Java Card Architecture for Autonomous yet Secure Evolution of Smart Cards Applications *

Olga Gadyatskaya, Fabio Massacci, Federica Paci, and Sergey Stankevich

DISI, University of Trento, Italy
{surname} @disi.unitn.it

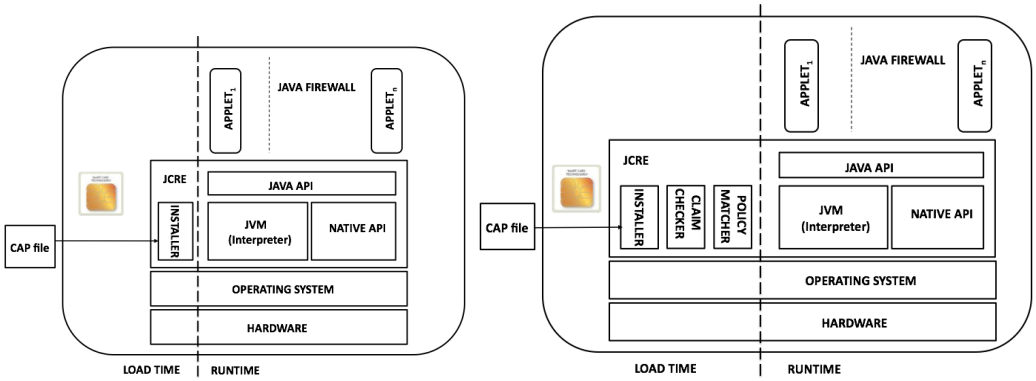
Abstract. Open multi-application smart cards that allow post-issuance evolution (i.e. loading of new applets) are very attractive for both smart card developers and card users. Since these applications contain sensitive data and can exchange information, a major concern is the assurance that these applications will not exchange data unless permitted by their respective policies. We suggest an approach for load time application certification on the card, that will enable the card to make autonomous decisions on application and policy updates while ensuring the compliance of every change of the platform with the security policy of each application's owner.

1 Introduction

Open multi-application environments such as PCs and mobile phones are widespread. The main characteristics of such environments are co-existence of several applications on one single platform and the possibility of these platforms to evolve by adding new applications or updating of existing ones in a fully distributed and autonomous way. Such applications can exchange data locally or remotely or access local APIs and in order to protect our security we regulate their accesses by more or less stringent mechanisms of access control locally enforced by the platform. The problem of this approach is that we may end up downloading something that turns out to be unusable. An alternative solution could be to shift the security checks at loading time. This approach requires applications to come with a manifest of their security-relevant actions and check whether their behavior is acceptable before installing the code. The idea was explored by Sekar et al. when the notion of *model-carrying code* was introduced [13], has been demonstrated in the *Security-by-Contract* (S×C) approach [3], and was adopted by W. Enck et al. for Android security [4].

Modern smart cards could be another example of an open multi-application environment. But even examples of potentially multi-stakeholders applications are de facto single one: a loyalty Miles& More Lufthansa credit card [9], could include a Lufthansa application for collecting miles and a Bank application; in fact it is just a credit card and mileage is calculated by the back-end system.

* Work partially supported by the EU under grant EU-FP7-FET-IP-SecureChange. We thank B. Chetali, Q. Nguyen, and I. Symplot-Ryl for useful discussions.



(a) Smart Card Architecture (b) Security by Contract Architecture

Fig. 1. Architecture Evolution for Self-Certification

In order to support autonomous evolution on multi-application smart cards, we need a way for the platform to verify that applications arriving on the platform comply with the policy which is dynamically created by combining the policies of already installed applications. The task can be more complicated when applications can also be removed and we want to avoid loss of functionalities.

In this paper we propose an addition to the Java Card security architecture based on the Security-by-Contract approach that preserves security of the smart card when the content of the card changes, so that the smart card itself can ensure security after an update, when new applications arrive, old ones are updated or removed, or their security policies are changed.

