Detecting Illegal Information Flow Using Abstract Interpretation and Model Checking

[Published in *Gemplus Developer Conference*, Montpellier, France, June 20–21, 2000.]

Pierre Bieber[1], Jacques Cazin[1], Abdellah El-Marouani[2], Pierre Girard[2], Jean-Louis Lanet[2], Rodolphe Muller[2], Virginie Wiels[1], and Guy Zanon[1]

[1] ONERA-CERT/DTIM
BP 4025, 2 avenue E. Belin, F-31055 Toulouse Cedex 4, France
{bieber, cazin, wiels, zanon}@cert.fr
[2] GEMPLUS
Avenue du pic de Bertagne, 13881 Gemenos cedex, France
{pierre.girard, jean-louis.lanet}@gemplus.com

Abstract. This paper describes the status of a joint project between Gemplus and ONERA. It presents an approach enabling a smart card issuer to verify that a new applet securely interacts with already loaded applets. A security policy has been defined that associates levels to applet attributes and methods and defines authorized flows between levels. We propose a technique based on model checking to verify that actual information flows between applets are authorized. In this paper, we focus on the development of the prototype of the analyzer.

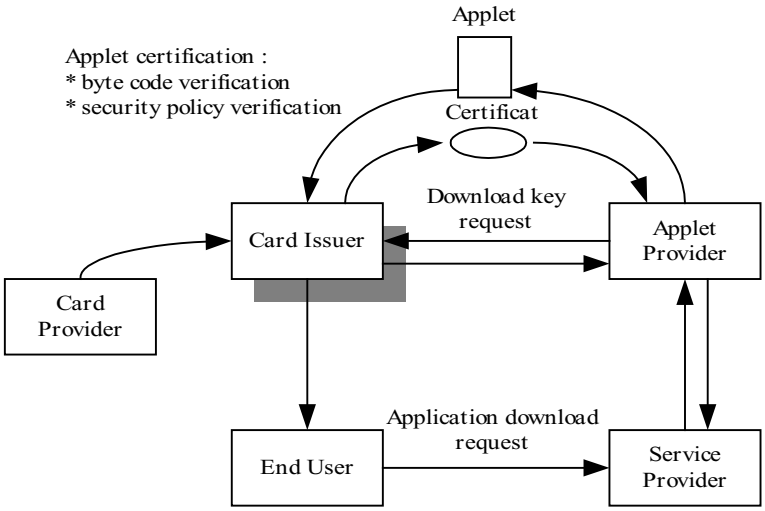
1 Introduction

Security is always a big concern for smart cards but it is all the more important with multi-application smart cards and post issuance code downloading. Opposed to mono-applicative smart cards where Operating System and application were mixed, multi-application smart cards have drawn a clear border between the operating system, the virtual machine and the applicative code. In this context, it is necessary to distinguish the security of the card (hardware, operating system and virtual machine) from the security of the application. The card issuer is responsible for the card security and the application provider is responsible for the applet security, which relies necessarily on the card security.

The physical security is obtained by the smart card media and its tamper resistance. The security properties that the OS guarantees are the quality of the cryptographic mechanisms (which should be leakage resistant, *i.e.*, resistant against side channel attacks such Differential Power Analysis), the correctness of memory and I/O management.

A Java Card virtual machine relies on the type safety of the Java language to guaranty the innocuousness of an applet with respect to the OS, the virtual

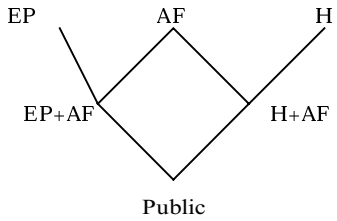
machine, and other applets. However, this is ensured by a byte-code verifier that is not on card, and extra mechanisms have been added. A secure loader checks before loading an applet that it has been signed (and therefore verified) by an authorised entity (namely the card issuer). The following scheme shows the role of the different participants. The card issuer or a Trusted Third Party (TTP) is responsible in delivering the certificate indicating the correctness of the verified applet. This verification concerns the type correctness and the card issuer security policy correctness [5].



Applet providers and end users cannot control that their information flow requirements are enforced inside the card because they do not manage it. Our goal is to provide techniques and tools enabling the card issuer to verify that new applets respect existing security properties defined as authorized information flows. If the applet provider wants to load a new applet on a card, it provides to the card issuer or to the TTP the byte code for this applet. The card issuer has a security policy for the card and security properties that must be satisfied. This security policy should enforce the confidentiality while taking into account data exchange between applets.

Actually, most of multi-application smart cards, in order to build cooperative schemes and to optimize memory usage, allow data and service sharing (*i.e.*, objects sharing) between applications. Beyond this point, there is a need for a card-wide security policy concerning all applications. A small example should clarify this point. When an application provider A decides to share (or more probably to sell) some data with an application provider B, it asks for guarantees that B is not able to resell those data or to make them available world-wide. For example, in Java, if one decides to store the exchanged information in a public static variable, this datum becomes readable by every one. This point is important and difficult to verify using traditional means.

