A tool for checking secure interaction in Java Cards

Marco Avvenuti, Cinzia Bernardeschi, Nicoletta de Francesco, Paolo Masci

A tool for checking secure interaction in Java Cards

M. Avvenuti, C. Bernardeschi, N. De Francesco, P. Masci
Dipartimento di Ingegneria dell'Informazione
Università di Pisa
Italy

***Abstract*—We present an approach based on a multilevel security policy and the theory of abstract interpretation for checking secure interaction between applications in Java Cards. The security policy is defined by the user, which assigns security levels to Java Card applications. Actual values are abstracted into security levels, and an abstract interpreter executes the bytecode of applications in the abstract domain. We show JCSI, a tool that implements the presented approach. JCSI can be used to check the binary code of Java Card applications before their installation on-card.**

I. Introduction

Java Cards are pocket-size cards equipped with an embedded microcontroller that supports the execution of a Java Virtual Machine. Typical applications include credit and loyalty systems, electronic cash, healthcare and government identification. In multi-applicative Java Cards, new applications can be downloaded and installed even after card issuance [4].

The Java Card system has built-in mechanisms to enforce security and protection. Indeed, Java Card applications are executed within a protected space, called *context*. Each application is associated with a unique context, and a Java Card component, called *firewall*, uses an access control mechanism to limit the access rights of applications. The basic rule enforced by the firewall is that applications can access only objects of their own context. Information exchange between applications is realised through special objects, called *shareable interface*s. Although firewall and runtime environment provide powerful security mechanisms, they are inadequate to protect applications against unauthorised information propagation. Indeed, in order to check secure interaction between applications, the analysis of all information flows among applications installed on-card is required. The Electronic Purse case study [3] is an example in which an unauthorised application avoids the rules of the firewall and is able to get access to a service provided by another application by using information gathered from a third application installed on-card.

The contribution of this work is an approach for the analysis of secure interaction in Java Cards. The approach is based on a multi-level security policy and the theory of abstract interpretation of the operational semantics [5], [6], [12]. Abstract interpretation is a method for analysing programs in order to collect approximate information about their runtime behaviour. It is based on non-standard semantics, i.e. a semantic definition in which a simpler abstract domain replaces the standard one, and the operations are interpreted on the new domain. The use of abstract interpretation allows, on one side, being semantics based, to accept as secure a wide class of programs, and, on the other side, being rule based, to be fully automated. The security policy assigns unique security levels to applications, and data are initially assigned to the security level of the application they belong to. Applications are executed in the abstract domain through a set of abstract rules that trace the flow of information. Java Card applications are analysed one at a time and the analysis is performed method per method. We associate a state to every instruction in the bytecode. The state takes into account the security level of local variables, method invocations and the heap. Instructions of the bytecode are executed by an abstract interpreter by means of a fixpoint iteration. A similar fixpoint iteration is used by the Bytecode Verifier, which checks type correctness of the Java bytecode [10].

We developed a tool suitable to analyse the binary code of Java Card applications, which are released to users when installing new services on-card. The complete Java Card instruction set is handled by our analysis.

II. Problem statement

The Java Card system relies on the Java Card Runtime Environment (JCRE), which is responsible for managing resources, executing programs and applying access control mechanisms (see Figure 1). JCRE consists of a native operating system, the Java Card Virtual Machine (JCVM) and the Application Programming Interfaces (APIs), which define the set of services provided by the system. Java Card applications reside in a user space, and they can use JCRE services. Both applications and APIs are implemented as *packages*. A package contains the compiled form of a set of Java classes and interfaces. Interfaces define only methods. Shareable interfaces are special interfaces used to define methods that can be shared between different contexts.

JCRE assigns an unique identifier (*AID*) to packages installed on-card. Some classes contained in the package act like servers, and wait for commands dispatched by the JCRE. Such classes are called *applets*.

A. Firewall & object sharing mechanisms

The Java Card firewall limits the boundaries of applets. The basic rule enforced by the firewall is the following: applets may access only objects belonging to the context of their own package. Namely, at runtime, the firewall checks all operations, and denies the executed applet to access objects belonging to contexts different from the *current active context*, that is the context of the applet which is being executed.

The Java Card system provides mechanisms for sharing information and services between applets of different contexts. Sharing is based on Entry Point Objects (*EPO*s) and Shareable Interface Objects (*SIO*s). EPOs are objects of JCRE; examples of EPO classes are `APDU`, which stores applet commands, `AID`, which identifies applets installed on-card, and `JCSystem`, which provides methods suitable for a limited inspection of the call stack of shared methods. SIOs, on the other hand, are objects belonging to applets and they are used to implement shared objects interfaces. SIOs must always extend `Shareable`. SIOs can be accessed by applets belonging to any package. Moreover, pointers to SIOs can be passed from one applet to another, for example by saving a pointer to a SIO in a public field. Access policies to methods defined in the SIO are managed by the applet that implements the SIO.

