have described the protocols involved in the system, namely *initialization phase* (from hereon called `protocol #1`, see Fig. 3, 4(a) and 4(b)) and *Off-card Contract-Policy Matching* (`protocol #2`, see Fig. 2). `Protocol #1` is performed in a secure environment, this is because in this phase certificates and keys are generated, subsequently SC does not have means to establish any secure communication. Hence `protocol #1` is not secure, anyhow this does not invalidate the security of the system.

The key of the security of the system is then `protocol #2`. To analyze it we used the *LySa tool* [2]. The LySa tool is an automated tool for verifying security properties of protocols that use cryptography to protect network communication from tampering by malicious parties. Protocols modeled in the process calculus LySa are input to the tool. The LySa tool makes a fully automated program analysis that can guarantee confidentiality and authentication properties. Freshness and integrity are implicitly guaranteed whenever the Nonce is either confidential or can not be predicted and whenever the probability of two different messages colliding into the same hash is negligible.

The protocol has been checked against single session (i.e., one initiator and one responder at a time). This is sufficient to prove that the system is secure, since Java Card technology does not allow multiple sessions (one APDU-response for one APDU-command at a time). The analysis shows that the values known to the attacker are: *contract and policy order* message, certificates of parties involved, and all the encrypted messages (see Fig. 2). These messages does not

reveal any confidential information as long as cryptography is not broken. The protocol is therefore secure.

### 5.2 Cryptographic Algorithms and Key-Sizes

In security, standard cryptographic algorithms are considered secure as long as the key is large enough to guarantee that the system cannot be broken for the time data or information needs to be confidential. [27] extensively discuss recommended algorithms and related weaknesses, if any, along with recommended key lengths both for symmetric and asymmetric cryptography. In our implementation we chose AES-128 for symmetric and RSA-512 for the asymmetric (both values are limitations imposed by the simulator). According to [27], to break a 128 bit symmetric key you need $10^{16}$ years while for RSA-512 (whose security corresponds to 50 bit symmetric key [27]) time to break is significantly smaller, from 10 min to 1 hour. This is unacceptable since RSA is used for certificates, which are supposed to be valid for years. Considering that this size was imposed by the simulator (Section 4), for a real implementation we recommend to use RSA-2048 (whose security corresponds to 103 bit symmetric key, time to break $> 10^8$ years). Finally, to generate the *NONCE*, a Secure Random Number Generator has to be used instead of a Pseudo-Random one because the latter produces predictable numbers [28].

## 6 Discussion

Despite the previous design (and relative prototype), another version has been developed. In the latter, every entity uses two key pairs, and consequently two certificates, one for encryption and one for signature. This way, the signature (performed with the *private key for signature* of the sender) is encrypted by means of the *public key for encryption* of the receiver. Therefore, confidentiality in the signature is achieved thanks to the *public key for encryption* instead of the session key. The diagram of this design is depicted in Fig. 6. Since a new certificate has to be managed, some changes have been added: it is necessary to create two CSRs and store two certificates during the initialization, and also to exchange another one during the matching phase.

This approach is more secure, since it increases the number of keys an attacker needs to break in order to fully attack the system (i.e., the attacker has to find not only the session key but also to break RSA for both the certificates).

On the other hand, the space needed by the prototype using two RSA key pairs is more than 1.5 KB bigger than the one needed by the proposed prototype (taking in account the prototype built uses 512-bit RSA keys, this amount is expected to increase with a 2048-bit key). Thus, since smart cards are constrained resource devices and the security level provided by symmetric encryption is more than sufficient (Section 5). Hence the approach requiring less space was chosen as more suitable.

