Constraint-based security analysis for the Java Card firewall

Authors:	Marc Éluard and Thomas Jensen
Email:	`{eluard,jensen}@irisa.fr`

Date:	26 / 02 / 2002
Classification:	Public
Number:	SECSAFE-IRISA-002
Version:	1.2
Status:	Stable.

Abstract

This paper presents a constraint-based static analysis to prove security (confidentiality) properties of Java Card programs. We define a subset of the Java Card bytecode focussing on aspects of the Java Card firewall, method invocation, field access, variable access, shareable objects and contexts and present an analysis to compute an approximation of the set of possible value stored in each variable. To achieve this task, we introduce a new kind of constraints: quantified conditional constraints. This kind of constraints permits to generate the constraints for a program in a demand-driven fashion. In addition, it permit to model precisely the effects of the Java Card firewall by only producing a constraint if the corresponding operation is authorized by the firewall. The result of this analysis is a precise description of the object flow and of the security exceptions which can be thrown by the firewall.

Keywords: Static program analysis, smart cards, Java Card, security, firewalls, constraints, fixed point iteration.

1 Introduction

The Java Card language is a subset of Java, tailored to the limited resources available on today's smart cards. Java Card keeps the essence of Java, like inheritance, virtual methods, overloading, but leaves out features such as large primitive data types (`long`, `double` and `float`), characters and string, multidimensional arrays, garbage collection, object cloning, security managers [1, 10]. Given the security-critical application areas of Java Card, the language has been endowed with an elaborate security architecture. Applets are separated by a firewall which prevents one applet from accessing objects owned by another applet. Shareable objects and interfaces are used to provide communication between separated applets. A limited form of stack inspection allows a "server" applet (an applet who shares objects) to know the identity of a client which requested a particular service. These mechanisms (further detailed in section 2) facilitate the design of secure applications but do not themselves guarantee security. However, they offer the possibility to perform formal verification of application security using semantics and static program analysis.

This paper is concerned with verifying security properties pertaining to the confidentiality of data belonging to applets. The firewall guarantees that, by default, an applet of one package cannot access members of objects belonging to another package so, even if a foreign applet obtains a reference to an object with confidential informations this does not imply that information is leaked. Access can however be granted by declaring objects as *shareable* in which case methods marked shareable can be called from other applets. Thus, sensitive information can be leaked if a shared method returns it as a result of its invocation. Tracing the flow of information is further complicated by the fact that references to shared objects can be passed to other applets than the one to which it was originally transmitted. Thus, verifying that an object owned by applet A will never be accessible to applet B requires a precise inter-procedural analysis of the object flow of the Java Card program.

This paper presents a flow analysis for Java Card. The analysis is *constraint-based* in that for each instruction of the program it generates a set of constraints describing the data flow of the instruction. This technique is standard in flow and type analysis of object-oriented programs. We add a novelty, *quantified conditional constraints* (*QCC*s), that allows to generate the set of constraints of a program *on demand*. This way of generating constraints is useful and natural when analyzing object-oriented languages where the control flow and the data flow are inter-dependent. It generalizes the conditional constraints proposed by Palsberg and Schwartzbach [20] for object-oriented type analysis.

The paper is organized as follows. Section 2 introduces the central features of the Java Card 2.1.1 firewall and provides a detailed example. Section 3 defines our representation of the Java Card bytecode. The abstract domains used in the analysis are given in Section 4 and Section 5 defines the set of quantified conditional constraints generated for each type of instruction. Section 6 shows how these *QCC*s can be solved iteratively and Section 7 shows how the analysis performs on the example from Section 2. Section 8 and Section 9 discuss related works and directions for extending this work.

2 The Java Card firewall

The Java Card platform is a multi-application environment in which an applet's sensitive data must be protected against malicious access. In Java, this protection is achieved using class loaders and security managers to create private name spaces for applets. In Java Card, class loaders and security managers have been replaced with the Java Card firewall. The separation enforced by the firewall is based on the Java Card's package structure (the same as Java's) and the notion of *contexts* (in Java Card, this notion is called *group context*).

When an applet is created, the `JCRE` (*Java Card Runtime Environment*) gives it a unique applet identifier (AID) from which it is possible to retrieve the name of the package in which it is defined. If two applets are instances of classes coming from the same Java Card package, they are said to belong to the same context (which is identified by the package name). In addition to the contexts defined by the applets executed on the card, there is a special "system" context, called the `JCRE` context. Applets belonging to this context can access objects from any other context on the card. Thus, the set *Contexts* of contexts is defined by:

$$\textit{Contexts} = \{\ \texttt{JCRE}\ \} \cup \{\ \textit{pckg: pckg is a legal package name}\ \}$$

Every object is assigned a unique *owner context viz.*, the context of the applet which created the object. A method of an object is said to execute in the context of its owner[1]. It is with this context that the `JCRE` determines whether an access to another object will succeed. The firewall isolates the contexts in the sense that a method executing in one context cannot access any fields or methods of objects belonging to another context.

There are two ways in which the firewall can be circumvented: via `JCRE` entry points and via shareable objects. `JCRE` entry points are objects owned by the `JCRE` that have been specifically designated as objects that can be accessed from any context. The most prominent example is the APDU buffer in which commands sent to the card are stored. This object is managed by the `JCRE`, and in order to allow applets to access this object, it is designated as an entry point. Other entry points can be the elements of the table containing the

[1] In the case of static call, the execution is in the caller's context.

AIDs of the applets installed on the card. Entry points can be marked as *temporary*. References to temporary entry points cannot be stored in objects (this is enforced by the firewall).

Two applets in different contexts may want to share some information. Java Card offers a sharing mechanism, called *shareable objects*, that gives limited access to objects across contexts. An applet can allow another applet to access an object's methods from outside its context. The mechanism is restricted to methods and cannot be applied to fields. It uses a shareable interface that is an interface which extends `javacard.framework.Shareable`. In this interface, the applet gives the list of the method's signatures it wants to share. The class of the object to share must implement this interface. The "server" applet must define a method, `getShareableInterfaceObject`, called when an applet is asked to provide a shared object. It is passed as parameter the AID of the "client" applet which requested the shared object. This allows to share different objects with different client applets.

2.1 An example of sharing mechanisms

Figure 1 contains a example illustrating the sharing mechanisms of the firewall. We have 3 applets: Alice, Bob and Charlie. Alice implements a shareable interface `MSI` (we assume an interface `MSI` that extend `Shareable` in which the signature of the method `foo` is given) and is prepared to share an object `MSIO` (an instance of the class that implements the interface `MSI`) with Bob. When Alice receives a request for sharing (via a call to her method `getSIO`[2]) by the `JCRE`, she verifies that the caller is Bob. If it is Bob, she returns `MSIO` else she returns *Null*.

```
public interface MSI extends Shareable {
   Secret foo (); }
```

```
public class Alice extends Applet implements MSI {
   private Secret ObjectSecret;
   public Shareable getSIO (AID Client) {
      if (client.equals (BobAID))
         return (this);
      return null; }
   public Secret foo () {
      AID Client;
      Secret Response;
      Client = getPrevCxt();
      if (client.equals (BobAID))
         Response = ObjectSecret;
      return Response; } }
```

```
public class Bob extends Applet {
   public static MSI AliceObj;
   public void bar () {
      AliceObj = (MSI) getASIO
         (AliceAID); } }
```

```
public class Charlie extends Applet {
   private static MSI AliceObj;
   private static Secret AliceSecret;
   public void bar () {
      AliceObj = Bob.AliceObj;
      AliceSecret = AliceObj.foo (); } }
```

Figure 1: Example of shareable objects

Bob can ask for a shareable object from Alice using the `JCRE` method `getASIO`[3]. Assume now that Bob (inadvertently) leaks a reference to `MSIO` to the third applet Charlie. With this reference, Charlie can invoke the same methods of the `MSIO` object as Bob.

Alice has some doubts about Bob so she decides to verify, in each access to one of her shared methods, the identity of the caller. Java Card offers a method for obtaining the AID of the context in operation before

[2] In reality, this method is called `getShareableInterfaceObject` and is invoked by the Java Card run time system that mediates all requests for shared objects

[3] In reality, the method `JCSystem.getAppletShareableInterfaceObject`

the last context switch, here called `getPrevCxt`[4]. Using this information Alice can discover when applets from contexts other than Bob's attempt to access the `MSIO` object.

2.2 Limitations of the firewall

A potential difficulty with the Java Card firewall is that shareable objects can be accessed by any applet and not only by the applet to which the reference was given. Since references can be passed from one applet to another, this opens up the possibility for methods to share objects can be invoked by applets other than those for which they were intended. As illustrated by the example, Java Card has a limited form of the stack inspection mechanism that underlies the Java 2 security architecture for preventing such unwanted access. The Java Card `getPrevCxt` can be called to get access to the last context switch that took place. When a method is called from another applet, this context switch indicates the identity of the caller. This information can then be used to decide what value the method should return to the caller. However, it is up to the programmer to implement this correctly. If the security mechanisms provided by the language are not used properly, unwanted information flow can arise as a result of objects flowing from one applet to another. More generally, the security mechanisms provide by Java Card are sufficient to implement the wanted security policy. But there is nothing to prove that the implementation respect the wanted policy.

The rest of this paper is concerned with developing a static analysis that calculates, for each variable in an application, an approximation of the set of values that will be stored in this variable. This static approximation allows

- either to signal potential data flow from one applet to another that violates a given security policy,
- or, if no such flow is detected, to provide a proof that all data flow respects the policy.