In order to exemplify the object sharing mechanism, let us consider the scenario depicted in Figure 1. Assume that applets `A` and `B` belong to different contexts, and that `B` wants to invoke `foo()`, a shared method provided by `A`. The chain of method invocations is the following. *(1)* Applet `B` invokes `getAppletSIO(AID)`, implemented by a static EPO; `AID` identifies the applet implementing the shared object (`A`, in this case). *(2)* Method `getAppletSIO(AID)` dispatches a request for the shareable object to `A`. The request is done by triggering the invocation of method `getSIO(AID)` implemented by `A`, where `AID` identifies the applet that requested the shared object (`B`, in this case). *(3)* If applet `A` accepts the request, a pointer to the shareable object is returned to JCRE, which in turn returns the pointer to `B`. If the request is not accepted – this may happen, for example, when `A` uses an authentication policy and `B` is not authorised, or simply because `A` does not implement the `getSIO(AID)` – a null pointer is returned to JCRE, which in turn returns a null pointer to `B`. Applet `B`, after having received the pointer to the SIO implemented by `A`, may invoke method `foo()`. When method `foo()` is invoked, JCRE automatically performs a context switch: the current active context (i.e, the context of `B` in this case) is saved and subsequently restored when the execution of method `foo()` completes. Applet `A` is able to ensure the actual identity of the caller by using method `getPreviousContext()` of a static system EPO, which returns the identifier of the owner of the previous active context.

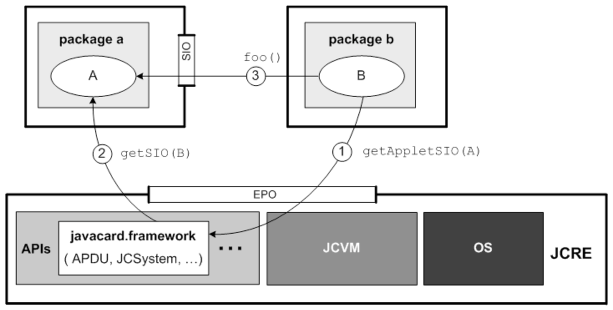


Fig. 1. The Java Card Environment.

B. Limits of the firewall

Let us consider the Electronic Purse case study [3], which considers the interactions between a purse applet and two loyalty applets (see Figure 2).

Purse. Purse 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. The transaction log has a limited dimension, thus Purse may overwrite old records to save new records. Purse offers a logfull service that can be subscribed by client applets that need lossless transaction logs. Client applets that are registered to the logfull service must implement a shareable interface defined by Purse (`LoyaltyPurseInterface`). This interface is used by Purse to invoke method `logFull()`, which notifies applets registered to the service 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`, which contains, among others, method `getBalance()` which can be invoked by other loyalty applets to get the current number of points collected; `LoyaltyPurseInterface`, which is needed by AirFrance since the loyalty applet subscribed the logfull service of Purse.

RentACar. RentACar is a loyalty applet. Similarly to AirFrance, RentACar implements a shareable interface `LoyaltyRentACarInterface` which contains, among others, method `getBalance()` which can be invoked by other loyalty applets to get the current number of points collected. RentACar is a client of Purse, as well, but RentACar is not registered to the logfull service, thus RentACar does not implement the shareable interface `LoyaltyPurseInterface`.

Assume that AirFrance requests RentACar the amount of points every time it is notified by Purse that the transaction log is full. Namely, assume that the implementation of `logFull()` in AirFrance has an invocation to method `getTransaction()` of Purse followed by `getBalance()` of RentACar. In this case RentaACar, upon receiving the invocation of `getBalance()`, may infer that Purse is going to over-write the transaction log, and may benefit from the logfull service of Purse even without subscribing to the service. Purse is not able to detect such problem. Moreover, this problem cannot be avoided by the firewall, since both AirFrance and RentACar are allowed to invoke `getTransaction()` to retrieve the transaction log.

The Electronic Purse shows an an example of secure interaction violation caused by nested calls to shareable interface methods between different packages. There are several other

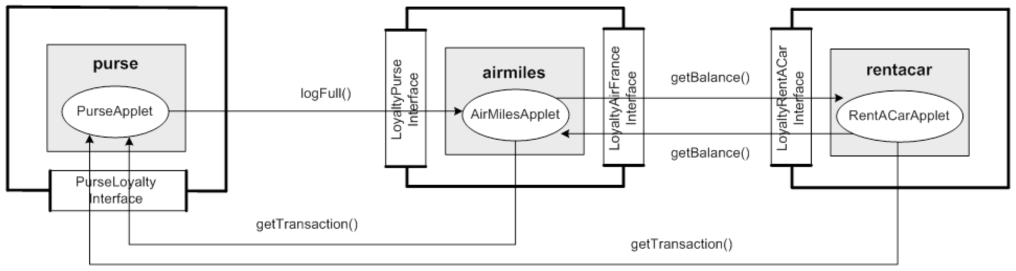


Fig. 2. The Electronic Purse.

methods to avoid the firewall policy (see Subsection III-C).

III. The proposed approach

In order to analyse secure interaction, we use the theory of secure information flow in a program [11]. We assume the following 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. $(\Sigma, \subseteq)$ 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 analysis operates over security levels in Σ. For any element p of P, the singleton set $\{p\}$ denotes information whose security level is that of package p. The set $\{p_1, p_2\}$ denotes information that depends on both packages p_1 and p_2. The minimum of Σ is the empty set.

