Extended Abstract: Embeddable Security-by-Contract Verifier for Java Card*

Olga Gadyatskaya, Eduardo Lostal, and Fabio Massacci

DISI, University of Trento,
via Sommarive, 14, Povo 0, Trento, Italy, 38123
{surname} @disi.unitn.it

Abstract. Modern multi-application smart cards based on the Java Card technology can become an integrated environment where applications from different providers are loaded on the fly and collaborate in order to facilitate lives of the cardholders. This initiative requires an embedded verification mechanism to ensure that all applications on the card respect the application interactions policy.
The Security-by-Contract (S×C) approach for loading time verification consists of two phases. During the first phase the loaded bytecode is verified to be compliant with the supplied contract. Then, during the second phase the contract is matched with the smart card security policy. In the paper we report about implementation of a S×C prototype, present the memory statistics that justifies the potential of this prototype to be embedded on an actual device and discuss the Developer S×C prototype that can be run on a PC.

1 Introduction

Multi-application smart cards are an appealing business scenario for both smart card vendors and smart card holders. Applications interacting on such cards can share sensitive data and collaborate, while the access to the data is protected by the tamper-resistant integrated circuit environment. In order to enable such cards a security mechanism is needed which can ensure that policies of each application provider are satisfied on the card. Though a lot of proposals for access control and information flow policies enforcement for smart cards exist [1, 6, 9], they fall short when the cards can evolve after issuance. The scenario of a dynamic and unexpected post-issuance evolution of a card, when applications from potentially unknown providers can be loaded or removed, is very novel.

For a dynamic scenario, traditionally, run-time monitoring is the preferred solution. But smart cards do not have enough computational capabilities for implementing complex run-time checks. Thus the proposal to adapt the Security-by-Contract approach (initially developed for mobile devices [2]) for smart cards appeared. In the Security-by-Contract (S×C) approach each application supplies

* This paper is a short version of [4]. It provides the high-level engineering aspects of the research results.

on the card its contract, which is a formal description of the application behavior. The contract is verified to be compliant with the application code, and then the system can ensure that the contract matches the security policy of the card.

The S×C framework deployed on the card ensures that all the loaded applications interact in compliance with the security policy of each application provider. In comparison with the existing works aiming at enforcing application interaction policies in a dynamic setting [3, 5], we improve the state of the art in the following (1) the S×C prototype was implemented to be integrated with an actual device, taking into account the memory usage restrictions, (2) we have developed the full eco-system of the S×C verifier based on the standard Java Card tools and specifications available, (3) we have implemented also a version for developers that can be run on a Windows-based PC.

The rest of the paper is structured is follows. Section 2 contains a brief overview of the Java Card technology and then we outline the S×C solution for Java Card (Section 3) emphasizing the changes to the platform. The design and implementation details are outlined in Section 4 For on-card prototypes small memory footprint is a must, we therefore present the memory usage statistics (for non-volatile memory and RAM) that demonstrates feasibility of the embedded implementation (Section 5). We conclude with Section 6.

2 The Java Card Platform

Java Card is a popular middleware for multi-application smart cards that allows post-issuance installation and deletion of applications. Application providers develop *applets* (Java Card applications) in a subset of Java. Full description of the Java Card language is provided in the official specifications [8]. Figure 1 presents the architecture of an integrated circuit with the Java Card platform installed and the application loading process. The architecture comprises several layers including device hardware, an embedded operating system (native OS), the Java Card run-time environment (JCRE) and the applications installed on top of it. Important parts of the JCRE are the Java Card virtual machine (JCVM) (its Interpreter part), the Installer, which is an entity responsible for post-issuance installation and removal of applications and the Loader, that comprises a set of API to access the loaded bytecode.

Applications are supplied on the card in packages. The source code of a package is converted by the application providers into class files and then into a CAP file. The CAP file is, essentially, an optimized Java Card bytecode, it consists of several efficiently organized components each carrying specific information. The CAP file is transmitted onto a smart card, where it is processed and linked.

