Java Card Secure Information flow tool
JCSI Version 1.2, April 2012
User Manual

Marco Avvenuti
Cinzia Bernardeschi
Nicoletta De Francesco
Paolo Masci

Department of Information Engineering
University of Pisa, Italy

Contents

1 Introduction

A Java Card applet is a smart card application written in the Java programming language, according to a set of conventions (ISO 7816) that regulate programs running in small footprint devices. Java Card applets are compiled by the standard Java compiler [1]. An off-card component, the Converter, takes the compiler's output and performs checking, optimization, name resolution and linking. A compact representation of the applet code is saved in the CAP (Converted APplet) file.

CAP files are the loading units in Java Card. They include the necessary information to install and execute applets in the Java Card Runtime Environment (JCRE) [2]. CAP files are very similar to class files. Nevertheless, while a large number of tools to analyze class files is available, the software community still lacks of similar tools for CAP files.

The JCSI tool performs a static analysis of CAP files in order to study the information flow generated by the code.

Included in JCSI, the DeCAP tool enables the analysis by converting a CAP file into a structure conforming to the Java Card Virtual Machine Language (JCVML) specification.

2 JCSI

2.1 GUI description

The JCSI's graphical user interface (Figure 1) includes a status window (1), three function buttons (2, 3, 4), one tool button (5) and one help button (6). The analysis of a CAP file is carried out through the following steps:

1. Load the CAP file and the necessary Export files (Subsection 2.2)
2. Set the parameters for the analysis (Subsection 2.3)
3. Perform the analysis (Subsection 2.4)
4. View the log file (Subsection 2.5)

2.2 Loading the CAP file and the Export files

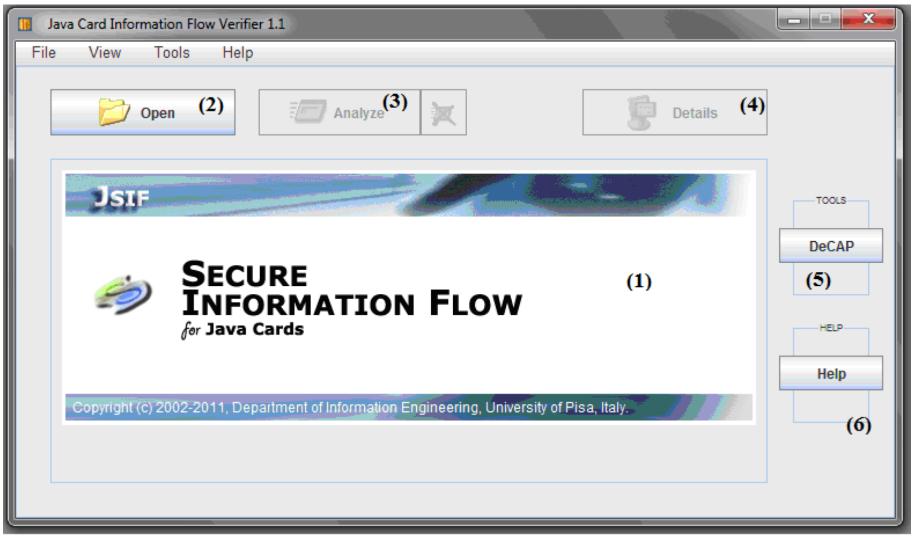
To load a CAP file and the needed Export files, click on the `Open` button. A new window `Open Files` will pop-up, showing the following buttons:

Open: a file browser window is opened to choose the desired CAP (Figure 2).

Add/Remove: a file browser window is opened to add/remove the Export files of the package (Figure 3). To automatically load JCRE API Export Files, select `Automatically Load JCRE API Export Files`.

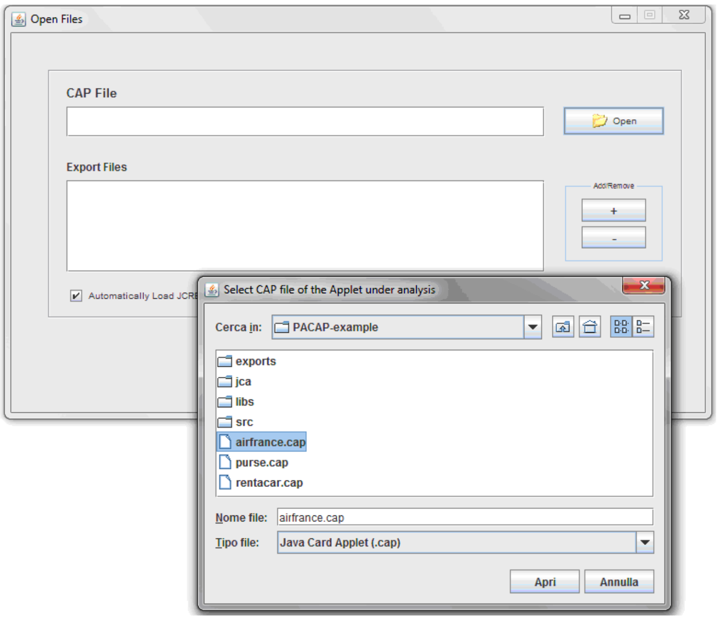
Ok: to confirm the chosen files. Missing Export files, if any, are listed as in Figure 4.

When the CAP file and all the necessary Export files have been successfully loaded, the main GUI is shown telling the user that the analysis of the information flow can start (Figure 5).



Copyright (c) 2002-2011, Department of Information Engineering, University of Pisa, Italy.

Figure 1: The JCSI tool main window.

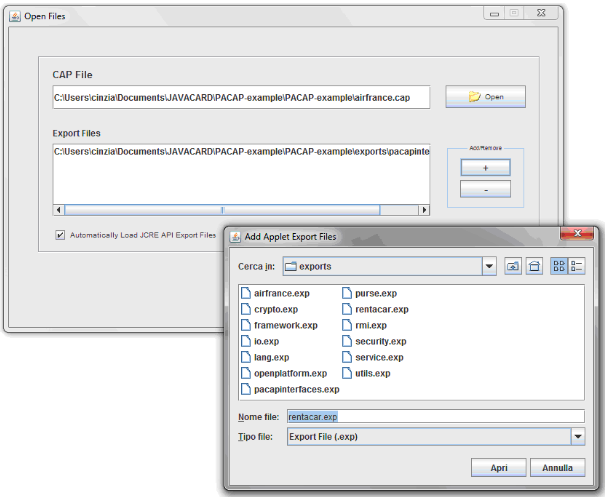


Cerca in: PACAP-example

Nome file: airfrance.cap

Tipo file: Java Card Applet (.cap)

Figure 2: Loading a CAP file.