We use the following notation. Given a package p, $Import(p)$ is the set of imported methods, i.e., methods invoked by p which belong to a SIO of another package, $Export(p)$ is the set of exported methods, i.e., the set of methods of a SIO implemented by p, and $Internal(p)$ is the set of internal methods of p. Given a method mt, $Packages(mt)$ is the set of packages that implement mt. Given a class τ, we use the syntax $\tau.f$ to denote field f of class τ. We use $[\tau$ to denote arrays of type τ.

A set of user packages satisfies secure interaction if methods shared between package p and package p' depend at most on the security levels of p and p' ($\subseteq \{p, p'\}$). The analysis checks the following constraints:

- *Imported methods*: the method *call* is correct. If the method belongs to a SIO of p', this means that the calling environment and parameters depend at most on p and p'.
- *Exported methods*: the method *return* is correct. If the method is invoked from p', this means that the return depends at most on p and p'.

Packages are analysed one at a time. When we verify a package, we have an `ambient` file that maintains the *security level* of methods, objects, and arrays in the heap.

A. Ambient file

Heap. 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.

When analysing package p, the security level of class fields is initially set to the security level of p. The level of class fields will be updated during the analysis according to the flow of information and dependencies between instructions in the program. When the analysis completes, the security level of class field $\tau.f$ consists of the maximum security level of field f of all objects of class τ. We abstract from array instances and indexes in the `ambient` file. The security level of arrays is initially set to the security level of the analysed package. When the analysis completes, the security level of $[\tau$ is the maximum security level saved into all arrays of type $[\tau$.

Methods. The security level of a method characterises how the method is called in terms of the maximum security level of method's actual parameters, calling environment and return. The `ambient` file saves the security level of all methods of a package[1]. Each method is denoted by an expression of the form $mt_p^{p'}(\tau_1, \ldots, \tau_n)\tau; \tau'$, where $\tau_1, \ldots, \tau_n$ are the arguments, τ is the return and τ' is the calling environment; p is the package that implements mt and p' is the package that invokes mt.

Let $\mathcal{M}$ be the set of methods defined in the `ambient` file. For internal methods, a single instance of the method is inserted in the `ambient` file with levels of arguments, return and calling environment equal to the level of the package: $mt_p^p(p, \cdots, p)p; p \ \forall mt \in Internal(p)$.

Methods imported by p are inserted in the `ambient` file with security levels for parameters and calling environment equal to the level of package p. The level of the method return in the `ambient` file is computed as the *lub* between packages implementing such method, since we cannot statically derive the actual implementation that will be invoked at run-time:
$mt_{p'}^p(\{p\}, \cdots, \{p\})S; \{p\} \ \forall mt \in Import(p)$, where
$S = lub(\bigcup_{p' \in Packages(mt)} \{p, p'\})$

For every method exported by p, an instance of the method is inserted in the `ambient` file for every package that invokes such method. For the method instance invoked by package p', the level of method's parameters, return and calling environment is $\{p, p'\}$:
$mt_p^{p'}(\{p, p'\}, \cdots, \{p, p'\})\{p, p'\}; \{p, p'\} \ \forall mt \in Export(p)$

Note that method $getSIO()$ is implicitly exported by applets that implement such method. Moreover, the invocation

[1]Note that an interface can be declared in a package and implemented by other packages. Moreover, the same interface can be implemented by many packages.

of $getSIO()$ is triggered by $getAppletSIO()$. As a consequence, an instance of $getSIO()$ for all packages that invoke $getAppletSIO()$ is inserted in the `ambient` file:
$getSIO_p^{p'}(\{p,p'\},\cdots,\{p,p'\})\{p,p'\};\{p,p'\}$ $\forall p'$ such that $getAppletSIO() \in Import(p')$

B. Analysis of a package

The analysis of a package is based on an iterative process that, starting from the initial `ambient` file D^0, verifies all methods in $\mathcal{M}$ that are implemented by the package. The list of methods to be analysed is maintained. Whenever a security level changes in the `ambient` file, all methods must be re-verified. The analysis uses an abstract interpreter, named Method Security Checker (MSC), to verify a single method. The analysis is carried out with the algorithm shown in Figure 3. Given a method $mt \in \mathcal{M}$ implemented by p 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 $D' = D$, another method is analysed; if $D' \neq D$, all methods are verified again starting from the `ambient` file D'. The verification terminates when, starting from an `ambient` file, all methods are analysed and the new `ambient` file is unchanged.

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

Fig. 3. Main steps of the analysis of a package

C. Analysis of a method

Basics

JVML is a stack based assembly language: there is an operand *stack* and a *memory* containing local variables (registers). The language has typed instructions (for example, τ`load` x pushes the content of type τ of register x onto the operand stack) and includes subroutines and exceptions. The bytecode of a method mt is a sequence of JVML instructions B_{mt}. We assume instruction are numbered starting from 1 and we use $B_{mt}[i]$ to denote instruction i of the bytecode.