The analysis is based on a standard constraint-based type analysis for Java-like languages, but is modified to keep an accurate account of the Java Card contexts. Indeed, since the security of an applet to a large extent relies on the use of the `getPrevCxt` method, the analysis must be able to model calls to this method precisely.

3 A representation of Java Card bytecode

To simplify the presentation we work with a "three-address" representation of Java Card bytecode where arguments and results of a bytecode instruction are fetched and stored in local variables instead of being popped and pushed from a stack. This format is similar to the intermediate language *Jimple* used in the Java tool *Soot* [23] and the transformation of code into this format is straightforward. We furthermore assume that the constant pool has been expanded *i.e.* that indices into the constant pool have been replaced by the corresponding constant. For example, the bytecode instruction `invokevirtual` takes as parameter the signature of the method called, rather than an index into the constant pool. The formal representation of Java Card bytecode can be found in [18].

3.1 Notations

The term $\mathcal{P}(X)$ denotes the power set of X: $\mathcal{P}(X) \equiv \{S \mid S \subseteq X\}$. A product type $X = A \times B \times C$ is sometimes treated as a labelled record: with an element x of type X we can access its fields with the names of its constituent types ($x.A$, $x.B$ or $x.C$). A list is defined by enumeration of its elements: $x_1 :: \cdots :: x_n$. List elements can be directly accessed giving their position ($v(i)$ for the i^{th} element). Lists can be concatened:

[4] In reality, this method is called `JCSystem.getPreviousContextAID`

$(x_1 :: \cdots :: x_n) ::: (x_m :: \cdots :: x_p) = x_1 :: \cdots :: x_n :: x_m :: \cdots :: x_p$. X^* denotes the type of finite lists, whose elements are of type *X*. The symbol $\rightarrow$ is used to form the type of partial functions: $X \rightarrow Y$. The $\bar{v} \in \bar{E}$ notation denotes the formula $v_1 \in E_1 \wedge \cdots \wedge v_n \in E_n$.

3.2 Types

Our program representation is a modified version of that of Bertelsen [5, 6]. Before introducing abstract types, we define some basic types. Id_p, Id_{ci}, Id_f and Id_m are respectively the types of qualified name of a package, of a class or an interface, of a field and of a method[5] Id_v is the type of (unqualified) names of variables. To extract name information of an identifier, we use the notation $\lceil Id \rceil^x$, where *Id* is a qualified name and *x* the type of the projection[6]. We assume a set `AID` which contains the possible applet identifiers of the applets installed on a card. This set contain a special AID for the `JCRE` (`JCRE` $\in$ `AID`).

General types Primitive values are abstract in their type (`boolean`, `short`, `byte` and `int`). A type can be a *Primitive*, the name of a class or of an interface, or an AID.

Classes A class or an interface descriptor consists of the set of the associated access modifiers ($\mathcal{P}(Mod_{ci})$[7]), the name of the class or interface (Id_{ci}), the name of the direct superclass or the names of direct super-interfaces (*Ext*), the name of the interfaces that the class implements (*Imp*), the name of its package (Id_p), field declarations (*Fld*), method declarations and implementations (*Mtd*). A class must have a superclass, the default being `java.lang.Object`, but an interface can have zero, one or more super-interfaces. Only a class can implement an interface, so for an interface this field is the empty set.

The fields of a class are described by a map from field names (Id_f) to a set of access modifiers ($\mathcal{P}(Mod_f)$) together with a type descriptor (*Type*). The type descriptor defines what type of values can be stored in the field.

Methods The methods are described by a map that to a method signature (*Sig*) associates a method descriptor (Desc_m). This structure consists of the set of the associated access modifiers ($\mathcal{P}(Mod_m)$), the code of the method (*Code*), a description of the formal parameters (*Param*), optionally a description of the variable used for to return a value (*Res*) and the local variables of the method (*Varl*). A signature is the name of the method (Id_m) and the list of type descriptors for its parameters ($Type^*$). Code is a list whose elements consist of a program counter value (Pc[8]) and the instruction at this address (*Bytecode*). The set of local variables is the list of all variable names (Id_v) with their type descriptor (*Type*).

Bytecode In this paper we only consider a small set of bytecodes which is sufficient to illustrate the different features of our analysis. In the following, T_i range over local variables and S_i is used to give the list of the type of the parameter for a call.

The main departure from standard bytecode is the introduction of the construct `ifAID` $T \in S$ *BCinst*. This specialized if-instruction takes as argument a variable *T* that contains an AID, a set $S \in \mathcal{P}($`AID`$)$ and

[5] The qualified name of an entity is the complete name. For a class, it is *p.c* where *p* is the name of the package and *c* the (unqualified) name of the class. For a method (*c.m*) or a field (*c.f*), it is the qualified name of the class and the (unqualified) name of the method or field.

[6] To extract a (unqualified name), we use *p* for a package, *c* for a class or an interface, *m* for a method and *f* for a field. To extract a qualified name, we combinate the symbols. For example, we use $\lceil Id \rceil^{p.c}$ to extract the qualified of the class (or the interface) in the qualified name *Id*.

[7] The access modifier `Interface` is used to specify that the declaration is for an interface.

[8] We assume furthermore a set *Pc* of program counter. A program counter identifies an instruction within the whole class hierarchy (*i.e.* it is relative to a class hierarchy and not just a method).

executes the instruction `BCinst` if the AID belongs to the set S. We have introduced this instruction to make explicit how the analysis takes information about AIDs into account. Ordinary bytecode can be transformed into using the `ifAID` instruction by identifying those conditional instructions that make test of the form $Aid \in S$. Most of such tests are syntactically explicit in Java Card source programs or can be identified by simple intra-procedural flow analysis.

$$Bytecode = \texttt{ifAID}\ T \in S\ BCinst \mid BCInst$$

The Java Card bytecode is transformed into a "three-address" like language. We will not describe this program transformation any further.

$$\begin{array}{rcl} BCinst & = & T := \texttt{getstatic}\, f \\ & \mid & T_0 := \texttt{invokeinterface}\, m\ T_1\ T_2 \cdots T_n\ S_2{::}\cdots S_n{::}S_{n+1} \\ & \mid & T := \texttt{invokestatic}\ getPrevCtx \\ & \mid & T_1 := \texttt{load}\, T_2 \\ & \mid & T := \texttt{new}\, Id_c \\ & \mid & \texttt{putstatic}\, f\, T \\ & \mid & T_1 := \texttt{store}\, T_2 \end{array}$$

$T{:}=$`getstatic`f loads value contained in the static field f of the class $\lceil f \rceil^{p.c}$ and stores it in T. $T_0{:}=$`invokeinterface` $m\ T_1\ T_2 \cdots T_n\ S_2{::}\cdots{::}S_n{::}S_{n+1}$ invokes the interface method m with the signature $S_2{::}\cdots{::}S_{n+1}$ on the object contained in T_1 with parameters $T_2 \cdots T_n$ and the result is stored in the variable T_0. $T{:}=$`invokestatic` *getPrevCtx* retreives the AID of the last active context before the last context switch and stores it in T. $T_1 := $ `load` T_2 loads value contained in T_2 and stores it in T_1. $T{:}=$`new` C stores a reference to the object created at this program point in T. `putstatic` $f\ T$ loads value contained in the variable T and stores it in the static field f of the class $\lceil f \rceil^{p.c}$. $T_1 := $ `store` T_2 loads value contained in T_2 and stores it in T_1.

3.3 Auxiliary functions on the class hierarchy

We define three predicates to detemine if a class member is visible from a given instruction. We have *CI_Visibility?* for a class or an interface, *Method_Visibility?* for a method and *Field_Visibility?* for a field. We must keep this test in the constraint because in some cases, like for the modifier `protect`, we need information about dynamic value.

$$\begin{array}{rcl} CI\_Visibility? & : & Id_c \times Id_{ci} \times E_{ci} \rightarrow Boolean \\ Method\_Visibility? & : & Id_c \times Id_c \times Desc_m \times E_{ci} \rightarrow Boolean \\ Field\_Visibility? & : & Id_c \times Id_f \times E_{ci} \rightarrow Boolean \end{array}$$

The function *Lookup* models the dynamic search of methods underlying the virtual method calls. It takes as arguments the signature of a method, the class in which the method is declared, the class in which the invokation are made and the class hierachy. It returns the fully qualified method name of the implementation of the method designated by the signature.

$$Lookup : Sig \times Id_{ci} \times Id_{ci} \times E_{ci} \rightarrow \mathcal{P}(Id_m)$$

A full description of the Java visibility rules and method resolution would be quite lengthy due to the non-trivial semantics of these two language features. We refer instead to the literature [12, 16, 14].

4 Abstract domains

Owners and contexts An object is owned by an applet (or the `JCRE`) thus its owner is given by an AID. We add to this the name of the package (*i.e.*, the context *cf.* Section 2) of the applet, hence the set *Owner* of objects owners was defined above as $Id_p \times Aid$.

The context in which a method is executed is a pair *(Prev,App)* where the first component *Prev* is the last active context before the last context switch and the second component *App* is the *caller* context (*i.e.*, the active context before the call). Formally we define:

$$Context \quad : \quad Owner \times Owner$$

Values We are primarily interested in modelling the object structure and ownership so we abstract primitive values such as booleans and integers to their type.

