



This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

<http://www.elsevier.com/copyright>

Contents lists available at [SciVerse ScienceDirect](#)

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

JCSI: A tool for checking secure information flow in Java Card applications

Marco Avvenuti^a, Cinzia Bernardeschi^{a,*}, Nicoletta De Francesco^a, Paolo Masci^b^a Department of Information Engineering, University of Pisa, 56126 Pisa, Italy^b School of Electronic Engineering and Computer Science, Queen Mary University of London, E1 4NS London, United Kingdom

ARTICLE INFO

Article history:

Received 3 May 2011

Received in revised form 30 April 2012

Accepted 17 May 2012

Available online 26 May 2012

Keywords:

Java card

Java bytecode

CAP file

Secure information flow

Abstract interpretation

ABSTRACT

This paper describes a tool for checking secure information flow in Java Card applications. The tool performs a static analysis of Java Card CAP files and includes a CAP viewer. The analysis is based on the theory of abstract interpretation and on a multi-level security policy assignment. Actual values of variables are abstracted into security levels, and bytecode instructions are executed over an abstract domain. The tool can be used for discovering security issues due to explicit or implicit information flows and for checking security properties of Java Card applications downloaded from untrusted sources.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

Java Cards are pocket-size cards equipped with an embedded micro-controller that supports the execution of a Java Virtual Machine (Chen, 2000). They are typically used in credit and loyalty systems, electronic cash, health-care and e-government.

A Java Card application consists of a set of applets bundled into a package. In multi-applicative Java Cards, new applications can be installed after card issuance. In order to enforce security and protection, applications are executed within protected spaces, called *contexts*. Each application is associated with a unique context. A component of the Java Card system, denominated *firewall*, uses an access control mechanism to enforce security policies. The basic rules enforced by the firewall are: (i) each applet can access only objects belonging to the context of the applet; (ii) information exchange between applets belonging to different contexts can be performed only through specific shared objects, denominated *shareable interfaces*. Applications providing shared resources are supported by the Java Card system with mechanisms suitable to customize the access policy. For instance, limited inspection of the call stack for checking the identity of the application willing to use the shared resource.

Although powerful, access control mechanisms are not sufficient to avoid unauthorized disclosure of information (Smith, 2007). The Electronic Purse case study (Cazin et al., 2000) is a well-known example that shows how a Java Card application can exploit information propagation for overriding access control policies.

In this work, we present a tool, denominated Java Card Secure Information (JCSI), for analyzing information flows in Java Cards. The tool implements a binary code disassembler for Java Card applications, and a data flow analysis (Lam and Ullman, 2007) based on a multi-level security policy and the theory of abstract interpretation (Cousot and Cousot, 1992). The multi-level security policy is used to associate security levels to applications, and the theory of abstract interpretation is used to re-define the semantics of bytecode instructions over a lattice of security levels. The lattice is given by the powerset of the applications' levels. The analysis uses a set of rules for detecting information flows in the bytecode. Applications are analyzed one at a time, and the analysis is carried out on a per-method basis. This enables a modular analysis similar to that performed by the Java bytecode verifier, which aims to check type-correctness of Java bytecode (Leroy, 2001). We use an *ambient* file to store and propagate security levels of methods (i.e., methods' arguments, return, and calling environment), and the security levels of objects in the heap. With this approach, information flows in the bytecode are assessed by checking that the security level of the applications' shared resources do not exceed the level specified in the security policy.

The ultimate aim of this tool is to help developers understand how information is propagated in different multi-applicative scenarios. On the one hand, developers can model different multi-applicative scenarios simply by customizing the security levels of

* Corresponding author: Tel.: +39 050 2217541; fax: +39 050 2217600.

E-mail addresses: m.avvenuti@ing.unipi.it (M. Avvenuti),
cinzia.bernardeschi@ing.unipi.it (C. Bernardeschi),
nicoletta.defrancesco@ing.unipi.it (N. De Francesco),
paolo.masci@eecs.qmul.ac.uk (P. Masci).

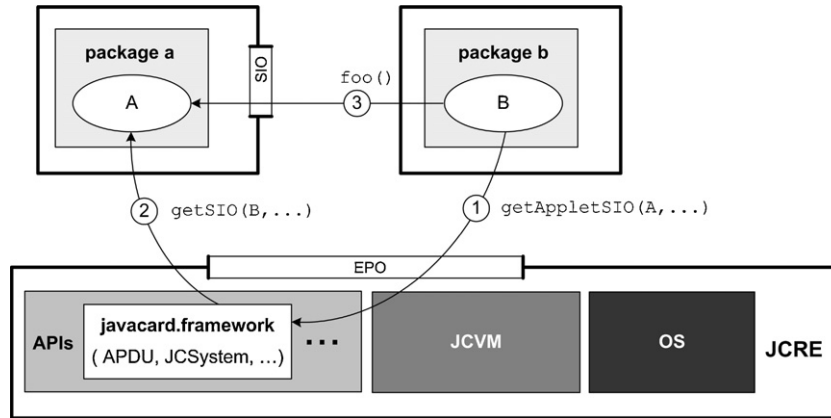


Fig. 1. The Java Card System.

methods and objects in the heap. On the other hand, developers can also study how information stored in specific variables propagates within the bytecode.

The contribution of this work is twofold: (i) we extend the approach for checking information flow in the bytecode defined in our previous work (Avvenuti et al., 2003; Barbuti et al., 2004), which covered only a limited subset of the Java language; (ii) we developed a tool that covers the Java Card 2.2.2 instruction set and that allows the user to define different security policies. The tool embeds a CAP file disassembler and visualizer, which enables to identify the precise cause of information flow. A preliminary version of the tool has been presented in Avvenuti et al. (2009). The current version of JCSI is available at <http://www.eecs.qmul.ac.uk/masci/JCSI>.

The rest of the paper is organised as follows. Section 2 describes the Java Card system. Section 3 explains the information flow problem in Java Cards. Section 4 presents the architecture of the JCSI tool, and describes in detail how the analysis is performed. Section 5 reports examples of application of JCSI. Section 8 concludes the paper.

2. Java Card System

The Java Card System is a platform for executing applications. The system relies on the Java Card Runtime Environment (JCRC) for managing resources, executing programs and applying access control mechanisms. The JCRC consists of a native operating system (OS), a Java Card Virtual Machine (JCVM) and a number of Application Programming Interfaces (APIs). Java Card applications reside in a user space and they can use JCRC services (see Fig. 1).

Java Card applications and JCRC's APIs are bundled into *packages*, which are data structures that store the compiled bytecode of Java classes and interfaces. A package¹ is uniquely identified and selected by an application identifier (*AID*), which is specified in the CAP file.

The Java Card firewall enforces access control mechanisms on applets. In order to enforce the access control rules, the firewall checks all operations performed by applets at run-time, and enables information exchange between applets belonging to different contexts only through specific shareable objects: Entry Point Objects (EPOs) and Shareable Interface Objects (SIOs). EPO objects belong to the JCRC's context, and they provide methods for exchanging messages (e.g., to request access to a resource), and for customizing access control rules (e.g., to identify the identity of another application). SIO objects, on the other hand, belong to applications'

context, and they provide methods to define the functionalities of applications' shared objects.

Java Card bytecode. Java Card applications are composed of applets, and they are compiled into binary CAP (Converted Applet) files, which contain an executable representation of the classes and interfaces defined in the applets. Methods defined in the applets are encoded as sequences of Java Card *bytecode* instructions. The semantics of Java Card bytecode instructions is defined in the Java Card Virtual Machine Language, which is an assembly language for Virtual Machines with an operand *stack* and a *memory* of local variables (registers). Instructions are typed: for example, *iload* (where *i* is an abbreviation for *int*) loads an integer onto the stack, while *aload* (where *a* stands for address of the *Object*) loads a reference which may point to any class and interface type, or array type. The instruction set includes, among others, construct for defining sub-routines and exception handlers.

In this work, we consider the complete Java Card 2.2.2 instruction set, which is summarized in Fig. 2. Let \mathcal{T} denote all types. \mathcal{T} includes the set $B = \{\text{boolean}, \text{short}, \text{byte}, \text{int}\}$ of primitive (basic) types, the set $\mathcal{C}' = \mathcal{C} \cup \{\text{Object}\}$ of user defined classes, together with the pre-defined *Object* class, the set \mathcal{I} of user defined interfaces and the set \mathcal{A} of array types. In the instruction set, we let $B' = B \cup \{\text{Object}\}$. Given a class $c \in \mathcal{C}$, we use the syntax $c.f:\tau$ to denote field f (with type τ) of class c , the syntax τ to denote arrays of type τ and the syntax $\tau_0.mt(\tau_1, \dots, \tau_n):\tau_r$ to denote the method mt of the class or interface $\tau_0 \in \mathcal{C} \cup \mathcal{I}$ with arguments of type τ_1, \dots, τ_n and return type τ_r .

In the following, given a method mt , B_{mt} denotes the finite sequence of bytecode instructions of mt . Given a set $\mathcal{L} = \{0, 1, \dots\}$ of instruction addresses, we use $B_{mt}[i]$, $i \in \mathcal{L}$, to indicate the i -th instruction in the sequence, being $B_{mt}[0]$ the entry point. The sub-sequence mt is omitted when clear from the context.

3. Information flow in Java Cards

Given a program with variables partitioned into two disjoint sets of high security (i.e., confidential) and low security (i.e., public) variables, the program has *secure information flow* if observations of the final value of low security variables do not reveal information about the initial values of high security variables (Bell and Padula, 1973; Denning, 1976; Denning and Denning, 1977).

In order to exemplify the concept of secure information flow, consider the following situations. Assume that y is a variable that stores confidential data (i.e., a high security variable), and x a public variable (i.e., a low security variable). In order to have secure information flow, programs should not contain instructions that assign y to x , which is called *explicit information flow*.