We briefly recall basic concepts of secure information flow in a program [11]. A program, with variables partitioned into two disjoint sets of high and low security, has secure information flow if observations of the final value of low security variables do not reveal any information about the initial value of high security variables. Assume `y` is a high security variable and `x` a low security one. Examples of violation of secure information flow in a high level language are: `(1) x:=y` and `(2) if y=0 then x:=0 else x:=1`. Statement `(1)`

```
1  iload   y
2  ifcmpz  5
3  iconst  1
4  goto    6
5  iconst  0
6  istore  x
7  return
```

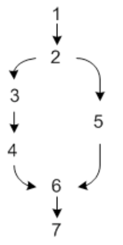


Fig. 4. A simple bytecode and its CFG

```
1  aload      x          1  aload      x
2  aload      y          2  aload      y
3  getfield   B.f2       3  getfield   B.f2
4  putfield   A.f1       4  ifge       6
5  return                5  invoke     mt
                         6  return
   (a)                      (b)
```

Fig. 5. Secure information flow violation (a) Objects (b) Methods

contains an *explicit* information flow from `y` to `x`, statement `(2)` contains an *implicit* information flow: in both cases, the final value of `x` reveals information on the value of the higher security variable `y`.

Our analysis considers information flow in the Java bytecode. Since the bytecode is unstructured, implicit flows are handled through the control flow graph (CFG) and the notion of immediate postdominator (*ipd*). Consider the bytecode in Figure 4, which corresponds to statement `(2)`. The branching instruction `ifcmpz j` causes the beginning of an *implicit flow*: if the conditional instruction is at position i, then the implicit flow affects all instructions belonging to paths from i to $ipd(i)$. Note that $ipd(i)$ is the first instruction which is not affected by the implicit flow, since it represents the point where the two branches join together. Let us consider the graph in Figure 4. We have that $ipd(2) = 6$, and both instructions `iconst 1`[2] and `iconst 0` are in the scope of the control instruction. The bytecode violates secure information flow, since there is an implicit flow from the high security variable y to the constants `1` and `0`, and instruction `istore` x saves a high constant into the low variable x.

An example of secure information flow violations through objects is shown in Figure 5(a). Assume register x contains a reference to an object of class `A`, while register y contains a reference to an object of class `B`. Moreover, assume that the security level of x, y and `A.f1` is low, while the level of `B.f2` is high. The code represents an explicit information flow from an high security field to a low security field. Figure 5(b) shows an example of secure information flow violation through a method invocation. Let mt be a method of low security level class `A`. Method mt is invoked or not according to the value of a high security field.

[2]Instruction `iconst 1` pushes the constant `1` onto the stack.

Our strategy

The semantics of the language has been enhanced to convey the level of information flow during execution. This is accomplished *i)* by annotating each value with the level of the information flows, both explicit and implicit, which the value depends on, and *ii)* by executing instructions under a *security environment*, which represents the least upper bound of the security levels of the open implicit flows when an instruction is executed.

Data are pairs (k, σ), where k is the value and σ is the security level. We model a state of the program execution as a tuple $\langle \sigma, i, m, st \rangle$, where σ is the security level of the environment, i is the address held by the program counter, m is the memory representing the current state of registers and st is the current state of the operand stack. In the state $\langle \sigma, i, m, st \rangle$, instruction i must be executed. For example, the rule for τ`load` x pushes onto the stack a value with security level equal to the least upper bound between the security level of $m(x)$ and the environment and assigns $i+1$ to the program counter:

$$\tau\textbf{load} \quad \frac{B[i] = \tau\texttt{load}\, x \qquad m(x) = (k, \sigma')}{\langle \sigma, i, m, st \rangle \longrightarrow \langle \sigma, i+1, m, (k, \sigma \cup \sigma') \cdot st \rangle}$$

The standard concatenation operator is $\cdot$ and the empty stack is represented by the symbol λ. We assume that the top of the stack appears on the left hand-side of the sequence (i.e., given $st = s_1 \cdots s_n$, element s_1 is the top of the stack).

We define an abstract semantics that abstracts from actual values and maintains only annotations on security levels. Any memory m is abstracted into an abstract memory M such that any variable x holds the security level of $m(x)$. Similarly, an abstract operand stack St keeps the security levels of the items in the corresponding stack st. For example, the rule for τ`load` is the following:

$$\tau\textbf{load} \quad \frac{B[i] = \tau\texttt{load}\, x \qquad M(x) = \sigma'}{\langle \sigma, i, M, St \rangle \longrightarrow \langle \sigma, i+1, M, (\sigma \cup \sigma') \cdot St \rangle}$$

A method is executed in the abstract domain starting from the security level assigned to arguments and calling environment in the `ambient` file.

Note that an invoke instruction updates the security level of the invoked method in the `ambient` file, and the security level returned by the invoked method is taken from the `ambient` file. A read and/or write operation of object fields or arrays refers to the security levels saved in the `ambient` file.