CAP File

C:\Users\cinzia\Documents\JAVACARD\PACAP-example\PACAP-example\airfrance.cap

Export Files

C:\Users\cinzia\Documents\JAVACARD\PACAP-example\PACAP-example\exports\pacapinte

☑ Automatically Load JCRE API Export Files

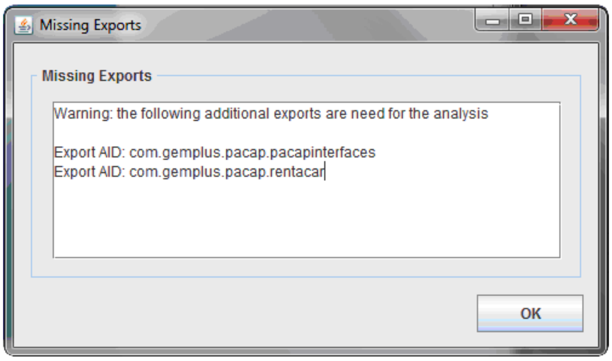
Cerca in: exports

airfrance.exp
crypto.exp
framework.exp
io.exp
lang.exp
openplatform.exp
pacapinterfaces.exp
purse.exp
rentacar.exp
rmi.exp
security.exp
service.exp
utils.exp

Nome file: rentacar.exp

Tipo file: Export File (.exp)

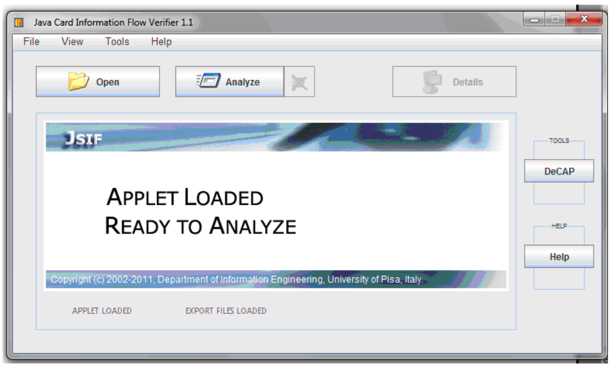
Figure 3: Loading Export files.



Warning: the following additional exports are need for the analysis

Export AID: com.gemplus.pacap.pacapinterfaces
Export AID: com.gemplus.pacap.rentacar

Figure 4: Missing Export files window.



APPLET LOADED
READY TO ANALYZE

Copyright (c) 2002-2011, Department of Information Engineering, University of Pisa, Italy.

APPLET LOADED

EXPORT FILES LOADED

Figure 5: Ready to analyze a CAP file.

2.3 Setting the parameters for the analysis

The verification process starts by clicking on the `Analyze` button. The setting of parameters for the analysis follows three main steps:

Step 1. Configuring packages' names

1. The `JCSI Analysis Wizard` window is prompted (Figure 6) to allow the user to set the number of packages that are already installed on card. The CAP files for these packages are not required.
2. Through the `Package Names Editor` window (Figure 7) the user can choose the names for the packages. These names are used as security levels ($p_0, \cdots, p_n$, by default).

At the end of this step the user clicks on the `Next` button.

Step 2. Configuring the Ambient file

1. The `Analysis Setup Loader` window is prompted (Figure 8) to allow the user to choose whether to use the default Ambient file (`Use Default Ambient File` option), or to load the Ambient file from a previously saved configuration (`Load Custom Ambient File` option). In the latter case, a browser window is opened to select the file.
2. The `Ambient File Editor` window is prompted (Figure 9) to allow the user to view and customize the selected Ambient file. The Ambient file is organized in four sections: *Imported Methods*, *Internal Methods*, *Exported Methods* and *Heap Section*.

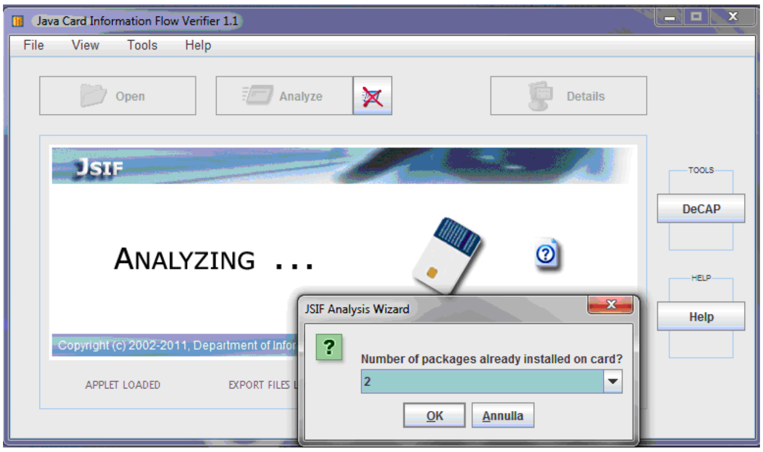
   For methods, the window shows the method's name together with information on the package that implements the method (`_name`) and the package that imports the method ($\wedge$`name`).
3. The user can either select a method from a Method section, or an object from the Heap Section:
   - The `View Selected` button shows the signature of the selected method, or the security level of the selected object. For methods, the security levels of arguments, return value, calling environment are shown. The security policy is also shown within angled brackets (Figure 10, e.g. ⟨ `airfrance` ⟩).
   - The `Edit Selected` button allows the user to modify the security levels and the security policy of methods (Figure 11), or the security levels of objects (Figure 12).

At the end of this step the user clicks on the `Next` button.

Step 3. Confirming Analysis Setup

The `Analysis Setup` window is prompted (Figure 13) showing the Ambient file. The user can set the following:

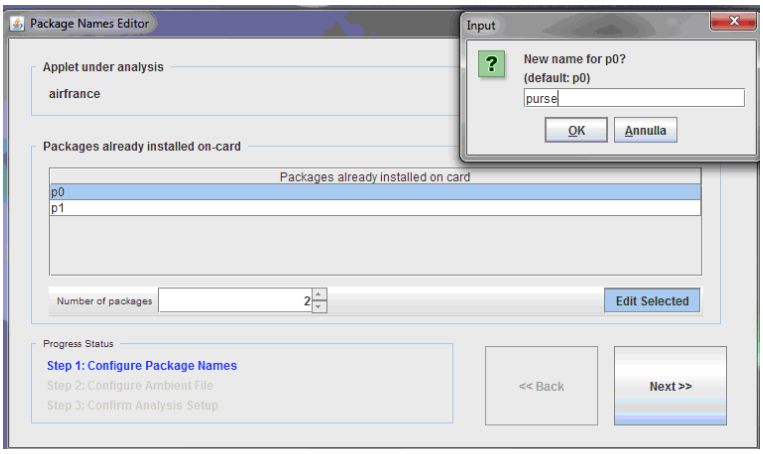
Save Ambient file: To save the Ambient file for future utilization (Figure 14)



Number of packages already installed on card?

2

Figure 6: Setting the number of packages already installed on card.