¹ In this paper we use the term package and application indifferently.

pop	Pop the top of the stack.
dup	Duplicate the top of the stack.
β op	$(\beta \in \mathcal{B})$ Takes two operands of type β from the stack, and pushes the result of type β onto the stack.
τ const d	$(\tau \in \mathcal{B}')$ Loads a constant d of type τ onto the stack.
τ load r	$(\tau \in \mathcal{B}')$ Loads the value of type τ from register r to the stack.
τ store r	$(\tau \in \mathcal{B}')$ Takes a value of type τ from the stack and stores it into register r .
ifcond L	$(L \in \mathcal{L})$ Takes a value of type int from the stack, and jumps to L if the value satisfies <i>cond</i>
goto L	$(L \in \mathcal{L})$ Jumps to L .
new τ	$(\tau \in \mathcal{C}' \cup \mathcal{I})$ Creates an instance of class τ and adds a reference to the created instance on top of the stack.
getfield $\tau_0.f:\tau$	$(\tau_0 \in \mathcal{C}', \tau \in \mathcal{T})$ Takes an object reference of class τ_0 from the stack; fetches field f (of type τ) of the object and loads the field on top of the stack.
putfield $\tau_0.f:\tau$	$(\tau_0 \in \mathcal{C}', \tau \in \mathcal{T})$ Takes a value of type τ and an object reference of class τ_0 from the stack; saves the value in field f of the object.
checkcast τ	$(\tau \in \mathcal{C}' \cup \mathcal{I})$ If the reference on top of the stack is of type τ leaves the stack unchanged, otherwise raises an exception.
newarray τ	$(\tau \in \mathcal{T})$ Creates an instance of an array of class τ and adds a reference to the instance on top of the stack.
raload	$(\tau \in \mathcal{B}')$ Takes a reference to an array and an integer index from the stack. The array reference is of type $[\tau]$. Loads on the stack the value, of type τ , stored at the index position in the referenced array.
rastore	$(\tau \in \mathcal{B}')$ Takes an array reference, an integer index and a value from the stack. The array reference is of type $[\tau]$, the value of type τ . The value is saved in the referenced array at the index position.
arraylength	Takes a reference to an array from the stack and places the length of the array on the stack.
instanceof τ	$(\tau \in \mathcal{C}' \cup \mathcal{I} \cup \mathcal{A})$ Takes a reference from the stack and pushes 1 on top of the stack if the reference is not null and is an instance of type τ , 0 otherwise.
invoke $\tau_0.mt(\tau_1, \dots, \tau_n):\tau_r$	$(\tau_0 \in \mathcal{C}' \cup \mathcal{I}, \tau_1, \dots, \tau_n, \tau_r \in \mathcal{T})$ Takes the values v_1, \dots, v_n (of types τ_1, \dots, τ_n) and an object reference of class τ_0 from the stack. Invokes method $\tau_0.mt$ of the object with actual parameters v_1, \dots, v_n ; places the method return value (of type τ_r) on top of the stack.
rreturn	$(\tau \in \mathcal{B}')$ Takes the value of type τ from the stack and terminates the method.
athrow	Takes an object reference from the stack and raises an exception.
jsr L	$(L \in \mathcal{L})$ Places the address of the successor on the stack and jumps to L .
ret r	Jumps to the address specified in register r .

Fig. 2. A representative subset of Java Card 2.2.2 instruction set.

Similarly, programs should not contain sequences of instructions that assigns values to x depending on the value of y , which is called *implicit information flow* (for example, if $(y>10)$ then $x=1$ else $x=2$). In this case, the value of x reveals information on the value of y .

In the Java (and Java Card) bytecode, given the richness of the language, explicit and implicit information flows are not due only to assignments and conditional branches, but also to instructions that dynamically create objects, access arrays, and invoke methods (Amtoft et al., 2006). Furthermore, when studying information flow in the Java bytecode, the problem is also complicated by the fact that the bytecode is unstructured, and approaches based on theory of graphs must be used to properly handle implicit information flows. In this work, we use the concepts of control flow graph and post-dominators (Ball, 1993).

A control flow graph (CFG) is a directed graph (V, L) , where V is a set of nodes, and $L \subseteq V \times V$ is a set of edges. Given a bytecode B , the CFG associated with B is built as follows: V contains one node for each instruction; L contains all edges (i, j) such that instruction j can be executed immediately after i , i.e., either j is the *natural* successor of i ($j=i+1$), or i is a branching instruction (goto, if, switch, etc.) and j is one of the targets. Note that conditional branching instructions can have two or more successors. Instruction i : ifcond L , for instance, has two successors: instructions

$i+1$, which is the natural successor, and L , which is the target of the conditional instruction. We assume the CFG has a final node, END, and an edge from each return instruction of the bytecode to node END.

In the CFG, given two nodes $i, j \in V$, j *postdominates* i , denoted by $j \text{ pd } i$, if $j \neq i$ and j is on every path from i to the final node. Node j *immediately postdominates* i , denoted by $j = \text{ipd}(i)$, if $j \text{ pd } i$ and there is no node r such that $j \text{ pd } r \text{ pd } i$.

For conditional branching instructions, the instruction itself is the point where an implicit information flow begins. Such implicit flow affects instructions belonging to all the paths from i to $\text{ipd}(i)$. The first instruction not affected by the implicit flow is $\text{ipd}(i)$, because it represents the joining point of all branches of the conditional instruction. In the following we use $\text{scope}(i)$ to denote the set of instructions belonging to the paths from i to $\text{ipd}(i)$ (note that i and $\text{ipd}(i)$ are not included in $\text{scope}(i)$).

Examples of information flow in Java Cards. We now show a situation to exemplify (i) how applets can use and customize access control policies when sharing information, and (ii) how the access control policies can be overridden. For the sake of clarity, we show the source code of applets, instead of their compiled bytecode.

Using access control policies to control information sharing. Assume that **A** and **B** are two applets belonging to different packages. Assume that **A** stores sensitive data in a private field *balance*,

```

file AInt.java
import javacard.framework.Shareable;
public interface AInt extends Shareable{
    public short foo(); }

file A.java
import javacard.framework;
import a.AInt;
public class A extends Applet implements AInt{
    private short balance;
    public Shareable getSIO(AID client, byte num){
        if(client.equals(B)) return this;
        return null;
    }
    public short foo(){
        AID client = getPreviousContextAID();
        if(client.equals(B)) return balance;
        return 0;
    }
}

```

Fig. 3. Package a.

```

file BInt.java
import javacard.framework.Shareable;
public interface BInt extends Shareable{
    public a.AInt bar(); }

file B.java
import javacard.framework;
import a.AInt;
public class B extends Applet implements BInt{
    private static AInt AObj;
    private short ABalance;
    private void work (){
        AObj = (AInt) (JCSys.getAppletSIO(A, 0));
        ABalance = AObj.foo();
    }
    public Shareable getSIO(AID client, byte num){
        return this;
    }
    public AInt bar(){return AObj;}
}

```

Fig. 4. Package b.

and that A wants to share `balance` with B, but not with other applets. To this end, A can implement a SIO (AInt), which contains method `foo()` for returning the content of `balance` only to B.

If B wants to invoke the method `foo()` of the SIO shared by A, the chain of method invocations imposed by the JCRE is the following (see Fig. 1):

- (1) Applet B must invoke `getAppletSIO(AID, byte)`, implemented in a static EPO of the JCRE; the first parameter (AID) identifies the applet implementing the shared object (A, in this case), and the second parameter (byte) selects, if necessary, a specific SIO (this parameter is used when applets share more than one SIO).
- (2) The JCRE dispatches a request for the shareable object to A by triggering the invocation of method `getSIO(AID, byte)` implemented by A, where the first parameter (AID) identifies the applet that requested the shared object (B, in this case) and `byte` defines a specific SIO.
- (3) If applet A accepts the request, a pointer to the shareable object is returned to the JCRE, which in turn returns the pointer to B. If the request is not accepted (this may happen, for example, when the applet that requested the SIO is not authorized), A returns a null pointer to the JCRE, which in turn returns a null pointer to B.

Applet B can invoke method `foo()` after receiving a non-null pointer to the SIO. When method `foo()` is invoked, the JCRE automatically performs a *context switch*: the *current active context* (i.e., the context of the applet being executed by the system – B, in this case) is saved, and the context of the owner of the SIO (A, in this case) becomes the new current active context. When the execution of the method completes, the context of the caller is set as the current active context.

In the Java Card System, an applet that obtains a reference to a SIO can pass the reference to applets of other packages. In order to enable SIOs' owners to determine the identity of the caller, the JCRE provides a static EPO, denominated `JCSys`, which provides a method for a limited inspection of the call stack (`getPreviousContext()`).

For instance, consider the code shown in Fig. 3. Method `getSIO(AID, byte)` of package a checks the identifier of the client,

and if it is different from B, then the method returns a null pointer. Moreover, in order to avoid that other applets use a copy of the SIO granted to B, method `foo()` checks the identity of the caller through `getPreviousContextAID()`, and returns the value of field `balance` only in the case the caller is actually B. The code of package b is shown in Fig. 4. B implements a private method `work()` that requests the SIO to A, saves it into the private static field `AObj` and invokes method `foo()` to update its private field `ABalance`.

Overriding access control policies. Let us consider an applet C, belonging to package c, that wants to override the access control policy imposed by A. If C requests the SIO directly to A, the request is rejected because C's identity is passed to A by JCRE and A checks the client identity before returning the SIO. Assume that B implements a SIO, named BInt, which includes method `bar()`. If C gets A's SIO by calling method `bar()`, C is able to invoke method `foo()` of A, but A does not return the value of field `balance` because the identity of the caller is checked in method `foo()`. The implementation of these possibilities is reported in Fig. 5.

Applet C, however, may gather information on the value of field `balance` as follows.

Method 1. B may send C the balance of A by implementing a method `get()` that returns B's private field `ABalance`:

```
public short get(){ return ABalance; }
```

Alternatively, B may send information on the balance of A by returning the value of a variable `x` that depends on `ABalance`:

```
public short get(){
    short x = (short)((ABalance <= 100)? 0 : 1);
    return x;
}
```

Method 2. A or B may perform conditional invocations to shared methods of C depending on the value of A's balance. Let CInt be a shareable interface of C and let `mh()` be a method of CInt; C can infer information on the amount of A's balance by checking if `mh()` has been invoked:

```
...
CObj = (CInt)(JCSys.getAppletSIO(C, 0));
if (ABalance > 100) CObj.mh();
...
```

```

file CInt.java
import javacard.framework.Shareable;
public interface CInt extends Shareable{
    public void mh();}

file C.java
import javacard.framework;
import a.AInt;
import b.BInt;
public class C extends Applet implements CInt{
    private static AInt AObject;
    private static BInt BObject;
    private short ASecret;
    public Shareable getSIO(AID client, byte num){
        return this;}
    public void work(){
        BObject = (BInt) JCSys.getAppletSIO(B, 0);
        AObject = BObject.bar();
        ASecret = AObject.foo();
    }
}

```

Fig. 5. Package c.

Method 3. Calls to JCRE's EPOs can contribute to information flows. If B makes a request for C's SIO depending on the value of A's balance, C may infer information because the invocation of `getAppletSIO(C, 0)` made by B triggers `getSIO(B, 0)`:

```

...
if (ABalance > 100)
    CObj = (CInt)JCSys.getAppletSIO(C, 0);
...

```

Similarly, if B returns a SIO to C depending on A's balance, C infers information on A through the release of the SIO from B. An implementation of `getSIO()` in B that triggers this information flow is the following:

```

public Shareable getSIO(AID cl, byte num){
    if(cl.equals(C) && ABalance > 100)
        return this;
    return null;
}

```

4. The JCSI tool

JCSI is a tool for checking secure information flow in the binary CAP files of Java Card applications. The analysis performed by the tool is grounded on our previous work (Barbuti et al., 2004), which we briefly summarize in the following.

