On Practical Information Flow Policies for Java-Enabled Multiapplication Smart Cards *

Dorina Ghindici, Isabelle Simplot-Ryl

IRCICA/LIFL, CNRS UMR 8022, Univ. Lille 1, INRIA Lille - Nord Europe, France
{dorina.ghindici, isabelle.ryl}@lifl.fr

Abstract. In the multiapplicative context of smart cards, a strict control of underlying information flow between applications is highly desired. In this paper we propose a model to improve information flow usability in such systems by limiting the overhead for adding information flow security to a Java Virtual Machine. We define a domain specific language for defining security policies describing the allowed information flow inside the card. The applications are certified at loading time with respect to information flow security policies. We illustrate our approach on the LoyaltyCard, a multiapplicative smart card involving four loyalty applications sharing fidelity points.

1 Introduction

Computer systems handle a considerable amount of data carrying sensitive information that should be protected from malicious users. Programs running on such systems may access data either to perform computations or to transmit it over an output channel. Thus they can violate the security of sensitive data either by releasing it to unauthorized users or by modifying it. In order to prevent such situations, tracing data manipulation throughout programs is mandatory.

Information flow analysis [16] consists in statically analyzing the code of a program in order to detect illicit data manipulations. Concretely, data manipulated by programs (e.g. objects, parameters) are tagged with security labels and all information flows are traced. The assignment `p:=s`, where `p` is a public, observable variable and `s` contains secret, confidential data, generates an *explicit* flow from secret to public data. The code `if(s) p:=0` contains an *implicit* flow of information as an external observer, who has knowledge about the control flow of the program, can learn information about the secret data `s`. Usually, information flow is associated with non-interference [9] which prevents all information flows from sensitive data to non-sensitive data. The examples above generate illegal information flows w.r.t non-interference.

Information flow analysis does not guarantee security by itself: it is a powerful mechanism that can be exploited to implement the desired security policies. The difficulty is to ensure that local checks (mechanisms) actually implement the global security policy. Information flow mechanisms are too coarse to express

* Funded by ANR SESUR SFINCS (ANR-07-SESU-012) project

desired policy, thus one of their common pitfalls is to define and verify complex policies, reflecting real attacking scenarios.

In this paper, we address the problems of defining security policies for information flow, enforcing them by an information flow analyser and helping the programmer to build safe applications in case the verification fails. The target devices are multiapplicative smart cards, running a Java Virtual Machine (JVM). The security policies express allowed data flow between applications, either due to code reuse or collaborations (e.g. commercial agreement). To support our approach, we consider the case study of LoyaltyCard, a multiapplicative smart card containing four fidelity applets. The main contributions of this paper are:

– to define a specific language for specifying information flow security policies,
– to present how the policies are enforced in a standard JVM,
– to make the information flow analysis practical w.r.t. software engineering, by adding information flow contracts and giving hints for helping programmers to develop safe applications.

The rest of the paper is structured as follows: Section 2 presents the LoyaltyCard example, while Section 3 introduces some aspects of information flow analysis in the context of open, small systems and identifies challenges. In Section 4 we define a domain specific language for information flow policies, while Section 5 presents a deployment and development environment. Section 6 discusses related work, while Section 7 summarizes our contributions.

2 LoyaltyCard Example

In this section we present LoyaltyCard, a multiapplicative Java-enabled smart card composed of four loyalty applications: two air companies (FlyFrance, FlyMaroc), a car renting company (MHz) and a hotel (Illtone). The applications implement loyalty services and can share information. Three of these applications form a group of partners: confidential data flow between partners is secure, while collaborations with external applications, as depicted in Figure 1, may lead to illegal flows of information.

Let us suppose that FlyFrance has a commercial agreement with MHz and Illtone, so part of FlyFrance points can be used to obtain MHz and Illtone loyalty points. On the other hand, FlyFrance does not want FlyMaroc to learn any information about the fidelity status of its clients (e.g. the number of miles, or the status: gold, silver client, etc). FlyMaroc has also an agreement with MHz and offers a discount, based on the fidelity status of the MHz client. Suppose that, when asked by FlyMaroc, MHz returns not only its fidelity points, but also fidelity points of its partners (FlyFrance). FlyMaroc can infer, through MHz, information about the FlyFrance fidelity points, as depicted in Figure 1. In such a way, an illegal information flow is established. Illtone also offers a discount for MHz clients, but this time the flow of information is allowed, as Illtone is one of the partners of FlyFrance.

