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 life[1]. 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).

[1]http://java.sun.com/javacard/specs.html

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

Copyright (c) IARIA, 2010
ISBN: 978-1-61208-100-7

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]).

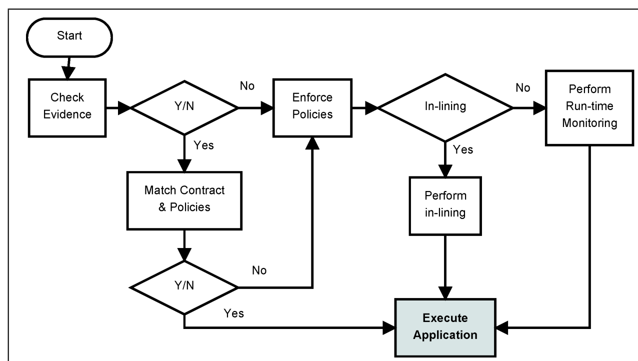

Figure 1. S×C Workflow

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.

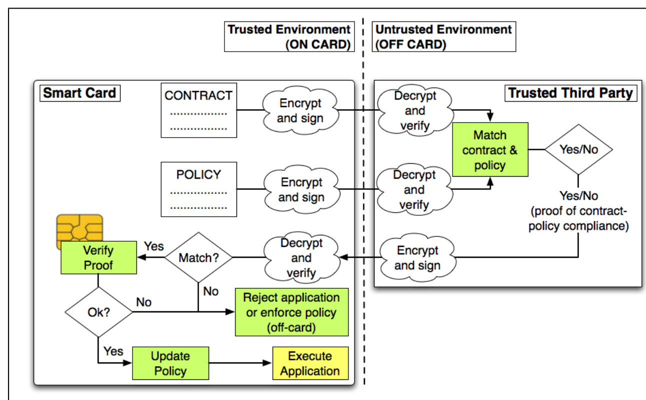

Figure 2. Off-Card S×C Contract-Policy Matching

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

Copyright (c) IARIA, 2010 ISBN: 978-1-61208-100-7

means of APDU-response messages.

Initialization Phase. This phase is divided into three different steps: *Certificate Signing Request* (CSR) building [11], certificates issuing, and finally certificates and policy storage. As shown in Fig. 3 the first step consists in building the CSR for the two certificates 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 `SPrKSCEnc(SCEncCSR)` means that the CSR for encryption is signed (`S`) with private key (`PrK`) of `SC` for encryption (`Enc`). Messages throughout all figures are likewise.