Starting from the semantics of the language, we define a set of inference rules in structured operational semantics style (Siveroni, 2004). Then, we define an enhanced semantics of the language to keep track of the information flow during program execution. Such an enhanced semantics was obtained by annotating values with information flow levels, and by executing instructions under a *security environment*, which takes into account the security level of the implicit flows. We proved that the approach is able to detect unauthorized disclosure of confidential information.

The approach we previously proposed applies only to a limited subset of the Java bytecode, which does not include, among others, array, objects and method calls. The tool described here extends our previous work in order to cover the complete Java Card 2.2.2

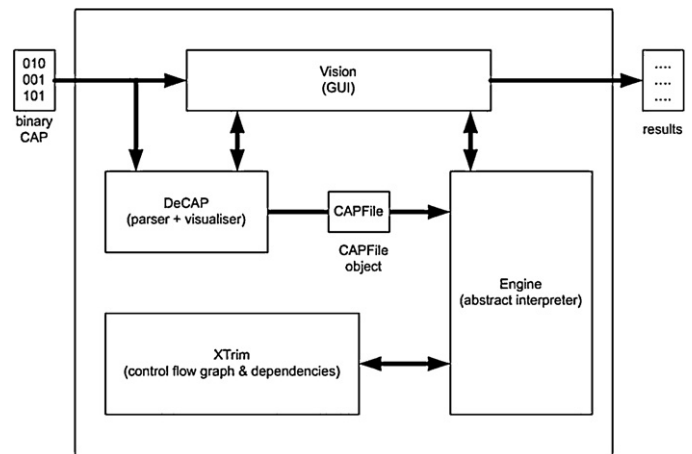


Fig. 6. JCSI architecture.

instruction set. The tool has been developed in the Java Programming Language.

In order to perform the data flow analysis, the bytecode of the CAP file must be type-correct – this can be checked, for instance, through the Java Card bytecode Verifier (Leroy, 2001). The data flow analysis is performed on one CAP file at a time.

The tool implements an analysis wizard which guides the user during the analysis, which is carried out through the following in three main steps:

- **Step 1: multi-applicative scenario setup.** The tool prompts to identify the number and names of the packages already installed on card. Such an information will be used by the tool to assign *security levels* to methods and objects in the heap. In particular, each package name is used as place-holders for the security level of such a package.
- **Step 2: multi-level security policy setup.** The tool prompts to setup a multi-level security policy for methods, fields, and objects in the heap. The multi-level security policy can be either automatically generated by the tool, or loaded from previously saved policies. In any case, the tool allows the user to customize all policy parameters in order to perform different analyses. The tool stores the policy in a configuration file, named *ambient file*. During the data flow analysis, the *ambient file* is used to propagate security levels among objects in the heap, fields and methods (internal, imported, and exported). Partial analyses of the bytecode can be performed, e.g., by including in the *ambient file* only a subset of the implemented methods.
- **Step 3: data flow analysis execution.** The tool shows how information is propagated through the bytecode during the data flow analysis. The tool constantly updates the level of objects in the heap, fields and methods in the *ambient file* in order to propagate information flows. The fixpoint of the iterative data flow is reached when all methods specified in the *ambient file* have been analyzed and the security levels in the *ambient file* are unchanged.

At the end of the analysis, the security levels in the *ambient file* register data dependencies. The tool reports if the CAP file satisfies the multi-level security policy described in the *ambient file*. If the CAP file does not satisfy the policy, the tool reports a detailed log of the bytecode instructions violating the policy.

The software architecture of the tool is shown in Fig. 6. The tool is composed of four main modules: Vision, Engine, XTrim and DeCAP.

Vision. This module implements the graphical user interface of the tool (Fig. 7). The user interface provides the following

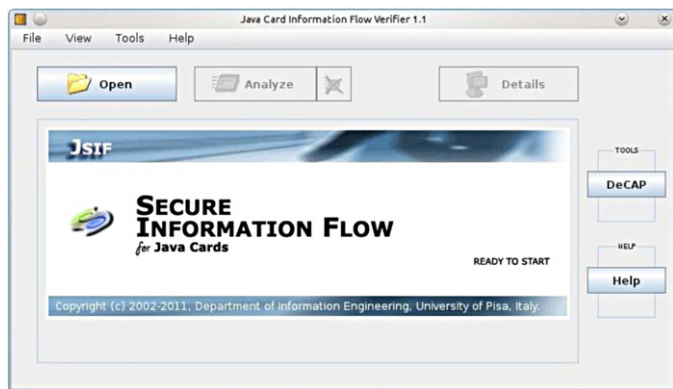


Fig. 7. JCSF's graphical user interface.

functionalities: select the package to be analysed; load export files; specify number and names of packages already installed on the card; load, save, view and edit ambient files (see Fig. 8); choose the stopping rule for the analysis, (i.e., either stop at the first violation, or at the fixpoint).

Engine. This module implements the abstract interpreter that performs the data flow analysis of the bytecode. At the end of the abstract execution, the Engine compares the security levels of methods belonging to sharable interfaces against the defined security policy. A detailed discussion of the module is reported in the following.

XTrim. This module generates the control flow graph of the bytecode for each method defined in the CAP file under analysis. XTrim is also responsible of computing the scope of conditional branching instructions.

DeCAP. A CAP file disassembler and visualizer. DeCAP is a tool that provides a set of APIs suitable to read binary CAP files used by Java Cards. This tool is invoked by the Engine in order to parse CAP files. Binary CAP files are represented through a Java class called *CapFile*. DeCAP also provides a GUI that enables users to explore the structure of CAP and export files, and to visualize their binary content in a more comprehensive mnemonic format (see Fig. 9).

In the following, we explain the security model and the ambient file. Then, we give details of the algorithm implemented by the Engine to perform the analysis.

4.1. Security model

We define a set P of security levels, one for each package. We consider the powerset $\Sigma = 2^P$, i.e., the set of all subsets of P , ordered by subset inclusion. (Σ, \subseteq) is a complete lattice (every pair of elements of Σ has both a greatest lower bound, *glb*, and a least upper

bound, *lub*). The *lub* is given by the union (\cup) and the *glb* is given by the intersection of subsets (\cap). Given $A \subseteq B$, $A \cup B = B$ and $A \cap B = A$. The minimum of Σ is the empty set (i.e., $\{\}$).

The analysis operates over security levels in Σ . At the beginning of the analysis each data is annotated with a security level. For instance, $\{p\}$ denotes a secret of p , the empty set $\{\}$ denotes public data. At the end of analysis, the singleton set $\{p\}$ denotes information that depends only on package p , the set $\{p_1, \dots, p_n\}$ denotes information that depends on private data of packages p_1, \dots, p_n .

4.2. Ambient file

The *ambient* file has two sections: the *heap section*, which stores and propagates security levels of objects in the heap, and the *methods section*, which is dedicated to methods. Each entry in the *ambient* file specifies the security levels held by the item and, optionally, the security policy that must be checked.

Heap section. The heap is a private resource of the package. The *ambient* file maintains a security level for each class field and for each array type. Objects are abstracted into classes. Arrays are abstracted into array types. Arrays of objects and array of arrays are all abstracted into array of references type. These abstractions are due to the fact that the bytecode does not contain enough information to keep track of object instances and specific elements in arrays. Assuming we are analyzing the package p , the tool initializes the security levels to the default value equal to $\{p\}$. This level can be customized by the user to model the desired policy.

The security level of each class field $c.f$ is initially set to $\{p\}$. The level of class fields will be updated during the analysis according to the flow of information and the dependencies between instructions in the program. When the analysis completes, the security level of $c.f$ is the maximum security level of the field f of all objects of class c . Similarly, the security level of each array type is set to the security level $\{p\}$. When the analysis completes, the security level of arrays of type τ is the maximum security level saved into all array instances of type τ .

Methods section. Let \mathcal{M} denote the set of all methods occurring in package p . Methods in \mathcal{M} can be either *internal methods* (i.e., methods defined in p – they can be invoked only by applets belonging to the packages), *imported methods* (i.e., methods defined in other packages – they are invoked by p), or *exported methods* (i.e., methods defined in p that can be used by other package). The level of a method characterizes how the method is called in terms of the level of the method's actual parameters, return and calling environment.

The syntax we use for denoting an entry for a method mt is the following: $mt_p^p(\tau_0, \tau_1, \dots, \tau_n)\tau_r; \tau_e \langle S \rangle$, where p is the package that implements mt , p' is the package that invokes mt , τ_0 is the implicit parameter (*this*), τ_1, \dots, τ_n are the method's arguments, τ_r is the return, and τ_e is the calling environment. The security level $S \in \Sigma$ specifies the security policy that must be satisfied by mt .

The policy is optional. When the security policy is not specified for a method, the tool does not check any constraint on such a method. This is useful for modeling situations where any information flow is allowed through the method. Examples include invocation of JCRE methods and invocation of methods within third-party collaboration scenarios.

4.2.1. Default initialization of the methods section

The default initialization of the *ambient* file configures the analysis for the worst-case scenario: any package already installed on card may implement methods imported by the applet under analysis and may invoke exported methods of the applet under analysis. The default policy enforces that information shared between two packages only depends on these two packages.

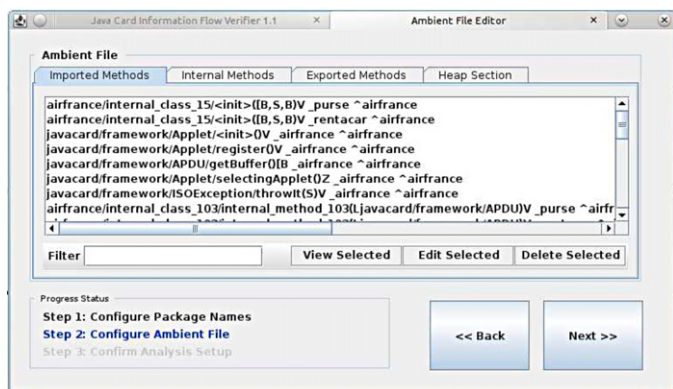


Fig. 8. Ambient file editor.

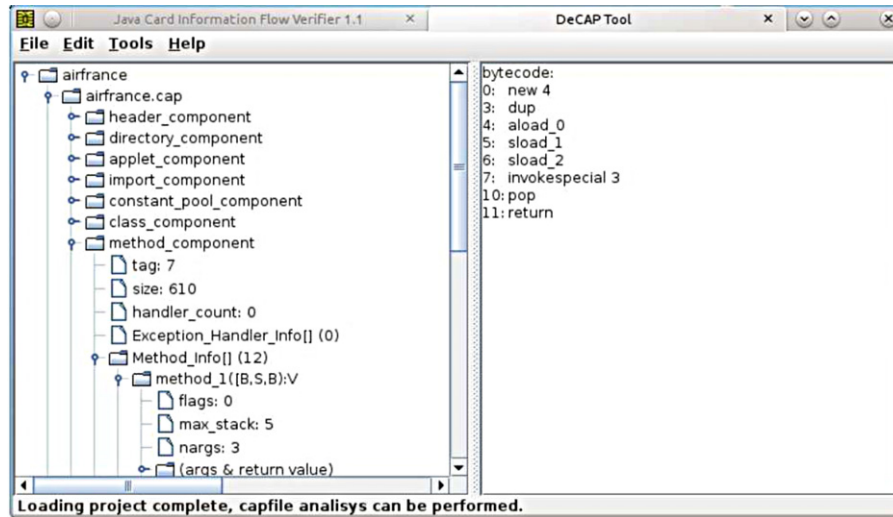


Fig. 9. DeCap tool.