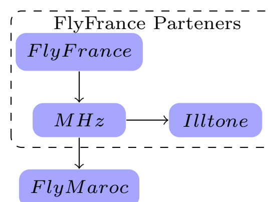

Fig. 1. Illegal information flow

We want to be able to show that the implementations of the FlyFrance, MHz, FlyMaroc and Illtone enforce information flow policies, e.g. each program shares data only with trusted applications.

```
class FlyFrance {
  private int miles;
  [..]
  public void updates() {
    int i=0;
    for(;i<noLoyalties;i++)
      update(loyalties[i]);
  }

  void update(Loyalty l){
    l.update(miles);
  }
}
```

```
class MHz extends Loyalty{
  private int points;
  private int ppoints;
  [..]
  public void update(int p){
    this.ppoints += p;
  }
  public int getLevel_() {
    if(points+ppoints>GOLD)
      return LEVEL_GOLD;
    return LEVEL_SILVER;
  }
}
```

```
class FlyMaroc {
  private int oldLevelMhz;
  [..]
  int makeGetLevel(MHz h) {
    int newLevel = h.getLevel();
    if(oldLevelMHz!=newLevel){
      print("level changed!");
      oldLevelMHz = newLevel;
      return newLevel;
    }
    return ERROR;
  }
}
```

Fig. 2. Excerpt from the Java implementation of LoyaltyCard

Figure 2 shows an extract of the Java code of `FlyFrance`, `FlyMaroc` and `MHz` classes. The confidential data of `FlyFrance` is stored in the field `miles`. The method `update` in `FlyFrance` updates the points of its partners (`MHz` and `Illtone`). `MHz` stores its partner points in field `ppoints`. Method `MHz.getLevel()` returns the fidelity level of `MHz`, based on `MHz` points and on partner points. This method is called by `FlyMaroc` in order to offer a discount, which leads to an unexpected flow of information to an application untrusted by `FlyFrance`. The `MHz.getLevel()` method is also called by `Illtone`, but in this case, the flow of information is authorized as `FlyFrance` has an agreement with `Illtone`.

3 Embedded Security and Information Flow

Ubiquitous computing is evolving towards post issuance and automatic execution of untrusted code. Executing untrusted code implies many security risks. For example, a malicious applet running on your mobile phone or smart card can do a lot of harm: it can disclose confidential information, financial data, address book, social security and medical files, etc. Moreover, if the system runs multiple applications, which share data, then it must ensure data confidentiality for each application by controlling the underlying information flow.

3.1 Information Flow model

In [6], a compositional information flow analysis enforcing non-interference for Java programs running on small, open embedded systems has been presented. The analysis consists in statically interpreting JVM bytecode and inferring types representing all possible information flows that may occur when executing the program. The behaviour of a program, in terms of information flow, is defined using contracts: a security contract [1] guarantees the maximal information flow that may occur while executing the program. In this paper, we enrich this framework with security policies which relax non-interference by describing allowed flows of information between application.

We now briefly describe the information flow model, which is needed for better understanding the identified challenges and the proposed solutions. For more details on the model, please refer to [6].

Information sources and security levels As in classical information flow, each field is annotated by a security level: s for secret, sensitive data and p for public, observable data. This work is based on the idea that confidential data in small objects (e.g. loyalty points, PIN code) typically resides in instance fields of objects [10]. We prevent information flow from high-security to low-security instance fields. In order to have a reasonable size of information flow annotations for embedded systems, the analysis is *field independent* but *security level sensitive*: all the fields of an object having the same security level are modeled as having the same location. Thus, considering the security levels s and p, each object o is modeled as two sub-objects (parts): a secret part (o^s) and a public part (o^p).

In the LoyaltyCard example, the confidential data (`FlyFrance` fidelity points), is stored in the field `FlyFrance.miles` and the field `ppoints` of its partners (`MHz` and `Illtone`). Hence these fields have security level s.

The flow relation We now define the flow relation. We say that there is a flow from a to b if an observer of b can learn information about a.

Considering our split of objects and the dichotomy of Java types (elementary types and object types), the flows between two elements a and b have the form $a^{\wp(p,s)} \xrightarrow{\texttt{r/v/i}} b^{\wp(p,s)}$, where `v` denotes a flow arising from an assignment of primitive type, `r` a reference flow (an alias), `i` an implicit flow; s and p denote the security levels, secret or public, while $\wp(p,s)$ denotes subsets of $\{p,s\}$.

For example, if an object `a` has a field `s`, of type `int`, labeled with security level s and `b` has field `p` with security level p, the code `b.p = a.s` generates a value flow from the secret part of `a` to the public part of `b`, denoted by $b^p \xrightarrow{\texttt{v}} a^s$. The code `if(a.s) b.p=0` generates an implicit flow $b^p \xrightarrow{\texttt{i}} a^s$.

The security contract of a method A security contract[19] carries relevant information for a later usage of the method: it contains flows, potentially generated by the execution of the method, between sources of information flow (abstract values) visible outside the method. We identify thus the following abstract values:

- the parameters of a method m,
- the return value of the method, denoted by the abstract value R,
- input/output channels: all the channels are abstracted by a single value, IO,
- static fields, which are modeled as fields of a single object, denoted by the abstract value $Static$,
- exceptions: all thrown values flow to the abstract value Ex.