We adopt a common approach (going back to at least [13]) to model dynamically allocated data in which all objects created by the same `new` instruction are identified by one object. We shall refine this slightly by distinguishing objects by keeping the owner as part of the abstract object. More precisely, a reference to an object (*Obj*) is abstracted into the instruction that created the object and the owner of the object. We suppose we have a special *Null* reference.

$$\begin{array}{rcl} Ref & : & Pc \times Owner \\ Ref & = & Ref_i \cup \{\ Null\ \} \end{array}$$

We have three kinds of values manipulated by the instructions: references, primitive values and applet identifiers.

$$Value \quad = \quad Ref \cup Primitive \cup Aid$$

Concerning the value in memory, we can have a class instance (*Obj*) which contains the name of the class (Id_{ci}), the owner of this instance (*Owner*), boolean flags indicating whether or not it is a `JCRE` entry point or a temporary `JCRE` entry point (*cf.* Section 2) and the set of fields (*Fldv*), which maps a field name to a set of values. For us, the real values manipulated in a program can be object, primitive and AID.

$$\begin{array}{rcl} RValue & = & Obj \cup Primitive \cup Aid \\ Obj & = & Id_{ci} \times Owner \times JCREep \times tJCREep \times Fldv \\ Owner & = & Id_p \times Aid \\ Fldv & = & Id_f \to \mathcal{P}(Value) \end{array}$$

Firewall checks The checks made by the firewall are formalised through a collection of predicates. Covering all bytecode instructions would require eight different predicates; we use two of these predicates during the presentation of the analysis:

- The predicate *AccessInterface?* validates the access to methods of an object.

  $$AccessInterface? : Object \times Object \times Id_i \times E_{ci} \to boolean.$$

  The first object represents the current context, the second is the object on which the call is made and Id_i is the name of the interface which declared the method called. The access is authorized if and only if the context represented by the first object is the context of the `JCRE` or if the contexts of the two objects are the same or if the second object is a `JCRE` entry point or if the class of the second object implements a shareable interface and Id_i extends a shareable interface.

- The predicate *AccessPutstatic?* checks the validity of the access to a static field of a class.

  AccessPutstatic? : Object $\times$ *RValue* $\rightarrow$ *boolean.*

  The object represents the current context that wants to store the value is the static field. The access is only authorized if the context represented by the object is the context of the `JCRE` or if the value is neither a global object nor a temporary `JCRE` entry point.

5 Flow analysis

In this section we describe a data flow analysis to approximate the part of a program's behaviour relevant to security verifications. The main information calculated by our analysis is an approximation of the objects stored in the variables of the program. More precisely, we calculate the following information:

$\mathcal{V}$ $[\![var,m,ctx]\!] \in \mathcal{P}(Value)$	:	the set of values stored in the variable *var* of method *m* when this method is called in context *ctx*,
$\mathcal{SF}$ $[\![Id_{ci}]\!] : Id_f \rightarrow \mathcal{P}(Value)$	:	the values of the static fields of a class name,
mem : *Value* $\rightarrow$ *RValue*	:	an approximation of the memory, it maps a value manipulated by the instruction (like a reference) to a real value (like an object),
$\mathcal{C}$ $[\![m,ctx]\!] \in \mathcal{P}(Ref)$	:	the set of objects on which a call to method *m* in context *ctx* is made.

It is important to analyse methods for each calling context since this is the information available to the firewall at run-time. An analysis that does not exactly model this information would have poor precision. This information serves two purposes: it permits constructing a control flow graph (by resolving which method is called at a given virtual method call) and it makes explicit if an object owned by an applet is stored in a variable accessible by another applet.

An intra-procedural analysis is required in order to approximate the behaviour of each server applet when it receives a request for a shared object. This analysis is orthogonal to the analysis presented in this paper and will not be described here. We shall assume the function:

$$Return\_SIO : \texttt{AID} \times \texttt{AID} \rightarrow \mathcal{P}(Ref_i).$$

It takes the AID of a server and the AID of a client and returns a safe approximation of the set of objects that the server will share with the client (the set that it returns is equal to or bigger than the set returned during the execution).

5.1 Quantified conditional constraints

The analysis will be specified in constraint-based style. We introduce a new type of constraints, the *quantified conditional constraints* (*QCCs*) that can be considered as a constraint scheme from which actual constraints can be generated.

The first kind of constraints used in static analysis is the simple constraint (*SC*). It is used to model the flow and the modification of information. A simple constraint has a form:

$$\text{Expression} \subseteq \text{Variable}$$

An extention of this kind of constraint was used by Palsberg and Schwartzbach [20] for type analysis. They take a simple constraint and add a condition under which the constraint is valid. Such a *conditional* constraint has the form:

$$\text{Class} \in \text{Variable}_1 \rightarrow \text{Expression} \subseteq \text{Variable}_2$$

The Variable_2 have Expression as possible value if and only if Class is a possible value for Variable_1. The simple constraint models an instruction of a method and the condition model the fact that this method can effectivelly be called.

This kind of constraints solves the problem that the constraints to be generated depend on the actual data flow of the program. The solution has the drawback that it has to generate all possible constraints from the outset and then test for each iteration and for each constraint whether it should be taken into consideration. In the following, we propose to generate the constraints set in an incremental fashion where constraints are only added once the data flow analysis has actually established that the constraints will be activated.

We propose to extend this kind of constraints in the following two ways:

- have more possible conditions to model some verification made by the environment (like the firewall checks or the visibility rules),
- produce dynamically the system based on the current value of each variable (instead of generating constraints for all possible values of the domain of the variable).

This new kind of constraints is called *quantified conditional constraints* and has a form:

$$\forall\, v_1, \cdots, v_n \in S_1, \cdots, S_n : \mathit{cond}(v_1, \cdots, v_n) \longrightarrow \mathit{cstr}(v_1, \cdots, v_n)$$

Here, *cstr* is a set of constraints parameterised on $v_1, \cdots, v_n$ and *cond* are conditions on the value $v_1, \cdots, v_n$. Evaluation of such a *QCC* results in a set of constraints for each value $v_1, \cdots, v_n \in S_1, \cdots, S_n$ satisfying the condition *cond*. In our analysis, the *QCC*s have a particular structure, as shown below.

- The set *S*, used in the quantification, can be the value of a variable ($\mathcal{V}\, [\![x,m,ctx]\!]$), the set of objects on which a call is made ($\mathcal{C}\, [\![m,ctx]\!]$), the result of the *Lookup* or a constant set.
- The condition *cond* is a conjunction of conditions. It can be a test on the visibility, a firewall check or a test for membership of a constant set.
- A constraint *const* is a set of simple constraint *SC*. *SC* have a form: $Exp \subseteq Var$. *Exp* can be a variable, a constant set, a dereferencing of the memory or the set of the values of a static field. *Var* can be a variable, a dereferencing of the memory or the set of the value of a static field.

$$\begin{array}{rl}
QCC: & \forall\, \overline{value} \in \overline{S} : cond(\overline{value}) \longrightarrow cstr(\overline{value}) \\
S: & \mathcal{V}\, [\![x,m,ctx]\!] \mid \mathcal{C}\, [\![m,ctx]\!] \mid Lookup\ (Sig,\ Id_{ci},\ Id_{ci},\ E_{ci}) \mid Const\ Set \\
cond: & H_1 \wedge \cdots \wedge H_n \\
Condition\ (H): & CI_Visibility?\ \ (Id_c,Id_{ci}) \mid Method_Visibility?\ \ (Id_c,Id_c,Desc_m) \mid Field_Visibility? \\
 & (Id_c,Id_f) \mid AccessInterface?\ \ (Object,Object,Id_i) \mid AccessPutstatic?\ \ (Object,RValue) \\
 & \mid value \in Const\ Set \\
cstr: & \mathcal{P}(SC) \\
Constraint\ (SC): & Exp \subseteq Var \\
Exp: & Const\ Set \mid \mathcal{V}\, [\![x,m,ctx]\!] \mid \mathcal{SF}\, [\![Id_{ci}]\!](Id_f) \mid \mathcal{C}\, [\![m,ctx]\!] \mid mem(Ref).Fldv(Id_f) \mid Return_SIO \\
 & (Aid,Aid) \\
Var: & \mathcal{V}\, [\![x,m,ctx]\!] \mid \mathcal{SF}\, [\![Id_{ci}]\!](Id_f) \mid \mathcal{C}\, [\![m,ctx]\!] \mid mem(Ref).Fldv(Id_f)
\end{array}$$

5.2 Analysis

The analysis generates, for each method and for an execution context *ctx*, a set of *QCC*s that describes the data flow of the method in this context. The set of constraints for a method is the union of the set of constraints for each instruction. The function to analyse an instruction is:

$$\mathcal{A}_{Inst} : Inst \times Id_m \times Context \rightarrow \mathcal{P}(QCC)$$

This function takes three parameters: the instruction to analyse, the current method, and the context in which the method is executing. An instruction is just a program counter and the bytecode instruction at this address. In the following we define this function for each bytecode instruction.

getstatic The `getstatic` instruction loads a value stored in a static field of a class or interface and stores it into a local variable.
$\mathcal{A}_{Inst}$ *((pp,*$T :=$ `getstatic` f*), m, ctx)* =

$$\begin{array}{l}\forall\,(r) \in \mathcal{C}\,[\![m, ctx]\!] : \mathit{Field\_Visibility?}(mem(r).Id_{ci}, f, E_{ci}) \\ \quad \rightarrow \{\, \mathcal{V}\,[\![T, m, ctx]\!] \supseteq \mathcal{SF}\,[\![\lceil f\rceil^{p.c}]\!](f)\,\}\end{array}$$

The value in the field *C.f* is stored in the local variable *T* if and only if the field exists and the field is visible at instruction *Inst*.

