A Load Time Policy Checker for Open Multi-Application Smart Cards

Nicola Dragoni, Eduardo Lostal
DTU Informatics, Technical University of Denmark
Email: ndra@dtu.dk, edlo@imm.dtu.dk

Olga Gadyatskaya, Fabio Massacci, Federica Paci
DISI, University of Trento
Email: surname@disi.unitn.it

***Abstract*—Applications on multi-application smart cards contain sensitive data and can exchange information. Thus a major concern is that these applications should not exchange data unless permitted by their respective policy. As modern smart cards allow post-issuance installation and removal of applications, traditional approaches for information flow analysis are not suitable.**

We suggest the Security-by-Contract approach for loading time application certification on the card, that will enable the stakeholders with the means to ensure the compliance of every update of the card with their security policy. We describe an extension of the card security architecture to deal with verification for different types of updates and present a Java Card prototype implementation of the Policy Checker with performance measurements.

***Index Terms*—Smart cards security, application certification, policy models, information exchange.**

I. Introduction

Though smart card technology is mature enough to support multiple applications from different providers, multi-application smart cards are not widely adopted. One of the main reasons for such rare adoption is that the mechanism to install or update applications on smart cards and the security mechanism that controls the interactions between applications are not flexible.

The most popular solution for smart cards now is GlobalPlatform (GP) [8] on top of Java Card [10]. Before smart cards are released to users, vendors have to get a certificate which guarantees that their proprietary implementation of GP is compliant with Common Criteria [11].

When an application is installed or updated on the card, it is important to make sure that the application is compliant with the platform policies. Thus, vendors have to recall the smart card from the users and obtain a new certificate. The security mechanism is also not flexible: since the security policy governing the interactions of an application with other applications on the card is embedded in the application code, an update of the policy requires to change the application code and to re-install it on the card. Thus, vendors have to obtain a new certificate also when security policies only change.

We propose a framework to verify at the loading time that newly installed applications on the card comply with the platform policies, thus smart cards do not need to be certified again. The framework is based on the Security-by-Contract (S×C) approach illustrated in Figure 1. Each application comes with its *contract* that specifies the functionalities provided by

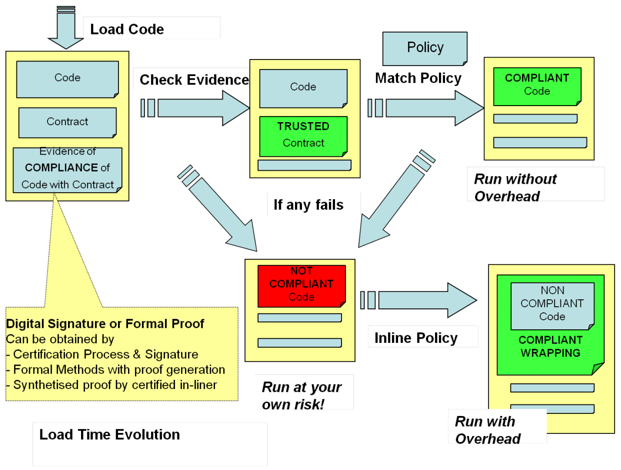

Fig. 1. Security-by-Contract for Load Time Evolution

the application and the functionalities of other applications the application can try to invoke. The contract also specifies which applications are authorized to invoke the application's functionalities and which functionalities provided by other applications, the application needs to call. A PolicyChecker evaluates the contract against the platform policy that is the union of all the contracts of the applications installed on the card. We present a proof-of-concept implementation of the PolicyChecker component of the S×C framework and give the performance evaluation results. The experimental results show that the PolicyChecker implementation does not consume a lot of memory and thus is suitable for memory limited devices such as smart cards.

The paper is organized as follows. The Java card architecture with the S×C components is introduced in Section II. We define the application contract and the policy in Section III. The prototype implementation of the PolicyChecker component is presented in Section IV. We discuss related work and conclude the paper in Section V.

