Can We Support Applications' Evolution in Multi-application Smart Cards by Security-by-Contract?

Nicola Dragoni[1], Olga Gadyatskaya[2], and Fabio Massacci[2]

[1] DTU Informatics, Technical University of Denmark, Denmark
ndra@imm.dtu.dk
[2] DISI, University of Trento, Italy
{gadyatskaya,massacci}@disi.unitn.it

Abstract. Java card technology have progressed at the point of running web servers and web clients on a smart card. Yet concrete deployment of multi-applications smart cards have remained extremely rare because 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. Yet the current security models and techniques do not support this type of evolution. We propose in this paper to apply the notion of *security-by-contract* (S×C), that is a specification of the security behavior of an application that must be compliant with the security policy of the hosting platform. This compliance can be checked at load time and in this way avoid the need for costly run-time monitoring. We show how the S×C approach can be used to prevent illegal information exchange among several applications on a single smart card platform, and to deal with dynamic changes in both contracts and platform policy.

Keywords: Security-by-Contract, multi-application smart card, illegal information exchange, security policy enforcement.

1 Introduction

Multi-application smart cards aim at making it possible to run several applications from different providers on the same smart card and to dynamically load and remove applications during the active life of the card. With the advent of new web enabled cards the industry potential is huge. However, despite of the large number of research papers on the topics there are few to none real-life deployments.

One reason is the lack of solutions to an old problem [14]: the control of interactions among applications. While many techniques can be used to check information flow (e.g. [2,12,8]) if we know and install all applications at once before distributing the card to the public, the natural business model is the asynchronous loading and updating of applications by different parties. Hence we need a method to check interactions at load- or run-time.

P. Samarati et al. (Eds.): WISTP 2010, LNCS 6033, pp. 221–228, 2010.

Applications run in dedicated *security domains* [10]. The name is evocative of a separate space (such as in a virtual machine) but in reality a domain just supports security services such as key handling, encryption, decryption, digital signature generation and verification for their providers' (Card Issuer, Application Provider or Controlling Authority) applications, and a number of functions to download and make an application executable.

The control of the communications between the applications and the applications and the platform rests on the Java Run-time Environment (JRE) [11]. The basic model of the GP security is the same behind the confinement of standard Java applets [7]: to deal with the untrusted code we can exploit the mechanism of permissions to control execution of potentially dangerous or costly functionality, such as starting various types of connections. However permissions do not solve the issue of interactions. On the card we can also have a firewall security mechanism that isolates each applet from the other applets within of its own space (a "context" in the Java Card jargon). The result is that the internal operations of an applet have no effect on other applets embedded on the card. Still, applications can interact in this environment by explicitly implementing sharing methods callable via an API, application *service* in Java Card 3.0 specification [11] or Global Services in the GP specs.

In the GP registry every application has an entry, which includes its ID (AID) and Global Services recorded for this application (if any). But GP does not solve the problems of illegal information exchange even for the applications from different security domains and all inter-application interactions are pushed to lower level – JRE, or even hardware. If application A knows shareable interface of application B, then it may use it for its own purposes, and there is no means for B or B security domain owner to prevent it, unless special controls are hacked into the Java firewall. However this prevents completely the asynchronous download or update of different applications. Moreover, there is no uniform policy language to express security policies for GP and it is even difficult to show that the security policies of two versions of a smart card by the same vendor are inter-operable. There are business solutions for multi-application smart cards on top of GP and Java Card from Venyon Oy, Gemalto and companies alike developed for banking, transport and mobile operators. But typical solution from such companies is only responsible for handling loading of card customer applications, security domain key handling and management and removal of applications [1], thus such a solution is only an improvement of GP, but it is not dealing with certification of new applications on the card, checks of compliance of new applications with the initial card security policy and checks if the removal of some application is even possible and will not break the work of others remaining on the card.

What remains out of reach is a secure way to deploy new applications on the multi-application smart card once it is in the field. A costly manual review is necessary. Owners of different trust domains would like to make sure their applications cannot be accessed by new applications added after theirs. Actually, currently 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 after that changes and they do not need to obtain a new certificate. Otherwise no smart card customers (banks, governments, airline companies, etc) will accept to issue such cards for their needs. The natural consequence is that there are essentially no open multi-application smart cards, though both the GP and Java Card specifications support them.

In the next section we present the S×C framework for open multi-application smart card security and show how it can solve the problem of Common Criteria compliance certification, introducing also the hierarchy of application models, and discuss the lowest level with its limitations (§3). Finally we summarize related works and conclude (§4).