D. The abstract interpreter

The abstract interpreter MSC performs an abstract execution of the bytecode of a method according to a data flow analysis. Given a method mt, each instruction $i \in B_{mt}$ is assigned a state $\langle Q_i, D \rangle$, which represents the state in which instruction i is executed. D is the `ambient` file and Q_i is an abstraction of the JCVM's state *before* the execution of i. Q_i is a triple (E, M, St), where $E \in \Sigma$ is the security level of the environment, $M : \mathit{Registers} \rightarrow \Sigma$ is a mapping from local registers to security levels (the memory) and $St \in \Sigma^\star$ (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). In the following, we use Q to denote the set of states Q_i.

A partial order relation on the domain Q is defined. This relation is induced from the ordering relation among security levels. For example, the bottom element of memories is assigned to all registers. Given two memories M_1 and M_2, $M_1 \subseteq M_2$ iff, for every register x, $M_1(x) \subseteq M_2(x)$. Given two stacks St_1 and St_2, $St_1 \subseteq St_2$ iff each item in St_1 is $\subseteq$ of the corresponding item in St_2. Given two states $Q_i = (E, M, St)$ and $Q'_i = (E', M', St')$, $Q_i \subseteq Q'_i$ iff $E \subseteq E', M \subseteq M', St \subseteq St'$. The least upper bound operation on Q is defined point-wise on memories, stacks and environments: $(E, M, St) \cup (E', M', St') = (E \cup E', M \cup M', St \cup St')$.

The abstract interpreter is based on a set of rules for instructions. The rules defines a relation $\rightarrow \subseteq \langle Q, D \rangle \times \langle Q, D \rangle$. Figure 6 show some rules. Given Q, the notation $Q[Q_i \cup = (\sigma, M, St)]$ is used to denote a new state which is equal to Q except for the entry Q_i, that is set equal to the least upper bound between Q_i and (σ, M, St). The rules are defined for all bytecode instructions. For instance, rule **if** (for $B[i] = \texttt{if}\, L$) updates the successors Q_{i+1} and Q_L taking into account the security level of the environment and the new operand stack. Moreover, it updates also the security environment of every instruction $j \in \mathit{scope}(i)$. Rule **getfield** leaves onto the stack the *lub* between the security level of a class field in the `ambient` file, the parameter on the stack and the environment. Exception and subroutines are handled with a proper modification of the CFG: for exceptions, the after state of protected instructions and implicit flows is propagated to the entry point of exception handlers; for subroutines, the after state of `ret` x is propagated to all possible return point of the subroutine.

Instructions to be verified are inserted into a worklist `WL`, initialised by inserting instruction 1. When instruction i is executed, the state Q_i is considered 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, since an instruction j can be the successor of more than one instruction, Q_j stores a state that merges all possible states in which j can be executed. When `WL` is empty, the verification of a method completes. The initial state $\langle Q^0, D^0 \rangle$ reflects the state of the JCVM on method entrance. The fixpoint is reached with a finite number of iterations and the termination is guaranteed independently from the order of application of the rules.

Figure 7 shows the result of the analysis applied to the bytecode in Figure 4. Q_1 consists of a memory with x and y, a low and a high variable, respectively, an empty operand stack and a low security environment. Instruction 1 loads y onto the operand stack. The after-state of 1 becomes the before-state of 2. Instruction 2 pops an element from the stack and updates the environment of instructions in its scope (3, 4 and 5) with the level of the implicit flow (high in this case). The after-state

$$\textbf{if}\quad \frac{B[i] = \texttt{if}cond\,L,\quad Q_i = (\sigma, M, k\cdot St),\quad Q_j = (\sigma', M', St'),\quad \gamma = \sigma \cup k \cup \sigma'}{\langle Q, D\rangle \to \langle Q[Q_{j,j\in scope(i)} \cup\!= (\gamma, M', St'), Q_{j\in\{i+1,L\}} \cup\!= (\gamma, M\cup M', St \cup St')], D\rangle}$$

$$\textbf{getfield}\quad \frac{B[i] = \texttt{getfield}\ \ \tau\texttt{.f},\quad Q_i = (\sigma, M, k\cdot St),\quad \gamma = \sigma\cup k \cup D(\tau.f)}{\langle Q, D\rangle \to \langle Q[Q_{i+1} \cup\!= (\sigma, M, \gamma\cdot St)], D\rangle}$$

Fig. 6. Examples of abstract execution rules

	E	$(M(x), M(y))$	St
Q_1	l	(l,h)	λ
Q_2	l	(l,h)	h
Q_3	h	(l,h)	λ
Q_4	h	(l,h)	h
Q_5	h	(l,h)	λ
Q_6	l	(l,h)	h
Q_7	l	(h,h)	λ
Q_{END}	l	(h,h)	λ

Fig. 7. An example of data-flow analysis

of 2 becomes the before-state of 3 and 5 (the successors of 2). When the data flow analysis completes, the low security variable x contains a high value.

IV. Detecting illegal flows in Java Cards

When the analysis terminates, the `ambient` file records the highest implicit flow for method's calls, the least upper bound of the security levels of data flowing into method parameters, arrays and objects in the heap during the execution of the package. Illicit flow of information can be detected looking at SIO methods in the `ambient` file, since interface methods must not exceed a given security level.

