The PACAP Prototype: a Tool for Detecting Java Card Illegal Flow

[Published in I. Attali and T. Jensen, Eds., *Java on Smart Cards: Programming and Security*, vol. 2041 of *Lecture Notes in Computer Science*, pp. 25-37, Springer-Verlag, 2001.]

Pierre Bieber[1], Jacques Cazin[1], Abdellah El-Marouani[2], Pierre Girard[2], Jean-Louis Lanet[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 Gémenos Cedex, France
{abdellah.el-marouani, pierre.girard, jean-louis.lanet}@gemplus.com

Abstract. This paper presents some practical issues of a joint project between Gemplus and ONERA. In this approach, a smart card issuer can 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 and we will present the first results.

1 Illegal Flow in Multi-applicative Smart Cards

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 (OS) and application were mixed, multi-application smart cards have drawn a clear border between the OS, 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 as 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 guarantee the innocuousness of an applet with respect to the OS, the virtual machine, and other applets. However, this is ensured by an off-card byte-code verifier, and extra mechanisms that 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). Figure 1 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].

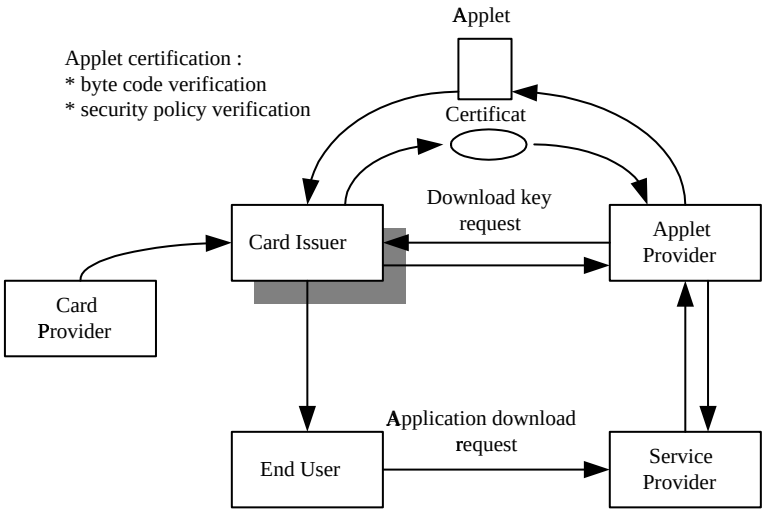

Fig. 1. Certification scheme

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.

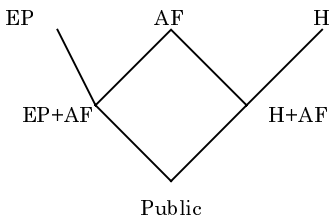

Fig. 2. The security policy lattice

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. We verify information flow between applets that share data through shareable interfaces. The sharing interface is the means to transfer information from an applet to another one in Java Card. The calls to a sharing interface can be

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

issued from an applicative command (process APDU) or an external call. We check in all the interactions if the level associated to all the variables (system and application) does not exceed the allowed sharing level.

Our tool verifies automatically if a set of applications correctly implements 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.

The abstraction mechanism and the invariant definition have been described in [13] and [14]. The tool needs as input, a representation of the lattice and the configuration (*i.e.*, the set of applets).

With this information the tool transforms the byte code into a formal semantics, adds the relevant invariants and performs the verification of the invariants.

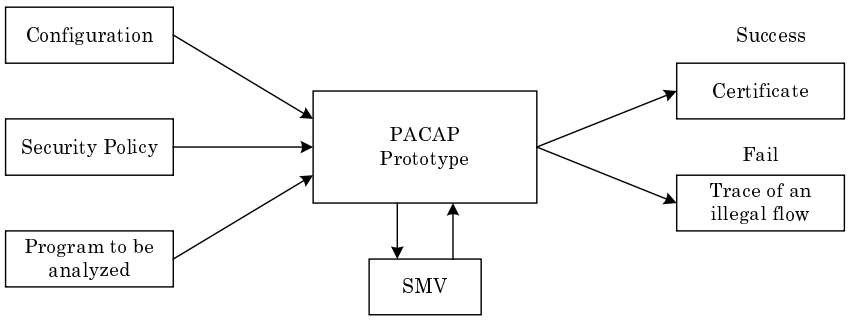

Fig. 3. Architecture of the prototype

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 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 generate only call graph that include an interface method, either as the root or as a leaf. The call graph that does not include such a method is not relevant here. Furthermore the call graph 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.