Applet under analysis

airfrance

Packages already installed on-card

Packages already installed on card
p0
p1

Number of packages 2

Progress Status

Step 1: Configure Package Names

Step 2: Configure Ambient File

Step 3: Confirm Analysis Setup

New name for p0?
(default: p0)

purse

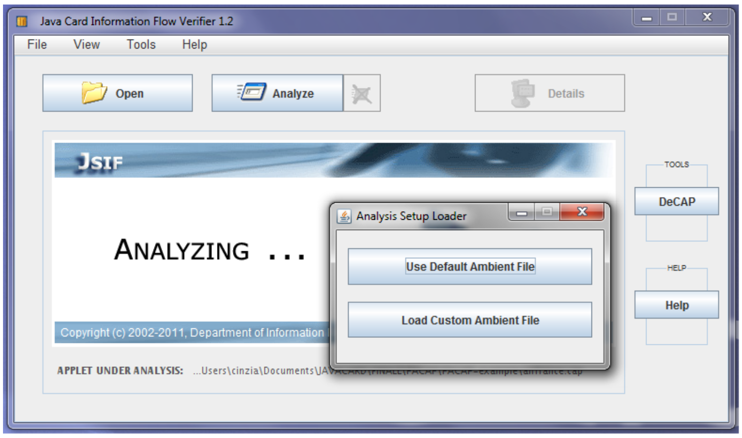
Figure 7: Choosing the names for packages already installed on card.

Log Ambient File Changes: If this option is selected, a text file is created that logs all changes occurring to the ambient file during the analysis. By default, the text file is stored in the same folder of the CAP file, and its name is given by the name of the CAP file under analysis followed by a timestamp.

Log Abstract Interpreter States: If this option is selected, a text file is created that logs a complete trace of the analysis performed by the tool. As for option "Log Ambient File Changes", by default, such text file is stored in the same folder of the CAP file, and its name is given by the name of the CAP file under analysis followed by a timestamp.

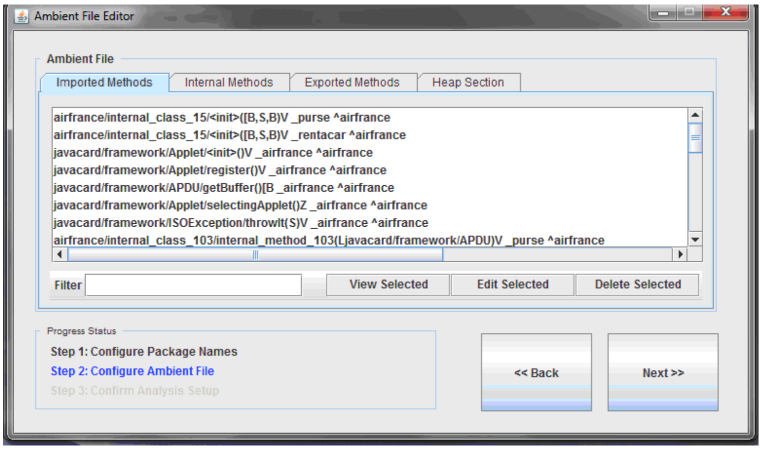
At the end of this step the user clicks on the `Next` button. The analysis starts as described in the next section.

In the tool, the following conventions are used:



APPLET UNDER ANALYSIS: ...Users\cinzia\Documents\JA

Figure 8: Loading the Ambient file.



```
airfrance/internal_class_15/<init>([B,S,B)V _purse ^airfrance
airfrance/internal_class_15/<init>([B,S,B)V _rentacar ^airfrance
javacard/framework/Applet/<init>()V _airfrance ^airfrance
javacard/framework/Applet/register()V _airfrance ^airfrance
javacard/framework/APDU/getBuffer()[B _airfrance ^airfrance
javacard/framework/Applet/selectingApplet()Z _airfrance ^airfrance
javacard/framework/ISOException/throwIt(S)V _airfrance ^airfrance
airfrance/internal_class_103/internal_method_103(Ljavacard/framework/APDU)V _purse ^airfrance
```

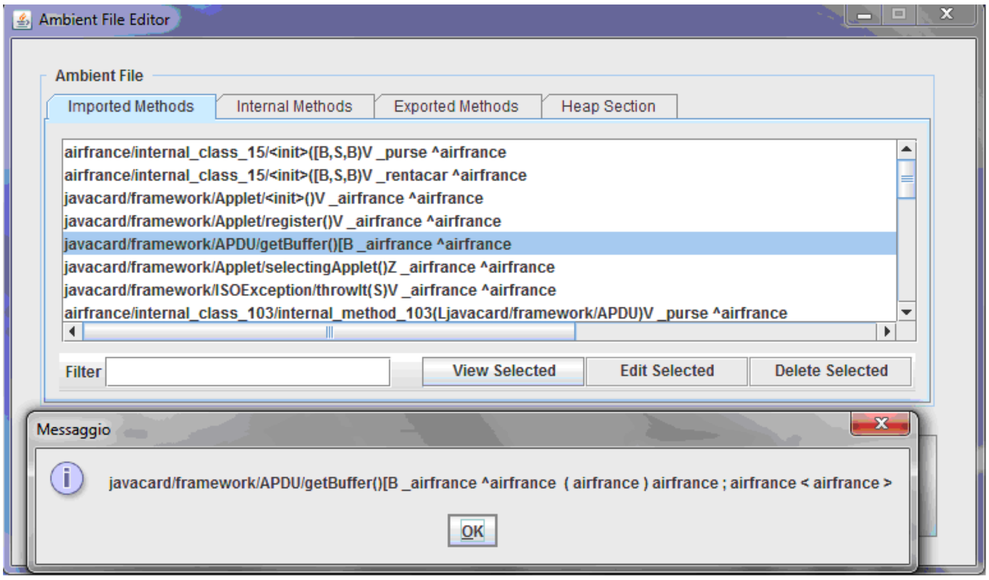
Step 1: Configure Package Names
Step 2: Configure Ambient File
Step 3: Confirm Analysis Setup

Figure 9: The Ambient file editor.

- Given two security levels `a` and `b`, notation `a + b` is used for { `a`, `b` }.
- The security policy ”all levels are allowed” can be expressed either by including the whole set of levels within the angled brackets or by omitting the policy from the Ambient file.
- The security level **.public.** is used to denote public information.
- The name of the method is shown as the method's number, as it appears in the CAP file. The symbolic name of the method can be retrieved by using the DeCAP tool.