The default initialization of the methods section is shown in Fig. 10. For internal methods, denoted as *Internal*(*p*), a single instance of each method is inserted with levels of arguments, return and calling environment set to *p*. As internal methods always satisfy the secure information flow, the security policy is omitted (i.e., any information flow is accepted for such methods).

For imported methods, denoted as *Import*(*p*), the worst case scenario is that any package on card may implement such methods. JCRE methods can be identified by the signature. Imported methods belonging to the JCRE, except *getAppletSIO*(), have default level *p* and the security policy is omitted (i.e., any information flow is allowed). Method *getAppletSIO*() has default level *p*, and security policy $\langle\{p\}\rangle$.

For other imported methods, the levels of parameters and calling environment are set to $\{p\}$. The level of the return is set to *lub* of all packages. The security policy for a method mt_p^p is set to $glb(\{p, p'\} \mid p' \text{ installed on card})$, where *p* is the applet under analysis and *p'* is the applet that implements the method.

For exported methods, denoted as *Export*(*p*), the worst case scenario is that any package installed on card may invoke such methods. This implies that the *ambient* file contains an item mt_p^p with level $\{p, p'\}$ for parameters, return and calling environment. The security policy for such an item is set to $\langle\{p, p'\}\rangle$.

$$\begin{aligned} &\forall mt \in \text{Internal}(p) : \\ &\quad mt_p^p(\{p\}, \dots, \{p\})\{p\}; \{p\} \\ &\forall mt \in \text{Import}(p) : \\ &\quad - mt \in \text{JCRE} \\ &\quad \quad \text{if } (mt \neq \text{getAppletSIO}()) \\ &\quad \quad \quad mt_p^p(\{p\}, \dots, \{p\})\{p\}; \{p\} \\ &\quad \quad \text{else} \\ &\quad \quad \quad \text{getAppletSIO}_p^p(\{p\}, \{p\})\{p\}; \{p\} \langle\{p\}\rangle \\ &\quad - mt \notin \text{JCRE} \\ &\quad \quad mt_p^p(\{p\}, \dots, \{p\})W; \{p\} \langle S \rangle \text{ where} \\ &\quad \quad \quad W = \text{lub}(\{p, p'\} \mid p' \text{ installed on card}) \\ &\quad \quad \quad S = \text{glb}(\{p, p'\} \mid p' \text{ installed on card}) \\ &\forall mt \in \text{Export}(p) : \\ &\quad \forall p' \text{ such that } p' \text{ installed on card} \\ &\quad \quad mt_p^p(\{p, p'\}, \dots, \{p, p'\})\{p, p'\}; \{p, p'\} \langle\{p, p'\}\rangle \end{aligned}$$

Fig. 10. Default initialization of the methods sections in the *ambient* file.

The security policy $\langle\{p, p'\}\rangle$, which guarantees that private information of a third package cannot be released by *p* to *p'*, also applies to the invocation of *getSIO*(), which is actually triggered by other packages through the *getAppletSIO*(). The default initialization includes an instance of method *getSIO*() for every other package *p'* on card.

4.2.2. Custom initialization of the methods sections

The default initialization of the *ambient* file is fully automatic. However, it does not take advantage of the knowledge of real interactions between packages. As a consequence, the analysis may lead to false positives that in practice may result over restrictive. This applies also to Java Card entry point methods for JCRE – such as methods for APDU-related instructions, the applet constructor, and the applet life-cycle methods *install*(), *select*(), *deselect*(), *process*(), *getSIO*() – to which the default initialization assigns the same policy as the methods invocable by user packages. In order to perform a more accurate and realistic analysis, the tool enables the user to change the levels and the security policy by manually editing the *ambient* file. In general, when the CAP files of the other applets installed on card are available, the user, assisted by the DeCAP tool, can look at those files for identifying which packages can potentially invoke the exported methods, and modify the *ambient* file accordingly.

A simple customization strategy can exploit information contained in the *Import components* of the other CAP files installed on card and the information contained in the *Export components*. By matching such an information, the user can identify which packages implement a given method, as well as which packages may potentially invoke a given method. Such a strategy is shown in Fig. 11. *Packages*(*mt*) denotes the set of packages that implement method *mt*.

For imported methods, the security policy is $\langle\text{glb}(\{p, \bar{p}\} \mid \bar{p} \in \text{Packages}(mt))\rangle$.

For exported methods, if *mt* is one of the entry point methods for JCRE and *mt* is not *getSIO*(), the user can release the security policy, as the invocation of such methods propagates information to JCRE only. For these methods, the *ambient* file contains an instance of *mt* with level *p* and without a security policy ($mt_p^p(\{p\}, \dots, \{p\})\{p\}; \{p\}$).

If the exported method is *getSIO*(), we need to identify all packages that may potentially request a shareable object by invoking method *getAppletSIO*() (see Section 3). To control the information flow, the *ambient* file must contain an instance of *mt* for

```

 $\forall mt \in \text{Internal}(p) :$ 
   $mt_p^p(\{p\}, \dots, \{p\})\{p\}; \{p\}$ 

 $\forall mt \in \text{Import}(p) :$ 
   $mt \in \text{JCRE}$ 
  if  $(mt \neq \text{getAppletSIO}())$ 
     $mt_p^p(\{p\}, \dots, \{p\})\{p\}; \{p\}$ 
  else
     $\text{getAppletSIO}_p^p(\{p\}, \{p\})\{p\}; \{p\} \langle \{p\} \rangle$ 
   $mt \notin \text{JCRE}$ 
   $mt_{p'}^p(\{p\}, \dots, \{p\})W; \{p\} \langle S \rangle$  where
     $W = \text{lub}(\{\{p, p'\} \mid p' \in \text{Packages}(mt)\})$ 
     $S = \text{glb}(\{\{p, p'\} \mid p' \in \text{Packages}(mt)\})$ 

 $\forall mt \in \text{Export}(p) :$ 
   $mt$  entry point for JCRE
  if  $(mt \neq \text{getSIO}())$ 
     $mt_p^p(\{p\}, \dots, \{p\})\{p\}; \{p\}$ 
  else
     $\forall p'$  such that  $\text{getAppletSIO}()$  invocable by  $p'$ 
     $\text{getSIO}_{p'}^p(\{p, p'\}, \{p, p'\})\{p, p'\}; \{p, p'\} \langle \{p, p'\} \rangle$ 
   $mt$  not entry point for JCRE
   $\forall p'$  such that  $mt$  invocable by  $p'$ 
     $mt_{p'}^p(\{p, p'\}, \dots, \{p, p'\})\{p, p'\}; \{p, p'\} \langle \{p, p'\} \rangle$ 

```

Fig. 11. Example of custom strategy for initialising the methods sections in the ambient file.

each package that invokes $\text{getAppletSIO}()$, with the security policy $mt_p^p(\{p\}, \dots, \{p\})\{p\}; \{p\} \langle \{p, p'\} \rangle$.

For all other exported methods, the policy is the same as the one defined by the default initialization.

4.3. Engine

In this section, we explain in detail the analysis implemented by the Engine module.

Iterative data flow analysis. The data flow analysis is performed on a per-method basis. The analysis starts from an initial ambient file D^0 , and checks all methods in \mathcal{M} that are implemented by the package under analysis. The set T of methods to be analyzed is maintained in a work-list.

The analysis uses an abstract interpreter, named Method Security Checker (MSC), for verifying a method with the algorithm shown in Fig. 12. Given a method $mt \in T$ and an ambient file D , MSC performs an abstract execution of the bytecode of mt with respect to the security levels in D and produces a new ambient file D' . If $D' = D$, then the fixpoint has been reached for that method, and another method is analyzed. If $D' \neq D$, all methods are verified

```

 $D := D^0$ 
 $T := \{mt \in \mathcal{M} \mid p \text{ implements } mt\}$ 
 $MT := T$ 
while  $(MT \neq \emptyset)$ 
  select  $mt \in MT$ 
   $MT := MT - \{mt\}$ 
   $D' := \text{MSC}(mt, D)$ 
  if  $(D' \neq D)$ 
     $D := D'$ 
     $MT := T$ 

```

Fig. 12. Iterative data flow analysis of a package (pseudo-code).

again starting from the ambient file D' . The verification terminates when, starting from an ambient file, all methods are analyzed and the ambient file remains unchanged.

Given a method mt , when verifying the bytecode B_{mt} , we use a global state $\langle D, Q \rangle$, where D is the ambient file, and Q is a table containing a row Q_i for every instruction $i \in \mathcal{L}$ of the method.

Given an ambient file D , an element t of D is denoted with D_t : $D_t = \sigma$ if t is a class field or an array type; $D_t = (\sigma_0, \sigma_1, \dots, \sigma_n)\sigma_r; \sigma_e$ if t is a method with n parameters $\sigma_1, \dots, \sigma_n$ plus the implicit parameter σ_0 , σ_r is the return and σ_e is the environment. For static methods, σ_0 is omitted.

Given Q , Q_i represents the state of JVM in which instruction i is executed and we will refer to it as the *before-state* of instruction i . Moreover, we will refer to the state generated after the execution of instruction i as the *after-state* of i . Q_i is a triple $\langle E, M, St \rangle$, where $E \in \Sigma$ is the security level of the environment, $M: \text{Registers} \rightarrow \Sigma$ is a mapping from local registers to security levels (the memory) and $St \in \Sigma^*$, where $*$ denotes the set of finite sequences over a set, is a mapping from the elements in the operand stack to security levels (the stack).

Q_{END} represents the final state reached after the execution of the last instruction of the method. The standard concatenation operator is \cdot and the empty stack is represented by the symbol λ . We assume that the values on top of the stack appear on the left hand-side of the sequence (i.e., given $st = s_1 \dots s_n$, the element s_1 is the top of the stack).

The initial state Q^0 reflects the state of JVM on method entrance. The operand stack is empty for every instruction. Q^0 assigns the bottom element of the lattice to the environment and to every register of all instructions but instruction 0. For instruction 0, the registers containing the method's actual parameters are initialized to the security level taken from the ambient file. The level of the environment of instruction 0 is equal to the level of the calling environment taken from the ambient file for the method.