Let Σ_m be the set of abstract values of a method m. We define now the security contract of a method m as

$$S_m = \{a \xrightarrow{\texttt{r/v/i}} b \mid a, b \in \Sigma_m \times \wp(p, s) \text{ and the execution of } m \text{ potentially generates a flow from } a \text{ to } b\}.$$

For example, considering that `FlyFrance.miles` and `MHz.ppoints` have security level s, the method `MHz.update` from the LoyaltyCard (Figure 2) generates a flow from the parameter `p` to a secret field of `this`. Thus, the security contract of the method is $S_{update} = \{this^s \xrightarrow{\texttt{v}} p\}$. The information flow analysis infers the security contracts in Figure 3 for the rest of the methods in Figure 2.

$$\begin{array}{lll} FlyFrance: & S_{update} & = \{l^s \xrightarrow{\texttt{v}} this^s\} \\ MHz: & S_{update} & = \{this^s \xrightarrow{\texttt{v}} p\} \\ MHz: & S_{getLevel} & = \{R \xrightarrow{\texttt{i}} this^{s,p}, R \xrightarrow{\texttt{v}} Static\} \\ FlyMaroc: & S_{makeGetLevel} & = \{this^p \xrightarrow{\texttt{i}} h^{s,p}, R \xrightarrow{\texttt{i}} h^{s,p}, R \xrightarrow{\texttt{v}} Static\} \end{array}$$

Fig. 3. Security contracts for LoyaltyCard

3.2 Challenges

The real challenge in information flow analysis is applying its results in practice. We identify some of the major problems in making the analysis usable for which we give solutions in the next sections.

Defining policies that explore security contracts In literature, confidentiality is often seen as a non-interference [9] problem, as public outputs cannot depend on secret inputs. Non-interference policies do not allow any flows from secret to public values, but only flows from secret to secret. Nevertheless, non-interference does not make any distinction between the source of secrets. It is a transitive and symmetric relation. Policies defined with such relation are too restrictive, and not the desired policies in most of the cases, and especially in multiapplicative smart cards [8]. Our aim is to refine non-interference by defining more complex intransitive and asymmetric policies. In the LoyaltyCard, the security policies of an applet running on the smart card (FlyFrance) allow secret information to be released to some other applet (MHz) but not to FlyMaroc.

In order to escape from non-interference strictness, we define, in Section 4, a specific language, which describes the allowed flow of information between applications. Programs are certified by verifying that the security contracts respect the desired security policies.

Integration to existing systems (Jvm) Another important challenge, which prevented information flow mechanisms from being used in real systems, is getting the certification process and information flow policies to correctly and easily interact with existing systems. JFlow [13] and Flow Caml [18] are powerful languages, that offer support to a reliable development by defining a new programming language which mixes source code and security policies in a coherent set. However, they do not address the problems raised by mobile code and open environments, and do not fit into the Java paradigm of dynamic class loading.

Integrating information flow policies for mobile code in Java-enabled small open embedded systems requires, at least, *(i)* separation of code and security policies and *(ii)* certification at load time. In Section 5.1 we present how the enforcement process of information flow policies is integrated in a JVM.

Developing safe interacting applications The security of a system depends on the security of each component. A key in computer security is not only detecting and preventing attacks, but also helping the developer to build safe applications, components. Contracts can be used as a support for creating secure applications, as developers can express, by their means, the expected behaviour of unknown or untrusted applications. In this paper, we express information flow in a program using contracts and we introduce an approach, based on reverse engineering, to help building applications that respect information flow policies.

4 A DSL for information flow policies

A domain-specific language (DSL) is a small, usually declarative, language that targets a particular kind of problem. The key characteristic of DSLs is their focused expressive power. DSLs are usually concise, offering only a restricted suite of notations and abstractions, thus adapted to express security policies.

4.1 DSL definition

In order to express security policies describing collaborations and information flows between applications, we thus define a domain-specific language in Figure 4. The security policies that can be expressed with the DSL are simple, but have enough power to model collaborations schemes in a smart card. The DSL was designed for multiapplication smart cards, but it can be extended to other applications, if necessary.

Multiapplicative smart cards allow data sharing and service sharing in order to optimize the use of resources (e.g. API) and to allow collaborative schemes (e.g. agreements or contracts between applications). In a smart card, the entities exchanging or sharing data are the applications, thus the DSL contains rules defining *trust relations* between applications.

Applications are addressed either by package or class names, using elements in the sets of terminals *Class* and *Package*; *Field* denotes a set of terminals containing field names.

S ::= $(Class|Package)$[,S]
F ::= $Field$[,F]
R_c ::= $Class$ **secret** F**;**
R_s ::= S **shares with** S**;**
R_p ::= S **strict secret ;**
P ::= $(R_c|R_s|R_n)$[,P]

Fig. 4. A DSL for information flow policies

As we consider that confidential data resides in class fields, rule R_c expresses the secrets of a class, by listing the fields that should remain confidential, and thus that have the security level s. The main rule of the DSL is R_s which describes the allowed information flows. For example, the signification of S_1 **shares with** S_2**;** is that all elements in S_1 can share their secrets with all elements in S_2.