2 Security-by-Contract for Smart Cards

In current setting, formal proof of card Common Criteria compliance includes a proof that all applications on the card satisfy the security policy of the card, provided together by all security domain owners and controlling authorities. So, if $A_1, \ldots, A_n$ are the applications on the smart card and the smart card policy $Policy = \bigcup_{i=1}^{n} Spec_{A_i}$, where $Spec_{A_i}$ is a policy provided by the owner of application A_i, then the compliance of applications to the policy can be written as $\bigcup_{i=1}^{n} A_i$ *satisfies* $Policy$. Accordingly, after new application B arrives on the platform the smart card vendor has to prove that

$$\bigcup_{i=1}^{n} A_i \cup B \text{ satisfies } Policy \cup Spec_B.$$

For this Common Criteria compliance problem, we want to show that *Security-by-Contract* methodology (S×C) for smart cards we propose in this paper can help smart card vendors with this issue. In the S×C setting each application has a contract, which describes its security-relevant behavior. Thus application B will arrive on the platform with its contract $Contract_B$. Then the S×C framework will prove (check) the compliance of the application B and its contract, and then check that $\bigcup_{i=1}^{n} Contract_{A_i} \cup Contract_B$ *satisfies* $Policy \cup Spec_B$. Since the contract is a formal model it is easier to show its compliance with another formal model of the same type ($Policy$), rather then try to show the compliance of applications and the policy. This is why we claim that new smart cards need the notion of Security-by-Contract.

In this paper we focus on contracts and policies describing the interactions among the different applications running on a single smart card platform. Intuitively, we say that an application A interacts with an application B if there is some information exchange between these applications. In this dynamic setting, we are primary interested in addressing two specific security challenges, namely *preventing illegal information exchange* among applications and dealing with *dynamic evolution of smart card platform*. In this paper due to limitations of space

we will consider only arrival of new application on the platform as a dynamic change. This type of change is the most important, since it allows openness of the platform. The other types of evolution, namely update or removal of an existing application and update of platform security policy will be considered in the future works. Thus the problem we want to solve:

– **Problem** P_1: new application should not interact with forbidden applications already on the smart card.

Sekar et al. [16] have proposed the notion of Model Carrying Code (MCC), that requires the code producer to establish a model regarding the safety of mobile code which captures its *security-relevant behavior*. The code consumers check their policies against the model associated with untrusted code to determine if this code will violate their policy. The major limitation was that MCC had not fully developed the whole lifecycle and had limited itself to finite state automata which are too simple to describe realistic policies. Even a simple, basic policy such as "Only access url starting with http" could not be addressed. The *Security-by-Contract (*$\mathsf{S}\times\mathsf{C}$*)* framework that we have developed for mobile code [4,3] builds upon the MCC to cope with more practical scenarios.

At *load time*, the target platform checks that the evidence is correct. Such evidence can be a trusted signature as in standard mobile applications [18], but now the digital signature does not just certify the origin of the code, but also bind together the code with a contract with the main goal to provide a semantics for digital signatures. An alternative evidence can be a proof that the code satisfies the contract (and then one can use PCC techniques to check it [13] or the techniques used by Ghindici et al on smart-cards [5]). Once we have evidence that the contract is trustworthy, the platform checks that the contract is compliant with the policy that our platform wants to enforce. If it is, then the application can be run without further ado. At run-time the JRE firewall can simply check that only the declared API in the contract can be called. The matching steps guarantee that the resulting interactions would be correct.

Security-by-Contract methodology proposed in [3,4] was created for mobile applications and did not take into account interactions among applications and illegal information flow problems. For the mobile platform there is no expectation that the platform owner will certify the absence of illegal information exchange between different applications. But multi-application smart card has different stakeholders, namely the owner of the platform, the owners of the applications and the user of the card. Now we enhance the $\mathsf{S}\times\mathsf{C}$ approach with information flow to deal with problems of sensitive data exchange of smart card applications. Another improvement of the $\mathsf{S}\times\mathsf{C}$ approach we suggest in this paper is enriching the contracts with "wishes". Contract can contain not only exact behavior of the application, but also what behaviors application owner consider as possible. There might be a desire to interact with some application X not yet present on a platform or a wish to disallow some application Y to receive sensitive data through indirect interactions with some application Z.

Another challenge we have to address is to find an appropriate language for specifying contracts (policies) describing possible (allowed) information exchange

among applications. To deal with computational limitations we propose a hierarchy of contracts/policies models for GP-based smart cards. The rationale is that each level of the hierarchy can be used to specify contracts and policies, but with different computational efforts and expressivity limitations.