In order to execute the analysis, the CFG of the method is derived from the bytecode, and the dependency between instructions is computed (i.e., for each instruction i , we compute $\text{scope}(i)$). The CFG defined in Section 3 is extended in order to consider information flow due to subroutines and exceptions.

Subroutines are sections of code shared between multiple execution paths. Instruction $\text{jsr } L$ jumps to instruction L , which is the first instruction of the subroutine, and places a return address onto the stack. Typically, the first instruction of the subroutine pops the return address from the stack and stores it in a register; this way, when the subroutine completes, a ret instruction can be used to fetch the return address from the register, and jump back to the instruction following $\text{jsr } L$. The actual return address can be determined only in real executions. Hence, in order to take into account all execution flows, we need model the $\text{ret } r$ instruction as a jump to all possible return address. To this end, the CFG is extended with edges from the $\text{ret } r$ to all possible return points.

Exception handlers are identified through an *exception table*, denoted by E . For each exception handler, E defines the entry point of the handler, the range of protected instructions, and the type of exception caught. Note that the entry point of an exception handler can only be reached by raising an exception (i.e., there are no explicit jumps to the entry point of the exception handler). This is a constraint enforced by the Bytecode Verifier. We denote bytecode sequences protected by exception handlers with triples $\langle i, j, k \rangle$, where i is the first instruction protected by the handler, j is the first instruction not protected by the handler, and k is the entry point of the exception handler. Since the exception type is only known at run-time and cannot be derived from the bytecode, we cannot distinguish between different types of exceptions thrown by an

instruction. In order to take into account all execution flows, we impose that all the exception handlers that protect instructions are possible successors of such instructions. Given an instruction i , we use $H(i)$ to denote the set of entry points of the exception handlers protecting i . The CFG is extended as follows. For every protected instruction i , we add an arc in the CFG from i to the entry point of every exception handler protecting i ($H(i)$). We add an arc from instructions that raise exceptions (`throw` and `throwIt()`) to the END node.

Abstract interpreter. The abstract interpreter executes bytecode instructions over the abstract domain of security levels. The rules of the abstract interpreter define a relation $\rightarrow \subseteq \langle D, Q \rangle \times \langle D, Q \rangle$.

When instruction at position i (hereafter, we will use the term “instruction i ” with the same meaning of “instruction at position i ”) gets executed, the after-state of i is propagated to every successor of i . The propagation is obtained through a merge operator, *lub*, between the after-state of i and the state of the successor of i . The *lub* operator is defined separately on memories, stacks and environments: $(\sigma, M, St) \cup (\sigma', M', St') = (\sigma \cup \sigma', M \cup M', St \cup St')$. The *lub* operator on memories is defined point-wise on memory registers. Given two memories M and M' , $M \cup M'$ is defined as: $M(r) \cup M'(r)$ for every register r . Similarly, the *lub* operator on stacks is defined point-wise on the elements in the same position. Note that, for each bytecode address i , the stack size when i gets executed is independent from the control flow. This is a property of all type-correct bytecodes generated by compilers. Such property is also enforced by the standard Bytecode Verifier (Leroy, 2001). The *lub* operator on methods stored in D is defined point-wise on method parameters (according to their position), calling environment and return: $(\sigma_0, \sigma_1, \dots, \sigma_n)\sigma_r; \sigma_e \cup (k_0, k_1, \dots, k_n)k_r; k_e = (\sigma_0 \cup k_0, \sigma_1 \cup k_1, \dots, \sigma_n \cup k_n)\sigma_r \cup k_r; \sigma_e \cup k_e$.

Instructions to be verified are inserted into a worklist WL , initialized with instruction 0. When WL is empty, the verification of a method completes. Whenever an instruction i is fetched from WL , instruction i is executed starting from its current state Q_i , and the after-state of instruction i is computed. The after-state is then merged with the before-state of every successor of the instruction. If the state of a successor j changes, or if a successor has not been visited yet, j is inserted in WL . Note that Q_j stores a state that merges all possible states in which j can be executed. Note that, when an instruction i is executed, every state Q_j corresponding to the entry point of an exception handler protecting the instruction is also updated. The stack at the entry point of exception handlers contains one operand. In the rules, this operand is set equal to the *lub* between the levels in the stack positions and the level of the security environment.

The data flow analysis used in this work is conservative, i.e., all packages that may potentially disclose confidential information are rejected, but also some secure packages might be rejected. This is due to the fact that during the abstract execution of a method, all branches of control instructions are checked, even those that in the real execution would have never been executed.

Rules of the abstract interpreter. A set of rules for various types of instructions is presented in the Appendix. In the following we describe the notation used for the rules and then we describe the implemented rules.

Notation. $Q[Q_i \cup (\sigma, M, St)]$ denotes a table Q' which is equal to Q except for entry i , that is set equal to the *lub* between Q_i and (σ, M, St) : $Q' = Q_i \cup (\sigma, M, St)$. A list of entries to be updated may be specified. Given $s \in \Sigma^*$, $D[D_t \cup s]$ denotes an *ambient* file D' which is equal to D except for entry t , that is set equal to the *lub* between D_t and s . For methods we have a sequence of security levels whose length depends on the number of parameters. $M[v/r]$ denotes a memory M' which differs from M only for register r , whose value is

$M'(r) = v$. Function $\hat{\cdot}$ maps an array type into either an array of basic types or an array of references

$$\hat{\tau} = \begin{cases} \tau & \text{if } \tau \in \text{Base} \\ \text{Reference} & \text{otherwise} \end{cases}$$

Function $\Pi_A(D)$ denotes the *lub* of the security levels of arrays in D

$$\Pi_A(D) = \cup_{\tau \in T} D(\hat{\tau})$$

Given an instruction i , $h(Q_i)$ is used to denote the after state of i in the case in which instruction i raises an exception. Specifically, $h(Q_i)$ is a state equal to Q_i except for the stack. The stack contains only one element, obtained as the *lub* of the security levels of stack positions and the environment. Given $Q_i = (\sigma, M, St)$ with $St = s_1 \dots s_n$, we have that $h(Q_i) = (\sigma, M, (\sigma \cup (\bigcup_{i \in \{1, \dots, n\}} s_i)))$.

In the rules, the security level of the result of an expression is computed by using both the security level of its operands and the security level of the environment (implicit flow). Each rule updates the before state of the entry point of exception handlers protecting the instruction.

Rule **op**, for example, inserts onto the stack the *lub* between two parameters taken from the stack and the environment of the instruction. A control instruction at program point i (rule **if**) upgrades the level of the environment of instructions belonging to *scope*(i) to the *lub* between the level saved in the *ambient* file and the level of the tested value on the stack (implicit flow). Rule **jsr** pushes the security level of the environment onto the stack. Rule **ret** updates the state of every possible return point (j_1, \dots, j_n) of the subroutine. Rule **throw** updates the before-state of END node, in addition to the entry point of exception handlers. Rule **new** pushes the security level of the environment onto the stack.

A partial order relation is defined on the domain of global states: given two global states $\langle Q, D \rangle$ and $\langle Q', D' \rangle$, $\langle Q, D \rangle \subseteq \langle Q', D' \rangle$ if and only if $Q \subseteq Q'$ and $D \subseteq D'$, where $Q \subseteq Q'$ if and only if $\forall i, Q_i \subseteq Q'_i$ and $D \subseteq D'$ if and only if $D_t \subseteq D'_t$ for every entry t . For memories and stacks, \subseteq is defined point-wise on registers and stack positions. For method entries, \subseteq is defined point-wise for elements in the same positions.

The data flow framework defined by our rules is monotone (Lam and Ullman, 2007). The proof is done by cases for any rule. This is a key point, because we are guaranteed that the analysis completes in a finite number of iterations, and independently from the order of application of the rules.

Analysis results. When the analysis of a package completes, the *ambient* file stores the highest security level of calling environment, actual parameters and return for each method. Illegal flows of information can be detected by looking at methods of shareable interface objects in the *ambient* file.

Checking exported methods. Given a security policy (S) for an exported method mt_p^p , the *return* level of the method must be $\subseteq S$.

Checking imported methods. Given a security policy (S) for an imported method mt_p^p , the *parameters*, *return*, and *calling environment* of the method must be $\subseteq S$.

5. Examples of analysis

In this section, we exemplify the analysis performed by the tool with two examples. In the first example, we show in detail a step-by-step application of the rules applied by the Engine of JCSI in a simple example derived from those described in Section 2. In the second example, we show the functionalities of the JCSI tool by using it for the analysis of the well-known Electronic Purse case study (Cazin et al., 2000). In both cases, we initialized the *ambient* file using the custom strategy shown in Fig. 11.

Example 1: Step-by-step execution. Let us consider the analysis package *b* where method `getSIO()` returns a SIO to *c* if and only

```
.method public getSIO(AID,byte)Shareable
.locals 3

0  aload r0;
1  aload r1;
2  getstatic b/B.C;
3  invokevirtual JCSys.AID.equals()
4  ifeq 11;
5  aload r0;
6  getfield ABalance;
7  push 100;
8  if_icmple 11;
9  aload r0;
10 areturn;
11 aconst_null;
12 areturn;
```

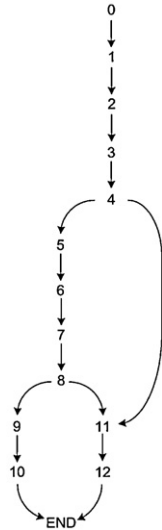


Fig. 13. Bytecode of `getSIO()` of B and its CFG.

if the balance of A is greater than a given threshold (shown at the end of subsection 2). Fig. 13 shows the bytecode of the method and its control flow graph. The bytecode uses 3 registers; on method entrance, the implicit parameter (applet B reference) is saved into register `r0`, the AID parameter is saved into register `r1` and the byte is saved into register `r2`.

Let us analyze `getSIO()` when it is invoked by package `c`. The corresponding entry in the `ambient` file is `getSIObc`. Assume method `foo()` has already been analyzed. In this case the security level of field `ABalance` in the `ambient` file is $\{a, b\}$. Verification of `getSIObc` starts from the initial global state $\langle D^0, Q^0 \rangle$ shown in Fig. 14. The before-state of instruction 0 is set according to the security levels of the method in the `ambient` file.

Instruction 0 is executed. Rule **load** pushes the *lub* of the security level of the environment and the register `r0` onto the stack. The after-state $\langle \{b, c\}, \langle \{b, c\}, \{b, c\}, \{b, c\} \rangle, \{b, c\} \rangle$ is propagated to instruction 1, the successor of 0. The before-state of 1 becomes $Q_1^0 \cup \langle \{b, c\}, \langle \{b, c\}, \{b, c\}, \{b, c\} \rangle, \{b, c\} \rangle$ and instruction 1 is inserted into the WL. When instruction 2 is executed, Rule **getstatic** pushes the *lub* between the security level of the environment and the level of the static field `b/B.C` saved in the `ambient` file onto the stack. Instruction 3 updates the entry corresponding to method `equals()`

in the `ambient` file, removes the parameters from the stack and pushes the return taken from the `ambient` file onto the stack.