By default, an element in S can share its secret with other elements having the same type (e.g. class A shares its secrets with all instances of class A). Rule $R_p = S$ **strict secret ;** refines the security policies by specifying that an element A in S must not share its secrets with other objects of type A. While the rule R_s defines type-based policies, rule R_p refers to instance-based policies, in the case when the instances have the same type.

The sharing relation can be associated to the *trust relation* defined in [8] by Girard: one application transmits its secrets only to trusted applications. As the trust relation, the sharing relation is neither symmetric nor transitive. If A **shares with** B then not necessarily B **shares with** A. An application would not trust another application only because one of its trusted applications does. Detecting data leaks due to transitivity, or propagation, is one of the main concerns of information flow security. Allowing transitivity would make no distinction between information flow and access control.

4.2 DSL verification

The certification process of an information flow policy has two parts:

1. verifying *simple class sharing*: an application gives its secrets only to trusted applications,
2. verifying *transitivity* (or data propagation): an application trusted by A does not share confidential data with applications untrusted by A.

Simple class sharing The rule P_i **shares with** P_j, where P_i and P_j are two packages, can be read as "any class in package P_i trusts any class in package P_j". Hence, the DSL in Figure 4 can be reduced to rules having the form A **shares with** B, where A and B are class names in *Class*. We can compute a function $share : Class \rightarrow \wp(Class)$ which associates to each class the classes it trusts. By default, a class trusts ifself, thus A **shares with** A**;** and $A \in share(A)$. Verifying the security policy of a class $A \in Class$ reduces to verifying that secrets of A flow only to elements in $share(A)$. Hence, we verify security policies at the granularity level of classes.

Transitivity Once a class A shares confidential data with a trusted class B, A loses control over its propagation. The secret of A becomes the secret of B. The policy of A holds if the policy of B is more restrictive: B does not share its secrets with applications untrusted by A. Formally, verifying transitivity can be summed up to verifying that, for all $B \in share(A)$, $share(B) \subseteq share(A)$ holds.

Security policies and security contracts We now show how policies defined using the DSL are enforced by the information flow analysis described in Section 3. Confidential data resides in object fields. Let $fields_s : Class \rightarrow \wp(Field)$ be a function that associates to each class the fields having the security level s, thus fields in the rule $R_c = Class$ **secret** *Field*; the function $fields : Class \rightarrow \wp(Field)$ gives all fields of a class.

Secrets can be made accessible either by direct access to fields or through method invocations and operations performed by the method. In order to prevent direct access, secret fields of a class A in $\mathit{fields}_s(A)$ must be declared using the Java access modifier *private*. This restricts the access to secret fields only in the class where they have been declared and thus to which they belong. Based on this, certifying a class A with respect to an information flow policy consists of verifying every method in A and methods that use the class A. Let *Method* be the set of methods names and $\mathit{methods} : \mathit{Class} \rightarrow \wp(\mathit{Method})$ a function that gives the list of methods for each class.

Let us remind that the information flow model in Section 3 computes, for each method $m \in \mathit{Method}$, a security contract S_m containing all the possible flow of information between abstract values in Σ_m (parameters, *IO*, *Ex*, *Static*, *R*). A flow is denoted by $a^{t_1} \xrightarrow{\texttt{r/v/i}} b^{t_2}$ with $t_1, t_2 \in \{s, p\}$ denoting the security level. Flows can be from public/secret parts of an abstract value to public/secret parts of another abstract value. Security is concerned with protecting flows from the secret parts to public/secret parts. As a general rule, flows from secret to public are forbidden, while flows from public to public are always allowed. A class A shares its secrets with classes in $\mathit{share}(A)$, thus only flows from secret parts of parameters of type A to parameters with type in $\mathit{share}(A)$ and to return (R) are allowed. Let $\mathcal{T}$ be a function which associates to an abstract value its definition type, in *Class*. The algorithm that verifies method in a class A is depicted in Figure 5. To permit the flows to return, we consider that $R \in \mathit{share}(A)$.

1: **for all** m in $\mathit{methods}(A)$ **do**
2: **if** $\exists a^p \xrightarrow{f} b^s \in S_m$ **then**
3: **return false**
4: **end if**
5: **for all** $a^s \xrightarrow{f} b^s \in S_m$ **do**
6: $t_1 = \mathcal{T}(a)$, $t_2 = \mathcal{T}(b)$
7: **if** $t_1 \notin \mathit{share}(t_2)$ **then**
8: **return false**
9: **end if**
10: **end for**
11: **end for**
12: **return true**