The interactions between applets from different packages are mediated by the JCRE firewall. If two applets belong to different packages, their *contexts* are different, and the Java Card firewall confines applet's actions to its designated context. Thus, normally, an applet can reach only objects belonging to its own context. The only applet's objects accessible through the firewall are methods

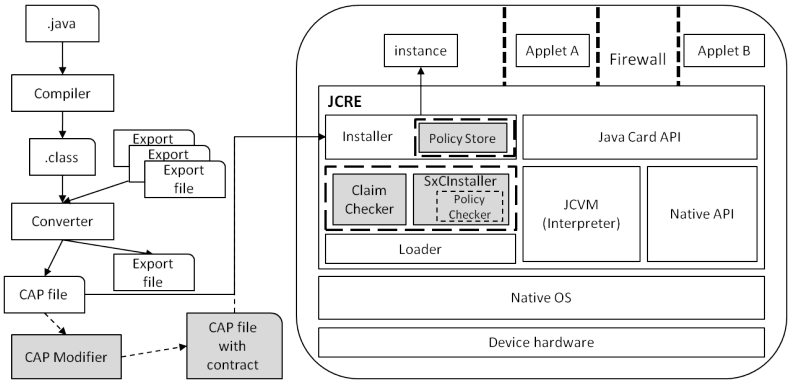

Fig. 1. *The Java Card architecture and the loading process. The white components and data structures belong to the standard Java Card platform. The grey components and data structures are additions introduced by the* S×C *scheme. The dashed lines denote the changes to the loading process.*

of specific *shareable interfaces*, also called *services*. A shareable interface is an interface that extends `javacard.framework.Shareable`.

An application A implementing some services is called a *server*. An application B that tries to call any of these services is called a *client*. A typical scenario of service usage starts with a client's request to the JCRE for a reference to A's object (that is implementing the necessary shareable interface). The firewall passes this request to application A, which decides if the reference can be granted or not. If the decision is positive, the reference is passed through the firewall and is stored by the client for further usage. The client can now invoke any method declared in the shareable interface which is implemented by the referenced object. During invocation of a service a *context switch* will occur, thus allowing invocation of a method of the application A from a method of the application B. A call to any other method, not belonging to a shareable interface, will be stopped by the Java Card firewall.

As all applet interactions inside one package are not controlled by the firewall and due to the fact that a package is loaded in one pass, we consider that one package contains only one applet and there is an one-to-one correspondence between packages and applications. Another important assumption for us is that packages do not implement shareable interfaces declared in other packages, this assumption can in fact be guaranteed by the S×C framework.

Currently the services access control enforcement on Java Card is embedded into the application code. Traditionally, the server will receive an AID (unique application identifier) of the client requesting its service from the JCRE and check that this client is authorized before granting it the reference to the object (that can implement multiple services). Once the object reference is received, the client can access all the services within this object and it can also leak the

object reference to other parties. The S×C framework checks the authorizations for each service access, thus the object reference leaks are no longer a security threat. In Java Card the controls for checking the invocation context can also be embedded directly in the code of each service. We argue that this approach is not satisfactory, as the access control list of authorized clients can be updated only through complete removal and reinstallation of the server applet. When the server package is imported by other (client) packages on the card the server removal is not possible. The embedded S×C verifier can enforce the same service access control policies in a flexible fashion when each server can update its policy without reinstallation.

In a CAP file all object types and methods are referred to by their tokens which are used by the JCRE for on-card linking. A service s is identified as a tuple $\langle A, I, t\rangle$, where A is the AID of the package providing the service s, I is a token for a shareable interface where the service is defined and t is a token for the method in the interface I. The correct service tokens can be obtained from the Export file of a package (produced by the Converter) or from the CAP file.

The JCRE imposes some restrictions on method invocations in the application bytecode. Only the opcode `invokeinterface` allows to perform the context switch between two different packages. Thus, in order to collect all potential service invocations we analyze the bytecode and infer from the `invokeinterface` instructions possible services to be called. More details are available for the interested reader in [4].

3 Security-by-Contract for Java Cards

In the Security-by-Contract scheme every application carries its contract embedded into the CAP file. Let $A.s$ be a service s declared in a package A. The contract consists of two parts: AppClaim and AppPolicy. AppClaim specifies provided and invoked services (Provides and Calls sets correspondingly). We say that the service $A.s$ is provided if applet A is loaded and service s exists in its code. Service $B.m$ is invoked by A if A may try to invoke $B.m$ during its execution. The AppClaim will be verified for compliance with the bytecode (the CAP file).

The application policy AppPolicy contains authorizations for services access (sec.rules set) and functionally necessary services (func.rules set). We say a service is necessary if a client will not be functional without this service on board. The AppPolicy lists applet's requirements for the smart card platform and other applications loaded on it. A functionally necessary service for applet A is the one which absence on the platform will crash A or make it useless. For example, a transport application normally requires some payment functionality to be available. If a customer will not be able to purchase the tickets, she would prefer not to install the ticketing application from the very beginning. It is required that for every application A $\mathsf{func.rules_A} \subseteq \mathsf{Calls_A}$. An authorization for a service access contains the package AID of the authorized client and the service tokens. The access rules have to be specified separately for each service and each client that the server wants to grant access.