A mandatory security policy is necessary to solve the problem of re-sharing shared objects as mentioned above [4]. The security policy should model the information flows between the applications that, themselves, reflect the trust relationships between the participants of the applicative scheme. The best candidate for such a mandatory policy appears to be a multilevel policy. This security model uses a set of security levels ordered in a complete lattice. With this security model, each applet is assigned a security level and each shared data has a specific security level. This lattice represents all the legal flows. For example, consider that the configuration to be checked includes an Air France loyalty applet (level AF) an Hertz loyalty applet (level H) and an electronic purse (level EP). When buying a flight ticket with the purse, you add miles to your loyalty program. Shared information from Air France and the electronic purse (level EP+AF) may be received by Air France applet and electronic purse applet. The same operation can be done when renting an Hertz car. This is represented by the following lattice.



To model that applets may only communicate through shared interfaces, direct flows between AF, H and EP are forbidden.

2 Applet Analysis

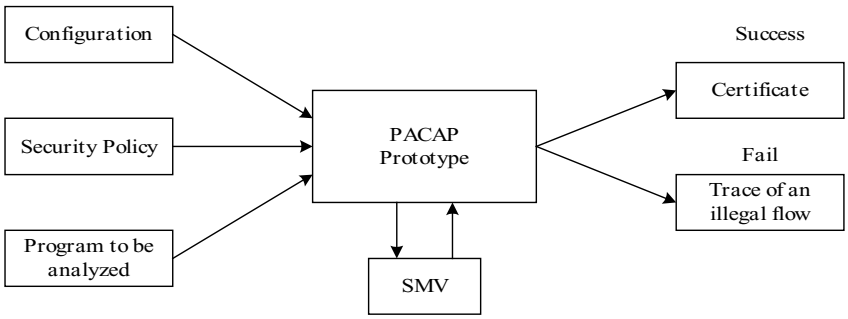
The PACAP project[1] aims at checking the data flows between objects on the card by static analysis prior to applets downloading, for a given configuration. Our tool verifies automatically if a set of applications implements correctly a given security policy. An application is composed of a finite number of interacting applets. Each applet contains several methods. For efficiency reasons, we want to limit the number of applets and methods analyzed when a new applet is downloaded or when the security policy is modified.

Our method to verify the security property on the application byte code is based on three elements:

- abstraction: we abstract all values of variables by computed levels;
- sufficient condition: we verify an invariant that is a sufficient condition of the security property;
- model checking: we verify this invariant by model checking.

[1] The PACAP project is partially founded by MENRT contract n°98B0252

The tool needs as input, a representation of the lattice and the configuration (*i.e.,* the set of applet). With this information the tool transforms the byte code into a formal semantics, adds the relevant invariants and performs the verification of the invariants.



The verification is done by an off the shelf model checker: SMV from Cadence Lab. If the verification failed (*i.e.*, an illegal flow has been discovered) a trace is provided in order to remove or to extract the proof of the illegal flow. In the case of a successful verification, a certificate can be provided as shown in the previous picture.

The transformation into a formal model is fully automatic. The tool computes all the call graphs of the application and it generates one SMV model per graph. Two kinds of methods will especially interest us because they are the basis of applet interactions: interface methods that can be called from other applets and methods invoking external methods of other applets. We have decided to analyze subsets of the call graph that include such interaction methods. Furthermore an analyzed subset only contains methods that belong to the same applet. For a given program, we consider as inputs results from external invocations and read attributes. We take as outputs parameters of external invocations and modified attributes. We thus associate security levels with applet attributes and with method invocations between applets.

The tool performs several preliminary operations before constructing the formal model. For example when abstracting the value of the variables by their corresponding levels, we loose information for subroutine treatment. When a Java `try catch finally` structure is used, it is compiled in a call to a subroutine with the return address stored in a local variable. Of course, with the abstraction, we cannot preserve this control flow. We have to inline the subroutines.

Then, the tool builds the formal models by translating the semantics of each byte code in the SMV language, preserving only the relevant information. When a call to a method interface or an access to an object field is encountered, we add the relevant invariant in the model. We consider a set of methods at a time. Our approach uses the modularity capabilities of SMV (the model checker tool) by building an SMV module for each method.

In the last phase, the tool builds a main module that contains instances of each of these method abstraction modules and describes the interconnections

between these modules. The interconnections consist in building the call stack, the parameter passing and exception treatments. We obtain a collection of formal models that can be verified using the SMV tool.

Of course, when a formulae can not be satisfied, the SMV tool provides a trace where all the different interactions between the variables are recorded. We have to extract from this trace the relevant information and to make the correlation with the program. Due to the abstraction, it is possible that detected illegal flow are not possible in the real world.