- **L0: Application as Services.** This level models applications as lists of required and available services. Essentially it is the current set-up of the GP.
- **L1: Allowed Control Flow.** This level provides a call graph $G_1(A)$ of the application, where vertices are the states of the application and edges represent invocations of different services. Then we can do a bit of history-based access control and more fine grained information exchange control.
- **L2: Allowed and Desired Control Flow.** This level adds to the previous one the notions of correct and error states. It can be necessary if we want to test that removal of an application (or a change in the policy) does not break other applications.
- **L3: Full Information Flow.** This level extends the previous one considering also the information flow among variables.

3 Contract and Policy as List of Services

At this level of abstraction we represent both contracts and policies by means of the services that application provides (to other applications) and of the services that application requires (from other applications). The list of available services can be comprised from the services with inherited Shareable Interface, the available services are called *global services* in the GP jargon. The list of required services can be made from the list of the OPEN *getService* calls [10].

Definition 1 (A rule). *A security policy rule* Rule *is represented by the following four fields:*

Application	*Name and security domain (company or package) of the application, in the form* name@domain.
Shares(A)	*Set of applications with which the application* A *may interact. We use the notation* A@D *to denote a specific application* A *in the domain* D, ∗@D *to denote all the applications in the domain* D, *and* ∗ *to denote that* A *shares with any application of any domain.*
Provides(A)	*List of services provided by the application* A. *For the sake of simplicity this list contains only the names of the services, that is a service* s.A@D *is simply denoted with* s.
Requires(A)	*List of services required by the application* A.

Given an application A belonging to a security domain D (company or package) we denote a service s of A with s.A@D.

Definition 2 (Contract). *A contract of an application is a policy rule.*

Definition 3 (Security Policy). *A security policy* $Policy$ *of a smart card is a non-ordered set of different policy rules:*

$$Policy = \{Rule_1, \ldots, Rule_n\}, \text{where } Rule_i \neq Rule_j, i \neq j.$$

A contract of an application A matches a platform policy if there is no illegal information exchange between the application A and applications already on the card. We need therefore to define what is an *illegal information exchange* at this level of abstraction. We start introducing the notion of *direct communication* between applications. The idea is that A is able to directly communicate with B if B provides some services that A intends to use.

Definition 4 (Direct Communication). *Application A directly communicates with an application B,* $A \rightarrow B$*, if* $Requires(A) \cap Provides(B) \neq \emptyset$*.*

The notion of illegal information exchange is built on top of the notion of direct communication. The intuition is that there is a potential illegal information exchange if an application A directly communicates with another application B that may directly communicate to some other applications *forbidden* for A, i.e., with some applications that A is not allowed to directly communicate. We need therefore to capture the notions of *allowed* and *forbidden information sharing* between two communicating applications.

Definition 5 (Allowed Information Sharing). *Let A be an application directly communicating with B. We say that there is an allowed information sharing between A and B, denoted* $A \overrightarrow{ok} B$*, if* $Shares(A) = Shares(B)$*.*

Definition 6 (Forbidden Information Sharing). *Let A be an application directly communicating with B. We say that there is a forbidden information sharing between A and B, denoted* $A \overrightarrow{no} B$*, if* $Shares(A) \neq Shares(B)$*.*

Definition 7 (Illegal Information Exchange). *Given a contract* $Contract_A$ *of an application A and a platform policy Policy there is an illegal information exchange, denoted* $Contract_A \leftrightarrow Policy$*, if there exist an application B described in Policy such that at least one of the following conditions is true:*

- $(A \rightarrow B) \wedge (A \overrightarrow{no} B)$
- $(B \rightarrow A) \wedge (B \overrightarrow{no} A)$

The notion of illegal information exchange is sufficient to address the problem P_1, the intuition is that when new application is loaded, its contract is checked against the platform policy in order to detect possible illegal information exchanges. We have an algorithm to perform this check, which is not presented in this paper due to lack of space.

4 Related Works and Conclusions

In [5] Ghindici et al. proposed a domain specific language for security policies describing the allowed information flow inside the card. Each application is

certified at loading time, having a information flow signature assigned to each method. Information flow in this framework is represented as relations between two method variables with annotations about the type of flow, for example, from secret to secret through a direct assignment. However these policies are too simple to capture the full scope of interesting policies.

In [9,17] Huisman et al. presented 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. To check that composition of two applets respects the behavioral safety property existing model checking techniques are used.

In [6] Girard suggested to associate security levels to application attributes and methods using traditional Bell/La Padula model and the security policies in this model define authorized flows between levels. This approach was further investigated in [2] by Bieber et al. and the technique based on model checking for verification of actual information flows was presented there. The same approach was also used by Schellhorn et al. in [15] for their formal security model for operating systems of multi-application smart cards.