invokeinterface The `invokeinterface` instruction makes a call to an interface method.
$\mathcal{A}_{Inst}$ *((pp,*$T_0 :=$ `invokeinterface` $p\ T_1\ T_2\ \cdots\ T_n\ S_2 :: \cdots :: S_n :: S_{n+1}$*), m, ctx)* =

$$\begin{array}{l}\forall\,(r, o, q) \in \mathcal{C}\,[\![m, ctx]\!] \times \mathcal{V}\,[\![T_1, m, ctx]\!] \times \mathit{Lookup}(sig, mem(o).\mathit{Type}, \lceil p\rceil^{p.c}, E_{ci}) \\ \qquad : \mathit{AccessInterface?}(mem(r), mem(o), \lceil p\rceil^{p.c}, E_{ci}) \\ \qquad \rightarrow \left\{\begin{array}{l}\mathcal{V}\,[\![T_1, m, ctx]\!] \subseteq \mathcal{V}\,[\![P_1, q, ctx']\!], \\ \ldots \\ \mathcal{V}\,[\![T_n, m, ctx]\!] \subseteq \mathcal{V}\,[\![P_n, q, ctx']\!], \\ \mathit{Init\_Var}(E_{ci}(\lceil q\rceil^{p.c}).Mtd((\lceil q\rceil^{m}, S_2 :: \cdots :: S_{n+1})).Varl, q, ctx') \\ \mathcal{C}\,[\![q, ctx']\!] \supseteq \{o\} \\ \mathcal{V}\,[\![T_0, m, ctx]\!] \supseteq \mathcal{V}\,[\![R, q, ctx']\!]\end{array}\right\}\end{array}$$

where we have used the following abbreviations:

$$\begin{array}{rcl} sig &=& (\lceil p\rceil^{m}, S_2 :: \cdots :: S_n) \\ P_1 :: \cdots :: P_n &=& (E_{ci}(\lceil q\rceil^{p.c}).Mtd)((q, S_2 :: \cdots :: S_n)).Param \\ ctx' &=& (Prev, App) \\ App &=& (mem(r).Prop.Id_p, mem(r).Prop) \\ Prev &=& \left\{\begin{array}{l} ctx.Prev \text{ if } ctx.App.Id_p = App.Id_p \\ ctx.App \text{ else}\end{array}\right. \\ R &=& (E_{ci}(\lceil q\rceil^{p.c}).Mtd)((q, S_2 :: \cdots :: S_n)).Res.Id_v \end{array}$$

We begin with the calculation of the set a possible method call and of the context in which the method call will be executed *(Prev,App)*. If the call is accepted by the firewall

$$\mathit{AccessInterface?}\ (mem(r), mem(o), \lceil p\rceil^{p.c}, E_{ci})$$

we add constraints to simulate the call. We create constraints to simulate the transfer of the actual parameters to the formal parameters:

$$\mathcal{V}\,[\![T_i, m, ctx]\!] \subseteq \mathcal{V}\,[\![(Param\_of(q))(i+1), n, (Prev, App)]\!]$$

We add a constraint to retrieve the value returned by the method called

$$\mathcal{V}\,[\![T_0, m, ctx]\!] \supseteq \mathcal{V}\,[\![R, q, (Prev, App)]\!]$$

And we add the object o in $\mathcal{C}\,[\![q,(Prev,App)]\!]$ to indicate that the method q was invoked on this object.

load The `load` instruction loads value contained in a variable and stores it in an other variable.
$\mathcal{A}_{Inst}$ *((pp,T_1 := `load` T_2), m, ctx) =*

$$\rightarrow \left\{ \mathcal{V} [\![T_1, m, ctx]\!] \supseteq \mathcal{V} [\![T_2, m, ctx]\!] \right\}$$

The values contained by the variable T_2 are transfered into the variable T_1.

new The `new` instruction simulates the creation of a new class instance and store a reference on it into a variable.
$\mathcal{A}_{Inst}$ *((pp,T := `new` c), m, ctx) =*

$$\begin{array}{l} \forall\, (r) \in \mathcal{C}\, [\![m, ctx]\!] \textbf{:}\ CI\_Visibility?(mem(r).Id_{ci}, c) \\ \quad \rightarrow \left\{ \mathcal{V} [\![T, m, ctx]\!] \supseteq \{(pp, r.Prop)\} \right\} \end{array}$$

If the class is visible by the instruction, we store in $\mathcal{V}$ [[T,m,ctx]] the reference on the created object.

putstatic The `putstatic` instruction stores a value in a static field.
$\mathcal{A}_{Inst}$ *((pp,`putstatic` f T), m, ctx) =*

$$\begin{array}{l} \forall\, (r, v) \in \mathcal{C}\, [\![m, ctx]\!] \times \mathcal{V}\, [\![T, m, ctx]\!] \\ \quad\quad \textbf{:}\ Field\_Visibility?(mem(r).Id_{ci}, f) \wedge AccessPutstatic?(mem(r), mem(v)) \\ \quad \rightarrow \left\{ \mathcal{SF}\, [\![\lceil f \rceil^{p.c}]\!](f) \supseteq \{v\} \right\} \end{array}$$

The value contained by the variable *T* is stored in the static field *f* of the class $\lceil f \rceil^{p.c}$ if the field is visible by the instruction and if the firewall accepts this access.

store The `store` instruction stores the value contained in variable T_2 in variable T_1.
$\mathcal{A}_{Inst}$ *((pp,T_1 := `store` T_2), m, ctx) =*

$$\rightarrow \left\{ \mathcal{V} [\![T_1, m, ctx]\!] \supseteq \mathcal{V} [\![T_2, m, ctx]\!] \right\}$$

The values contained by the variable T_2 are transfered into the variable T_1.

getPrevCtx The instruction `invokestatic` *getPrevCtx* makes a call on the static method *JCSystem.getPreviousContextAID*.
$\mathcal{A}_{Inst}$ *((pp,T := `invokestatic` getPrevCtx), m, ctx) =*

$$\left\{ \begin{array}{l} \forall\, (r) \in \mathcal{C}\, [\![m, ctx]\!] \textbf{:}\ ctx.App.Id_p = mem(r).Prop.Id_p \\ \quad\quad \rightarrow \left\{ \mathcal{V} [\![T, m, ctx]\!] \supseteq ctx.Prev.AID \right\} \\ \\ \forall\, (r) \in \mathcal{C}\, [\![m, ctx]\!] \textbf{:}\ ctx.App.Id_p \neq mem(r).Prop.Id_p \\ \quad\quad \rightarrow \left\{ \mathcal{V} [\![T, m, ctx]\!] \supseteq ctx.App.AID \right\} \end{array} \right\}$$

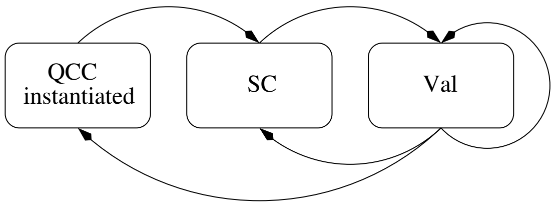
The method *getPrecCtx* can be used to find the AID of the active applet before the last context switch. The first constraint is used when the active context is the context of the caller, in this case they have the same previous context. The second constraint when the active context and the context of the caller are not the same, in this case the previous context is the context of the caller.

ifAID The *QCC*used in this construct is the one analysed for the *BCinst* instruction. A condition is added such that the constraints are only generated if the condition in the test is true.
Let $\mathcal{A}_{Inst}$ *((pp,$BCinst$), m, ctx)* = $\forall\, \bar{v} \in \bar{E}$: *cond* $\longrightarrow \{C\}$. Then
$\mathcal{A}_{Inst}$ *((pp,*`ifAID`$T \in SBCinst$*), m, ctx)* =

$$\forall\, (\bar{v}, a) \in \bar{E} \times \mathcal{V}\, [\![T, m, ctx]\!] : cond \wedge a \in S \longrightarrow \{\, C \,\}$$

6 Resolution

The resolution of quantified conditional constraints can be done iteratively as an ordinary fixpoint computation. The main difference is that the set of constraints and the value of variables in the constraints evolve together. Hence, the iteration sequence consists of triples *(qcc,sc,val)* where *qcc* is the current set of quantified conditional constraints instantiated for particular contexts, *sc* is the current set of simple constraints and *val* is a valuation that to each variable associates its current value.



Suppose given a program *P* with a set of applets (*Aplt*) and a set of methods (*Meth*). Let *Q* be the set of (uninstantiated) *QCC*s obtained by analysing *P*. During the resolution of *Q*, we compute the new set of instantiated *QCC*s, $\mathcal{P}(QCC)$, with the function $Eval_{QCC}$, the new set of simple constraints *SC*, $\mathcal{P}(SC)$, with the function $Eval_{SC}$ and the new valuation *Val* with the function $Eval_{Val}$, as defined below.