There are two implicit flow in the code; the first one starts at 4, the second one starts at 8. Both flows terminate at node END ($ipd(4) = ipd(8) = \text{END}$) and instructions 9, 10, 11, 12 depend on both implicit flows. The security level of the first implicit flow is equal to $\{b, c\}$ and the environment of instructions under the implicit flow is updated accordingly. The security level of the implicit flow at 8 is instead equal to $\{a, b, c\}$ because instruction 6 pushes onto the stack field `ABalance` whose level in the `ambient` file is equal to $\{a, b\}$. Both instructions 9 and 11 are executed in the environment $\{a, b, c\}$ and push onto the stack a constant whose security level is equal to $\{a, b, c\}$. As a consequence, instruction 10 and instruction 12 both return $\{a, b, c\}$. Rule **return** modifies the security level of the return of method `getSIObc` in the `ambient` file, which is set equal to $\{a, b, c\}$. Fig. 15 shows the global state $\langle Q, D \rangle$ at the end of the analysis of the method. Since $D \neq D^0$, the analysis of the whole set of methods is executed again starting from D .

The analysis result is that method `getSIO()` does not guarantee secure information flow. Indeed, from the content of the `ambient` file D at the end of the analysis, we can notice that the return of method `getSIObc` has security level $\{a, b, c\}$. This is a symptom of information flow that carries confidential information. The maximum level should be $\{b, c\}$, and $\{a, b, c\} \not\subseteq \{b, c\}$.

Example 2: The Electronic Purse. Let us consider the well-known Electronic Purse case study of PaCap (Cazin et al., 2000). The case study considers the interactions between a Purse applet and two Loyalty applets (see Fig. 16).

Purse. The Purse applet performs debit and credit operations in different currencies, plus some administration functions. To this purpose, Purse implements a shareable interface, `PurseLoyaltyInterface`, which contains method `getTransaction()` that can be invoked by loyalty applets to get transaction records stored in the transaction log. This log has a limited dimension, thus Purse may over-write old records to save new records. Client applets may need lossless transaction logs, hence Purse offers a log-full service that can be subscribed. Client applets that are registered to the log-full service must implement a shareable interface defined by `Purse(LoyaltyPurseInterface)`. This interface is used by Purse to invoke method `logFull()`, which notifies registered applets that the transaction log is going to be over-written.

AirFrance. AirFrance is a loyalty applet. This applet is a client of Purse and can interact with other loyalty applets. AirFrance implements two shareable interfaces: `LoyaltyAirFranceInterface`,

$D^0 =$	class fields, arrays	$Q^0 =$			
	<code>AObj = {b}</code> <code>ABalance = {a, b}</code> ...				
	static fields				
	<code>B.A = {b}</code> <code>B.C = {b}</code> ...				
	internal methods				
	<code>equals_b^b(({b}, {b})){b}; {b}</code> ...				
	imported methods				
	<code>getAppletSIO_b^b(({b}, {b})){b}; {b}</code> <code>foo_a^b(({a}, {b})){a, b}; {a, b}</code>				
	exported methods				
	<code>getSIO_b^c(({b}, {b}, {b}, {b})){b, c}; {b, c}</code> <code>getSIO_b^c(({a}, {a}, {a}, {a})){b, a}; {a, b}</code> <code>bar_b^c(({b}, {b}, {b}, {b})){b, c}</code>				

	<i>E</i>	<i>M</i>	<i>St</i>
0	$\{b, c\}$	$\langle \{b, c\}, \{b, c\}, \{b, c\} \rangle$	λ
1	$\{\}$	$\langle \{\}, \{\}, \{\} \rangle$	λ
2	$\{\}$	$\langle \{\}, \{\}, \{\} \rangle$	λ
3	$\{\}$	$\langle \{\}, \{\}, \{\} \rangle$	λ
..
12	$\{\}$	$\langle \{\}, \{\}, \{\} \rangle$	λ
END	$\{\}$	$\langle \{\}, \{\}, \{\} \rangle$	λ

Fig. 14. Initial global state $\langle D^0, Q^0 \rangle$ during the analysis of package `b`.

$D =$	class fields, array $AObj = \{b\}$ $ABalance = \{a, b\}$...	$Q =$		E	M	St
	static fields $B.A = \{b\}$ $B.C = \{b\}$...		0	$\{b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	λ
	internal methods $equals_b^b(\{b, c\}, \{b, c\})\{b, c\}; \{b, c\}$...		1	$\{b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	$\{b, c\}$
	$work_b^b(\{b\})\{b\}; \{b\}$		2	$\{b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	$\{b, c\} \cdot \{b, c\}$
	imported methods $getAppletSIO_b^b(\{b\}, \{b\})\{b\}; \{b\} \langle \{b\} \rangle$ $foo_a^b(\{a, b\})\{a, b\}; \{a, b\} \langle \{a, b\} \rangle$		3	$\{b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	$\{b, c\} \cdot \{b, c\} \cdot \{b, c\}$
	exported methods $getSIO_b^c(\{b, c\}, \{b, c\})\{a, b, c\}; \{a, b, c\} \langle \{b, c\} \rangle$ $getSIO_a^a(\{a, b\}, \{a, b\})\{a, b\}; \{a, b\} \langle \{a, b\} \rangle$ $bar_b^c(\{b, c\})\{b, c\}; \{b, c\} \langle \{b, c\} \rangle$		4	$\{b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	$\{b, c\}$
			5	$\{b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	λ
			6	$\{b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	$\{b, c\}$
			7	$\{b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	$\{a, b, c\}$
			8	$\{b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	$\{b, c\} \cdot \{a, b, c\}$
			9	$\{a, b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	λ
			10	$\{a, b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	$\{a, b, c\}$
			11	$\{a, b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	λ
			12	$\{a, b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	$\{a, b, c\}$
			END	$\{b, c\}$	$(\{b, c\}, \{b, c\}, \{b, c\})$	λ

Fig. 15. Global state (Q, D) at the end of the analysis of package b .

which contains, among others, method `getBalance()` which can be invoked by other loyalty applets to get the current number of miles collected; `LoyaltyPurseInterface`, which is needed because AirFrance is registered to the log-full service of Purse.

RentACar. RentACar is a loyalty applet. Similarly to AirFrance, RentACar implements a shareable interface `LoyaltyRentACarInterface` which contains methods that can be invoked by other loyalty applets to get the current number of miles collected (`getBalance()`). RentACar is also a client of Purse, but it is not registered for the log-full service, thus RentACar does not implement the shareable interface `LoyaltyPurseInterface`.

Assume that AirFrance requests RentACar the amount of miles every time Purse notifies AirFrance that the transaction log is full. In this case, the `logFull()` method implemented by AirFrance contains an invocation of method `getTransaction()` of Purse followed by an invocation of method `getBalance()` of RentACar. Applet RentACar, whenever observes an invocation of `getBalance()`, can infer that Purse is going to over-write the transaction log. Thus, even without subscribing to the log-full service, RentACar is able to benefit from such a service. Purse is not able to detect such information flow. Moreover, this flow cannot be detected by the firewall, because (i) both AirFrance and RentACar are allowed to invoke `getTransaction()` of Purse to retrieve the transaction log; (ii) AirFrance is allowed to invoke `getBalance()` of RentACar; (iii) RentACar is allowed to invoke `getBalance()` of AirFrance. The case study is an example of information flow caused by nested calls to methods of shareable interface objects between different packages.

Fig. 17 shows an excerpt of the initial ambient file generated by JCSI for the analysis of AirFrance. The tool represents $\{p1, p2\}$ as $p1+p2$. Let `airfrance`, `purse` and `rentacar` be the security levels

of AirFrance, Purse and RentACar packages, respectively. In the initial ambient file, method `logFull()` implemented by AirFrance and invoked by Purse has security level $\{\text{airfrance}, \text{purse}\}$, while method `getBalance()` implemented by RentACar and invoked by AirFrance has security level $\{\text{airfrance}, \text{rentacar}\}$.

The package does not guarantee secure information flow because the level of the calling environment of `getBalance()` is $\{\text{airfrance}, \text{purse}, \text{rentacar}\}$, while it should be $\{\text{airfrance}, \text{rentacar}\}$. This is due to the implementation of method `logFull()`, which invokes `getBalance()` depending on data provided by other packages (the environment is indeed $\{\text{airfrance}, \text{purse}\}$). According to rule **invoke**, environment, parameters and return of `getBalance()` become $\{\text{airfrance}, \text{rentacar}\} \cup \{\text{airfrance}, \text{purse}\} = \{\text{airfrance}, \text{purse}, \text{rentacar}\}$. Part of the analysis results for package AirFrance are shown in Fig. 18.

We analyzed also the other two packages. The result is that Purse guarantees secure information flow, while RentACar does not. Specifically, the analysis of RentACar shows that, in method `getBalance()`, RentACar invokes `getTransaction()` of Purse for obtaining the transaction log when the level of the calling environment depends on data provided by other packages.

6. Related work

Secure information flow in the bytecode has been studied in many works (e.g. Barthe and Rezk (2005), Genaim and Spoto (2005)). Here we summarize a number of works that are closely related to Java Cards. An extensive survey of the techniques applied for enforcing information flow security policies is not in the scope

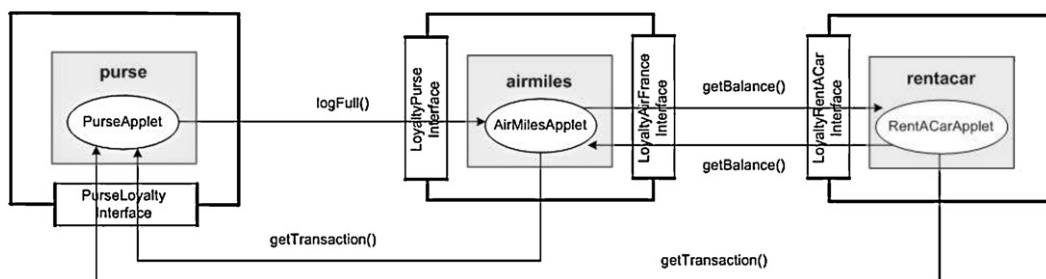


Fig. 16. The Electronic Purse.

```

<Internal methods>
% - install(array of byte,short,byte):void
method_1([B,S,B):V_airfrance^airfrance(airfrance,airfrance,airfrance)##;
    airfrance
% - <init>(array of byte,short,byte):void
method_15([B,S,B):V_airfrance^airfrance(airfrance,airfrance,airfrance,airfrance)##;
    airfrance
...
<Exported methods>
% - logFull()
method_294():V_airfrance^purse(airfrance+purse)##;
    airfrance+purse <airfrance+purse>
% - getBalance()
method_473():S_airfrance^rentacar(airfrance+rentacar) airfrance+rentacar;
    airfrance+rentacar <airfrance+rentacar>
% - updatePoints(byte,short)
method_503(B,S):V_airfrance^rentacar(airfrance+rentacar,airfrance+rentacar,airfrance+rentacar)##;
    airfrance+rentacar <airfrance+rentacar>
% - exchangeRate()
method_607():V_airfrance^purse(airfrance+purse)##;
    airfrance+purse <airfrance+purse>
...