Fig. 5. Certifying the policy of class A

Encapsulation The split of objects and the definition of secret/public part may open a door to bypassing security checks through encapsulation. For example the code `A.p.r=A.s`, where `s` and `r` have security level s and $\mathcal{T}(p) \notin \mathit{share}(A)$, generates a flow $A^s \xrightarrow{\texttt{V}} A^s$, allowed by our analysis, but illegal as the secret of A flows to an untrusted type (p). In order to avoid such leaks, we define the following encapsulation property: *All secret fields and sub-fields of a class A must be trusted by A*, where sub-fields refer to fields of fields and etc.

for all f in $\mathit{fields}_s(A)$ **do**
if $\mathcal{T}(f) \notin \mathit{share}(B)$ **then**
return false
end if
end for
for all f in $\mathit{fields}(A) \setminus \mathit{fields}_s(A)$ **do**
if $\mathit{scr}_C(\mathcal{T}(f))$ **then**
return $\mathit{enc}_{field}(\mathcal{T}(f), B)$
end if
end for
return true

Fig. 6. $\mathit{enc}_{field}(A, B)$

The verification of this property consists in unfolding the fields of each class and verifying that, for each secret field f, we have $\mathcal{T}(f) \in \mathit{share}(A)$. Let $enc_{field}(A, B)$ be a function which verifies, recursively, that all secret fields of class A are trusted by B ($enc_{field} : \mathit{Class} \times \mathit{Class} \rightarrow \{true, false\}$). The algorithm is depicted in Figure 6.

If we take in consideration that only few classes contain secret fields, we can label the classes containing only public fields and stop the unfolding when we meet such classes. Let $scr_C : \mathit{Class} \rightarrow \{true, false\}$ be a function which tests if a class contains some secret fields or not; $scr_C(A)$ refers not only to secrets defined in A but also to secrets defined in fields of A, etc.

Thus, to verify that a class A respects the encapsulation property, a call to $enc_{field}(A, A)$ is sufficient.

4.3 Example

Figure 7 presents the information flow policy for the LoyaltyCard presented in Section 2. The first three rules define the confidential data, while the last two rules define the allowed information flow. The policy respects transitivity, as the policies of applications trusted by FlyFrance (MHz, Illtone) are smaller than the policy of FlyFrance. The verification fails while trying to validate the method `makeGetLevel` defined in `FlyMaroc`, as it contains a flow $this^p \xrightarrow{\mathtt{i}} h^s$, where h denotes the MHz application.

FlyFrance **secret** *miles*;
MHz **secret** *ppoints*;
Illtone **secret** *ppoints*;
FlyFrance **shares with**
MHz,*Loyalty*,*Illtone*;
MHz **shares with** *Illtone*;

Fig. 7. Security policy for LoyaltyCard

4.4 Discussion

Conflict resolution While rule R_c and R_s are permissive, the rule R_p is restrictive and thus can generate conflicts. Let us consider the following policy for a class $x.A$, where x is the package to which A belongs:

$$x.A \textbf{ strict secret ;} x.A \textbf{ shares with } x.* \textbf{;}.$$

In the first rule, $x.A$ does not trust itself, while in the second rule $x.A$ trusts all classes in package x, and thus it trusts itself. To solve such conflicts, we consider that the rules R_p (**strict secret**) prevail over rules R_s (**shares with**). Thus, we first construct the function *share*, and only after we take into consideration the fact that a class is **strict secret** or not.

Support for overloading One of the most powerful attributes of object-oriented programming, and thus Java, is code reuse and factorisation, by the means of inheritance. But, apart from providing this powerful functionality, inheritance provides also means for leaking information. To prevent such leaks, we define some relations between policies of subclasses and superclasses.

The first restriction regards inherited fields: their security level cannot be changed by a subclass. Doing so, the security contracts of inherited methods

change, and the superclass must be reanalyzed. This is not convenient for our compositional approach, and for open systems. Nevertheless, a child class can declare new fields even with security level s.

While overloading a class, for example B extends A, the security policy of B must not only enforce security for B, but also for A and classes already verified using A. If the policy of B is greater than the policy of A, formally $share(B) \supseteq share(A)$, then the confidentiality of A is not respected anymore, as B can trust and share its secrets (and thus those of A) with classes which A does not trust. If the policy of B is smaller than the policy of A, formally $share(B) \subseteq share(A)$, in order to certify B we must reanalyse A, as A, and thus a part of B, have been certified using a greater policy. From these examples, we can conclude with: the security policy of a class B must be the same as the security policy of its superclass A, $share(B) = share(A)$.

The constraint above is too strict for API classes, which are *public classes* (we use this term to denote classes which do not contain secrets, hence classes for which scr_C returns $false$). In order to deal with API, we relax the policy above in the following way: the policy of a subclass must be the same as the policy of the inherited class only if the inherited class contains secret fields. Thus, the policy of a subclass can be any policy, if the inherited class is a public class. Problems may arise if we cast a public class to a class which contains secrets. To deal with such situations, we extend the flow signature with the types in which public classes are cast inside the method, and we take into consideration all these types while verifying the method.