II. Multi-application Smart Card Architecture

The Java Card architecture consists of several layers as illustrated in Fig. 2: device hardware, an embedded operating system (OS), a Java Card Runtime Environment (JCRE) on top of the embedded OS, and the applications (also called *applets*) installed on the smart card, that are written in (a subset of)

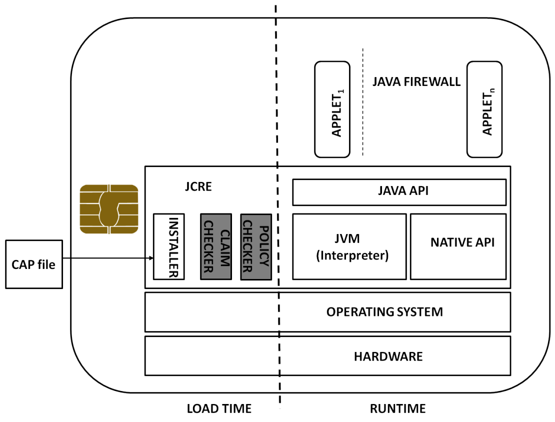

Fig. 2. Java Card with Security-by-Contract Architecture

the Java language [10].

The JCRE is responsible for managing and executing applets. It is composed of a Java Virtual Machine (JVM), native API, framework APIs, and an application installer. The installer downloads and installs the applications on the card. Before being loaded on the smart card, applets are converted into a CAP file, that is a binary executable representation of the Java classes which compose the applet. Having received the CAP file, 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 with the JCRE. Then the JVM can execute the code contained in the CAP file.

The Java Card applets installed on a smart card are isolated by the Java firewall. If an applet (*server*) wants to share data with another applet (*client*) from a different package, it has to implement a *shareable interface* defining the methods available though the firewall. These methods (or *services*) are the only methods of the server applet accessible through the firewall.

The firewall will pass the call for the shareable interface from the client to the server. In the current security model the server code includes an access control list of the applets that are allowed to call it. If the client is in the list, the access is granted and the server passes through the firewall a reference to an object implementing the necessary interface. If the client is not in the list, it only gets `null`. After the reference is received the client can use services of the server by invoking the corresponding methods of the obtained object. As we already mention, this security mechanism is not flexible, as any change in the security policy (for example, revocation of an access right) requires a change in the code of the applet. In order to address the shortcomings of the Java Card security architecture, we extend the JCRE based on the Security-by-Contract approach with two additional components: the ClaimChecker and the PolicyChecker(see Figure 2). This addition allows to preserve the security of the smart card when a new applet is installed, an old one is being updated or removed, or applet policies change. The basic idea of our proposal is that the ClaimChecker component checks that the code of the new applet is compliant with its contract, while the PolicyChecker verifies that the contract is compliant with the security policy on the card.

III. Security-by-Contract for Smart Cards

The contract in the S×C approach is a specification of an application that will help honest application providers to describe what their application is going to do and the platform to check that this application is well-behaved [2].

The $\mathsf{Contract}_\mathsf{A}$ of an applet A consists of two parts:

- The $\mathsf{Claim}_\mathsf{A} = \langle \mathsf{Provides}_\mathsf{A}, \mathsf{Calls}_\mathsf{A} \rangle$ describes the actual behavior of A in terms of provided and called services. $\mathsf{Provides}_\mathsf{A}$ is a set of services that applet A provides as a server. $\mathsf{Calls}_\mathsf{A}$ is a set of services of other applets that A can try to invoke. It is possible to compare $\mathsf{Claim}_\mathsf{A}$ with the actual code of A received by the card.
- The $\mathsf{AppPolicy}_\mathsf{A} = \langle \mathsf{Allows}_\mathsf{A}, \mathsf{Needs}_\mathsf{A} \rangle$ is declared by the application provider and it details the desirable behavior of other applications on the card with respect to the applet A. $\mathsf{Allows}_\mathsf{A}$ contains service access rules as pairs (s, B), where s is a service of A and B is some applet. $\mathsf{Needs}_\mathsf{A}$ is a set of functionally necessary services. Namely, it contains services of other applets that A needs in order to be functional.