3 The PACAP Prototype

The first step was to specify the translation rules between the Java byte code and the SMV language. In order to ease the final transformation several treatments must be done on the byte code: for example, subroutine elimination and end of conditional branch computation.

The call graph

We have to build an SMV model for each call graph that includes an access to a method of a shareable interface. The following figure shows the call graph of the `logfull` method. The purse calls this method through the Loyalty shareable interface. There are several possibilities of illegal information flow during this call. The theory will impose to verify the level of the passed and returned parameters of `logfull`, `IsThereTransaction`, `getTransaction`, `getIdlenght`, `getType` and `getReste` method call.

The resulting call graph is a tree: for each `invokeSpecial` and `invokeVirtual` byte code we need to develop the sub-tree while the `invokeStatic` and `invokeInterface` are leaves of the tree.

Subroutine elimination

Subroutines are a means to compile in an efficient way the Java try-finally blocks. Without this mechanism, it is necessary to duplicate the code of the exception treatment. In a subroutine call, the first action is to store the return address in a local variable. Unfortunately with our abstraction we loose this information. We manipulate only levels and never the contents of the variables. We have to duplicate all the code of the subroutine even for nested subroutines. We give hereafter an example of such an elimination. Of course we have to pay a particular attention to the exception table and all the conditional jumps.