In this paper we have proposed the *security-by-contract* ($\mathsf{S}\times\mathsf{C}$) framework as a possible security model for multi-application smart cards. The $\mathsf{S}\times\mathsf{C}$ approach improves the current literature by addressing problems related to the dynamic evolution of both applications and policies. Moreover, it is based on a hierarchy of models that allow to have benefits in terms of computational complexity or language expressivity. In particular, the first level of the hierarchy requires algorithms not more complicated than the usual smart card applications, while the mentioned approaches are usually based on complicated logic and model checking, all needing off-card activities. We have shown how the $\mathsf{S}\times\mathsf{C}$ approach can be used to check the absence of illegal information flow during loading of new application at the first level of a proposed hierarchy.

The key limitation of level L0 is that it does not capture the *actual* information exchange among the applications, but only the *possible* information exchange. In other words, we are not able to specify the concurrent behavior of an application, distinguishing for instance between the services that an application *might* need from the services that an application *strongly requires*. Another limitation of this level is the possibility of indirect communication between applications in current framework.

Future work will be focused on two main issues: (1) developing all the levels of abstraction (starting from the limitations of Level 0) and (2) extending the security relevant actions of both contracts and policies in order to consider also rules restricting the use of resource APIs (connection to the Internet, EEPROM memory, ...). Further development of level L0 includes capturing indirect communications in the contracts and enhancing contracts with the notion of "wish". Thus application providers will be able to declare in the contracts explicitly the exact behavior of the application extracted from the code and the desirable behavior of other applications on the platform concerning direct and indirect communications with their own application.

References

1. Venyon banking services, `http://www.venyon.com/banking`
2. Bieber, P., Cazin, J., Wiels, V., Zanon, G., Girard, P., Lanet, J.-L.: Checking secure interactions of smart card applets: Extended version. J. of Comp. Sec. 10(4), 369–398 (2002)
3. Desmet, L., Joosen, W., Massacci, F., Philippaerts, P., Piessens, F., Siahaan, I., Vanoverberghe, D.: Security-by-Contract on the .NET platform. Information Security Tech. Rep. 13(1), 25–32 (2008)
4. Dragoni, N., Massacci, F., Naliuka, K., Siahaan, I.: Security-by-Contract: towards a semantics for digital signatures on mobile code. In: López, J., Samarati, P., Ferrer, J.L. (eds.) EuroPKI 2007. LNCS, vol. 4582, pp. 297–312. Springer, Heidelberg (2007)
5. Ghindici, D., Simplot-Ryl, I.: On practical information flow policies for java-enabled multiapplication smart cards. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 32–47. Springer, Heidelberg (2008)
6. Girard, P.: Which security policy for multiplication smart cards? In: USENIX Workshop on Smartcard Technology. USENIX Association (1999)
7. Gong, L., Ellison, G., Dageforde, M.: Inside Java 2 platform security: architecture, API design, and implementation. Addison-Wesley, Reading (2003)
8. Hubbers, E., Oostdijk, M., Poll, E.: From finite state machines to provably correct java card applets. In: SEC 2003 (2003)
9. Huisman, M., Gurov, D., Sprenger, C., Chugunov, G.: Checking absence of illicit applet interactions: a case study. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 84–98. Springer, Heidelberg (2004)
10. GlobalPlatform Inc. GlobalPlatform Card Specification, Version 2.2. Specification 2.2, GlobalPlatform Inc. (2006)
11. Sun Microsystems. Runtime Environment Specification. Java Card™ Platform, Version 3.0, Connected edition. Specification 3.0, Sun Microsystems (2008)
12. Mostowski, W., Poll, E.: Malicious code on java card smart cards: attacks and countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008)
13. Necula, G.C.: Proof-carrying code. In: Proc. of the 24th ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang, pp. 106–119. ACM Press, New York (1997)
14. Sabelfeld, A., Myers, A.C.: Language-based information flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
15. Schellhorn, G., Reif, W., Schairer, A., Karger, P., Austel, V., Toll, D.: Verification of a formal security model for multiapplicative smart cards. In: Cuppens, F., Deswarte, Y., Gollmann, D., Waidner, M. (eds.) ESORICS 2000. LNCS, vol. 1895, Springer, Heidelberg (2000)
16. Sekar, R., Venkatakrishnan, V.N., Basu, S., Bhatkar, S., DuVarney, D.C.: Model-carrying code: a practical approach for safe execution of untrusted applications. In: Proc. of the 19th ACM Symp. on Operating Syst. Princ., pp. 15–28. ACM Press, New York (2003)
17. Sprenger, C., Gurov, D., Huisman, M.: Simulation logic, applets and compositional verification. Technical Report RR-4890, INRIA, 07 (2003)
18. Yee, B.S.: A sanctuary for mobile agents. In: Vitek, J., Jensen, C.D. (eds.) Secure Internet Programming. LNCS, vol. 1603, pp. 261–273. Springer, Heidelberg (1999)