```

Fig. 17. An excerpt of the `ambient` file of Airfrance.

of this paper. Readers interested in a fairly accurate survey on information flow in the bytecode can refer to (Sabelfeld and Myers, 2003).

For Java Cards, (Girard, 1999) introduced the concept of trust relationship between applications and the use of a multi-level security policy to detect illegal data sharing. The verification of information flow has been coped in Cazin et al. (2000) by using model checking. The approach is based on a security policy that defines the allowed flows of information between applets; the verification is done by the SMV model checker. A tool has been developed that computes all call graphs of the application and generates an SMV model per graph. The analysis implemented in JCSI has an advantage with respect to this work, as the implemented iterative data flow we use does not require the explicit construction of the complete abstract transition system (which is required in model checking approaches). An abstract transition system would generally result in a large number of states. For complex applications, the state space may become intractable because the same instruction can be executed in different states, with different security levels for memory, environment and the stack. Instead, during a data flow analysis, the number of states is limited by the number of bytecode instructions in each method.

Recently, a different approach has been proposed in Ghindici and Simplot-Ryl (2008). The authors define a domain specific language for defining security policies, use contracts as a support to the developers for expressing the expected behavior of applications. Then, they annotate the bytecode with proof elements which are verified at loading time by a custom class loader. A Security-by-Contract approach was also developed in Dragoni et al. (2011), where new applications to be installed on the card are first verified by a Policy Checker to check if they comply with the contracts and with the smart card security policy.

The analysis proposed in these approaches can be more accurate than the one implemented in our tool. The analysis implemented in our tool is monomorphic, in the sense that it is not possible to distinguish between instances and method invocations. The analysis could be improved by collecting in the `ambient` file, for each object instance and array instance, the program point of their creation and, for each method, the program point of method invocation. The ultimate aim of our paper, however, is not to

propose an innovative analysis technique, but to explore the effort needed for developing a fairly complete analysis tool to support Java Card applet developers. The functionalities of the CAP file disassembler are directly accessible from the user interface made ready-to-use – as such, that module can be conveniently used by developers for exploring the content of the compiled Java Card applications.

Other works apply static analysis for checking correctness properties of Java card applets, see for example Almaliotis et al. (2008), Albert et al. (2008), or formalize the security policy of the Java Card firewall. In Eluard et al. (2001), Eluard and Jensen (2002), for instance, a formal specification of the firewall is presented and an operational semantics of a subset of the Java Card language that includes the security checks of the firewall is defined. In Caromel et al. (2001) an analysis is proposed for detecting whether an access to shared objects violates the rules of the firewall.

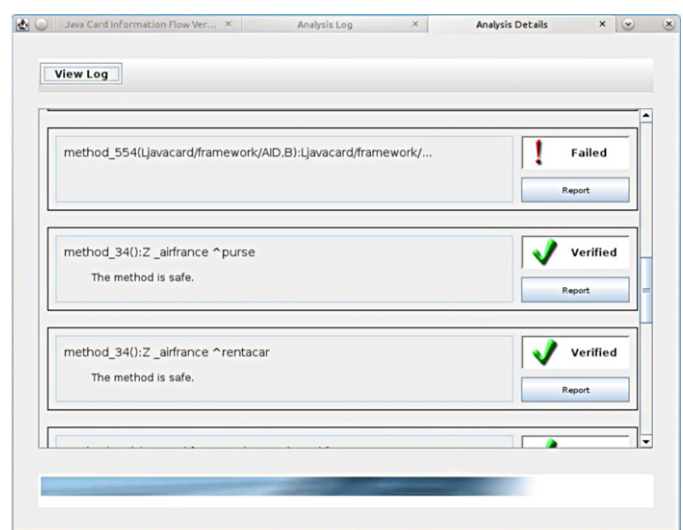


Fig. 18. Example of analysis results.

7. Discussion

The development of the JCSI tool for the analysis of compiled CAP files was more challenging than we thought in the first place. It started as a mere “programming exercise”, where one has to follow specifications and invest time and effort for implementing a tool set that is capable of handling the complete instruction set of the Java bytecode. Almost immediately, it turned out to be instead a continuous error-and-trial approach for obtaining a “right interpretation” of the JVM specification. In fact, although the JVM specification is quite precise about the format of the CAP/Export files, it does not always deliver a complete and clear view of the whole picture.

A topic we have found particularly challenging and error-prone has been the *token-based linking mechanism* of the Java Card system, which provides the rules for referencing objects and methods exported by other packages. Tokens are numbers used in CAP/Export files for identifying classes, methods and fields, instead of Unicode strings used in Java class files. Due to aliasing of token names, the way the Java Card compiler uses tokens to encode methods appears to be potentially fragile.

In order to explain this problem, we report a situation we faced while analyzing the package *Purse* of the PaCap example, where we need to resolve an external method reference in the bytecode of method `delLoyalty(Ljavacard/framework/AID)`. For resolving external references in method calls we apply the procedure outlined in the JVM specification step-by-step. Consider the following invocation in the bytecode of method `removeNotification(Ljavacard/framework/AID;)V` in *Purse*²:

```
17: invokevirtual 596
```

We aim to resolve the identity of method 596. According to the JVM specification, the argument of `invokevirtual` is an index to a constant pool item of type `CONSTANT.VirtualMethodref.info` (see instruction `invokevirtual`, JVM specification [JVM 2.2.2 \(2006\)](#)).

The entry in the constant pool provides the following information: a tag, whose value is always 3; a class item that represents the class that contains a declaration or definition of the virtual method (the class can be defined either in the current package or in an imported package); a token item that represents the virtual method token of the referenced method (Sections 6.7.2, 7.5.54 and 7.5.57 of the JVM specification [JVM 2.2.2 \(2006\)](#)). The fragment of the `purse.cap` CAP file (as disassembled by the DeCAP tool) corresponding to constant pool index 596 is as follows:

```
// purse.cap
constant.pool[596]
  CONSTANT.VirtualMethodref.info
    tag                : 3
    package.token       : 0
    class.token         : 0
    method.token        : 0
```

In this case, as the class is defined in an imported package, the DeCAP represents its reference as a pair, `package.token` and `class.token` (see Section 6.7.1 of the JVM specification [JVM 2.2.2 \(2006\)](#)). The value of `package.token` represents a package token defined in the Import component of the CAP file, which must be used to identify the AID of the imported package. In this case, the AID is `A0:0:0:0:62:0:1`, which identifies the `java.lang` package. The Export file of `java.lang` is `lang.exp`.

The value of `class.token` and `method.token` are then used to identify the class and the method in the Export file. The methods

belonging to the class are described by array `methods` of elements of type `Method.info` (see Section 5.7 in the JVM specification [JVM 2.2.2 \(2006\)](#)). In the considered export file (`lang.exp`), for class token 0 we have the following array:

```
// lang.exp (classes[0].methods)
Method.info[0]
  token                : 0
  access.flags         : PUBLIC
  name.index           : 0
  descriptor.index     : 1
Method.info[1]
  token                : 0
  access.flags         : PUBLIC
  name.index           : 2
  descriptor.index     : 3
```

There is aliasing of token names – the two `Method.info` structures report the same value (0) for `method.token`. Also, they have the same value for the `access.flags` (`public`). The procedure has lead us to a situation where the identification of the method is ambiguous, as the JVM specification does not provide information about how to resolve this situation.

The specification of instruction `invokevirtual` indicates that “The method must not be `<init>`, an instance initialization method, or `<clinit>`, a class or interface initialization method.” (see page 188, instruction `invokevirtual` in the JVM specifications 2.1.1 [JVM 2.2.2 \(2006\)](#)). We believed that this was a constraint enforced in the bytecode – Section 7.1 “Assumptions: The Meaning of Must” of the JVM specification [JVM 2.2.2 \(2006\)](#) suggested how to correctly interpret the text in the specification.

The workaround we used to correctly identify the method was then check all methods names indexed by the `Method.info` structured with identical token, and check whether only one of the candidate methods was *not* an initialization method. The `name.index` and `descriptor.index` in the `Method.info` structures are references to entries in the constant pool component of the Export file:

```
// lang.exp
constant.pool[0]      : <init>
constant.pool[1]      : ()V
constant.pool[2]      : equals
constant.pool[3]      : (Ljava/lang/Object)Z
```

The first `Method.info` structure is therefore related to method `<init> ()V`, and the second to `equals (Ljava/lang/Object)Z`. In this case, the method invoked is `equals (Ljava/lang/Object)Z`. Similar issues also occurred for other Export files. With the information we have at the moment, we don’t know whether there will be particular situations where this workaround will fail.

The issue described above raises the following questions: What are the actual rules implemented in the Java Card compiler/converter? Which aspects of the JVM specification are provided in a way that is open to different interpretations? We believe that these questions are worth of further investigation, as these gaps in the JVM specification may lead to implementations of the Java Card system that may potentially be exposed to unforeseen security breaches.

8. Conclusion

This paper describes a tool for studying the information flow in multi-applicative Java Cards. Based on a multi-level security policy and on the theory of abstract interpretation, the tool analyses the packages installed on card to check if information flow is secure according to a given policy. The tool can also be used to view the content of binary CAP files. The security policy is described in an ambient file, in term of security levels that are assigned to methods. The ambient file is automatically initialized by taking a

² The method name as visualized by the DeCap tool is `method_2953`.

worst-case scenario as default. In such a scenario, any applet installed on card may invoke methods exported by others, and any applet may implement methods of a given interface. The default policy enforces that information shared between two packages only depends on these two packages. The analysis setup can be manually customized by the user, that can modify the constraints posed by the default policy (e.g., allowing third party collaboration) and better select the method entries based on the real interaction patterns among packages. To this purpose, can be of great help the CAP file disassembler provided with the tool. The customization of the analysis setup is up to skilled users. As a future work, the usability of the tool can be improved by, for instance, taking advantage of domain specific languages to specify the security policy and set the entries of the ambient file.

Acknowledgements

This work was partially supported by Fondazione Cassa di Risparmio di Pisa, Italy (no. PR02-182) and by the European Commission through the Network of Excellence ReSIST (IST-026764). This journal paper is based on our previous conference publication (Avvenuti et al., 2009). The authors would like to thank the anonymous reviewers, as their comments helped improve the tool and the manuscript.