1: **if** $fields_s(B) \setminus fields_s(A) \cap fields(A) \neq \emptyset$ **then**
2:    **return false**
3: **end if**
4: **if** $fields_s(A) \neq \emptyset \wedge share(B) \neq share(A)$ **then**
5:    **return false**
6: **end if**
7: **return true**

Fig. 8. Certifying the policy of class B extends A

For example, let us consider that we have `C extends B`. There are 2 cases: Security issues arise when class B does not contain any secrets ($fields_s B = \emptyset$) and C declares secret fields ($fields_s(C) \neq \emptyset$). In this case, the leak occurs only when a cast from B to C is made inside a method m. To solve this problem, while analysing m, we store the types in which classes of type B are cast inside m; for example, if parameter p_1 of type B is cast in C or in D, then we associate a list to p_1 ($p_1 \Rightarrow (C, D)$). This list must be kept only for types which do not contain secret fields, thus for which the function scr_C does not hold. Flow signatures are extended with such lists. For simplicity, we do not consider this case in the algorithm presented below.

We can now extend the certification algorithm presented in Figure 5 to take into consideration overloading. The extension is presented in Figure 8. Readers should not confuse security policies with security contracts, for which we have different restrictions.

Extending policies (Declassification) The DSL and the security policies can be extended to express more detailed rules about the release of information. The current DSL expresses policies that apply to entire program, and does not specify

where the information release is permitted. We can define rules that delimit the methods where the information flow may occur, for example

$$R_m ::= S \textbf{ shares with } (\textbf{IO} \mid S) \textbf{ in } \mathit{Method}\textbf{;}$$

where *Method* represents a method name or a list of methods and **IO** is a keyword (terminal) standing for the abstract value IO. The declassification adds power of expression as it allows also to send data on input/output channels.

Declassification relaxes the security policies in certain method. To support polymorphism and dynamic class loading, all the overriding classes must agree on the declassification contract, e.g. the declassification rule must be defined by every class in the class hierarchy.

Information flow policies as contracts The DSL in Figure 4 allows the declarations of information flow policies for applications sharing confidential data. Not only this language has a declarative value, but it also has a contractual value. For example, with the rule $FlyFrance$ **shares with** MHz, FlyFrance imposes a contract to MHz: FlyFrance agrees to share its secrets with MHz only if MHz does not share its secrets with applications not trusted by FlyFrance. Thus, the policies defined using the DSL are contracts that applications must respect. An application accepts the contract of a trusted application only if it is smaller than its own contract. In order to deal with openness and overriding, the DSL imposes that the contracts of classes extending classes containing confidential data do not change, with respect to the contract of overridden class.

5 Integrating information flow in a development and deployment schema

Even if information flow is a well studied area, there are not enough mechanisms guaranteeing security for existing systems. The main difficulty for practical information flow is to integrate it in a real development and deployment schema.

5.1 Enforcing security policies for Jvm

We present here how information flow policies defined in previous section may be enforced by any JVM. As the compiled JVM bytecode is downloaded through an unsecured channel, the information flow certification must be done oncard, preferably at loading time in order to avoid run-time overhead. Both security contracts and policies must be enforced. As computing security contracts requires many resources (both in memory and time), we perform a two step analysis: *(i)* an external phase [6] (supposed to have access to infinite resources) which computes the type inference and annotates the bytecode with some proof elements, and *(ii)* an embedded phase [7], which verifies, at loading time, the security contracts obtained during the external phase. The verification operation is linear in code size and uses constant memory. This technique lies on the same simple idea as proof-carrying code [15] that it is easier to verify a result already computed. We deal here only with the verification of information flow policies. The verification of security contracts is described in [7].

In order to make the analysis practical and integrable with any existing JVM system, we *(i)* load policies to be certified as attributes of .class files; systems not enforcing information flow can ignore these attributes, and *(ii)* verify security policies with a custom class loader, that can be installed on any system.

Extending .class files with information flow security policies The policy of a class A is the list of classes with which it can share confidential data (denoted by $share(A)$). The .class attribute for the policy of A contains thus a list of class names. The class names are represented by their index in the ConstantPool of the class A. Considering that in a smart card the number of installed applications is not significant, thus the sharing policies are quite simple, the newly added attribute contains usually only few entries. The small size of the attribute is acceptable for a small system.

As classes are loaded one by one, it is possible to load A before loading all the classes used by A. While validating a class A, we also validate the policies of classes used in A. Thus, to be able to validate A, we also load the policies of classes used in A. If B is a class used by A, when loading A either *(i)* we take in consideration the policy of B, if B has already been loaded or *(ii)* use the policy of B that A announces and we keep it oncard, in a repository, in order to validate (and remove) it when B is loaded.

Verifying security policies using a custom class loader The loading process in a JVM is performed by the class loaders. In order to integrate the information flow analysis on any JVM, the verification is performed by custom a class loader ($SafeClassLoader$), which can be built in the single class loader of KVM or installed as a user-defined class loader for a standard JVM. The $SafeClassLoader$ must verify both security contracts, as described in [7], and information flow policies.