The function $Eval_{QCC}$ uses the current valuation to instantiate the *QCC*s in the set *Q* and adds the corresponding constraints to the current set of constraints. This is where the resolution becomes context-sensitive: if a method is not called in a particular context, no constraints for this method in that context are generated.

$$Eval_{QCC}: \mathcal{P}(QCC) \times Val \rightarrow \mathcal{P}(QCC)$$
$$Eval_{QCC}\ (qcc,val) = qcc \cup \bigcup_{\substack{m \in Meth \\ ctx \in Context}} \{\ ctr \mid o \in \mathcal{C}\, [\![m,ctx]\!] \wedge ctx' = CalcCtx\,(o,ctx,val) \wedge ctr \in \mathcal{A}_{Mtd}\,(m,ctx')\ \}$$

where the function for calculating the context of the call is given by

$$CalcCtx : Ref_i \times Context \times Val \rightarrow Context$$
$$CalcCtx\,(r,c,v) = (Prev, App)\ \textbf{where}$$
$$App = (v(mem)(r).Prop.Id_p,\ v(mem)(r).Prop)$$
$$Prev = \begin{cases} c.Prev\ \textbf{if}\ c.App.Id_p = App.Id_p \\ c.App\ \textbf{otherwise} \end{cases}$$

The function $Eval_{SC}$ uses the current valuation to verify the condition for each constraint in the set of instantiated *QCC*s and adds the corresponding simple constraints to the current set of constraints. This evaluation permits to restrict the production of the simple constraints that model the effect of an instruction that it "executed". We use the notation $[\![\text{Exp}]\!]_V$ to denote the evaluation of the expression *Exp* with the values contained by the valuation *V*.

$$Eval_{SC}\colon \mathcal{P}(QCC) \times \mathcal{P}(SC) \times Val \to \mathcal{P}(SC)$$
$$Eval_{SC}\ (qcc,sc,val) = sc \cup \{\ ctr[\overline{v}/\overline{x}] \mid \forall\, \overline{x} \in \overline{X} : cond \longrightarrow ctr \in qcc \wedge \overline{v} \in [\![\overline{X}]\!]_{val} \wedge cond[\overline{v}/\overline{x}]\ \}$$

The function $Eval_{Val}$ is the standard evaluation function associated to a constraint set. For every constraint $exp \subseteq var$ in the current constraint set *cs* we evaluate the expression with the current evaluation and add the new value in *val(var)*.

$$Eval_{Val}\colon \mathcal{P}(SC) \times Val \to Val$$
$$Eval_{Val}\ (sc,val) = val[var \mapsto val(var) \sqcup [\![exp]\!]_{val} \mid exp \subseteq var \in sc]$$

ALGORITHM 1

$\mathrm{Q} := \bigcup_{A \in Aplt} \mathcal{A}_{Class}\ (A)$;
$\mathrm{qcc}' := \mathcal{A}_{Class}\ (JCRE)_{(JCRE,JCRE)}$;
sc, sc$'$, qcc := $\emptyset$;
val := $\perp$;
val' := val_0[9] ;
while qcc $\neq$ qcc$'$ **or** sc $\neq$ sc$'$ **or** val $\neq$ val$'$ **do**
    qcc := qcc$'$; sc := sc$'$; val := val$'$;
    qcc$'$:= $Eval_{QCC}$ *(qcc,val)* ;
    sc$'$:= $Eval_{SC}$ *(qcc,sc,val)* ;
    val$'$:= $Eval_{Val}$ *(sc,val)* ;
endwhile

END

Proposition 6.1 *Algorithm 1 terminates with a correct solution to Q.*

The proof of Proposition 6.1 is an extension of the standard argument based on Tarski's theorem [24, 11] to take into account that some of the systems evolve (in a monotonic fashion!) during the computation. The formal proof can be found in [17].

Establishing a start state for the iteration requires special attention in Java Card because there is no `main` to initialize the analysis. The sequence of operation is given by the `JCRE` and the user. We model this interaction with the card by adding an artificial `JCRE` applet that is analysed like the other applets. For the `JCRE` we know its context (it is (`JCRE`,`JCRE`)) which permits the algorithm to produce the initial set of instantiated *QCC*s. The initial valuation val_0 links each element with its default value. For each $\mathcal{V}\ [\![\text{x,m,ctx}]\!]$ and $\mathcal{C}\ [\![\text{m,ctx}]\!]$ the default value is $\emptyset$. For each $\mathcal{SF}\ [\![Id_{ci}]\!]$ the default value is the function which links each static field of Id_{ci} with its default value ($\emptyset$ for a reference and $\{P\}$ for a primitive *P*). Finally, we initialize the abstract memory (*mem*) with the undefined abstract objects for each abstract reference.

7 An example analysis

In figure 2, we present a variation of the example given in section 2.1, in which the firewall and Alice can not prevent the flow of the Alice secret to Charlie. Here, Bob implements a shareable object and passes a

[9] The definition of the initial value val_0 comes after the algorithm.

`public class Bob extends Applet`
`      implements MSI2{`
`   private static MSI AliceObj;`
`private void bar () {`
`   AliceObj=(MSI) getSIO (AliceAID); }`
`public Secret foo2 () {`
`   return AliceObj.foo (); } }`
	`public class Charlie extends Applet {`
	`   private static MSI2 BobObj;`
	`   private static Secret AliceSecret;`
	`private void bar () {`
	`   BobObj=(MSI2) getSIO (BobAID); }`
	`private void foo3 () {`
	`   AliceSecret=BobObj.foo2 (); } }`

Figure 2: An example of illegal object flow

reference to it to Charlie. In this case, the invoke at *Alice.foo* is valid at runtime, because for Alice the caller is always Bob. Here, we only presents the transformation of this example in our language in the figure 3. The associated quantified conditional constraints as well as the complete resolution are given in the appendix A. During the resolution, each "variable" received the possible values taht it can contained. In this exemple, the important value is the secret of Alice (represented by the reference $(p_1, AliceAID)$) and the important variable is the static field `AliceSecret` of Charlie. The resolution give, as a part of the global solution, the following possible value for the static field of Charlie:

$$(\mathbf{p_1}, \mathbf{AliceAID}) \in \mathcal{SF}\ [\![\mathbf{Charlie}]\!](\mathbf{Charlie}.\mathbf{AliceSecret})$$

This result prove that there is an illegal object flow with the secret of Alice.

```
public class Alice extends Applet implements MSI {
   private Secret ObjectSecret;
   public Secret foo () {
      AID Client;
      Secret Response;
      1:T1:=invokestatic getPrevCxt
      2:Client:=store T1
      3:ifAID Client ∈ {BobAID} T2:=getstatic Alice.ObjectSecret
      4:Response:=store T2
      5:Alice.foo_Ret:=load Response
      return Alice.foo_Ret } }
```

`public class Bob extends Applet`
`      implements MSI2{`
`   private static MSI AliceObj;`
`public Secret foo2 () {`
`   6:T3:=getstatic Bob.AliceObj`
`   7:T4:=invokeinterface MSI.foo T3`
`   8:Bob.foo2_Ret:=store T4`
`   return Bob.foo2_Ret } }`
	`public class Charlie extends Applet {`
	`   private static MSI2 BobObj;`
	`   private static Secret AliceSecret;`
	`private void foo3 () {`
	`   9:T5:=getstatic Charlie.BobObj`
	`   10:T6:=invokeinterface MSI2.foo2 T5`
	`   11:putstatic Charlie.AliceSecret T6} }`

Figure 3: The translation of the three methods of the example in in our language

8 Related works

The formalisation of the Java Card firewall has been the object of several works. Motré [19] has formalized the firewall with the B method. She defines a machine for the firewall and an operation for each check of the firewall. This modeling provides a formal description of the firewall that is used to ensure that the firewall verifications are sufficient to fulfill the security policy. In addition, successive refinements lead to a reference implementation of the firewall. More traditional operational semantics for modeling the firewall checks have been given by Éluard *et al.* [18] and Siveroni *et al.* [22] show how to integrate this into an operational semantics for Java Card. For the modeling of the `JCRE` it is necessary to be able to "execute" the differents applets. We choose to follow the approach used by Attali *et al.* [3, 4] and model the `JCRE` by a program. With this approach, we can adapt the JCRE to obtain either exactly the execution we want or all possible executions.

Two works on the verification of applet sharing on Java Card are closely related to ours. Bieber *et al.* [8, 7], as part of the Pacap project [2], have defined an analysis of Java Card applets which can detect illegal information flow. Their approach is based on three elements: an abstraction of values of variables into a *level* that describes the sharing of the value, an invariant that is a sufficient condition the security property to hold and a model checker to verify the invariant. A lattice of levels is used to represent the sharing of objects. If an applet *A* is allowed to share some information with an applet *B*, the level *A*+*B* is entered into the lattice specifying the security policy. Each applet is represented by a call graph and each call graph is transformed into an SMV model. To work with a shareable object, an applet must call an interface method so only call graphs which include an interface method are taken into account. The invariant together with the control flow graphs are given to the SMV model checker for verification. The work presented here complements their work by providing a precise description of how these control and data flow graphs can be calculated, taking into account the firewall and the different calling contexts.

The analysis proposed by Caromel, Henrio and Serpette [9] has as aim to signal whether a security exception might (or will definitely) be raised by the firewall at execution of a set of applets. The analysis thus shares objectives with ours and calculates the same type of information. The differences between the analyses lie in the precision. Caromel *et al.* have opted for a simple, flow-insensitive analysis whereas we can obtain some flow sensitity through the choice of local variables in our three-address byte code. Instead of modelling the memory state explicitly, they use an alias analysis to track side effects of assigments. The control flow analysis in their analysis is a simple class hierarchy analysis, in contrast to our context-sensitive flow analysis. Indeed, their analysis does not analyse methods separately for each calling context and hence would not be able to deal with the call stack inspection as well as our analysis. Thus, the two analyses can be seen as two extremes of the design space for flow analysis for Java Card.

The quantified conditional constraints (*QCC*s) introduced in Section 5.1 are an extension of the conditional constraints (originally due to Reynolds [21]) that are used in the object-oriented type analysis defined by Palsberg and Schwartzbach [20]. In this analysis, conditions of the form $C \in \mathcal{V}(X)$ are used to guard the constraints generated from class C such that these are only evaluated when class C is actually used. However, it is still necessary to generate the constraints for every class in the hierarchy which leads to scalability problems. The *QCC*s, on the other hand, generates these constraints *on demand*: when the analysis discovers that a certain class or method is used, the corresponding constraints are generated and added to the current set of constraints.