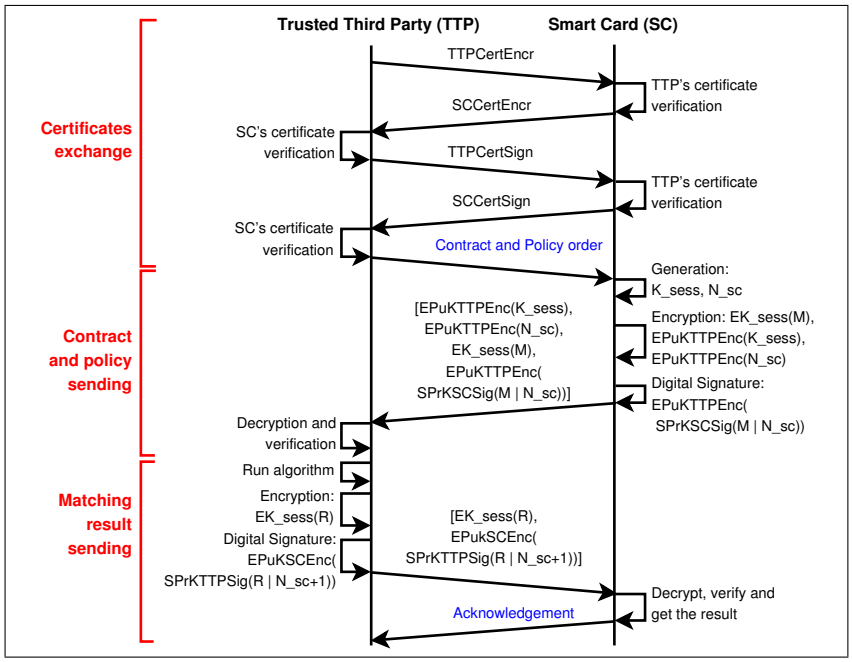



**Fig. 6.** Contract-Policy Matching Phase with Two Certificates

## 7 Conclusion

In this paper we have addressed a key open issue in the S×C framework for open multi-application smart cards, namely supporting the outsourcing of the contract-policy matching service to a TTP by securing the communication between the card and the TTP. The design of the $(\mathsf{SC})^2$ system as well as its optimized implementation have been presented. The solution provides confidentiality, integrity and mutual authentication altogether. Since smart cards are resource constrained devices, a memory analysis has also been presented to demonstrate the suitability of the framework for these devices. Finally, a security analysis and an alternative version of $(\mathsf{SC})^2$ have been discussed.

## References

1. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
2. C. Bodeia, M. Buchholtzb, P. Deganoa, F. Nielsonb, and H.R. Nielsonb. Static validation of security protocols. *Computer Security*, 13(3):347–390, 2005.
3. D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In *Formal Methods (FM'05)*, volume 3582 of *LNCS*, pages 91–106, Newcastle Upon Tyne, UK, July 2005. Springer-Verlag.

4. Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, Boston, MA, USA, 2000.
5. Multos Consortium. Multos developer's reference manual. mao-doc-tec-006 v1.45. Specification 1.45, 2009.
6. Z. Dawei and D. Wenrui. *Optimization of resolution on Java card*. Journal of Beijing University of Aeronautics and Astronautics. 2009.
7. L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-Contract on the .NET platform. *Information Security Tech. Rep.*, 13(1):25 – 32, 2008.
8. N. Dragoni, O. Gadyatskaya, and F. Massacci. Supporting applications' evolution in multi-application smart cards by security-by-contract [short paper]. In *Proc. of WISTP'10*, volume 6033 of *LNCS*, pages 221–228, 2010.
9. N. Dragoni, O. Gadyatskaya, and F. Massacci. Security-by-contract for applications evolution in multi-application smart cards. In *Proc. of NODES*, DTU Technical Report, 2010.
10. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Proc. of EUROPKI*, pages 297–312. Springer-Verlag, 2007.
11. L. Fournerie. An in-card bytecode optimization for objects management in java-based smart cards. In *Proc. of the 4th Gemplus Developer Conference*, 2002.
12. D. Ghindici and I. Simplot-Ryl. On practical information flow policies for java-enabled multiapplication smart cards. In *Proc. of CARDIS*, 2008.
13. P. Giambiagi and G. Schneider. Memory consumption analysis of java smart cards. In *Proc. of CLEI'05*, Cali, Colombia, October 2005.
14. O. Henninger, K. Lafou, D. Scheuermann, and B. Struif. Verifying X.509 Certificates on Smart Cards. In *World Academy of Science, Engineering and Technology*, volume 22, pages 25–28, Aug 2006.
15. GlobalPlatform Inc. GlobalPlatform Card Specification, Version 2.2. Specification 2.2, GlobalPlatform Inc., 2006.
16. ITU-T. ITU-T Rec. X.690 Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). Technical report, ITU, 2002.
17. ITU-T. ITU-T Rec. X.509 Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks. Technical report, ITU, 2005.
18. D.-W. Kim and M.-S. Jung. A study on the optimization of class file for java card platform. In *Information Networking: Wired Communications and Management*, volume 2343 of *LNCS*, pages 563–570. 2002.
19. RSA Laboratories. *PKCS #1 v2.1: RSA Cryptography Standard*. RSA Security Inc. Public-Key Cryptography Standards (PKCS), Jun 2002.
20. X. Leroy. Bytecode verification on java smart cards. *Softw. Pract. Exper.*, 32:319–340, April 2002.
21. Sun Microsystems. Runtime environment specification. Java Card$^{TM}$ platform, version 3.0, connected edition. Specification 3.0, Sun Microsystems, 2008.
22. M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 2560, June 1999.
23. G.C. Necula. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang.*, pages 106–119. ACM Press, 1997.
24. B.C. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *Communications Magazine, IEEE*, 32(9):33 –38, 1994.

25. M. Nystrom and B. Kaliski. PKCS #10: Certification Request Syntax Specification version 1.7. Technical report, RSA Security, Nov 2000. Internet RFC 2986.
26. R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D.C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proc. of SOSP-03*, pages 15–28. ACM, 2003.
27. N. Smart. ECRYPT II Yearly Report on Algorithms and Keysizes (2009-2010), March 2010.
28. W. Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education, 2002.
29. Sun Microsystems, Inc. *Development Kit User's Guide, Java Card Platform, Version 2.2.2*, March 2006.
30. B.S. Yee. A sanctuary for mobile agents. In J. Vitek and C.D. Jensen, editors, *Secure Internet Programming*, pages 261–273. Springer-Verlag, 1999.