Applications may define usage rules in Allows even for applications not yet present on the card. But if some application A declares that it needs a service of another application B, and B is not yet present on the card, we decline A the access to the card, since A will not be functional. An example of a card where functionality can be an issue is a public transportation card. A ticketing applet needs an access to some payment services, otherwise a customer is not able to buy new tickets and the ticketing applet becomes useless. We use notation $A.s$ for a service s of an applet A. In actual contracts applications are referred to by their AID (unique application identifiers) and services are referred to by their CAP file tokens [10].

We consider the following properties:

- *Stable Security* After an update of the card if an applet A can invoke a service s of an applet B, then A is authorized to do it : if $B.s \in \mathsf{Calls}_\mathsf{A}$ then $(s, A) \in \mathsf{Allows}_\mathsf{B}$.
- *Stable Functionality* After an update every applet is functional: if $B.s \in \mathsf{Needs}_\mathsf{A}$ then $s \in \mathsf{Provides}_\mathsf{B}$.

On multi-application smart cards the *security policy* is a collection of policies of each stakeholder on the card. When a new applet A arrives on the platform, first it is necessary to test whether $\mathsf{Claim}_\mathsf{A}$ meets the policy of the other stakeholders. Second, A will provide its own policy on the card, which, after ensuring its compliance with behaviors of other applets, will be incorporated into the platform policy. Thus security policy on the card $\mathsf{Policy} = \{\mathsf{Contract}_{\mathsf{A}_1}, \ldots, \mathsf{Contract}_{\mathsf{A}_n}\}$, where $A_1, \ldots, A_n$ are applications installed on the card.

A. Multi-application Card Example

We assume several stakeholders on the card: Bank, Transport, Sky and SASTravel. We use a notation $A@Provider$

TABLE I
CONTRACTS CASE STUDY

Application	Claim		AppPolicy	
	Provides	Calls	Needs	Allows
EMV@BANK	transaction fill_purse	-	-	(transaction, ePurse@BANK) (fill_purse, ePurse@BANK)
ePurse@BANK	payment account_balance	fill_purse transaction	fill_purse	(payment, jTicket@Transport) (account_balance, jTicket@Transport)
jTickect@Transport	buy_ticket	payment	payment	-
Weather@Sky	weather_info weather_RSS	-	-	(weather_info, eTicket@SASTravel) (weather_RSS, eTicket@SASTravel)
eTicket@SASTravel	-	weather_info weather_RSS	weather_info	-

to denote an application A belonging to a provider $Provider$. The following applications try to be installed: *EMV@BANK*, *ePurse@BANK*, *jTicket@Transport*, *Weather@Sky*, *eTicket@SASTravel*.

The details of the contracts are shown in Table I. It is easy to see that the applets *EMV, ePurse, jTicket, Weather* and *eTicket* provide compliant contracts and can be installed on the card.

IV. PolicyChecker PROTOTYPE IMPLEMENTATION

The PolicyChecker component accesses the received contract and maintains the policy (collected from installed applets). We developed a proof-of-concept prototype of the PolicyChecker implemented as an applet. Informally, the PolicyChecker has to check that the stable security and stable functionality properties will be maintained by the card after an update. If this will not be the case, the update is rejected. Otherwise it is accepted, and the policy is updated correspondingly (in case of installation a new contract is added to the policy and in case of removal the old contract is deleted). The specification of the PolicyChecker component, including algorithms, can be found in [4].

We have implemented a prototype of the PolicyChecker based on Java Card Technology (JC) 3.0.2 (Connected Edition) [10]. To implement the PolicyChecker prototype, we have used the card emulator provided by C-JCRE, a reference implementation of the JCRE [14]. The emulation environment allows the simulation of the persistent memory of the card (EEPROM), saving contents and restoring them in the subsequent session. The interaction with the simulator is done by means of scripts containing the Application Protocol Data Unit (APDU) commands to be sent to the card.