9 Conclusions and future work

In this paper, we have described a static analysis to compute an approximation of the values of the different variables used in a Java Card application. The static analysis uses an extension of the traditional constraint-based techniques to specifiy the analysis and relies on a demand-driven iteration scheme for generating and solving the constraints. The resolution gives us a description of the flow of objects between applets and of the behavior of the firewall. With this information, we construct the control flow graph associated with these applets. The security properties verified here are relatively simple and it is possible to formulate more sophisticated properties, *e.g.*, by using a suitable temporal logic [15] or finite automata [17].

In this article, we have not taken into account the flow of primitive values so we cannot detect if a secret primitive value is revealed. To verify this, owner information must be added to primitive values and primitive operations must be adjusted to calculate the possible owners depending on the values used in the operation as well as the applet which does the operation. Likewise, arrays can be taken into account using our formalism but we haven't detailed this here for lack of space—details about this extension can be found in the first author's PhD thesis [17].

Finally, for the moment the analysis does not take into account exceptions other than security exceptions. With the current abstraction of the primitive values it is clear that exceptions related to *e.g.*, array access (index-out-of-bound exceptions) can only be dealt with in a very approximate fashion. Exceptions form an integral part of the control-flow of an application so progress in this direction is highly desirable.

References

[1] Java Card 2.1.1. `http://java.sun.com/products/javacard/javacard21.html`, 2001.

[2] The PACAP project. `http://www.gemplus.com/smart/r_d/projects/pacap.htm`, 2001.

[3] Isabelle Attali, Denis Caromel, Carine Courbis, Ludovic Henrio, and Henrik Nilsson. Smart tools for Java Cards. In Josep Domingo-Ferrer, David Chan, and Anthony Watson, editors, *Smart Card Research and Advanced Application Conf. (CARDIS)*. Kluwer Academic Publishers, September 2000.

[4] Isabelle Attali, Denis Caromel, Carine Courbis, Ludovic Henrio, and Henrik Nilsson. An integrated development environment for Java Card. *Special issue on Smart Cards of the Journal Computer Networks*, 36(4):391–405, July 2001.

[5] Peter Bertelsen. Semantics of Java Byte Code. Technical report, Dep. of Information Technology, Technical University of Denmark, March 1997. Home page `http://www.dina.kvl.dk/~pmb/`, 2001.

[6] Peter Bertelsen. Dynamic semantics of Java bytecode. In *Workshop on Principles on Abstract Machines*, September 1998. Home page `http://www.dina.kvl.dk/~pmb/`, 2001.

[7] Pierre Bieber, Jacques Cazin, Pierre Girard, Jean-Louis Lanet, Virginie Wiels, and Guy Zanon. Checking secure interactions of smart cards applets. In *European Symp. on Research in Computer Security (ESORICS)*, volume 1895 of *Lecture Notes in Computer Science (LNCS)*, pages 1–16. Springer-Verlag, 2000.

[8] Pierre Bieber, Jacques Cazin, Abdellah El Marouani, Pierre Girard, Jean-Louis Lanet, Virginie Wiels, and Guy Zanon. The PACAP prototype : a tool for detecting Java Card illegal flow. In Isabelle Attali and Thomas Jensen, editors, *Java Card Workshop (JCW)*, volume 2041 of *Lecture Notes in Computer Science (LNCS)*, September 2000.

[9] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Context inference for static analysis of java card object sharing. In Isabelle Attali and Thomas Jensen, editors, *Smart Card Programming and Security (ESMART)*, volume 2140, September 2001.

[10] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, 2000.

[11] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Conf. on Functional Programming and Computer Architecture (FPCA)*, pages 170–181. ACM Press, June 1995.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000. 896 pages, `http://java.sun.com/docs/books/jls/index.html`, 2001.

[13] Paul R. Hudak. A semantic model of reference counting and its abstraction. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 3, pages 45–62. Ellis Horwood series in computer and their applications, 1987.

[14] Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Security and dynamic class loading in Java: A formalisation. In *Int. Conf. on Computer Languages (ICCL)*, pages 4–15, May 1998.

[15] Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Verification of control flow based security policies. In *Symp. on Security and Privacy*, pages 89–103. IEEE Computer Society Press, May 1999.

[16] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999. `http://java.sun.com/docs/books/vmspec/index.html`, 2001.

[17] Marc Éluard. *Analyse de sécurité pour la certification d'applications Java Card*. PhD thesis, Université de Rennes 1, December 2001. N. d'ordre : 2614.

[18] Marc Éluard, Thomas Jensen, and Ewen Denney. An operational semantics of the java card firewall. In Isabelle Attali and Thomas Jensen, editors, *Smart Card Programming and Security (ESMART)*, volume 2140, September 2001.

[19] Stéphanie Motré. Modélisation et implémentation formelle de la politique de sécurité dynamique de la Java Card. In *Approches Formelles dans l'Assistance au Développement de Logiciel (AFADL)*, pages 158–172. LSR/IMAG, January 2000.

[20] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[21] John C. Reynolds. Automatic computation of data set definitions. In *Information Processing*, volume 1, pages 456–461. North-Holland, August 1969.

[22] Igor Siveroni, Thomas Jensen, and Marc Éluard. A formal specification of the java card applet firewall. In Hanne Riis Nielson, editor, *Nordic Workshop on Secure IT-Systems (NordSec)*, Technical Report IMM-TR-2001-14. Technical University of Denmark, November 2001.

[23] Soot: a Java optimization framework. `http://www.sable.mcgill.ca/soot/`, 2001.

[24] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, June 1955.

A The complete analysis of the example in Figure 3

To facilitate the comprehension, an owner is noted with the name of the applet. For example, *Alice* denotes the pair *(AlicePackage, AliceAID)* and *JCRE* denotes *(JCRE,JCRE)*. The number of a constraint is the number of the line for which the constraint is created. An evaluation of a constraint is denoted with adding a prime on its name. A part of the solution is given by a constraint with no number. When a *QCC* produces more than one constraints, we add a letter to its number.

The analysis of the example in Figure 3 yields the following quantified conditional constraints:

1_1 $\forall(r) \in \mathcal{C}\,[\![Alice.foo, ctx]\!]\text{:}ctx.App.Id_p = mem(r).Prop.Id_p \longrightarrow \{\ \mathcal{V}\,[\![T_1, Alice.foo, ctx]\!] \supseteq ctx.Prev.AID\ \}$

1_2 $\forall(r) \in \mathcal{C}\,[\![Alice.foo, ctx]\!]\text{:}ctx.App.Id_p \neq mem(r).Prop.Id_p \longrightarrow \{\ \mathcal{V}\,[\![T_1, Alice.foo, ctx]\!] \supseteq ctx.App.AID\ \}$

2 $\longrightarrow \{\ \mathcal{V}\,[\![Client, Alice.foo, ctx]\!] \supseteq \mathcal{V}\,[\![T_1, Alice.foo, ctx]\!]\ \}$

3 $\forall(r,c) \in \mathcal{C}\,[\![Alice.foo, ctx]\!] \times \mathcal{V}\,[\![Client, Alice.foo, ctx]\!]\text{:}$
$Field\_Visibility?(mem(r).Id_{ci}, Alice.ObjetSecret, E_{ci}) \wedge c \in \{BobAID\} \longrightarrow$
$\{\ \mathcal{V}\,[\![T_2, Alice.foo, ctx]\!] \supseteq \mathcal{SF}\,[\![Alice]\!](Alice.ObjectSecret)\ \}$

4 $\longrightarrow \{\ \mathcal{V}\,[\![Response, Alice.foo, ctx]\!] \supseteq \mathcal{V}\,[\![T_2, Alice.foo, ctx]\!]\ \}$

5 $\longrightarrow \{\ \mathcal{V}\,[\![Alice.foo\_Ret, Alice.foo, ctx]\!] \supseteq \mathcal{V}\,[\![Response, Alice.foo, ctx]\!]\ \}$

6 $\forall(r) \in \mathcal{C}\,[\![Bob.foo2, ctx]\!]\text{:}$
$Field\_Visibility?(mem(r).Id_{ci}, Bob.AliceObj, E_{ci}) \longrightarrow$
$\{\ \mathcal{V}\,[\![T_3, Bob.foo2, ctx]\!] \supseteq \mathcal{SF}\,[\![Bob]\!](Bob.AliceObj)\ \}$

7 $\forall(r,o,q) \in \mathcal{C}\,[\![Bob.foo2, ctx]\!] \times \mathcal{V}\,[\![T_3, Bob.foo2, ctx]\!] \times$
$Lookup((MSI.foo, Secret), mem(o).Type, MSI, E_{ci})\text{:}$
$AccessInterface?(mem(r), mem(o), MSI, E_{ci}) \longrightarrow$
$$\left\{ \begin{array}{l} \mathcal{V}\,[\![T_4, Bob.foo2, ctx]\!] \supseteq \mathcal{V}\,[\![q\_Result, q, ctx']\!] \\ \mathcal{C}\,[\![q, ctx']\!] \supseteq \{o\} \end{array} \right\}$$

8 $\longrightarrow \{\ \mathcal{V}\,[\![Bob.foo2\_Ret, Bob.foo2, ctx]\!] \supseteq \mathcal{V}\,[\![T_4, Bob.foo2, ctx]\!]\ \}$

9 $\forall(r) \in \mathcal{C}\,[\![Charlie.foo3, ctx]\!]\text{:}$
$Field\_Visibility?(mem(r).Id_{ci}, Charlie.BobObj, E_{ci}) \longrightarrow$
$\{\ \mathcal{V}\,[\![T_5, Charlie.foo3, ctx]\!] \supseteq \mathcal{SF}\,[\![Charlie]\!](Charlie.BobObj)\ \}$