2.4 Performing the analysis

At the beginning of the analysis, the `JCSI Analysis Wizard` window allows the user to choose the stop condition (Figure 15):



```
airfrance/internal_class_15/<init>([B,S,B)V _purse ^airfrance
airfrance/internal_class_15/<init>([B,S,B)V _rentacar ^airfrance
javacard/framework/Applet/<init>()V _airfrance ^airfrance
javacard/framework/Applet/register()V _airfrance ^airfrance
javacard/framework/APDU/getBuffer()[B _airfrance ^airfrance
javacard/framework/Applet/selectingApplet()Z _airfrance ^airfrance
javacard/framework/ISOException/throwIt(S)V _airfrance ^airfrance
airfrance/internal_class_103/internal_method_103(Ljavacard/framework/APDU)V _purse ^airfrance
```

```
javacard/framework/APDU/getBuffer()[B _airfrance ^airfrance ( airfrance ) airfrance ; airfrance < airfrance >
```

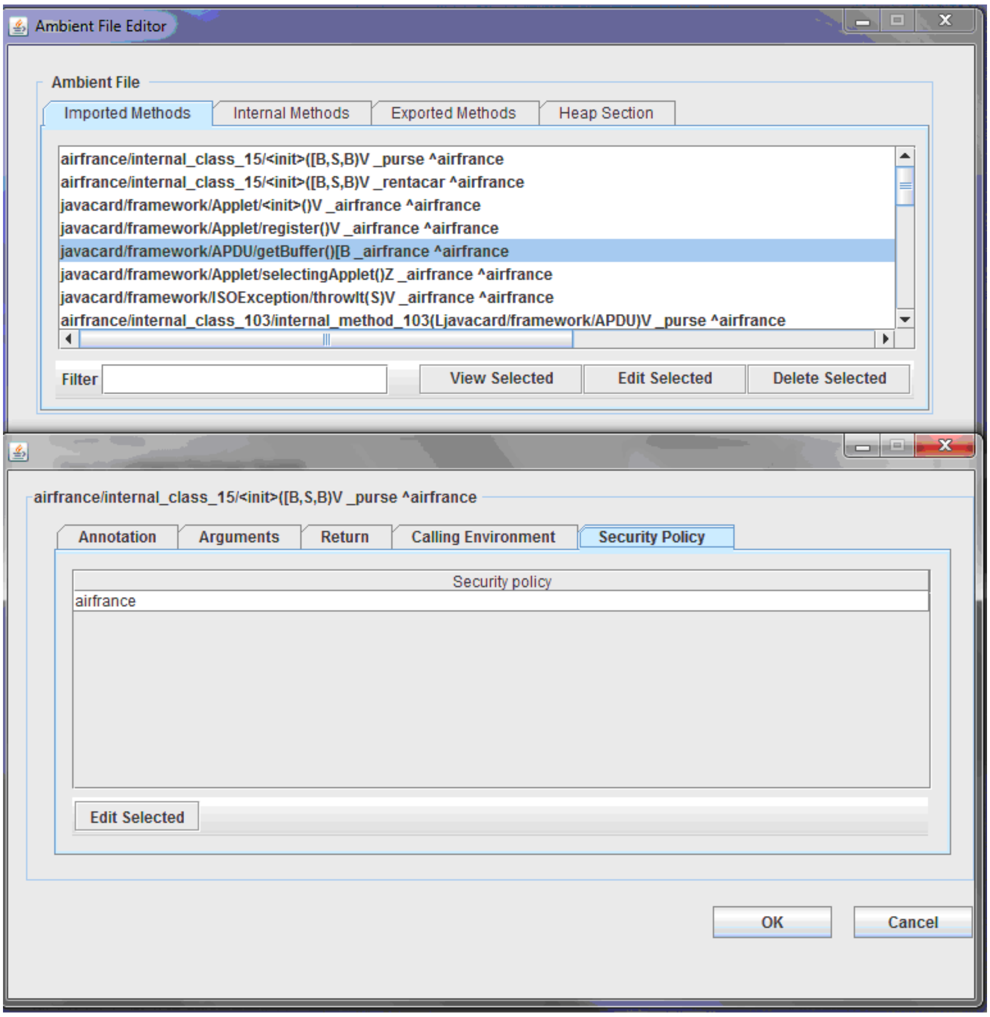
Figure 10: Selecting and viewing a method.

Stop at first violation: With this option set, the data-flow analysis stops as soon as the security levels calculated by the abstract interpreter violate the security policy specified in the Ambient file.

Stop at fixpoint: With this option set, the data-flow analysis stops at the fixpoint of the analysis (the security levels of the Ambient file are left unchanged by a run of the abstract interpreter).

After having selected the stop condition, click on the `OK` button to start the analysis. When the stop condition occurs, the result of the analysis are reported in the `Java Card Information Flow Verifier 1.2` main window (Figure 16).

The `Analysis Log` window reports the Ambient file updated according to the abstract interpreter rules. At the end of this phase, the Ambient file reports the maximum security level for methods and heap section.

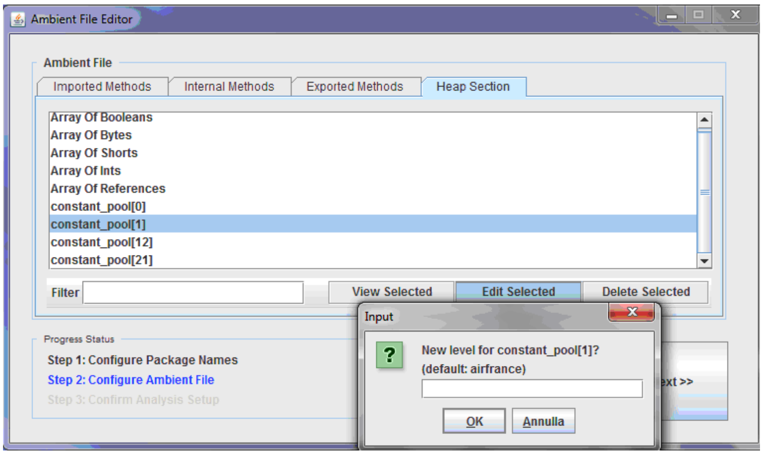


airfrance/internal_class_15/<init>([B,S,B)V _purse ^airfrance
airfrance/internal_class_15/<init>([B,S,B)V _rentacar ^airfrance
javacard/framework/Applet/<init>()V _airfrance ^airfrance
javacard/framework/Applet/register()V _airfrance ^airfrance
javacard/framework/APDU/getBuffer()[B _airfrance ^airfrance
javacard/framework/Applet/selectingApplet()Z _airfrance ^airfrance
javacard/framework/ISOException/throwIt(S)V _airfrance ^airfrance
airfrance/internal_class_103/internal_method_103(Ljavacard/framework/APDU)V _purse ^airfrance

airfrance/internal_class_15/<init>([B,S,B)V _purse ^airfrance

Security policy: airfrance

Figure 11: Changing the security policy of a method.