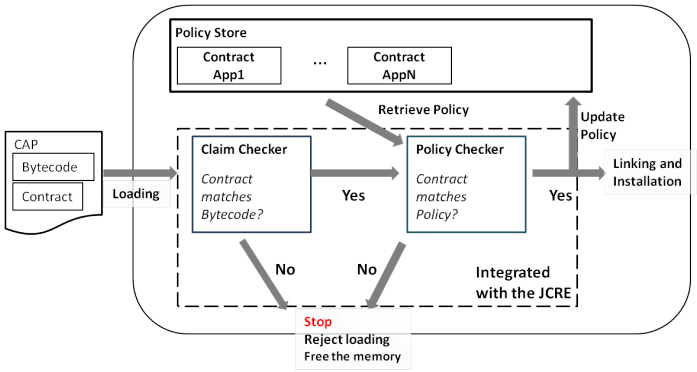

Fig. 2. *The Security-by-Contract workflow for loading.*

Contracts are delivered on the card within Custom components of the CAP files. CAP files carrying Custom components can be recognized by any Java Card Installer, as the Java Card specs require. More details on the structure of the Contract Custom component that we proposed are available in [4]. To ease the contract creation we have developed the CAP Modifier tool with a user-friendly graphical interface allowing to edit any section of the contract, save already created contracts as files for future usage and embed the created contracts into CAP files. The tool takes the CAP file generated with the standard Java Card tools and appends the Contract Custom component within it, modifying the Directory component of the CAP file accordingly (as the specification requires).

The S×C framework deployed on the card consists of two main components integrated with the platform: the ClaimChecker and the PolicyChecker. The ClaimChecker performs extraction of the contract and verifies that it is compliant with the application code. Then the PolicyChecker ensures that the security policy of the card, composed by all the contracts of currently loaded applications, is compliant with the contract. Another addition to the platform is the PolicyStore component. The PolicyStore appears due to the fact that only components implemented in Java Card (applets and the Installer) can allocate space in EEPROM (mutable persistent memory), that is the only type of memory suitable to store the security policy across the card sessions. The PolicyStore is a class in the Installer. Figure 1 depicts the S×C prototype embedded into the Java Card platform. Figure 2 summarizes the S×C workflow for loading, as the most interesting case, emphasizing the role of each component.

4 Implementation of the S×C Prototype

We have implemented the S×C prototype in C, as it is a standard language for smart card platform components implementation and the Loader API we had knowledge of was written in C. The main components of the S×C prototype are:

SxCInstaller. This component is an interface with the Installer. SxCInstaller calls the ClaimChecker that in a positive case (contract and bytecode are compliant) will return the address of the contract in the Contract Custom Component of the CAP file being loaded. The SxCInstaller also comprises (for memory saving reasons) the PolicyChecker component.

ClaimChecker. This component is called by SxCInstaller. It carries out the check for the compliance between the contract and the CAP file. The check is carried out after parsing the CAP file. We used a part of the standard Loader API, called in the current paper CAPlibrary, that contains functions to access the beginning and the length of each CAP file component. Using the functions of the CAPlibrary library for CAP file parsing on-card, this component gets the initial address of the components it needs from which it can eventually parse the rest of the components. If the result is positive, the ClaimChecker will return the address of the contract of the application in the Contract Custom component.

The ClaimChecker component has to establish that the services from $\mathsf{Provides}_\mathsf{A}$ exist in package A and the services from $\mathsf{Calls}_\mathsf{A}$ are indeed the only services that A can try to invoke in its bytecode. The goal of the ClaimChecker algorithm is to collect each `invokeinterface` opcode with its parameters (the method t and the Constant Pool index I). Then the collected set is compared with the set Calls of the contract. We emphasize that operands of the`invokeinterface` opcode are known at the time of conversion into a CAP file and thus are available directly in the bytecode. All methods of the application are provided in the Method Component of the application's CAP file, an entry for each method contains an array of its bytecodes. Exported shareable interfaces are listed in the Export component of the CAP file and flagged in the Class component. The strategy for the ClaimChecker is to ensure that each service listed in the Provides set is meaningful and no other provided services exist. More details of the ClaimChecker algorithm can be found in [4].