10 $\forall(r,o,q) \in \mathcal{C}\,[\![Charlie.foo3, ctx]\!] \times \mathcal{V}\,[\![T_5, Charlie.foo3, ctx]\!] \times$
$Lookup((MSI2.foo2, Secret), mem(o).Type, MSI2, E_{ci})\text{:}$
$AccessInterface?(mem(r), mem(o), MSI2, E_{ci}) \longrightarrow$
$$\left\{ \begin{array}{l} \mathcal{V}\,[\![T_6, Charlie.foo3, ctx]\!] \supseteq \mathcal{V}\,[\![q\_Result, q, ctx']\!] \\ \mathcal{C}\,[\![q, ctx']\!] \supseteq \{o\} \end{array} \right\}$$

11 $\forall(r,v) \in \mathcal{C}\,[\![Charlie.foo3, ctx]\!] \times \mathcal{V}\,[\![T_6, Charlie.foo3, ctx]\!]\text{:}$
$Field\_Visibility?(mem(r).Id_{ci}, Charlie.AliceSecret, E_{ci}) \wedge$
$AccessPutstatic?(mem(r), mem(v)) \longrightarrow$
$\{\ \mathcal{SF}\,[\![Charlie]\!](Charlie.AliceSecret) \supseteq \mathcal{V}\,[\![T_6, Charlie.foo3, ctx]\!]\ \}$

We assume that the JCRE calls a method of Charlie (like *process*) and this method calls the method *Charlie.foo3*. This call can be transformed into a constraint and solved [10]:

$\mathcal{C}\,[\![\mathbf{Charlie.foo3}, (\mathbf{JCRE}, \mathbf{Charlie})]\!] \supseteq \{(\mathbf{p_4}, \mathbf{CharlieAID})\}$

We assume that the beginning of the resolution has found the following values:

$\mathcal{SF}\,[\![\mathbf{Alice}]\!](\mathbf{Alice.ObjectSecret}) \supseteq \{(\mathbf{p_1}, \mathbf{AliceAID})\}$
$\mathcal{SF}\,[\![\mathbf{Bob}]\!](\mathbf{Bob.AliceObj}) \supseteq \{(\mathbf{p_2}, \mathbf{AliceAID})\}$
$\mathcal{SF}\,[\![\mathbf{Charlie}]\!](\mathbf{Charlie.BobObj}) \supseteq \{(\mathbf{p_3}, \mathbf{BobAID})\}$

The constraints 9, 10 and 11 can be instantiated with the context *(JCRE,Charlie)*.

[10] This call is made on the instance applet *Charlie*

$9'$ $\forall(\mathsf{r}) \in \mathcal{C}\,[\![\mathsf{Charlie.foo3}, (\mathsf{JCRE}, \mathsf{Charlie})]\!]$:
$\quad Field\_Visibility?(mem(\mathsf{r}).Id_{ci}, \mathsf{Charlie.BobObj}, E_{\mathsf{ci}}) \longrightarrow$
$$\left\{ \begin{array}{l} \mathcal{V}\,[\![\mathsf{T}_5, \mathsf{Charlie.foo3}, (\mathsf{JCRE}, \mathsf{Charlie})]\!] \supseteq \\ \mathcal{SF}\,[\![\mathsf{Charlie}]\!](\mathsf{Charlie.BobObj}) \end{array} \right\}$$

$10'$ $\forall(\mathsf{r}, \mathsf{o}, \mathsf{q}) \in \mathcal{C}\,[\![\mathsf{Charlie.foo3}, (\mathsf{JCRE}, \mathsf{Charlie})]\!] \times$
$\mathcal{V}\,[\![\mathsf{T}_5, \mathsf{Charlie.foo3}, (\mathsf{JCRE}, \mathsf{Charlie})]\!] \times$
$Lookup((\mathsf{MSI2.foo2}, \mathsf{Secret}), mem(\mathsf{o}).\mathsf{Type}, \mathsf{MSI2}, E_{\mathsf{ci}})$:
$\quad AccessInterface?(mem(\mathsf{r}), mem(\mathsf{o}), \mathsf{MSI2}, E_{\mathsf{ci}}) \longrightarrow$
$$\left\{ \begin{array}{l} \mathcal{V}\,[\![\mathsf{T}_6, \mathsf{Charlie.foo3}, (\mathsf{JCRE}, \mathsf{Charlie})]\!] \supseteq \mathcal{V}\,[\![\mathsf{q\_Result}, \mathsf{q}, \mathsf{ctx}']\!] \\ \mathcal{C}\,[\![\mathsf{q}, \mathsf{ctx}']\!] \supseteq \{\mathsf{o}\} \end{array} \right\}$$

$11'$ $\forall(\mathsf{r}, \mathsf{v}) \in \mathcal{C}\,[\![\mathsf{Charlie.foo3}, (\mathsf{JCRE}, \mathsf{Charlie})]\!] \times$
$\mathcal{V}\,[\![\mathsf{T}_6, \mathsf{Charlie.foo3}, (\mathsf{JCRE}, \mathsf{Charlie})]\!]$:
$\quad Field\_Visibility?(mem(\mathsf{r}).Id_{ci}, \mathsf{Charlie.AliceSecret}, E_{\mathsf{ci}}) \wedge$
$\quad AccessPutstatic?(mem(\mathsf{r}), mem(\mathsf{v})) \longrightarrow$
$$\left\{ \begin{array}{l} \mathcal{SF}\,[\![\mathsf{Charlie}]\!](\mathsf{Charlie.AliceSecret}) \supseteq \\ \mathcal{V}\,[\![\mathsf{T}_6, \mathsf{Charlie.foo3}, (\mathsf{JCRE}, \mathsf{Charlie})]\!] \end{array} \right\}$$

The condition for the constraint $9'$ is verified so the evaluation produce a simple constraint and its evaluation produce the result:

$9''$ $\mathcal{V}\,[\![\mathtt{T}_5, \mathtt{Charlie.foo3}, (\mathtt{JCRE}, \mathtt{Charlie})]\!] \supseteq \mathcal{SF}\,[\![\mathtt{Charlie}]\!](\mathtt{Charlie.BobObj})$
$\mathcal{V}\,[\![\mathbf{T_5}, \mathbf{Charlie.foo3}, (\mathbf{JCRE}, \mathbf{Charlie})]\!] \supseteq \{(\mathbf{p_3}, \mathbf{BobAID})\}$

We evaluate $10'$ to obtain two simple constraints and the result for the second one:

$10''_a$ $\mathcal{V}\,[\![\mathtt{T}_6, \mathtt{Charlie.foo3}, (\mathtt{JCRE}, \mathtt{Charlie})]\!] \supseteq \mathcal{V}\,[\![\mathtt{Bob.foo2\_Ret}, \mathtt{Bob.foo2}, (\mathtt{JCRE}, \mathtt{Charlie})]\!]$
$10''_b$ $\mathcal{C}\,[\![\mathtt{Bob.foo2}, (\mathtt{JCRE}, \mathtt{Charlie})]\!] \supseteq \{(\mathtt{p}_3, \mathtt{BobAID})\}$
$\mathcal{C}\,[\![\mathbf{Bob.foo2}, (\mathbf{JCRE}, \mathbf{Charlie})]\!] \supseteq \{(\mathbf{p_3}, \mathbf{BobAID})\}$

The constraints 6, 7 and 8 can be instantiated with the context *(JCRE,Charlie)*.

$6'$ $\forall(\mathsf{r}) \in \mathcal{C}\,[\![\mathsf{Bob.foo2}, (\mathsf{JCRE}, \mathsf{Charlie})]\!]$:
$\quad Field\_Visibility?(mem(\mathsf{r}).Id_{ci}, \mathsf{Bob.AliceObj}, E_{\mathsf{ci}}) \longrightarrow$
$\quad\quad \{\ \mathcal{V}\,[\![\mathsf{T}_3, \mathsf{Bob.foo2}, (\mathsf{JCRE}, \mathsf{Charlie})]\!] \supseteq \mathcal{SF}\,[\![\mathsf{Bob}]\!](\mathsf{Bob.AliceObj})\ \}$

$7'$ $\forall(\mathsf{r}, \mathsf{o}, \mathsf{q}) \in \mathcal{C}\,[\![\mathsf{Bob.foo2}, (\mathsf{JCRE}, \mathsf{Charlie})]\!] \times \mathcal{V}\,[\![\mathsf{T}_3, \mathsf{Bob.foo2}, (\mathsf{JCRE}, \mathsf{Charlie})]\!] \times$
$Lookup((\mathsf{MSI.foo}, \mathsf{Secret}), mem(\mathsf{o}).\mathsf{Type}, \mathsf{MSI}, E_{\mathsf{ci}})$:
$\quad AccessInterface?(mem(\mathsf{r}), mem(\mathsf{o}), \mathsf{MSI}, E_{\mathsf{ci}}) \longrightarrow$
$$\left\{ \begin{array}{l} \mathcal{V}\,[\![\mathsf{T}_4, \mathsf{Bob.foo2}, (\mathsf{JCRE}, \mathsf{Charlie})]\!] \supseteq \mathcal{V}\,[\![\mathsf{q\_Result}, \mathsf{q}, \mathsf{ctx}']\!] \\ \mathcal{C}\,[\![\mathsf{q}, \mathsf{ctx}']\!] \supseteq \{\mathsf{o}\} \end{array} \right\}$$