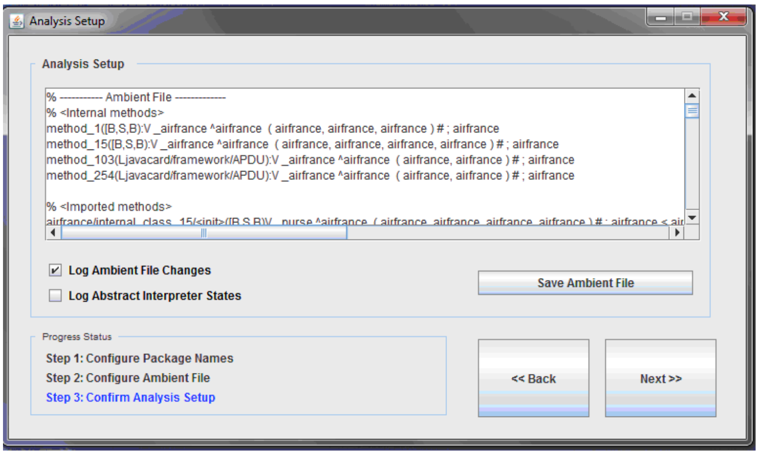


Array Of Booleans
Array Of Bytes
Array Of Shorts
Array Of Ints
Array Of References
constant_pool[0]
constant_pool[1]
constant_pool[12]
constant_pool[21]

New level for constant_pool[1]?
(default: airfrance)

Step 1: Configure Package Names
Step 2: Configure Ambient File
Step 3: Confirm Analysis Setup

Figure 12: Changing the security level of an object.



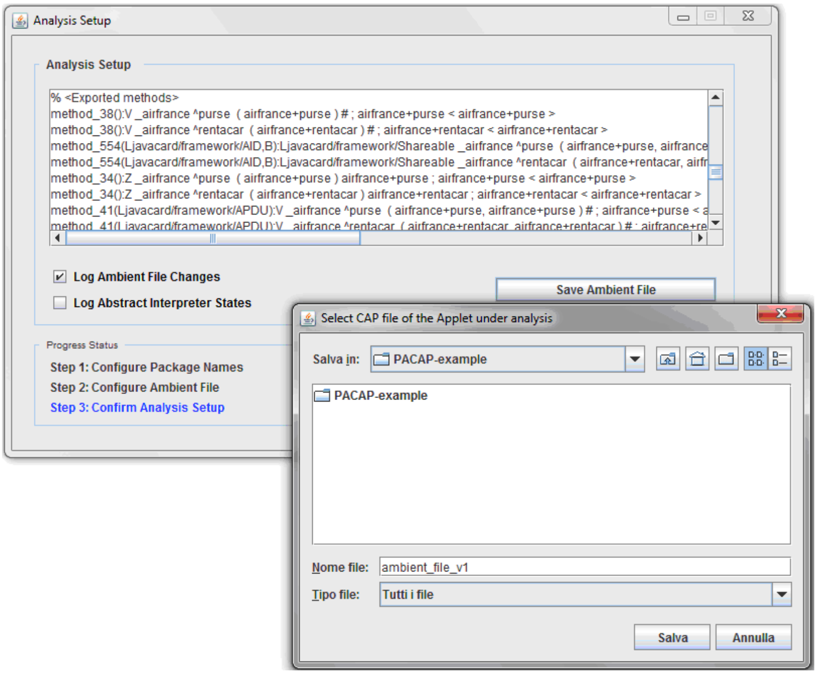
```
% ---------- Ambient File ------------
% <Internal methods>
method_1([B,S,B):V _airfrance ^airfrance  ( airfrance, airfrance, airfrance ) # ; airfrance
method_15([B,S,B):V _airfrance ^airfrance  ( airfrance, airfrance, airfrance, airfrance ) # ; airfrance
method_103(Ljavacard/framework/APDU):V _airfrance ^airfrance  ( airfrance, airfrance ) # ; airfrance
method_254(Ljavacard/framework/APDU):V _airfrance ^airfrance  ( airfrance, airfrance ) # ; airfrance

% <Imported methods>
airfrance/internal_class_15/<init>([B,S,B)V _purse ^airfrance  ( airfrance, airfrance, airfrance, airfrance ) # ; airfrance < air
```

☑ Log Ambient File Changes

☐ Log Abstract Interpreter States

Step 1: Configure Package Names
Step 2: Configure Ambient File
Step 3: Confirm Analysis Setup

Figure 13: Confirming the Analysis Setup.



```
% <Exported methods>
method_38():V _airfrance ^purse  ( airfrance+purse ) # ; airfrance+purse < airfrance+purse >
method_38():V _airfrance ^rentacar  ( airfrance+rentacar ) # ; airfrance+rentacar < airfrance+rentacar >
method_554(Ljavacard/framework/AID,B):Ljavacard/framework/Shareable _airfrance ^purse  ( airfrance+purse, airfrance
method_554(Ljavacard/framework/AID,B):Ljavacard/framework/Shareable _airfrance ^rentacar  ( airfrance+rentacar, airfr
method_34():Z _airfrance ^purse  ( airfrance+purse ) airfrance+purse ; airfrance+purse < airfrance+purse >
method_34():Z _airfrance ^rentacar  ( airfrance+rentacar ) airfrance+rentacar ; airfrance+rentacar < airfrance+rentacar >
method_41(Ljavacard/framework/APDU):V _airfrance ^purse  ( airfrance+purse, airfrance+purse ) # ; airfrance+purse < a
method_41(Ljavacard/framework/APDU):V _airfrance ^rentacar  ( airfrance+rentacar, airfrance+rentacar ) # ; airfrance+re
```