Appendix A. Some rules of the abstract interpreter

pop	$\frac{B[i]=pop, Q_i=(\sigma, M, k \cdot St)}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
dup	$\frac{B[i]=dup, Q_i=(\sigma, M, k \cdot St), \gamma=\sigma \cup k}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
op	$\frac{B[i]=\alpha op, Q_i=(\sigma, M, k_1 \cdot k_2 \cdot St), \gamma=\sigma \cup k_1 \cup k_2}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
const	$\frac{B[i]=\alpha const, Q_i=(\sigma, M, St)}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
load	$\frac{B[i]=\tau load r, Q_i=(\sigma, M, St), \gamma=\sigma \cup M(r)}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
store	$\frac{B[i]=\tau store r, Q_i=(\sigma, M, k \cdot St), \gamma=\sigma \cup k}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
if	$\frac{B[i]=if cond L, Q_i=(\sigma, M, k \cdot St), Q_j=(\sigma, M', St'), \gamma=\sigma \cup k}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
goto	$\frac{B[i]=goto L, Q_i=(\sigma, M, St)}{(Q, D) \rightarrow (Q[Q_i \cup (\sigma, M, St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
new	$\frac{B[i]=new \tau, Q_i=(\sigma, M, St)}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \sigma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
getfield	$\frac{B[i]=getfield \tau.f, Q_i=(\sigma, M, k \cdot St), \gamma=\sigma \cup k \cup D(\tau.f)}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
putfield	$\frac{B[i]=putfield \tau.f, Q_i=(\sigma, M, k_1 \cdot k_2 \cdot St), \gamma=\sigma \cup k_1 \cup k_2}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, St), Q_{j \in H(i)} \cup h(Q_i)], D[D_{\tau.f} \cup \gamma])}$
checkcast	$\frac{B[i]=checkcast \tau, Q_i=(\sigma, M, k \cdot St), \gamma=\sigma \cup k}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
newArray	$\frac{B[i]=newArray \tau, Q_i=(\sigma, M, k \cdot St), \gamma=\sigma \cup k}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
aload	$\frac{B[i]=\tau load, Q_i=(\sigma, M, k_1 \cdot k_2 \cdot St), \gamma=\sigma \cup k_1 \cup k_2 \cup D(\tau)}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
astore	$\frac{B[i]=\tau store, Q_i=(\sigma, M, k_1 \cdot k_2 \cdot k_3 \cdot St), \gamma=\sigma \cup k_1 \cup k_2 \cup k_3}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, St), Q_{j \in H(i)} \cup h(Q_i)], D[D_{\tau} \cup \gamma])}$
arraylength	$\frac{B[i]=arraylength, Q_i=(\sigma, M, k \cdot St), \gamma=\sigma \cup k \cup \mathbb{A}(D)}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
instanceof	$\frac{B[i]=instanceof \tau, Q_i=(\sigma, M, k \cdot St), \gamma=\sigma \cup k}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$
invoke	$\frac{B[i]=invoke mt() Q_i=(\sigma, M, k_0 \cdot k_n \cdot St), D_{mt}=(\sigma_0, \sigma_1, \dots, \sigma_n) \sigma_r; \sigma_e, \gamma=\sigma_e \cup \sigma}{(Q, D) \rightarrow (Q[Q_{i+1} \cup (\sigma, M, \sigma_r \cup \gamma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D[D_{mt} \cup (k_0 \cup \gamma, k_1 \cup \gamma, \dots, k_n \cup \gamma) \gamma; \gamma])}$
return	$\frac{B[i]=\tau return, Q_i=(\sigma, M, k \cdot St), D_{mt}=(\sigma_0, \sigma_1, \dots, \sigma_n) \sigma_r; \sigma_e, \gamma=\sigma \cup k}{(Q, D) \rightarrow (Q[Q_{END} \cup (\sigma, M, \lambda), Q_{j \in H(i)} \cup h(Q_i)], D[D_{mt} \cup (\sigma_0, \sigma_1, \dots, \sigma_n) \gamma; \sigma_e])}$
athrow	$\frac{B[i]=\tau throw Q_i=(\sigma, M, k \cdot St), \gamma=\sigma \cup k}{(Q, D) \rightarrow (Q[Q_{END} \cup (\gamma, M, \lambda), Q_{j \in H(i)} \cup h(Q_i)], D)}$
jsr	$\frac{B[i]=jsr L, Q_i=(\sigma, M, St)}{(Q, D) \rightarrow (Q[Q_i \cup (\sigma, M, \sigma \cdot St), Q_{j \in H(i)} \cup h(Q_i)], D)}$

ret
$$\frac{B[i]=ret r, Q_i=(\sigma, M, St)}{(Q, D) \rightarrow (Q[Q_{j_1}, \dots, Q_{j_n} \cup (\sigma, M, St), Q_{j \in H(i)} \cup h(Q_i)], D)}$$

References

- Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D., 2008. COSTA: a cost and termination analyzer for Java bytecode. In: Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode). Electronic Notes in Theoretical Computer Science. Elsevier, Budapest, Hungary.
- Almaliotis, V., Loizidis, A., Katsaros, P., Louridas, P., Spinellis, D., 2008. Static program analysis for Java Card applets. In: CARDIS '08: Proceedings of the 8th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications. Springer-Verlag, Berlin, Heidelberg.
- Amtoft, T., Bandhakavi, S., Banerjee, A., 2006. A logic for information flow in object-oriented programs. SIGPLAN Notices 41, 91–102.
- Avvenuti, M., Bernardeschi, C., De Francesco, N., Masci, P., 2003. Java bytecode verification for secure information flow. SIGPLAN Notices 38, 20–27.
- Avvenuti, M., Bernardeschi, C., De Francesco, N., Masci, P., 2009. A tool for checking secure interaction in Java cards. In: Waeselynyck, H. (Ed.), Proceedings of the 12th European Workshop on Dependable Computing, EWDC 2009. Toulouse, France, p. 8. <http://hal.archives-ouvertes.fr/hal-00380664/en/>.
- Ball, T., 1993. What's in a region? Or computing control dependence regions in near-linear time for reducible control flow. ACM Letters on Programming Languages and Systems 2 (1–4), 1–16.
- Barbuti, R., Bernardeschi, C., De Francesco, N., 2004. Checking secure information flow in assembly code by abstract interpretation. The Computer Journal 47 (1), 25–45.
- Barthe, G., Rezk, T., 2005. Secure information flow for a sequential Java virtual machine. In: ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI'05. ACM.
- Bell, D.E., Padula, L.J.L., 1973. Secure computer systems: mathematical foundations and model. Technical Report M74-244, MITRE Corporation, Bedford, Massachusetts.
- Caromel, D., Henrio, L., Serpette, B., 2001. Context inference for static analysis of Java Card object sharing. In: In, I., Attali, T., Jensen (Eds.), Conference on Research in Smart Cards, E-smart 2001. LNCS 2140, Springer-Verlag, pp. 43–57.
- Cazin, J., El-Marouani, A., Girard, P., Lanet, J., Wiels, V., Zanon, G., 2000. The PACAP prototype: a tool for detecting illegal flows. In: Java Card Workshop Proceedings, Cannes.
- Chen, Z., 2000. Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison Wesley.
- Cousot, P., Cousot, R., 1992. Abstract interpretation frameworks. Journal of Logic and Computers 2, 511–547.
- Denning, D., 1976. A lattice model of secure information flow. Communications in ACM 19 (5), 236–243.
- Denning, D., Denning, P., 1977. Certification of programs for secure information flow. Communications in ACM 20 (7), 504–513.
- Dragoni, N., Lostal, E., Gadyatskaya, O., Massacci, F., Paci, F., 2011. A load time policy checker for open multi-application smart card. IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2011), 153–156.
- Eluard, M., Jensen, T., 2002. Secure object flow analysis for Java card. In: Fifth Smart Card Research and Advanced Application (CARDIS'2002) Conference Proceedings. USENIX, pp. 97–110.
- Eluard, M., Jensen, T., Denne, E., 2001. An operational semantics of the Java Card Firewall. In: In, I., Attali, T., Jensen (Eds.), Conference on Research in Smart Cards, E-smart 2001. LNCS 2140, Springer-Verlag, pp. 95–110.
- Genaim, S., Spoto, F., 2005. Information flow analysis for Java bytecode. In: Cousot, R. (Ed.), Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17–19, 2005, Proceedings. Vol. 3385 of Lecture Notes in Computer Science. Springer-Verlag, pp. 346–362.
- Ghindici, D., Simplot-Ryl, I., 2008. On practical information flow policies for Java-enabled multiapplication smart cards. In: Grimaud, G., Standaert, F.-X. (Eds.), Smart Card Research and Advanced Applications. LNCS 5189, Springer-Verlag, pp. 32–47.
- Girard, P., 1999. Which security policy for multiapplication smart cards? In: USENIX Workshop on Smartcard Technology, pp. 21–28.
- Lam, A., Ullman, S., 2007. Compilers: Principles, techniques and tools. Addison Wesley.
- Leroy, X., 2001. Java bytecode verification: an overview. In: 13th International Conference on Computer Aided Verification, LNCS 2102, Proceedings, pp. 265–285.
- Sabelfeld, A., Myers, A., 2003. Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21 (1).
- Siveroni, I., Formal methods for smart cards 2004. Operational semantics of the Java Card virtual machine. Journal of Logic and Algebraic Programming 58 (1–2), 3–25.
- Smith, G., 2007. Principles of secure information flow analysis. In: Christodorescu, M., Jha, S., Maughan, D., Song, D., Wang, C. (Eds.), Malware Detection. Vol. 27 of Advances in Information Security. Springer, US, pp. 291–307, 10.1007/978-0-387-44599-1.13.
- Virtual Machine Specification, Java Card Platform, Version 2.2.2, <http://www.oracle.com/technetwork/java/javame/javacard>.
- Marco Avvenuti is associate professor with the Department of Information Engineering of the University of Pisa. He graduated in Electronic Engineering (1989) and received his PhD in Information Engineering from the University of Pisa (1993). His

current research interests are in the areas of mobile and pervasive computing and wireless sensor networks. He is a member of the IEEE Computer Society.

Cinzia Bernardeschi received the Laurea degree in Computer Science in 1987 and the PhD degree in 1996, both from the University of Pisa. She is associate professor with the Department of Information Engineering of the University of Pisa. Her research interests are in the area of software engineering, dependable systems and application of formal methods for specification and verification of safety-critical systems.

Nicoletta De Francesco is full professor of Computer Engineering at the Department of Information Engineering of the University of Pisa. Her current research

interests are in the area of formal verification of distributed systems by model checking and temporal logic. Recently she is studying the application of abstract interpretation and model checking techniques to check secure information in mobile code.

Paolo Masci obtained his PhD in 2008 from the Department of Information Engineering, University of Pisa, and his dissertation was on software solutions for battery-aware reconfiguration of wireless sensor networks. His research interest are in the field of automated reasoning. He is currently working on formal specification and verification of interactive medical devices.