Classes are loaded one by one. Once the security contracts of the class have been verified, the $SafeClassLoader$ validates the information flow policy, using the security contracts. The difficulty may arise from the fact that the loaded class A wants to share its secret with a class B not yet loaded. As the class is not present in the system, we do not have its security policy and we cannot verify the transitivity, formally $share(B) \subseteq share(A)$. In order to verify this condition when B is loaded, we keep a repository with rules having the form $share(B) \subseteq share(A)$. If B is used by another class C, the rule $share(B) \subseteq share(C)$ must be added to repository. In this case, the final rule kept in the repository is $share(B) \subseteq share(A) \cap share(C)$, as the policy of B should be more restrictive than both policies of A and C. Thus, when the load B, we also verify that $share(B) \subseteq X$ with X denoting the intersection of security policies of classes that trust B. Moreover, we verify that the loaded class has the same policy as its super class: $share(B) = share(B')$ with B extends B'.

Verifying encapsulation The same problem may arise when verifying encapsulation: A has a field of type B, but B is not yet loaded. In order to verify while loading B that all secret fields of B are trusted by A, we keep the following rule

to the repository: $enc_{field}(B, A)$. When loading B, if a rule $enc_{field}(B, A)$ is found in the repository, than the function $enc_{field}(B, A)$ (see Figure 6) is executed. If the test succeeds, the rule is deleted from repository and the loading process continues, by performing other checks.

The result of $enc_{fields}(B, A)$ depends on $scr_C(B)$ (the function which tests if B or fields B contain secret fields). The value returned by $scr_C(B)$ depends also on fields of B. Hence, the final value of $scr_C(B)$ can be computed only when all fields, fields of fields, etc. have been loaded. To ensure the correctness of scr_C computation, we extend the repository with rules of type $scr_C(B) = scr_C(C_1) \vee scr_C(C_2) \vee \ldots \vee scr_C(C_n)$, where $C_1 \ldots C_n$ represent the type of fields of B not yet loaded. This rule is deleted from repository when a class C_i is loaded with $scr_C(C_i) = true$ or when all classes $C_1 \ldots C_n$ are loaded. Moreover, to avoid the computation of scr_C each time when it is needed, the known values of scr_C are stored on the card, in a special repository.

The algorithm verifying encapsulation at loading time is similar to the external one (enc_{field})presented in Figure 6, except that it must also verify that the class $\mathcal{T}(f)$ has been loaded; if not, if should add $enc_{field}(B, A)$ to the repository.

5.2 Reverse Engineering tool

The certification must be done oncard due to the fact that the applications are loaded using an unsecured channel and must be adapted to the limited resources of the system. In the same time, the external analysis is supposed to be done on an system offering of unlimited resources comparing with a small system. Thus optimization and complexity are not an issue. Moreover, the external resources can be used for other purposes, for example for offering an easy development environment to programmers.

Security must be insured for different attacks against computing systems, for both deliberate or accidental attacks. Information flow insecurity may arise from malicious, untrusted code or from our own code. In the later case, the insecurity is due to bad conception of the application or to bad implementation. When the information leak comes from a bad implementation due to human error, it is not always obvious for the developer to correct the application in order to make it safe. The development environment should detect illicit flows and help the developer to correct his mistakes by offering all the necessary information.

The point of failure in the program certification is not usually the real source of information leak. For example, the certification of LoyaltyCard fails while analysing the method `FlyMaroc.makeGetLevel`. But the illegal information flow comes from the implementation of method `getLevel` in class `MHz`, where the computation of fidelity level for `MHz` takes into consideration the points of partners.

To detect the failure source, we propose a backward iterative algorithm, which, at each step, tries to detect an information flow in a method. The algorithm is similar to tracking thrown exceptions in Java programs. Let us assume that we have a recursive method $detect(m, f, pc)$ which detects where the flow f occurred in method m by performing a backward analysis starting from the program point pc. If the flow f was created due to another flow f_1, the method

detect(m, f_1, pc) is called recursively. If the flow f was created in a method m_1 invoked at pc, the algorithm calls *detect*(m_1, f, pc_f), where pc_f is the program counter corresponding to the return instruction in method m_1.

This approach is memory consuming and thus cannot be performed oncard, but it can explore the unlimited resources of the external analyser.

6 Related Work

Information flow [16] has been largely studied in the last decades and many models have been proposed [2, 10]. Unfortunately, these models are mostly theoretical and almost impossible to apply in practice. Complex programming systems [13, 18] enforcing information flow security exist, but they failed in showing how they can be successfully applied to real problems [20]. Most of the systems enforce standard non-interference and expressive, useful information flow security policies lack. The PACAP framework [4] involves a technique based on model checking to verify interactions for Java smart-cards, but the verification is limited to predefined scenarios, and it cannot be trusted in an open environment.