2 Security-by-Contract Smart Card Architecture

Java Card architecture [10] consists of several layers as illustrated in Fig. 1(a):

- a device hardware,
- an embedded operating system (OS),
- a Java Card Runtime Environment (JCRE) on top of the embedded OS,
- the applications that are installed on the smart card, that are called *applets*.

Applets before being loaded on a smart card are converted into a CAP file, that is a binary executable representation of the Java classes that compose the applet. The JCRE is responsible for managing and executing applets. It is composed by a Java Virtual Machine (JVM), native API, framework APIs, and an application installer. The installer downloads and installs the applications on the card. To load an application, the installer interacts with an off-card installation program which transmits the CAP file. Once received the CAP file, the installer, first, checks the signature of the file to ensure integrity and to prove the identity of the application provider. Then, the installer saves the content of

the file into the card's persistent memory, resolves the links with other applets already present on the card, creates an instance of the applet and registers it to the JCRE. Then JVM, which consists of a bytecode interpreter, executes the code contained in the CAP file.

The applets installed on the smart card are isolated by the Java firewall. The firewall allows only applets that belong to the same context to access the respective methods. If an applet (*server*) wants to share data with another applet (*client*) from a different context, it has to implement a *shareable interface* which defines a set of methods that are available to other applets. These methods (they are also called *services*) are the only methods of the server applet that are accessible through the firewall. The JCRE will pass the call to the shareable interface method from the client applet to the server applet. In the current Java Card security model the server has an access control list of the applets that are allowed to use this method, embedded into the server code. If the client is in the list, the access is granted, otherwise the client gets `null`. If the client has been updated, the server will still grant it an access, though now the server cannot really be sure of the trustworthiness of the client.

We have identified the requirements for an extension of the security mechanisms on Java Card in the presence of evolution from the requirements of the GlobalPlatform (GP) specification [8]. GP is a middleware, that can run on top of Java Card and provide more security mechanisms for applets management (for inter-application communications GP relies on the JCRE). GP specification provides explicit requirements for maintaining security (in terms of services access control enforcement) and functionality of the applications on the card during evolution.

The basic idea of our proposal is to add a contractual component to each applet detailing its security policy and its claims on the usage of other resources (services) on the platform (the latter can be also extracted from the CAP file). The architecture we propose is provided on Fig. 1(b). It is based on the addition of two components to the JCRE, the ClaimChecker and the PolicyChecker.

When a new applet has to be loaded on the card, the terminal sends the CAP file of the applet to the installer. The CAP file contains the binary code of the applet and its Contract. When the installer receives the CAP file of the new applet the ClaimChecker verifies that the claims are compliant with the applet's code. If this is the case, the PolicyChecker checks the applet's contract against the platform policy $\mathcal{P}$. If the PolicyChecker has returned True then the installer finalizes the loading and creates an instance of the applet. Otherwise, the applet is rejected. On the S×C-enhanced platform, when a method of the applet is called by another applet, the Java Card firewall simply checks that the method belongs to the shareable interface of the applet (this check is also a firewall's responsibility on the standard Java Card) and does not perform the run-time checks of the applet's privileges for an access to the services, since it was done at the loading time.

The security model behind the concepts in the architecture assumes that the card can be represented as a tuple $\langle \Delta_{\mathcal{A}}, \Delta_{\mathcal{S}}, \mathcal{A}, \mathsf{shareable}(), \mathsf{invoke}(), \mathsf{sec.rules}(),$

$\mathsf{func.rules}()\rangle$, where $\Delta_{\mathcal{A}}$ is a domain of applications; $\Delta_{\mathcal{S}}$ is a domain of services; $\mathcal{A} \subseteq \Delta_{\mathcal{A}}$ is a set of applications installed (deployed) on the platform. The function $\mathsf{shareable_A}$ defines the actual shareable interfaces of applet A available on the platform and the function $\mathsf{invoke_A}$ the set of services called by A.