☑ Log Ambient File Changes

□ Log Abstract Interpreter States

Step 1: Configure Package Names

Step 2: Configure Ambient File

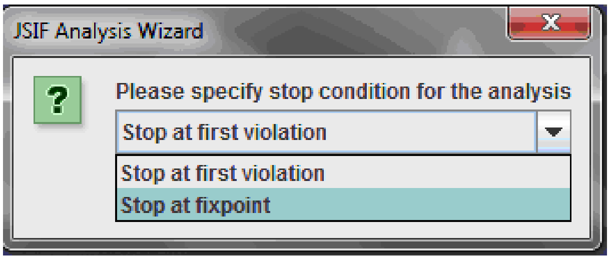
Step 3: Confirm Analysis Setup

Salva in: PACAP-example

Nome file: ambient_file_v1

Tipo file: Tutti i file

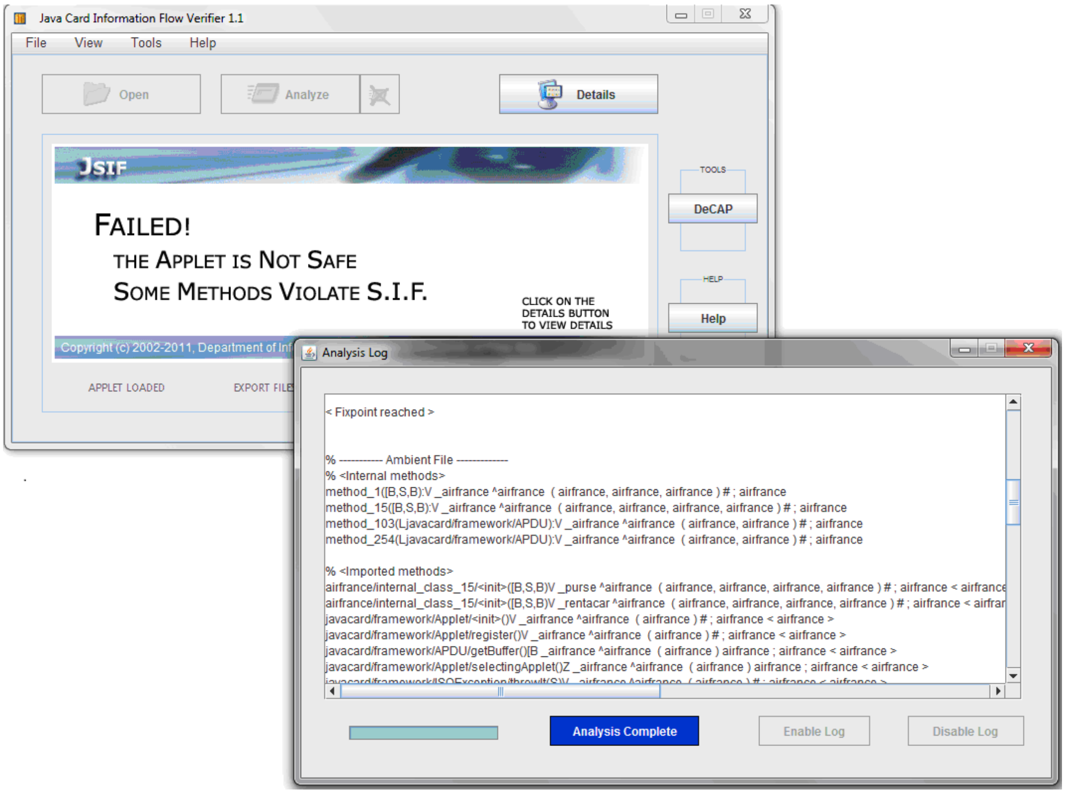
Figure 14: Saving the Ambient file.



Please specify stop condition for the analysis

Stop at first violation

Figure 15: Choosing the stop condition for the analysis.



FAILED!
THE APPLET IS NOT SAFE
SOME METHODS VIOLATE S.I.F.

CLICK ON THE
DETAILS BUTTON
TO VIEW DETAILS

```
< Fixpoint reached >


% ----------- Ambient File -------------
% <Internal methods>
method_1([B,S,B):V _airfrance ^airfrance  ( airfrance, airfrance, airfrance ) # ; airfrance
method_15([B,S,B):V _airfrance ^airfrance  ( airfrance, airfrance, airfrance, airfrance ) # ; airfrance
method_103(Ljavacard/framework/APDU):V _airfrance ^airfrance  ( airfrance, airfrance ) # ; airfrance
method_254(Ljavacard/framework/APDU):V _airfrance ^airfrance  ( airfrance, airfrance ) # ; airfrance

% <Imported methods>
airfrance/internal_class_15/<init>([B,S,B)V _purse ^airfrance  ( airfrance, airfrance, airfrance, airfrance ) # ; airfrance < airfrance
airfrance/internal_class_15/<init>([B,S,B)V _rentacar ^airfrance  ( airfrance, airfrance, airfrance, airfrance ) # ; airfrance < airfrar
javacard/framework/Applet/<init>()V _airfrance ^airfrance  ( airfrance ) # ; airfrance < airfrance >
javacard/framework/Applet/register()V _airfrance ^airfrance  ( airfrance ) # ; airfrance < airfrance >
javacard/framework/APDU/getBuffer()[B _airfrance ^airfrance  ( airfrance ) airfrance ; airfrance < airfrance >
javacard/framework/Applet/selectingApplet()Z _airfrance ^airfrance  ( airfrance ) airfrance ; airfrance < airfrance >
javacard/framework/ISOException/throwIt(S)V _airfrance ^airfrance  ( airfrance ) # ; airfrance < airfrance >
```

Figure 16: Analysis completion.

2.5 Viewing the details of the analysis

Click on the `Details` button of the `Java Card Information Flow Verifier 1.2` main window. The `Analysis Details` window will pop up, showing the results of the analysis (Figure 17). In this window:

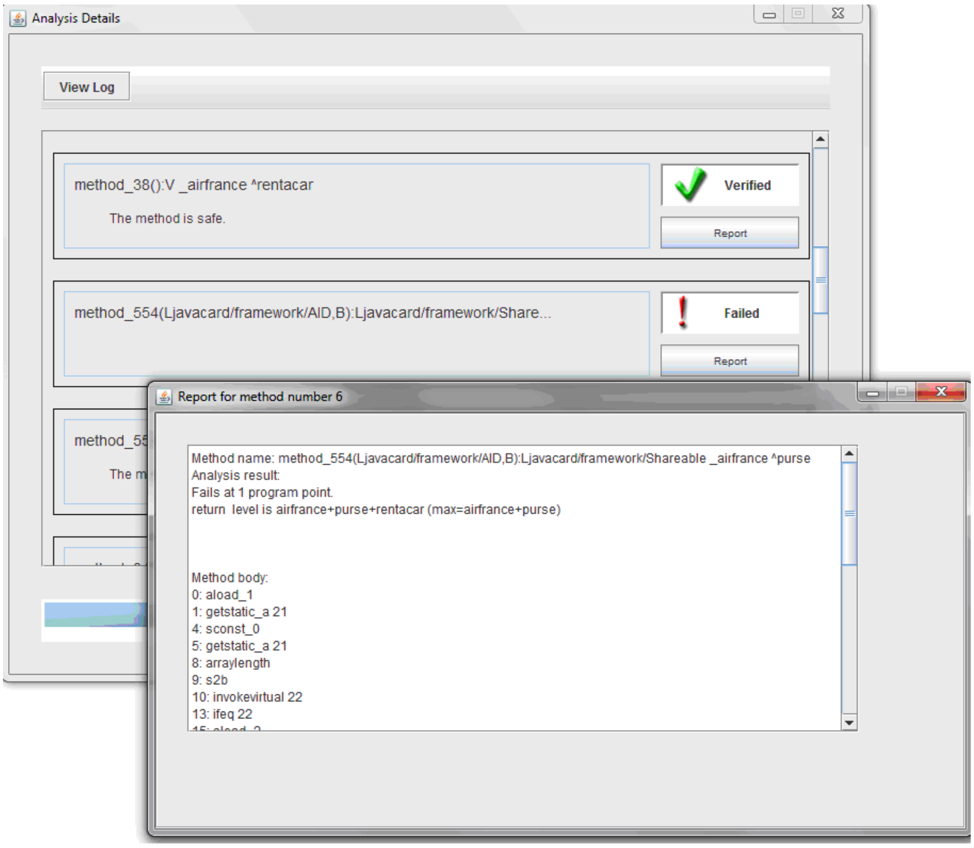
View Log: Click this button to view the Log file.