1) **Exported Methods.** For every exported method $mt_p^{p'}(\tau_1, \cdots, \tau_n)\tau; \tau'$ in the `ambient` file, the security level of the return must be $\subseteq \{p + p'\}$. This way we assess that the method is not releasing information to package p' that depends on packages different form p and p'. In the case of $getSIO()$, we check that the release of the SIO between two packages p and p' (p provides the SIO, and p' requests the SIO) does not depend on a third package.
2) **Imported Methods.** For every imported method $mt_{p'}^{p}(\tau_1, \cdots, \tau_n)\tau; \tau'$ in the `ambient` file, the security level of method parameters and calling environment must be $\subseteq glb(\bigcup_{\bar{p}\in Packages(mt)}\{p, \bar{p}\})$. This way we assess that a method invocation does not release information to package p' that depends on packages different form p and p'.

The approach proposed in this work is conservative, in the sense that all insecure packages are rejected, but we may also reject some secure packages, since in the abstract analysis all branches of control instructions are checked, even those that in the real execution would have never been executed. Moreover, when a shareable interface method is invoked, we assume that every package implementing the method can be invoked. The precision of the approach can be enhanced by annotating method invocations in the bytecode with information about the callers' identity. Such information can be provided by package developers.

V. The JCSI tool

JCSI is a tool that analyses programs compiled into *CAP* files, which is the standard Java Card binary file format. When a CAP file is analysed, JCSI assumes that it is type correct, i.e. it assumes that the CAP file has already been analysed by the off-card verifier provided by the Java Card platform.

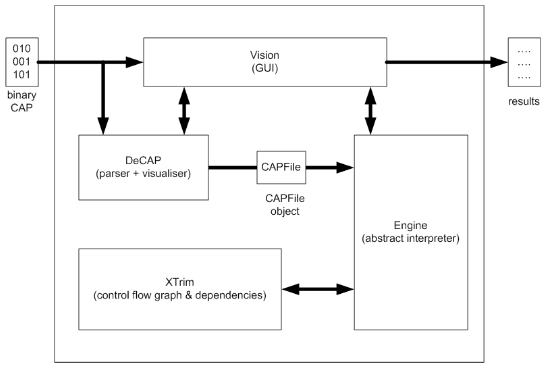


Fig. 8. Software modules of the tool.

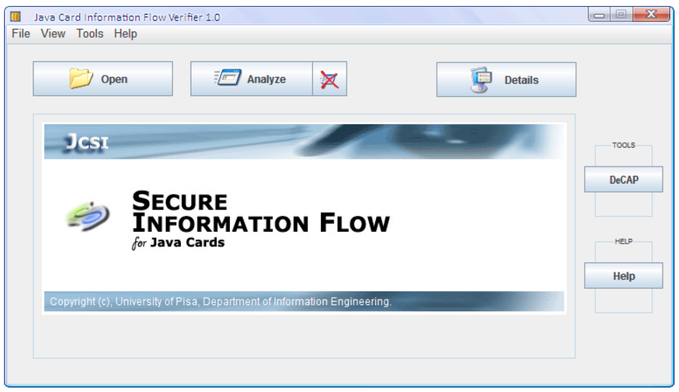
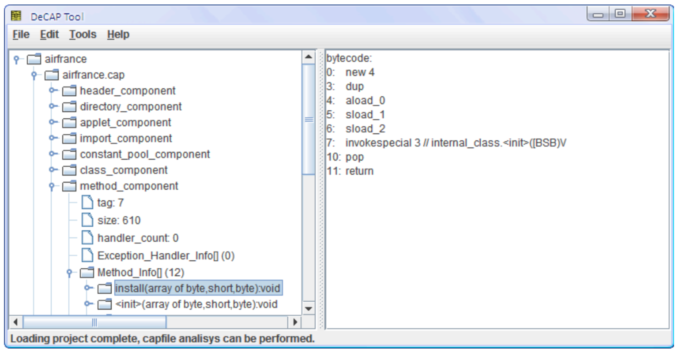

Java Card Information Flow Verifier 1.0
File View Tools Help
Open
Analyze
Details
JCSI
SECURE
INFORMATION FLOW
for Java Cards
Copyright (c), University of Pisa, Department of Information Engineering.
TOOLS
DeCAP
HELP
Help

Fig. 9. GUI of JCSI.

JCSI is composed of the following main software modules (see Figure 8):

Ængine: implements the abstract interpreter MSC of the bytecode according to the process in Figure 3.

XTrim: generates the control flow graph of the bytecode of each method, and takes into account exception handlers



```
airfrance
  airfrance.cap
    header_component
    directory_component
    applet_component
    import_component
    constant_pool_component
    class_component
    method_component
      tag: 7
      size: 610
      handler_count: 0
      Exception_Handler_Info[] (0)
      Method_Info[] (12)
        install(array of byte,short,byte):void
        <init>(array of byte,short,byte):void
```

```
bytecode:
0:  new 4
3:  dup
4:  aload_0
5:  sload_1
6:  sload_2
7:  invokespecial 3 // internal_class.<init>([BSB)V
10: pop
11: return
```

Loading project complete, capfile analisys can be performed.