$8'$ $\longrightarrow \{\ \mathcal{V}\,[\![\mathsf{Bob.foo2\_Ret}, \mathsf{Bob.foo2}, (\mathsf{JCRE}, \mathsf{Charlie})]\!] \supseteq \mathcal{V}\,[\![\mathsf{T}_4, \mathsf{Bob.foo2}, (\mathsf{JCRE}, \mathsf{Charlie})]\!]\ \}$

The condition for the constraint $6'$ is verified and the constraint $8'$ has no condition so the evaluation produce two simple constraints:

$6''$ $\mathcal{V}\,[\![\mathtt{T}_3, \mathtt{Bob.foo2}, (\mathtt{JCRE}, \mathtt{Charlie})]\!] \supseteq \mathcal{SF}\,[\![\mathtt{Bob}]\!](\mathtt{Bob.AliceObj})$
$8''$ $\mathcal{V}\,[\![\mathtt{Bob.foo2\_Ret}, \mathtt{Bob.foo2}, (\mathtt{JCRE}, \mathtt{Charlie})]\!] \supseteq \mathcal{V}\,[\![\mathtt{T}_4, \mathtt{Bob.foo2}, (\mathtt{JCRE}, \mathtt{Charlie})]\!]$

We evaluate $6''$ to obtain:

$\mathcal{V}\,[\![\mathbf{T_3}, \mathbf{Bob.foo2}, (\mathbf{JCRE}, \mathbf{Charlie})]\!] \supseteq \{(\mathbf{p_2}, \mathbf{AliceAID})\}$

We evaluate $7'$ to obtain two simple constraints and the result for the second one:

$7''_a$ $\mathcal{V}\,[\![\mathtt{T}_4, \mathtt{Bob.foo2}, (\mathtt{JCRE}, \mathtt{Charlie})]\!] \supseteq \mathcal{V}\,[\![\mathtt{Alice.foo\_Ret}, \mathtt{Alice.foo}, (\mathtt{Charlie}, \mathtt{Bob})]\!]$
$7''_b$ $\mathcal{C}\,[\![\mathtt{Alice.foo}, (\mathtt{Charlie}, \mathtt{Bob})]\!] \supseteq \{(\mathtt{p}_2, \mathtt{AliceAID})\}$
$\mathcal{C}\,[\![\mathbf{Alice.foo}, (\mathbf{Charlie}, \mathbf{Bob})]\!] \supseteq \{(\mathbf{p_2}, \mathbf{AliceAID})\}$

The constraints 1_1, 1_2, 2, 3, 4 and 5 can be instantiated with the context *(Charlie,Bob)*.

$1'_1$ $\forall(\mathsf{r}) \in \mathcal{C}\,[\![\mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!]\mathbf{:}(\mathsf{Charlie}, \mathsf{Bob}).\mathsf{App}.Id_p = mem(\mathsf{r}).\mathsf{Prop}.Id_p \longrightarrow$
$\{\ \mathcal{V}\,[\![\mathsf{T}_1, \mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!] \supseteq (\mathsf{Charlie}, \mathsf{Bob}).\mathsf{Prev.AID}\ \}$

$1'_2$ $\forall(\mathsf{r}) \in \mathcal{C}\,[\![\mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!]\mathbf{:}(\mathsf{Charlie}, \mathsf{Bob}).\mathsf{App}.Id_p \neq mem(\mathsf{r}).\mathsf{Prop}.Id_p \longrightarrow$
$\{\ \mathcal{V}\,[\![\mathsf{T}_1, \mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!] \supseteq (\mathsf{Charlie}, \mathsf{Bob}).\mathsf{App.AID}\ \}$

$2'$ $\longrightarrow \{\ \mathcal{V}\,[\![\mathsf{Client}, \mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!] \supseteq \mathcal{V}\,[\![\mathsf{T}_1, \mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!]\ \}$

$3'$ $\forall(\mathsf{r}, \mathsf{c}) \in \mathcal{C}\,[\![\mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!] \times \mathcal{V}\,[\![\mathsf{Client}, \mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!]\mathbf{:}$
$Field\_Visibility?(mem(r).Id_{ci}, Alice.ObjetSecret, E_{ci}) \wedge \mathsf{c} \in \{\mathsf{BobAID}\} \longrightarrow$
$\{\ \mathcal{V}\,[\![\mathsf{T}_2, \mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!] \supseteq \mathcal{SF}\,[\![\mathsf{Alice}]\!](\mathsf{Alice.ObjectSecret})\ \}$

$4'$ $\longrightarrow \{\ \mathcal{V}\,[\![\mathsf{Response}, \mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!] \supseteq \mathcal{V}\,[\![\mathsf{T}_2, \mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!]\ \}$

$5'$ $\longrightarrow \{\ \mathcal{V}\,[\![\mathsf{Alice.foo\_Ret}, \mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!] \supseteq \mathcal{V}\,[\![\mathsf{Response}, \mathsf{Alice.foo}, (\mathsf{Charlie}, \mathsf{Bob})]\!]\ \}$

The constraints $2'$, $4'$ and $5'$ have no condition and the constraint $1'_2$ has a valid condition. For the constraint $1'_1$, its condition is false. We obtain four simple constraints:

$1''_2$ $\mathcal{V}\,[\![\mathtt{T}_1, \mathtt{Alice.foo}, (\mathtt{Charlie}, \mathtt{Bob})]\!] \supseteq (\mathtt{Charlie}, \mathtt{Bob}).\mathtt{App.AID}$

$2''$ $\mathcal{V}\,[\![\mathtt{Client}, \mathtt{Alice.foo}, (\mathtt{Charlie}, \mathtt{Bob})]\!] \supseteq \mathcal{V}\,[\![\mathtt{T}_1, \mathtt{Alice.foo}, (\mathtt{Charlie}, \mathtt{Bob})]\!]$

$4''$ $\mathcal{V}\,[\![\mathtt{Response}, \mathtt{Alice.foo}, (\mathtt{Charlie}, \mathtt{Bob})]\!] \supseteq \mathcal{V}\,[\![\mathtt{T}_2, \mathtt{Alice.foo}, (\mathtt{Charlie}, \mathtt{Bob})]\!]$

$5''$ $\mathcal{V}\,[\![\mathtt{Alice.foo\_Ret}, \mathtt{Alice.foo}, (\mathtt{Charlie}, \mathtt{Bob})]\!] \supseteq \mathcal{V}\,[\![\mathtt{Response}, \mathtt{Alice.foo}, (\mathtt{Charlie}, \mathtt{Bob})]\!]$

We evaluate $1''_2$ to obtain:

$\mathcal{V}\,[\![\mathbf{T_1}, \mathbf{Alice.foo}, (\mathbf{Charlie}, \mathbf{Bob})]\!] \supseteq \{\mathbf{BobAID}\}$

We evaluate $2''$:

$\mathcal{V}\,[\![\mathbf{Client}, \mathbf{Alice.foo}, (\mathbf{Charlie}, \mathbf{Bob})]\!] \supseteq \{\mathbf{BobAID}\}$

We evaluate $3'$ to obtain a simple constraint and we evaluate it:

$3''$ $\mathcal{V}\,[\![\mathtt{T}_2, \mathtt{Alice.foo}, (\mathtt{Charlie}, \mathtt{Bob})]\!] \supseteq \mathcal{SF}\,[\![\mathtt{Alice}]\!](\mathtt{Alice.ObjectSecret})$
$\mathcal{V}\,[\![\mathbf{T_2}, \mathbf{Alice.foo}, (\mathbf{Charlie}, \mathbf{Bob})]\!] \supseteq \{(\mathbf{p_1}, \mathbf{AliceAID})\}$

In order, we evaluate $4''$ and $5''$ to have the results:

$\mathcal{V}\,[\![\mathbf{Response}, \mathbf{Alice.foo}, (\mathbf{Charlie}, \mathbf{Bob})]\!] \supseteq \{(\mathbf{p_1}, \mathbf{AliceAID})\}$
$\mathcal{V}\,[\![\mathbf{Alice.foo\_Ret}, \mathbf{Alice.foo}, (\mathbf{Charlie}, \mathbf{Bob})]\!] \supseteq \{(\mathbf{p_1}, \mathbf{AliceAID})\}$

We evaluate $7''_a$ following by $8''$ to obtain:

$\mathcal{V}\,[\![\mathbf{T_4}, \mathbf{Bob.foo2}, (\mathbf{JCRE}, \mathbf{Charlie})]\!] \supseteq \{(\mathbf{p_1}, \mathbf{AliceAID})\}$
$\mathcal{V}\,[\![\mathbf{Bob.foo2\_Ret}, \mathbf{Bob.foo2}, (\mathbf{JCRE}, \mathbf{Charlie})]\!] \supseteq \{(\mathbf{p_1}, \mathbf{AliceAID})\}$

We evaluate $10''_a$, we produce a simple constraint with $11'$ and we evaluate it:

$\mathcal{V}\,[\![\mathbf{T_6}, \mathbf{Charlie.foo3}, (\mathbf{JCRE}, \mathbf{Charlie})]\!] \supseteq \{(\mathbf{p_1}, \mathbf{AliceAID})\}$

$11''$ $\mathcal{SF}\,[\![\mathtt{Charlie}]\!](\mathtt{Charlie.AliceSecret}) \supseteq \mathcal{V}\,[\![\mathtt{T}_6, \mathtt{Charlie.foo3}, (\mathtt{JCRE}, \mathtt{Charlie})]\!]$
$\mathcal{SF}\,[\![\mathbf{Charlie}]\!](\mathbf{Charlie.AliceSecret}) \supseteq \{(\mathbf{p_1}, \mathbf{AliceAID})\}$