The implementation consists of three main classes: *Data Structures*, *Algorithms*, and *APDU Communication*. The two key classes within *Data Structures* class are *Contract* and *Policy* representing a contract and a policy respectively. The *Contract* class has three attributes: a String for the application ID, a *Claim* and an *AppPolicy* data structures. A *Claim* structure contains two vectors of Strings, one for *Calls* and one for *Provides*. *AppPolicy* contains a String for the application ID, a vector of *Allows* and a vector of Strings for *Needs*. *AppPolicy* contains the application ID because it is possible to update only this field. The class *Allows* has two String attributes to identify the service ID and the application ID.

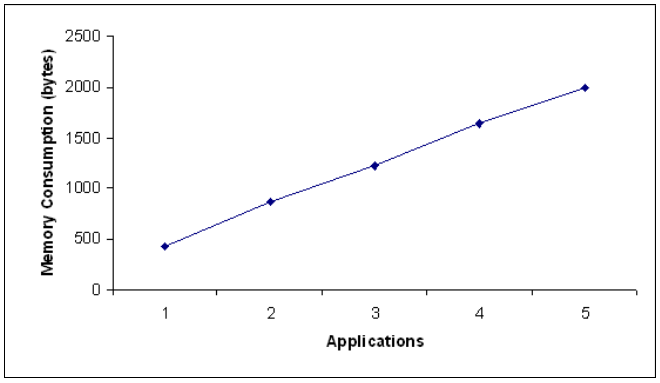

Fig. 3. Memory Usage After the Addition of Each Application's Contract

The *Policy* class has two attributes: a Vector of *Dependents* (*PolNeeds*) and a Vector of *PolAllows*. *Dependents* contains a String for the application ID and a vector of Strings for the applications that need some services that application ID provides (we reorganize the security policy for convenience of implementation). Finally, *PolAllows* class has as attributes a String for the application ID and a Vector of *Allows*.

The *Algorithms* class contains the implementation of all the contract-policy compliance verification algorithms. In a nutshell, the implementation directly follows the specifications provided in [4], so algorithms are basically implemented as a set of nested loops operating on the Vector data structure. To optimize time and memory consumption, we have used native functions (in particular related to Vector operations).

The *APDU Communication* class supports the communication with the off-card entity by means of the exchange of APDU messages. It receives the APDU commands from an off-card entity and processes them, deciding the operation to be carried out according to the instruction byte (INS) [9]. Appropriate algorithms from the *Algorithms* class are called then to manage the addition, update and removal operations.

We have evaluated the overall S×C approach with respect to memory usage that is an important parameter to show the feasibility of the approach on limited-resource constrained device such as smart cards. We focus only on the EEPROM memory because the applications and data are dynamically loaded there.

The prototype requires 11 kB of free memory on the card to run. Since current smart cards use between 64 and 128 kB memories (and this is expected to increase), the prototype only occupies a small amount of the smart card memory space.

Since in the S×C framework both contracts and policy are stored on the card, another key measure is the extra memory space required to store them. We evaluate the prototype by assuming that the applets introduced in the section III-A *EMV@BANK*, *ePurse@BANK*, *jTicket@Transport*, *Weather@Sky*, *eTicket@SASTravel* are installed one by one.

Figure 3 shows the memory usage after the addition of each contract. The extra memory space consumed by the S×C framework (contracts and policy) is 2.2 kB. Table II shows

TABLE II
DETAILS OF THE MEMORY USAGE FOR THE CASE STUDY

	Claim	AppPolicy	Contract	PolAllows	PolNeeds	System
EMV@BANK	104	144	272	96	96	
ePurse@BANK	112	144	280	96	112	
jTicket@Transport	104	122	240	56	56	
weather@Sky	104	144	272	96	56	
eTicket@SASTravel	96	112	232	56	56	
Total	1352			860		2220

the memory space (in bytes) required to store different parts of the application contracts. In the Table, constant variables are taken into account and this explains why the additions are not exact. This means, for instance, that *Contract* for *EMV@BANK* application is the addition of the *Claim* and the *AppPolicy* parts plus some constant variables added in the source code.