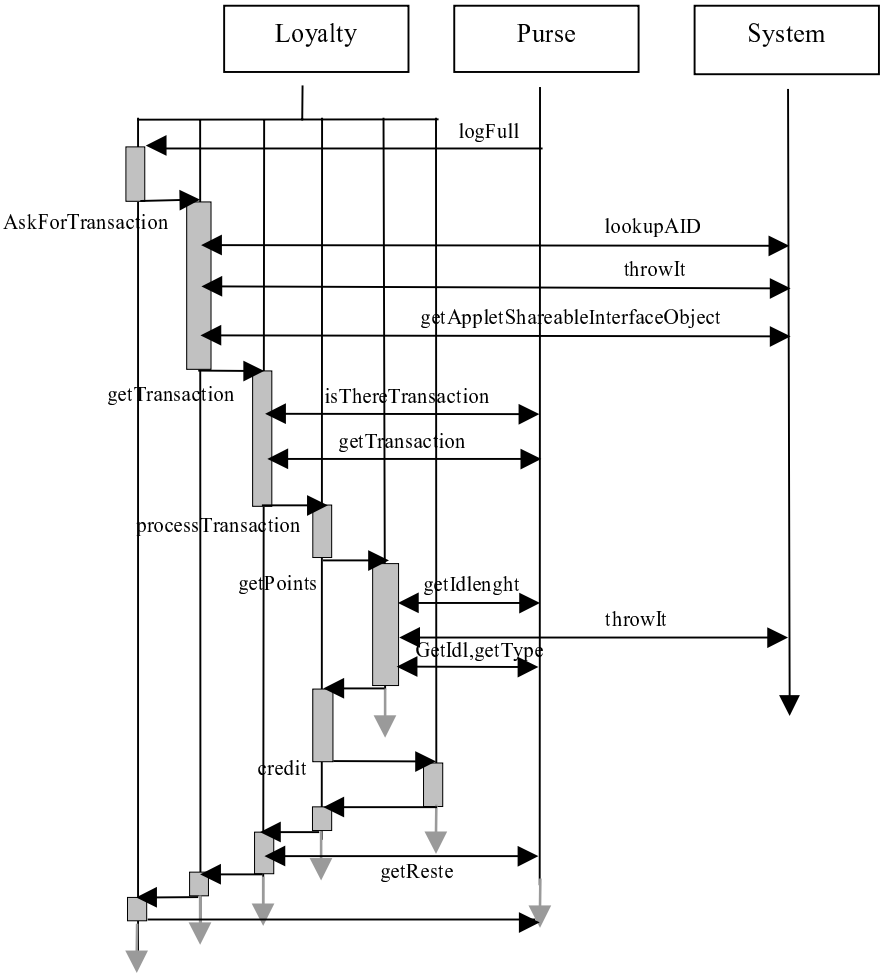

Fig. 4. `Logfull` call graph

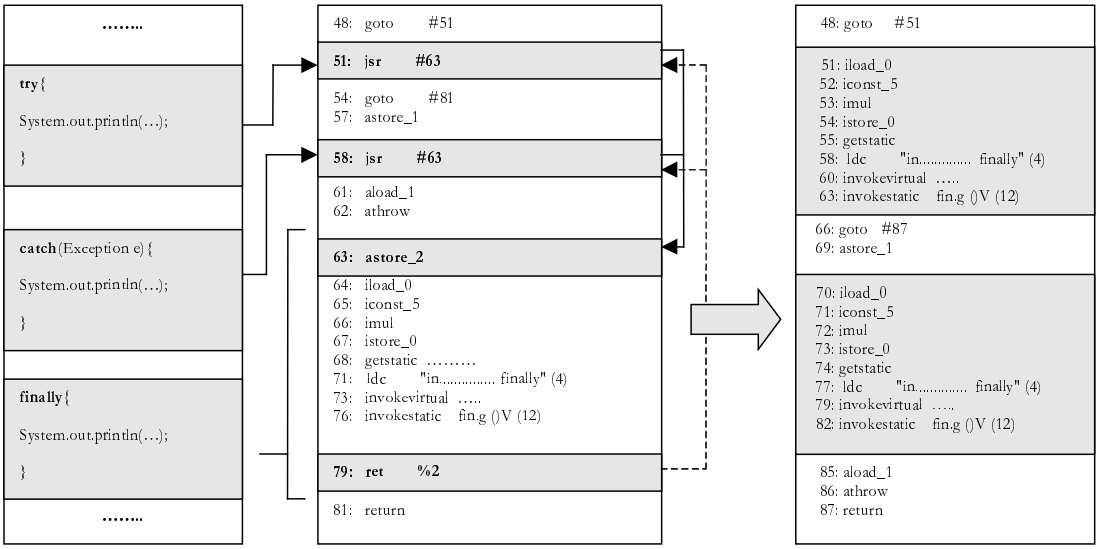

Fig. 5. Subroutine elimination

Implicit dependency

Dependency can be explicit or implicit. For example, if in a branch statement, one raises systematically an exception, it is possible when catching the exception to infer the value of the conditional. This can be a means to transfer illegally information. For this purpose we have to adjust the level of the state variables to the level of the conditional. When information can be infered (when both branch join) we have to release the previous level of the state variables. But computing the `endif` (the join point) can be very difficult. According to [15], it is possible to find structural 2-way conditional and to determine easily the `endif`. A problem arises with unstructured conditionals having abnormal entry or abnormal exit. Abnormal exit occurs when a break or a continue is inserted in one path. In this case we choose a conservative solution by never reducing the level. Of course this can lead to non existing illegal flow detection. When compound conditions (at the Java level) are used this lead at the byte code level to an abnormal entry. In this case, we have to duplicate the code and store the value of the system variables into a stack. For example, in the following Java program it seems obvious that the system variables must be restored before t2. The compound condition (a or not b) is made of two conditionals that are overlapped. It is more difficult at the byte code level to define exactly the end of the compound condition. The solution proposed by [15] uses code duplication and provides only one join point where the system variables have to be restored.

```
public static void
main(String args[])
       {
       boolean a= true;
       boolean b = true;
       int var1= 0;
       int var2= 0;
       int var3 =0;
       if (a||!b)
        var1=1;  //t1
       var2=2;   //t2
       var3=3;   //t3
       return;
```

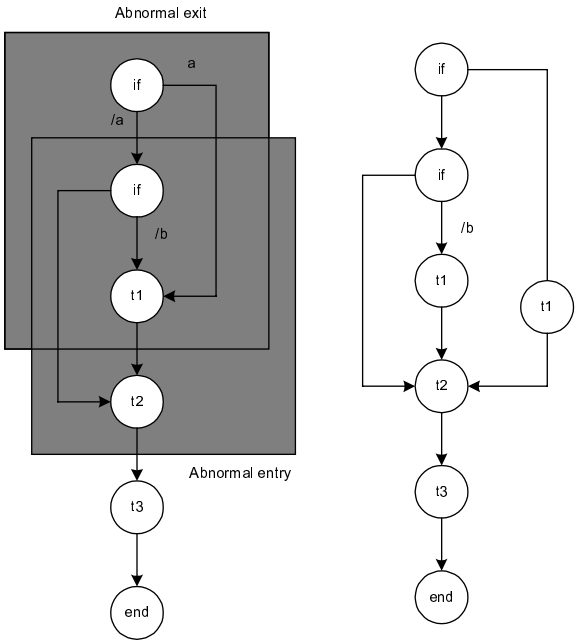

Fig. 6. Compound condition and code duplication

Exception treatment

The exception mechanism modifies control flow and thus can be a means to illegally transmit information. We have to translate into SMV the possibility that byte code can generate new control paths. Two kinds of exception will not be taken into account here: `OutOfMemoryException` and `StackOverflowError`. For those exceptions, the virtual machine send back an APDU with an error code and reinitializes the frame. Some byte code can generate one or more exceptions as shown in the following figure.