Due to the lack of space we only present the security policy data structures just to give a flavor of this part of the system. The security policy stored on the card consists of the contracts of the currently loaded applications. A contract in the form supplied on the card is a space-consuming structure (each AID can occupy up to 16 bytes). Therefore we have resolved to store the security policy on the card in a bit vectors format. The current data structure for security policy assumes there can be up to 10 loaded applets, each containing up to 8 provided services. Thus the security policy is a known data structure called Policy with a fixed format, the bits are taking 0 or 1 depending if the applet is loaded or the service is called/provided. The Mapping object maintains correspondence between the number the applet gets in the on-card security policy structure and the actual AID of the package, and between the provided service token and the number of this service in the policy data structure. The other two objects that are part of the on-card security policy are the MayCall list and WishList list, containing the potential future authorizations, necessary for a case when a loaded application carries a security rule for some application not yet on the card and the services that are called by applications but are not yet on the

card, because the server is not yet loaded, or because the current version of the server does not provide this service correspondingly. The PolicyStore, being written in Java Card, has to communicate the security policy to the PolicyChecker component (the SxCInstaller) that will run the contract-policy compliance check. This communication is implemented through usage of a native API.

The Developer S×C Prototype. The S×C prototype for experimenting on a PC comprises the same components: the SxCInstaller, the ClaimChecker and the PolicyStore, which in the Developer version is packaged as an applet. The communication between the SxCInstaller and the PolicyStore applet is emulated by using files. The S×CDeveloper prototype emulates deployment of the PolicyStore applet on a card using the Java Card development kit from Oracle. For the purposes of independent functionality testing we have implemented the CAPlibrary library and the necessary Installer functionality following the JCRE specifications. The Developer prototype accepts as input CAP files with the contract embedded into the Custom component by the CAP Modifier tool, runs the verification algorithms and outputs the results, notifying also which of the checks failed during verification (if any, otherwise it reports successful loading of an applet and updates the current security policy). Thus developers can create and embed different contracts and try the verification process.

5 Evaluation

In this section we report the memory measurements of the prototype carried out in the University of Trento. The details of the industrial evaluation performed by Trusted Labs (commissioned by Gemalto) can be found in [7]. The most important characteristics for an on-card component are RAM and non-volatile memory (NVM) consumption. NVM space is required to store the prototype and the necessary data (the security policy) across the card sessions. RAM memory is used to store the temporary data while the verification is performed.

We have explored two metrics for the NVM usage estimations off-device: the size of the object files in C compiled on a PC and the number of lines of code (LOCs). The ClaimChecker component object file compiled with the MinGW compiler tools occupies 6.5 KB, the ClaimChecker has 170 LOCs (.h + .c). The SxCInstaller object file occupies 7.3 KB, this component includes 178 LOCs. The PolicyStore applet CAP file (exact on-device measure) occupies 6KB.

RAM usage is also very important, as over-consumption of RAM by the prototype can lead to the denial of service. The higher is the RAM consumption, the lower is the level of interoperability of the prototype; because some cards cannot provide a significant amount of RAM for the verifier which has to run in the same time with the Installer. We have used a temporary array of 255 bytes to store the necessary computation data. 255 bytes is a small temporary memory buffer which ensures the highest level of interoperability for the prototype.

6 Conclusions

In the paper we have presented the S×C prototype for the Java Card-based smart cards. The prototype can be used to perform load time verification of Java Card applets and enforce service access control policies in a flexible way. The prototype is integrated on the card with the Loader API which provides direct access to the received bytecode. The prototype is able while parsing the bytecode to extract the application contract and to compare it with the actual code of the application. Then the received contract is transformed into the memory-efficient on-card format and compared with the security policy of the device. The memory statistics demonstrates feasibility of embedding the proposed prototype on an actual device. We have also developed the Developer S×C prototype that can be demonstrated on a usual PC without actual device, all the on-card functions necessary to process CAP files were implemented using the Java Card specifications.

Acknowledgements. We thank Quang-Huy Nguyen and Boutheina Chetali for insights on Java Card. This work was partially supported by the EU under grants EU-FP7-FET-IP-SecureChange and FP7-IST-NoE-NESSOS.

References

1. 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.
2. 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.
3. A. Fontaine, S. Hym, and I. Simplot-Ryl. On-device control flow verification for java programs. In *ESSOS'2011*, volume 6542 of *LNCS*, pages 43–57. Springer.
4. O. Gadyatskaya, E. Lostal, and F. Massacci. Load time security verification. In *ICISS'2011*, volume 7093 of *LNCS*, pages 250–264. Springer.
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. 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.
7. P. Capelastegui B. Chetali O. Delande M. Felici F. Innerhofer-Oberperfler V. Meduri F. Paci S. Paul B. Solhaug A. Tedeschi M. Angeli, G. Bergmann. D1.3: Report on the industrial validation of SecureChange solutions. *SecureChange EU project public deliverable, www.securechange.eu*, 2012.
8. Sun Microsystems. Virtual Machine and Runtime Environment. Java Card^{TM} platform. Specification 2.2.2, Sun Microsystems, 2006.
9. 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.