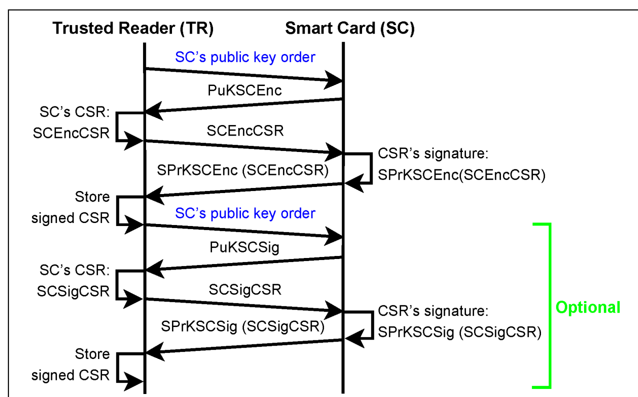

Figure 3. CSRs Building

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

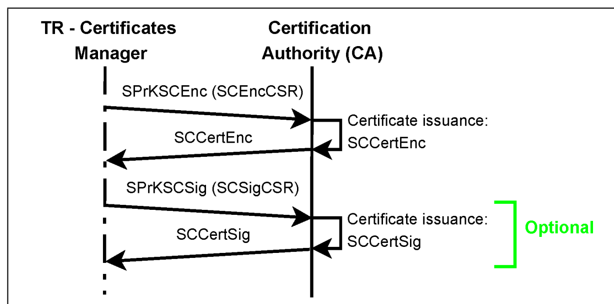

Figure 4. Certificates Issuing

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

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

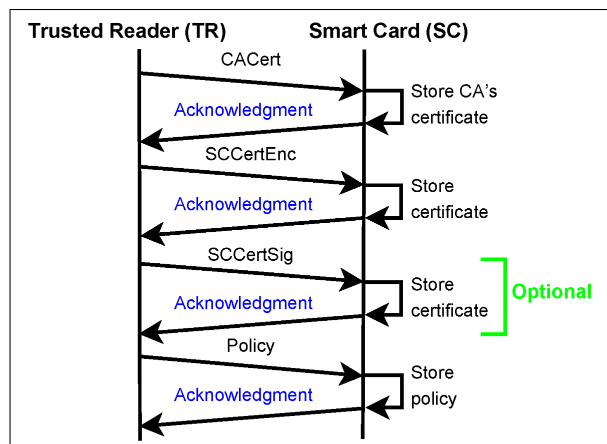

Figure 5. Storage of Keys and Certificates on Smart Card

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*.

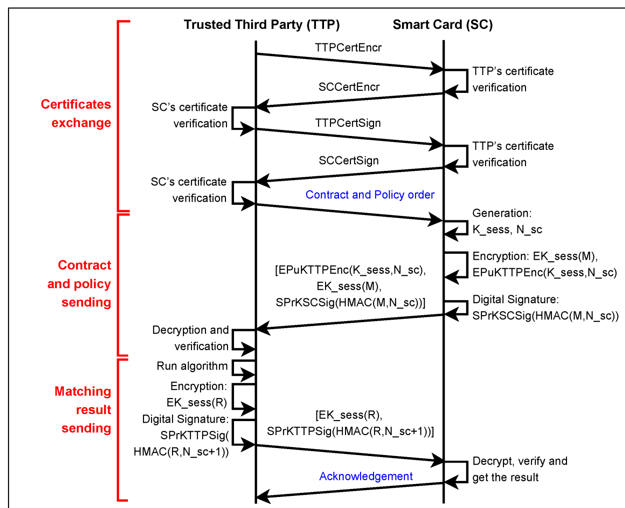

Figure 6. Protocol for Off-Card Contract-Policy Matching

In the first part TTP and SC exchange their own pair of certificates (one for encryption and one for the signature) and then respectively check the validity of those. Particularly, the SC checks them against CA certificate stored during *Initialization phase*. If the certificates 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

Copyright (c) IARIA, 2010 ISBN: 978-1-61208-100-7

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

[1] N. Dragoni, O. Gadyatskaya, and F. Massacci, “Supporting applications’ evolution in multi-application smart cards by security-by-contract,” in *Proc. of WISTP*, 2010, pp. 221–228.

[2] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney, “Model-carrying code: a practical approach for safe execution of untrusted applications,” in *Proc. of SOSP-03*. ACM, 2003, pp. 15–28.

[3] 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*. Springer-Verlag, 2007, pp. 297–312.

[4] 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.*, vol. 13, no. 1, pp. 25 – 32, 2008.

[5] 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.

[6] D. Vanoverberghe, P. Philippaerts, L. Desmet, W. Joosen, F. Piessens, K. Naliuka, and F. Massacci, “A flexible security architecture to support third-party applications on mobile devices,” in *Proc. of ACM Comp. Sec. Arch. Workshop*, 2007.

[7] B. Yee, “A sanctuary for mobile agents,” in *Secure Internet Programming*, J. Vitek and C. Jensen, Eds. Springer-Verlag, 1999, pp. 261–273.

[8] G. Necula, “Proof-carrying code,” in *Proc. of the 24th ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang.* ACM Press, 1997, pp. 106–119.

[9] D. Ghindici and I. Simplot-Ryl, “On practical information flow policies for java-enabled multiapplication smart cards,” in *Proc. of CARDIS*, 2008.

[10] ITU-T, “ITU-T Rec. X.509,” 2005.

[11] M. Nystrom and B. S. Kaliski, “PKCS #10: Certification Request Syntax Specification version 1.7,” RFC 2986, 2000.

Copyright (c) IARIA, 2010 ISBN: 978-1-61208-100-7