We translate the possibility of each byte code to generate an exception, by creating a non determinism in the choice of the next byte code to be executed. For the virtual machine, when an exception is raised during a byte code interpretation, the state of the system is not affected by the byte code. For this, we have to indicate to all incoming paths to a given instruction the possibility to execute this instruction or to raise exceptions. Then we have to model the modification of the control flow which is local to the method (exception handler or exception propagation) and in the call graph to indicate how exceptions are raised to the caller. We describe hereafter an example on how exceptions are treated locally in the model of the method.

In this example, after the first instruction it is possible to execute the invokeSpecial instruction or to raise a NullPointerException. The method 108, called with invokeSpecial_108 can raise a SecurityException or an Arithmetic Excep-

Byte code Java	Exception
aaload,baload,bastrore,sastore	NullPointerException, SecurityException, ArrayIndexOutofBoundException
aastore	NullPointerException, ArrayIndexOutofBoundException, ArrayStoreException, SecurityException
anewarray	NegativeArraySizeException
arraylength,athrow,getfield-<t>,getfield_<t>_this, getfield_<t>_w,invokeinterface,invokevrtual, putfield_<t>,putfield_<t>_this,putfield_<t>_w	NullPointerException,SecurityException
checkcast	ClassCastException,SecurityException
instanceof,putstatic_<t>	SecurityException
invokespecial	NullPointerException
sdiv,srem	ArithmeticException

Fig. 7. Byte code exceptions

```
case {
  (active & exeI=NoExecution) : {
    (next (pc),bc) := switch (pc) {
        -1 : (-1,nop);
        0 : ({pc+1,4},load);
        1 : (pc+1,invokeSpecial_108);
        2 : (pc+1,load);
        3 : (-1,ret);
        4 : (4,athrow_NullPointerException);};}
  (active & exeI=SecurityException) : {
    (next (pc),bc) := switch (pc) {
        5 : (5,athrow_SecurityException);
    default : (5,nop);};}
  (active & exeI=ArithmeticException) : {
        6 : (7,load);
        7 : (-1,ret);
    default : (6,nop);};}
  (∼active : {next(pc) := pc; bc := skip;}
```

Fig. 8. Model of the exception mechanism in SMV

tion. This is modelled with the incoming exception variable exeI. The first one is raised while the second is locally handeled.Using these variables allows the exceptions to be transferred from a method to the caller.

4 Results

The PACAP case study

The key point of such a tool is its ability to deal with real smart card applications. To verify the scalability of the prototype we have developed a set of communicating applets. They provide all the functionalities of smart card applets. They have all the administrative commands to initialize and personalize the applet. We paid a particular attention to key administration. The three applets are written in Java, and they have been compiled and converted into cap file in order to be downloaded into Java cards. The size of the purse cap file is around 30 ko which represents a big application for smart cards. By analyzing some methods of the loyalty package, we obtain for the average size of the methods 58 byte codes, the maximum 281 byte codes and the minimum 4. This provides an idea of the PACAP application.

The following picture shows the applet and their shareable interfaces. For example, the Loyalty applet will generate four SMV models one for each method of the interface and the `ProcessAPDU` command of the Loyalty. This command represents the calls from the terminal.