The functions $\mathsf{sec.rules}()$ and $\mathsf{func.rules}$, define respectively the security policy and the functionality policy of each application. For every applet $A \in \mathcal{A}$ $\mathsf{sec.rules_A(s)}$ defines for each service of applet A which other applets on the platform are authorized to call it. The functional rules $\mathsf{func.rules_A}$ specify the set of services on the platform that A needs in order to be functional (e.g. a transport applet normally needs a payment applet in order to be useful).

The contract $\mathsf{Contract_A}$ of an application A includes the following sets: $\mathsf{Provides_A}$ (a set of services that applet A has), $\mathsf{Calls_A}$ (a set of services of other applets that A calls), $\mathsf{sec.rules_A}$ (authorizations for A's services access) and $\mathsf{func.rules_A}$ (set of functionally necessary services for A).

The ClaimChecker for an application B with a contract $\mathsf{Contract_B}$ will return true if $\mathsf{shareable_B}$=$\mathsf{Provides_B}$ and $\mathsf{invoke_B}$=$\mathsf{Calls_B}$. A PolicyChecker algorithm for platform Θ and changed application B will allow the update if for all applications $A \in \mathcal{A}$ and services $s \in \mathcal{S}$

- **Security on contract level**: if service s of applet B is in $\mathsf{Calls_A}$ then A is authorized by B to call s ($(s, A) \in \mathsf{sec.rules_B}$).
- **Functionality on contract level**: if service s of applet B is in $\mathsf{func.rules_A}$ then $s \in \mathsf{Provides_B}$.

The main idea behind the security model is that if the ClaimChecker and the PolicyChecker are sound and they returned true for $\forall A \in \mathcal{A}$ then the platform Θ is secure.

3 Policy Checker Implementation

We have implemented the PolicyChecker for installation of a new application, as the most interesting and representative case, using Sun's Java Card simulator for Java Card 2.2.2 specification [10]. Contracts are implemented as instances of `Contract` Java class that are included in the CAP file of the applet. The PolicyChecker has been implemented as `Card` Java Card applet. The `Card` class has a field `Card.Pool` that stores the platform policy. `Card.Pool` is an instance of `Map` Java class that associates with each applet on the platform its contract. The applet is identified by a String that contains the AID while the contract is an instance of the `Contract` class. The method `validateContract(Contract)` of the class `Card` has as input parameter the `Contract` of a new applet and returns the result of the evaluation of `Contract` against the platform policy stored in field `Card.Pool`. If `Contract` is compliant with the platform policy, the new applet is installed, otherwise it is rejected.

We have tested the feasibility of installing the PolicyChecker on the card as an additional applet. In particular, we have evaluated the communication overhead associated with the installation of the PolicyChecker in terms of number

of APDU commands exchanged between the terminal and the card to load the CAP file. In fact, when the Java classes are converted into the CAP file, the terminal converts them into data sequences (APDUs), which then are used to upload the code and make it selectable. This is a good indication of the cross-platform memory footprint of the applet as default APDUs are up to 255 bytes. The CAP file generated for the PolicyChecker applet contains 18 Java classes and generates 118 APDUs commands to load the applet on the card. Thus, installing the PolicyChecker on the card does not require more APDUs than the installation of a normal applet. We have also evaluated the overhead of installing an applet along with its contract. We generated a `Contract` class for a sample *Transport* applet (T for short). The $\mathsf{Contract}_\mathsf{T}$ contains 3 services in $\mathsf{Provides}_\mathsf{T}$, 3 in $\mathsf{Calls}_\mathsf{T}$, 4 authorized applets in $\mathsf{sec.rules}_\mathsf{T}$ and 2 services in $\mathsf{func.rules}_\mathsf{T}$. The $\mathsf{Contract}_\mathsf{T}$ is rather complicated comparing with a contract an average smart card application can produce (usually applets have up to 2 services). The Java Card representation of the $\mathsf{Contract}_\mathsf{T}$ is only 7 APDUs.