Several works have developed policies for downgrading data [17]. JFlow [13], a powerfull programming language, implemented as an extension of the Java language, implements the decentralized label model (DLM) [14] which uses the notion of ownership; data can be release only by one of the owners only if all the owners agree. This approach is similar to our contracts on declassification: data can be released in a method if all classes in the hierarchy agree on the release. JFlow adds reliability to software implementation, but not to deployment and linking on a platform. Moreover, source programs must be annotated with security labels, and hence they must be re-coded. Many other forms and systems that declassify information have been presented [5, 12] but most of them are certified by a security type system and are based on the assumption that policies are known statically at compile time. All these work have solid theoretical foundations, but failed to be successfully applied in practice.

On the other hand, many domain specific languages and practical systems expressing security policies exist [3, 11], but they do not address information flow issues and most of the time they are dynamically enforced. Domain specific languages [11] limit themselves to specifying access control rules and do not address data propagation.

7 Conclusion

Motivated by the LoyaltyCard example, we present an approach to detect illegal information flows in multiapplicative smart cards. The desired security policies are specified using a simple, but expressive domain specific language and are enforced are loading time. On the one hand, this work bridges the gap between information flow models and current running systems. While the foundations of information flow models are solid, their practical side is still to be proved. Our approach limits the overhead for adding information flow security to existing JVM, as security labels and policies are separated from the code, and the illegal

information flow is detected by a custom class loader, installed on any JVM. On the other hand, our work bridges the gap between information flow security requirements and actual security policies, which do not take into consideration data propagation due to information flow.

References

1. AISSA, N. B. H., GHINDICI, D., GRIMAUD, G., AND SIMPLOT-RYL, I. Contracts as a support to static analysis of open systems. In *Proc. 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07)*.
2. AVVENUTI, M., BERNARDESCHI, C., AND FRANCESCO, N. D. Java bytecode verification for secure information flow. *ACM SIGPLAN Notices 38*, 12 (2003), 20–27.
3. BAUER, L., LIGATTI, J., AND WALKER, D. Composing security policies with polymer. In *Proc. ACM SIGPLAN Conf. PLDI'05*, pp. 305–314.
4. BIEBER, P., CAZIN, J., GIRARD, P., LANET, J.-L., WIELS, V., AND ZANON, G. Checking secure interactions of smart card applets: extended version. *J. Comput. Secur. 10*, 4 (2002), 369–398.
5. DAM, M., GIAMBIAGI, P. Confidentiality for mobile code: The case of a simple payment protocol. In *Proc. 13th IEEE CSFW'00*, pp. 233–244.
6. GHINDICI, D., GRIMAUD, G., SIMPLOT-RYL, I. Embedding verifiable information flow analysis. In *Proc. Conf. Privacy, Security and Trust (PST'06)*, pp. 343–352.
7. GHINDICI, D., GRIMAUD, G., AND SIMPLOT-RYL, I. An information flow verifier for small embedded systems. In *Proc. WISTP'07* , *LNCS* 4462, pp. 189–201.
8. GIRARD, P. Which security policy for multiapplication smart cards? In *USENIX Workshop on Smartcard Technology* (1999), pp. 21–28.
9. GOGUEN, J. A., AND MESEGUER, J. Security policies and security models. In *IEEE Symp. Security and Privacy* (1982), pp. 11–20.
10. HANSEN, R. R., PROBST, C. W. Non-interference and erasure policies for java card bytecode. In *6th Intl. Workshop on Issues in the Theory of Security (WITS'06)*.
11. HASHII, B., MALABARBA, S., PANDEY, R., BISHOP, M. Supporting reconfigurable security policies for mobile programs. *Comput. Networks 33*, 1-6 (2000), 77–93.
12. LI, P., AND ZDANCEWIC, S. Downgrading policies and relaxed noninterference. *ACM SIGPLAN Notices 40*, 1 (2005), 158–170.
13. MYERS, A. C. JFlow: practical mostly-static information flow control. In *Proc. 26th ACM SIGPLAN-SIGACT Symp. POPL'99*, pp. 228–241.
14. MYERS, A. C., AND LISKOV, B. A decentralized model for information flow control. In *Proc. 16th ACM Symp. SOSP'97* pp. 129–142.
15. NECULA, G. C. Proof-carrying code. In *Proc. 24th ACM SIGPLAN-SIGACT Symp. POPL'97*, pp. 106–119.
16. SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21*, 1 (2003), 5–19.
17. SABELFELD, A., SANDS, D. Dimensions and principles of declassification. In *Proc. 18th IEEE CSFW'05*, pp. 255–269.
18. SIMONET, V. Flow Caml in a nutshell. In *Proc. APPSEM-II* (2003), pp. 152–165.
19. WIN, B. D., PIESSENS, F., SMANS, J., AND JOOSEN, W. Towards a unifying view on security contracts. In *Proc. SESS'05*, pp. 1–7.
20. ZDANCEWIC, S. Challenges for information-flow security. The 1st Intl. Workshop on Programming Language Interference and Dependence (PLID'04).