Fig. 10. DeCap tool.

and subroutines. XTrim is also responsible of computing the scope of control instruction.

Vision: the GUI of JCSI (see Figure 9). Users can *i)* select the package to be analysed and the context of packages stored on-card, *ii)* start/stop the full analysis, *iii)* view the analysis results and a brief summary for each analysed method, *iv)* view the full log of the analysis.

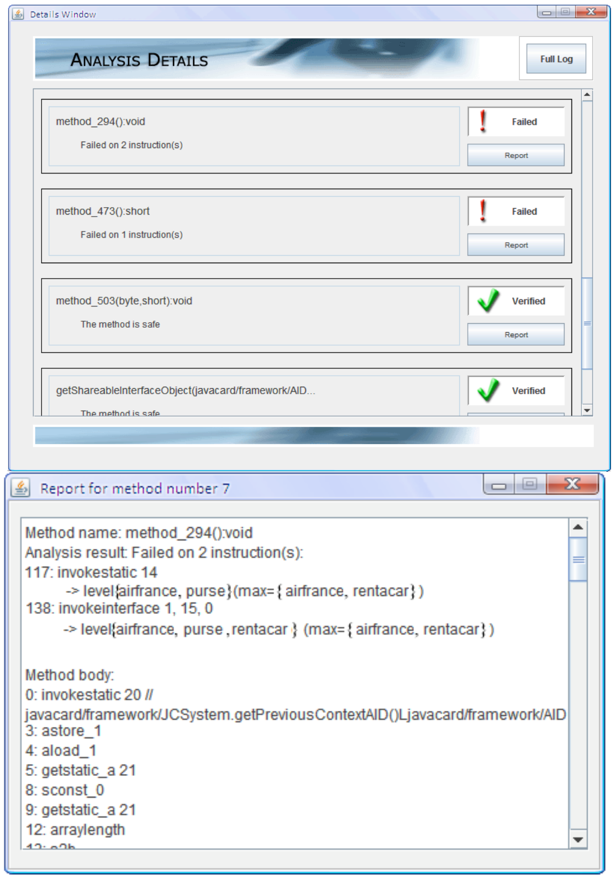
DeCAP: a CAP file disassembler and visualiser (see Figure 10). DeCAP is a tool that provides a set of APIs suitable to read binary CAP files used by Java Cards. This tool is invoked by ÆEngine in order to parse CAP files. Binary CAP files are represented through a Java class called *CapFile*. DeCAP also provides a GUI that allows users to explore the structure of CAP and export files, and to visualise their binary content in a more comprehensive mnemonic format.

JCSI performs the analysis of a CAP file in four main steps. First, unique security levels are automatically assigned to packages and the $Import$ and $Export$ of a package are automatically computed (optionally, they can also be specified by the user). Second, security levels are assigned to methods and class fields and array types in the `ambient` file. Third, the abstract interpretation of the bytecode is performed following the verification method described in Section III-B. Fourth, the tool reports the final result at the end of the analysis, which shows the secure interaction properties described in Section IV. If a package does not guarantee secure interaction, a detailed report on methods and instructions causing the violation is shown.

We checked the Electronic Purse with JCSI. In the following, we show an excerpt from the initial `ambient` file for the analysis of AirFrance.

```
------- Interface Methods (Exported)
interface purse/LoyaltyPurseInterface
 - method_294():void  //%--logFull
   @method_294():{airfrance,purse};{airfrance,purse}
interface purse/LoyaltyPurseInterface
 - method_607():void  //%--exchangeRate
   @method_607():{airfrance,purse};{airfrance,purse}
interface airfrance/LoyaltyAirFranceInterface
 - method_473():short  //%--getBalance
   @method_473():{airfrance,rentacar};{airfrance,rentacar}
interface airfrance/LoyaltyAirFranceInterface
 - method_503(byte,short):void  //%--updatePoints
   @method_503({airfrance,rentacar},{airfrance,rentacar})
    :{airfrance,rentacar};{airfrance,rentacar}

---- Internal Methods
 - install(array of byte,short,byte):void
   @install(airfrance,airfrance,airfrance)
    :airfrance;airfrance
 - <init>(array of byte,short,byte):void
   @init(airfrance,airfrance,airfrance)
    :airfrance;airfrance
 //%-- more initialisations omitted
```



method_294():void
Failed on 2 instruction(s)

method_473():short
Failed on 1 instruction(s)

method_503(byte,short):void
The method is safe

getShareableInterfaceObject(javacard/framework/AID...
The method is safe

```
Method name: method_294():void
Analysis result: Failed on 2 instruction(s):
117: invokestatic 14
      -> level{airfrance, purse}(max={ airfrance, rentacar} )
138: invokeinterface 1, 15, 0
      -> level{airfrance, purse ,rentacar } (max={ airfrance, rentacar} )

Method body:
0: invokestatic 20 //
javacard/framework/JCSystem.getPreviousContextAID()Ljavacard/framework/AID
3: astore_1
4: aload_1
5: getstatic_a 21
8: sconst_0
9: getstatic_a 21
12: arraylength
```

Fig. 11. Example of analysis results.