In summary, the prototype needs around 11 kB to store the code on the card, while the extra space required to store the case study applications is 2.2 kB. This means a small extra load taking into account the security benefits. For these reasons, we can conclude that the S×C framework can be implemented on smart cards.

As for the ClaimChecker implementation, it is hindered by the fact that this component needs to have direct access to the bytecode of the received applet in order to capture provided and received services. Thus it has to be integrated with the installer, which is a proprietary component developed usually in C, its implementation details are not available.

V. RELATED WORK AND CONCLUSIONS

Ghindici et al. [5] propose a domain specific language for security policies capturing the allowed information flow inside Java-based small embedded devices. Each application is certified at loading time, having an information flow signature assigned to each method, describing the relations between method variables annotated with type of the flow. These policies also capture the interactions between applications in the system and are able to detect sensitive data leaks.

Huisman et al. present a formal framework and a tool for compositional verification of application interactions on a multi-application smart card [7]. The approach is based on maximal applets construction w.r.t structural safety properties, simulating all the applets respecting these properties. Model checking techniques are then used to check if a composition of two applets respects some behavioral safety property.

Girard in [6] suggests to associate security levels to application attributes and methods, using the traditional Bell/La Padula model. Bieber et al. adopt this approach in [1] and propose a technique based on model checking for verification of actual information flows. The same approach is used by Schellhorn et al. in [13] for a formal security model for operating systems on multi-application smart cards.

Outside of the smart cards domain, the techniques for loading time policy enforcement in multi-application environments were investigated also for mobile platforms. Ongtang et al. propose the Saint framework for the Android platform to impose requirements on services usage on other applications during installation time and run-time [12]. Applications on Saint-enabled Android platforms can define their own permissions and demand the fulfillment of certain requirements by both their callers and callees. The Kirin framework is developed for Android by Enck et al. [3], it checks the permissions application requests at installation time in order to capture possibly dangerous combinations of permissions and suggest a user not to install a dangerous software.

In this work we have presented the *Security-by-Contract* framework as a security solution for open multi-application smart cards, which can be used in a dynamic environment when applications come and go and the security policy can be changed. The framework has been specified in terms of services used and provided by applications on the card. The main benefit of the approach is the loading time verification which ensures that at the runtime the policy of each applet will be satisfied. We reported about prototype implementation of the PolicyChecker component of the framework that demonstrated feasibility of the approach. The prototype is implemented as an applet on the Java Card 3.0.2 emulator. We have tested it on several applets: the tests have shown that the prototype requires only a small percentage of smart card memory to run.

ACKNOWLEDGEMENT

This work is partially supported by the EU under grant EU-FP7-FET-IP-SecureChange.

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] 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.

[3] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of CCS 2009*, pages 235–245.

[4] O. Gadyatskaya, F. Massacci, and A. Philippov. Implementation requirements and specification of the Policy Checker component. Report DISI-11-455, University of Trento, 2011.

[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.

[6] P. Girard. Which security policy for multiplication smart cards? In *USENIX Workshop on Smartcard Technology*, 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, Version 2.2. Specification 2.2, 2006.

[9] International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). ISO 7816: Smart Card Standard, 1995. Part 4: Interindustry Commands for Interchange.

[10] Sun Microsystems. Runtime environment specification. Java Card^{TM} platform, Connected edition. Specification 3.0, 2008.

[11] I. Narasamdya and M. Périn. Certification of smart-card applications in Common Criteria. In *SAC '09*, pages 601–608. ACM, 2009.

[12] 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.

[13] 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.

[14] Sun Microsystems, Inc. *Development Kit User's Guide, Java Card Platform, Version 3.0.2 Connected Edition*, December 2009.