For each method, the window shows:

Verified, if the method satisfies the security policy.

Failed, if the method violates the security policy.

Report: Click this button to view detailed information about the method (signature, reason of the failure, bytecode).



method_38():V _airfrance ^rentacar

The method is safe.

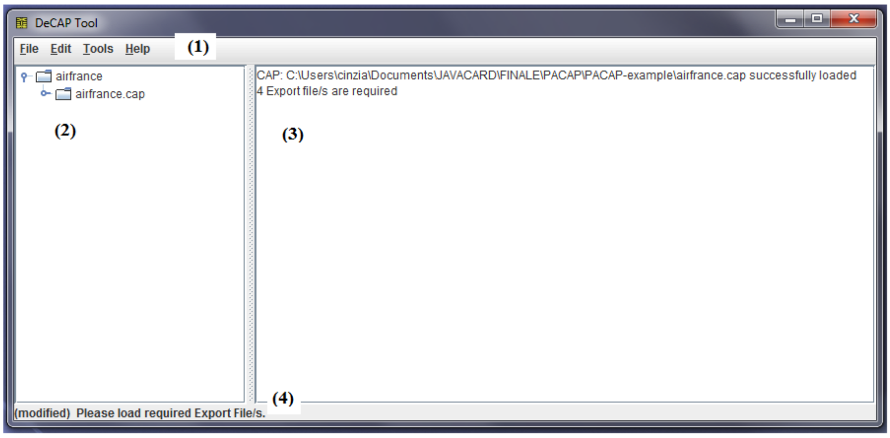
method_554(Ljavacard/framework/AID,B):Ljavacard/framework/Share...

```
Method name: method_554(Ljavacard/framework/AID,B):Ljavacard/framework/Shareable _airfrance ^purse
Analysis result:
Fails at 1 program point.
return  level is airfrance+purse+rentacar (max=airfrance+purse)



Method body:
0: aload_1
1: getstatic_a 21
4: sconst_0
5: getstatic_a 21
8: arraylength
9: s2b
10: invokevirtual 22
13: ifeq 22
```

Figure 17: Details of the analysis.

3 The DeCAP tool

DeCAP is a CAP file disassembler and visualizer tool. The tool simplifies bytecode inspection and analysis by resolving information coded in tokens and showing it in a mnemonic format. The graphical user interface of the tool reports all the information stored in binary CAP and export files.



```
CAP: C:\Users\cinzia\Documents\JAVACARD\FINALE\PACAP\PACAP-example\airfrance.cap successfully loaded
4 Export file/s are required
```

```
(modified)  Please load required Export File/s.
```

Figure 18: The DeCAP tool window.

The DeCAP window (Figure 18) is made of a menu (1), a CAP component browser (2), a description window (3) and a status bar (4). The CAP component browser is an expandable tree which simplifies the view of the CAP file components. The description window shows a text description of the highlighted component. The status bar guides the user during the CAP file inspection.

3.1 Menu

Functions are grouped into four categories:

File :

Load CAP File: To load a CAP file and begin a new working session. In the description window a message signals the correct loading of the CAP file and requests the user to load the Export files. The root of the CAP component browser tree has the same name as the loaded CAP file.

Add Export File: To load an Export file. Note that the Export files associated to the chosen CAP file are loaded automatically if they are stored in the same directory and if the JC_HOME setting (see Edit – Preferences) refers to a correctly installed version of Java Card SDK.

Exit: To exit from DeCAP.

Edit :

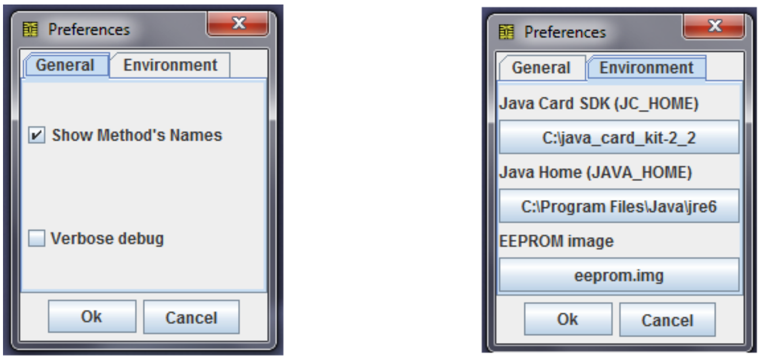
Preferences: To set program options. In the **General** tab (Figure 19 (1)) the `Show Method's Name` checkbox enables the name lookup, within the Export files of methods invoked, when displaying bytecode informations. The `Verbose debug` option enables debug messages relative to this tool development. The **Environment** tab (Figure 19 (2)) lets the user choose a Java Card SDK, a Java Runtime Environment and the filename of the `cref` EEPROM image file.

Tools :

System log: To show the application log.

Cap file analysis: A set of functions to gather information on CAP file structure.

Help: Program version and license information, and this help document.



☑ Show Method's Names

□ Verbose debug

Java Card SDK (JC_HOME)

C:\java_card_kit-2_2

Java Home (JAVA_HOME)

C:\Program Files\Java\jre6

EEPROM image

eeprom.img

(1) **(2)**

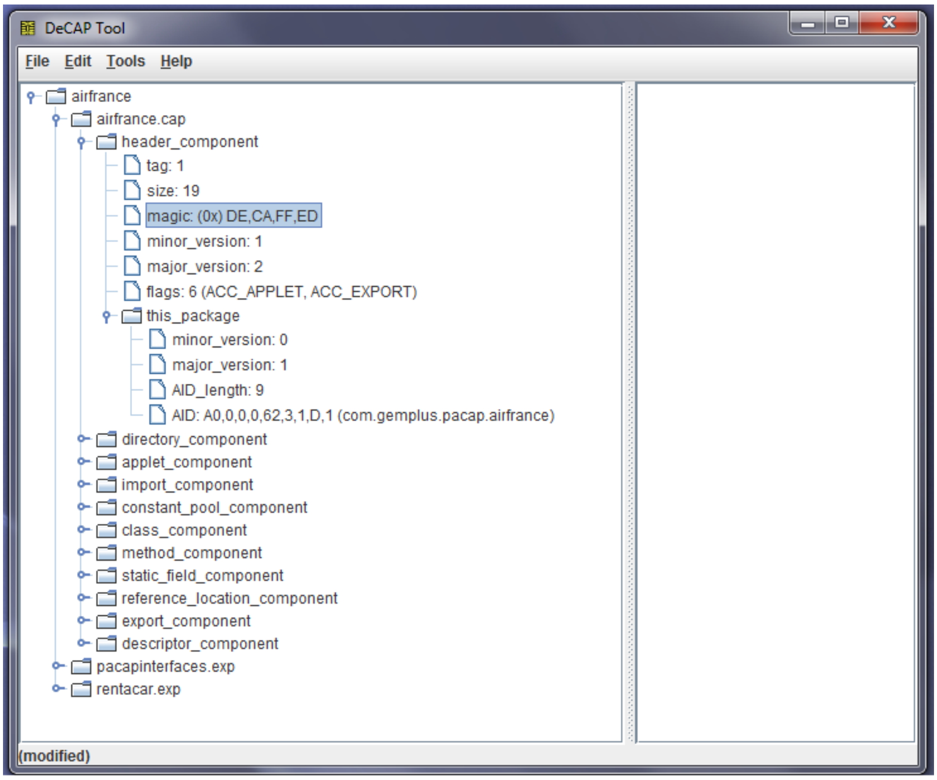
Figure 19: Preference dialogs: General tab (1), Environment tab (2).