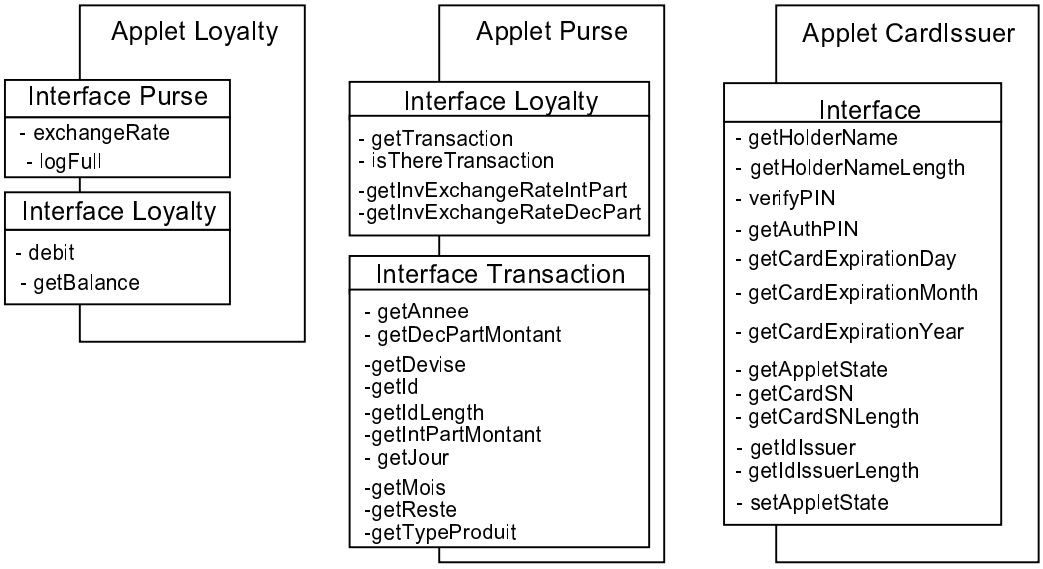

Fig. 9. The PACAP Case study

The `ProcessAPDU` command can be split into several call graphs. This command is a dispatcher (a huge switch case) that only transmits the parameter to the *ad hoc* method and sends back the result of the method. We obtain 51 SMV models for the Loyalty application.

The next table shows information about the size of four of the 51 generated SMV models for the package Loyalty.

	Number of methods	Number of SMV lines	Number of properties
Debit	5	1033	2
Logfull	9	1900	12
ExchangeRate	10	2144	13
getBalance	2	339	1

Fig. 10. Characteristics of some models

Model analysis

Most of the properties are verified within 10 seconds while a third of them needs around 8 minutes to be verified. Some properties are not verified. For example, in the logfull call graph, the following property does not hold:

invoke_isThereTransaction: assert G (bc = invoke_isThereTransaction -> (lpc | stck[stckP + 2] | stck[stckP + 1] -> method[0]))

It corresponds in the getTransactions method to the verification of the two parameters of the called method isThereTransaction. Those parameters are on top of the stack and their level must not exceed the level of the shared interface purse loyalty.

The detected flow is not really illegal. This is due to the policy used to assign levels to the variables. All the variables of an applet have the level of the applet. Unfortunately such a policy must be more accurately tuned. All the variables that are transferred to an interface method must be declassified to the interface level. After modifying our policy, all the properties hold.

The next step consists in developing hostile applets and verifying that illegal flow are correctly detected by the PACAP prototype.

5 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]. One approach has used model checking (with SMV) to specify the correct typing of Java byte code [10]. 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 *al.* [3] 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 [11] 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 [9] 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 [7] 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 [8] focuses on integrity property by controlling exclusively write operations to locations of references of sensitive objects such as files or network connections.

6 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 implemented 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.

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.

We demonstrated the ability of the tool to verify real life models and to check non trivial properties. But it seems difficult to obtain a fully automated tool. In fact, when a counterexample is given by SMV it is difficult to isolate the illegal flow and to identify in the source code the origin of the problem. We expect in a close future to provide a more friendly user interface with enough annotations in the SMV model to track the flow in the source code.

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 suffi-

cient 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. S. N. Freund and J. 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 œuvre d'une analyse statique de code en vue de la vérification d'applications sécurisées*. Ph.D. thesis, ENSAE, 1996.
4. P. Girard. *Which security policy for multi application smart cards?* In USENIX workshop on smart card technology, 1999.
5. P. Girard, J.-L. Lanet. *New Security Issues raised by Open Cards*. Information Security Technical Report, 4(2):19–27, 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, modeling 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. J. Posegga and H. 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. R. Stata and M. Abadi. *A type system for Java byte code subroutines*. In Proceeding of 25th Symposium on Principles of Programming Languages, 1998.
13. P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. *Checking Secure Interactions of Smart Card Applets*. In ESORICS 2000, Toulouse, September 2000.
14. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J-L. Lanet. *Electronic Purse Applet Certification*. In Workshop on Secure Architectures and Information Flow, London, December 1999. `<http://www.elsevier.nl/gej-ng/31/29/23/57/show/Products/notes/cover.htt>`.
15. C. Cifuentes, *Reverse Compilation Techniques*, Ph.D. Thesis, Queensland University of Technology, 1994.