At the end of the analysis of the Electronic Purse, we obtain that Purse satisfies secure interaction, while AirFrance and RentACar violate secure interaction. Indeed, we recall that, when AirFrance invokes `getBalance()` in the code of `logFull()`, there is a secure interaction violation because there is an implicit flow from `logFull()` to `getBalance()`. The level of the calling environment is {`purse, airfrance, rentacar`}. Similarly, the level of the calling environment is {`purse, airfrance, rentacar`} when RentACar invokes `getTransaction()` in the code of `getBalance()` JCSI performs the analysis of AirFrance in a few minutes on a desktop computer, and produces the report shown in Figure 11 at the end of the analysis. The JCSI tool and the case study are available at http://www.ing.unipi.it/∼o1833499/JCSI/.

VI. Related work and conclusions

Several works deal with the formalisation of the Java Card firewall in order to find out its limits in protecting sensitive data of packages. In [8], [7] 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 [2] an analysis is proposed for detecting whether an access to shared objects violates the rules of the firewall.

Other works statically check secure interaction by using the theory of secure information flow. The reader can refer to [11] for a survey of the techniques applied for enforcing information flow security policies in programs. In [3], a tool based on a model checking technique has been developed to check information flow in Java Cards. The tool is based on a security policy that defines the allowed flows of information between applets and the verification is done with the SMV model checker. The tool computes all call graphs of the application and generates an SMV model per graph. In [9] a case study is presented, where a combination of static program analysers are employed to check the source code of a smart card applet. More recent works propose tools that apply analysis techniques that do not guarantee sound and complete analyses [13], [1].

The main contribution of this work is the provision of an approach to statically check secure interaction in Java Cards based on abstract interpretation of the operational semantics. The proposed approach is conservative, which means that all codes that are not secure are guaranteed to be rejected. To verify a package, only the import component of the other packages is required. We reduce the complexity of the analysis by using a data-flow analysis. Moreover, the analysis scales up, since packages are analysed separately by a method per method verification.

We provide a tool that implements the approach, and is suitable to analyse the binary code of Java Card applets, which are released to users when installing new services on-card.

A limit of our approach is that the analysis is monomorphic, in the sense that it is impossible to distinguish between object instances. This is due to the design choice of modelling corresponding fields of different objects of the same class by a unique (the highest) security level. However, this is not a real limitation, since applets rarely create more than few objects. Similarly, we model array and method invocation without taking into account array instances and program points of method invocation. All the above limitations can be overridden by collecting in the `ambient` file, for each object instance and array instance, the program point of its creation, and, for each method, the program point of method invocation.

Acknowledgement

This work was supported by Fondazione Cassa di Risparmio di Pisa, Italy (N. PR02-182) and by the European Commission through the Network of Excellence ReSIST (IST-026764).

References

[1] V. Almaliotis, A. Loizidis, P. Katsaros, P. Louridas, and D. Spinellis. 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*, pages 17–31, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] D. Caromel, L. Henrio, and B. Serpette. Context inference for static analysis of Java Card object sharing. In *In I.Attali and T.Jensen (Eds): Conference on Research in Smart Cards, E-smart 2001*, pages 43–57. LNCS 2140, Springer-Verlag 2001.

[3] J. Cazin, A. El-Marouani, P. Girard, J.L. Lanet, V. Wiels, and G. Zanon. The PACAP prototype: a tool for detecting illegal flows. In *Java Card Workshop Proceedings*. Cannes, September 2000.

[4] Z. Chen. Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison Wesley, 2000.

[5] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[6] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Comp.*, 2:511–547, 1992.

[7] M. Eluard and T. Jensen. Secure Object Flow Analysis for Java Card. In *Fifth Smart Card Research and Advanced Application (CARDIS'2002) Conference Proceedings*, pages 97–110. USENIX, 2002.

[8] M. Eluard, T. Jensen, and E. Denne. An Operational Semantics of the Java Card Firewall. In *In I.Attali and T.Jensen (Eds): Conference on Research in Smart Cards, E-smart 2001*, pages 95–110. LNCS 2140, Springer-Verlag 2001.

[9] B. Jacobs, C. Marché, and N. Rauch. Formal Verification of a Commercial Smart Card Applet with Multiple Tools . In C. Shankland C. Rattray, S. Maharaj, editor, *Algebraic Methodology and Software Technology (AMAST 04)*, volume 3116 of *LNCS*, pages 241–257. Springer, July 2004.

[10] X. Leroy. Java bytecode verification: an overview. In *13th International Conference on Computer Aided Verification, LNCS 2102, Proceedings*, pages 265–285, July 2001.

[11] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE journal on selected areas in communications*, 21(1), 2003.

[12] D.A. Schmidt. Abstract interpretation of small-step semantics. In *5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages Proceedings*, 1996.

[13] D. Spinellis and P. Louridas. A framework for the static verification of API calls. *J. Syst. Softw.*, 80(7):1156–1168, 2007.