3.2 The component browser

The CAP file and the Export files are represented in an expandable tree structure that allows the user to select nodes and view leaf values. After a CAP file has been loaded, the root of the tree automatically takes the name of the loaded CAP file.

An example is shown in Figure 20. Component `airfrance.cap` is expanded into the following subcomponents: `header_component`, `directory_component`, `applet_component`, `import_component`, `constant_pool_component`, `class_component`, `method_component`, `static_field_component`, `reference_location_component`, `export_component` and `descriptor_component`.

The `header_component` is split into various subcomponents. Among them, the subcomponent `this_package` contains the AID of the Cap file.



```
airfrance
  airfrance.cap
    header_component
      tag: 1
      size: 19
      magic: (0x) DE,CA,FF,ED
      minor_version: 1
      major_version: 2
      flags: 6 (ACC_APPLET, ACC_EXPORT)
      this_package
        minor_version: 0
        major_version: 1
        AID_length: 9
        AID: A0,0,0,0,62,3,1,D,1 (com.gemplus.pacap.airfrance)
    directory_component
    applet_component
    import_component
    constant_pool_component
    class_component
    method_component
    static_field_component
    reference_location_component
    export_component
    descriptor_component
  pacapinterfaces.exp
  rentacar.exp
```

Figure 20: The component browser.

3.3 The description window

The `description window` displays general, multi-line information on the selected CAP file. The following procedure displays the bytecode representation of a method.

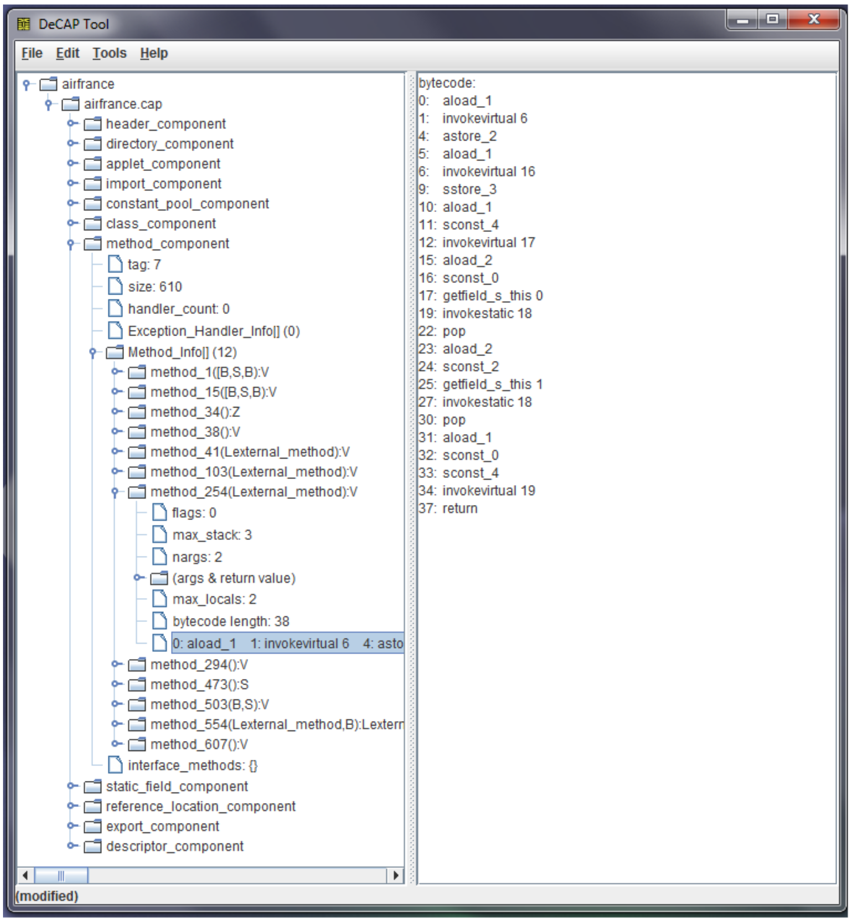
Referring to Figure 20, with a double click expand the node `method_component`, then go to the `Method_info` node and expand it. For each method, a node will be shown to represent the root of a subtree containing the method's information. To show the complete bytecode of the method, expand the *bytecode* attribute of the method, i.e. the node labeled with the first bytecode instruction.

As an example, Figure 21 shows the bytecode of method `method_254`, whose length is 38. Instruction are numbered following the program counter value. For `Invoke` instructions, the token is also displayed.

References

[1] Chen, Z. (2000) Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison-Wesley Longman Publishing Co., Inc, 2000.

[2] Java Card Platform Specification. Sun Microsystem. http://java.sun.com/products/javacard



```
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)
        method_1([B,S,B):V
        method_15([B,S,B):V
        method_34():Z
        method_38():V
        method_41(Lexternal_method):V
        method_103(Lexternal_method):V
        method_254(Lexternal_method):V
          flags: 0
          max_stack: 3
          nargs: 2
          (args & return value)
          max_locals: 2
          bytecode length: 38
          0: aload_1    1: invokevirtual 6    4: asto
        method_294():V
        method_473():S
        method_503(B,S):V
        method_554(Lexternal_method,B):Lextern
        method_607():V
      interface_methods: {}
    static_field_component
    reference_location_component
    export_component
    descriptor_component
```

```
bytecode:
0:  aload_1
1:  invokevirtual 6
4:  astore_2
5:  aload_1
6:  invokevirtual 16
9:  sstore_3
10: aload_1
11: sconst_4
12: invokevirtual 17
15: aload_2
16: sconst_0
17: getfield_s_this 0
19: invokestatic 18
22: pop
23: aload_2
24: sconst_2
25: getfield_s_this 1
27: invokestatic 18
30: pop
31: aload_1
32: sconst_0
33: sconst_4
34: invokevirtual 19
37: return
```

Figure 21: The bytecode representation in the description window.