3 State of the Art

A lot of work has been going on about the analysis of security properties of Java byte code. The major part of this work is concerned with properties verified by SUN byte code verifier like correct typing, no stack overflow, etc. Among this work, two kinds of approaches can be distinguished depending on the technique used for the verification. Most of the approaches are based on static analysis techniques, particularly type systems [2, 16]. One approach has used model checking (with SMV) to specify the correct typing of Java byte code [14]. Recently, several researchers investigated the static analysis of information flow properties quite similar to our secure dependency property but, to our knowledge, none of them applied their work on Java byte-code. Girard et alii [7] defined a type system to check that a program written in a subset of the C language does not transfer high level information in low level variables. In [3], the typing relation was related to the secure dependency property. Volpano and Smith [15] proposed a type system with a similar objective for a language that includes threads. They relate their typing relation to the non-interference property. The secure dependency property was compared with non-interference in [1]. Myers and Liskov [13] propose a technique to analyze information flows of imperative programs annotated with labels that aggregate the sensitivity levels associated with various information providers. One of the interesting feature is the declassify operation that allows providers to modify labels. They propose a linear algorithm to verify that labels satisfy all the constraints. A few pieces of work deal with other kind of security properties. In [8] the authors propose an automatic method for verifying that an implementation using local security checks satisfies a global security property. However, their approach is limited to control flow properties such as Java Virtual Machine stack inspection. The work described in [10] focuses on integrity property by controlling exclusively write operations to locations of references of sensitive objects such as files or network connections.

4 Conclusion

In this paper, we have presented an approach for the certification of applets that have to be loaded on a Java card. The security checks we propose are complementary to the security functions already present on the card. The applet firewall controls the interaction between two applets, while our analysis has a

more global view and is able to detect illicit information flow between several applets.

To verify our approach, we have developed a real application containing a purse and two loyalty programs with several data exchanges. The complete analysis of the PACAP application generates 35 different SMV models involving globally about 100 methods.

To check all the possible interactions on the example of the electronic purse, we have to verify all the different models. Since the prototype is not yet finished, we cannot provide a lot of results. For example, the complete (non simplified) version of the logfull interaction contains 5 methods, the verification of the eight properties takes 7.04s user time, 0.16s system time on an ultra 10 sun station.

Automation of the production of models is thus mandatory for the approach to be practicable. Such an automation is relatively straightforward providing that preliminary treatments are made to prepare the model construction, such as construction of the call graph, method name resolution, etc. However, a complete automation is hardly possible: an interaction with the user will be needed for the definition of security policy, level assignments and the trace analysis.

As a conclusion, it is clear that the Java Card is a powerful framework to develop and to deploy applications. But the security mechanisms are not sufficient to prevent some kind of attacks of the system as presented here. We believe that abstract interpretation and verification through a model checker is an efficient means to guarantee that a given security policy is correctly implemented by applications.

References

1. P. Bieber and F. Cuppens. *A Logical View of Secure Dependencies.* Journal of Computer Security, 1(1):99–129, 1992.
2. Stephen N. Freund and John C. Mitchell. *A type system for object initialization in the Java byte code language.* In Proceedings of OOPSLA 98, 1998.
3. P. Girard. *Formalisation et mise en oeuvre d'une analyse statique de code en vue de la vérification d'applications sécurisées.* PhD thesis, ENSAE, 1996.
4. Pierre Girard. *Which security policy for multiapplication smart cards?* In USENIX workshop on smartcard technology, 1999.
5. Pierre Girard and Jean-Louis Lanet. *Java Card or How to Cope with the new Security Issues raised by Open Cards?* In GDC 99, Paris, June 1999.
6. C. O'Halloran, J. Cazin, P. Girard, and C. T. Sennett. *Formal Validation of Software for Secure Systems.* In Anglo-french workshop on formal methods, modelling and simulation for system engineering, 1995.
7. T. Jensen, D. Le Metayer, and T. Thorn. *Verification of control flow based security policies.* In Proceedings of the 20th IEEE Security and Privacy Symposium, 1999.
8. X. Leroy and F. Rouaix. *Security properties of typed applets.* In Proceedings of POPL, 1998.
9. A.C. Myers and B. Liskov. *A decentralized model for information flow control.* In Proceedings of the 16th ACM symposium on operating systems principles, 1997.
10. Joachim Posegga and Harald Vogt. *Off line verification for Java byte code using a model checker.* In Proceedings of ESORICS, number 1485 in LNCS. Springer, 1998.

11. G. Smith and D.M. Volpano. *Secure information flow in a multi-threaded imperative language*. In Proceedings of POPL, 1998.
12. Raymie Stata and Martin Abadi. *A type system for Java byte code subroutines*. In Proc. 25th Symposium on Principles of Programming Languages, 1998.