The Journal of Systems and Software 85 (2012) 2479–2493

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.

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).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.05.061

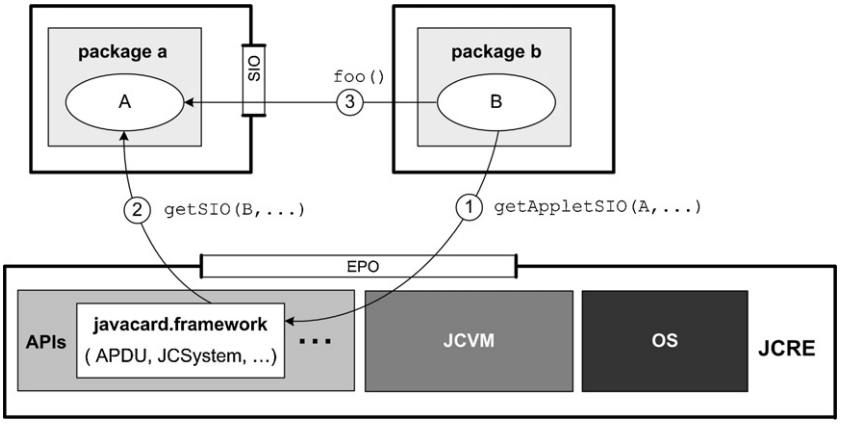

Fig. 1. The Java Card System.

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 (JCRE) for managing resources, executing programs and applying access control mechanisms. The JCRE 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 JCRE services (see Fig. 1).

Java Card applications and JCRE's APIs are bundled into *packages*, which are data structures that store the compiled bytecode of Java classes and interfaces. A package[1] 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 (*EPO*s) and Shareable Interface Objects (*SIO*s). EPO objects belong to the JCRE'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 $\mathcal{B} = \{boolean, short, byte, int\}$ of primitive (basic) types, the set $\mathcal{C}' = \mathcal{C} \cup \{Object\}$ 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 $\mathcal{B}' = \mathcal{B} \cup \{Object\}$. 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 $[\tau$ to denote arrays of type τ and the syntax $\tau_0.mt(\tau_1, \ldots, \tau_n) : \tau_r$ to denote the method mt of the class or interface $\tau_0 \in \mathcal{C} \cup \mathcal{I}$ with arguments of type $\tau_1, \ldots, \tau_n$ 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 $\mathcal{L} = \{0, 1, \ldots\}$ 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 subscript 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.