We do not show here how to construct a ClaimChecker. An example can be found in Ghindici et al. [5]. The ClaimChecker they have built is working on more complex information flow models and it can be restricted to our Contract model.

4 Related Works and Conclusions

Ghindici et al. [5] propose a domain specific language for security policies capturing the information flow within small embedded systems. In the framework they propose each application is certified at loading time, having a information flow signature assigned to each method, describing the flow relations between method variables. Huisman et al. [7] present a formal framework and a tool set for compositional verification of application interactions on a multi-application smart card. Their method is based on construction of maximal applets, w.r.t structural safety properties, simulating all the applets respecting these properties. Model checking techniques can be then used to check whether a composition of two applets A and B respects some behavioral safety property.

Girard in [6] suggests to associate security levels (clearances) to application attributes and methods, using traditional Bell/La Padula model. Bieber et al. adopt this approach in [2] and propose a technique based on model checking for verification of actual information flows. The same approach is used by Schellhorn et al. in [12] for their formal security model for operating systems of multi-application smart cards. Avvenuti et al. in [1] propose a tool for off-card verification of Java bytecode files, that could be later installed on the card, their method explores the multi-level policy model and the theory of abstract interpretation.

Outside of the smart cards domain, the techniques for policy enforcement in multi-application environment are investigated also for mobile platforms and operation systems. Ongtang et al. in [11] have proposed the Saint framework for Android mobile platform applications to impose requirements on the usage of their services on other applications during installation time and run-time.

Applications on a Saint-enabled Android platform can define permissions and demand fulfillment of certain requirements by both their callers and callees. The Kirin framework mentioned above was developed for Android by Enck et al. [4], it can check permissions application requests at installation time in order to capture possibly dangerous combinations of permissions and warn the user.

In this paper we have proposed an extension of the Java Card security mechanisms for open multi-application smart cards that makes it possible to do verify updates on the card. This extension adds two components to the JCRE, the ClaimChecker and the PolicyChecker. In a nutshell, all applications are arriving with specifications of their behavior and their requirements on other applications on the platform. These requirements merged together create platform security policy. The card can check autonomously whether they are acceptable and then either reject or accept the change.

References

1. M. Avvenuti, C. Bernardeschi, N. De Francesco, and P. Masci. A tool for checking secure interaction in Java Cards. In *Proc. of EWDC2009*, 2009.
2. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J-L. Lanet. Checking secure interactions of smart card applets: Extended version. *J. of Comp. Sec.*, 10(4):369–398, 2002.
3. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: towards a semantics for digital signatures on mobile code. In *Proc. of EuroPKI-07*, volume 4582 of *LNCS*, pages 297 – 312. Springer-Verlag, 2007.
4. W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of CCS 2009*, pages 235–245. ACM, 2009.
5. D. Ghindici and I. Simplot-Ryl. On practical information flow policies for java-enabled multiapplication smart cards. In *Proceedings of CARDIS 2008*, volume 5189 of *LNCS*, pages 32–47. Springer-Verlag, 2008.
6. P. Girard. Which security policy for multiplication smart cards? In *USENIX Workshop on Smartcard Technology*. USENIX Association, 1999.
7. M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: a case study. In *FASE'04*, volume 2984 of *LNCS*, pages 84–98. Springer-Verlag, 2004.
8. GlobalPlatform Inc. GlobalPlatform Card Specification. Specification 2.2, 2006.
9. Lufthansa. Miles&More credit cards. On the web at http://www.miles-and-more.com.
10. Sun Microsystems. Runtime environment specification. Java Card^{TM} platform, version 2.2.2. Specification 2.2.2., Sun Microsystems, 2006.
11. M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Proceedings of ACSAC 2009*, pages 340–349, 2009.
12. G. Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, and D. Toll. Verification of a formal security model for multiapplicative smart cards. In *ESORICS'00*, volume 1895 of *LNCS*. Springer-Verlag, 2000.
13. 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 the 19th ACM Symp. on Operating Syst. Princ.*, pages 15–28, 2003.