Radboud Universiteit Nijmegen

M T

Finding bugs in Java

Author:
M V

Supervisors:
E P
W M

January 15, 2007

ABSTRACT

Nowadays, developing high-quality software is a crucial issue in most fields we deal with, especially in those with high-reliability requirements (e.g. banking applications). Improving the quality of our software can be pretty easy by using static analysis techniques.

Static analysis techniques are automatic strategies for examining software programs, while checking certain properties. In the same way that compilers find errors in programs, these techniques find potential problems ignored by compilers. Their main advantage is that they do not actually execute the analyzed programs.

In this thesis, we discuss about such techniques, as well as describing in detail two custom source code analyzers written for specific purposes. One of them scans Java programs looking for pieces of code where the “this” reference is published before completing its construction. The other examines Java Card applications where the objects used with some Java Card API methods must obey certain properties, such as being *transient*, *persistent*, etc.

ACKNOWLEDGEMENTS

I would like to express my gratitude to those who, in one way or another, have helped me during my research:

First of all, I wish to thank Erik Poll and Wojciech Mostowski for their interest in supervising me, a foreign student they did not know at all, as well as their continuous helpful explanations. I am especially grateful to my parents, whose lifelong effort has enabled me being here today. Thanks to Elena, Eva and Luis too, for encouraging me throughout these months I have been abroad. And last, but not least, I would like to thank Marta for teaching me the meaning of *generosity*.

Sincerely, I do appreciate you all.

Contents

1 Summary

Bug-free applications are increasingly needed every day. For this reason, several techniques in charge of helping us to develop higher-quality software have been developed over the years. These strategies include:

- Code reviews: analyze an application by hand. Its main disadvantage is the high costs they involve.

- Automatic techniques: analyze an application automatically. Two strategies can be distinguished:

  - Dynamic techniques: they only analyze those program parts which are executed.
  - Static techniques: they analyze every program part, although it might never be run.

Code reviews are frequently used, but they are quite hard and expensive to apply. Therefore our goal is developing automatic analysis techniques, specifically static ones, in order to help code reviewers to find defects (bugs) in source code.

In this thesis, we discuss about such static analysis techniques, as well as describing in detail two custom detectors which look for certain defects in Java [20] and Java Card [21] source code (one detector for each platform). Our final aim is adding these custom detectors to the set of detectors used by the tool FindBugs [28].

FindBugs is a static analysis tool that examines Java class/JAR files looking for potential problems by matching their bytecodes against a list of bug patterns. A bug pattern is a code idiom that is likely to be an error.

The first custom detector, which is discussed in Chapter 4, tries to find pieces of Java source code where the “this” reference may escape before completing its construction. This could be particularly hazardous since other threads may access the “this” data structures before being initialized.

In order to find such hazards, our detector examines the Java bytecodes of the analyzed application. Furthermore, whenever it finds a piece of code which follows a bugpattern, it issues a warning.

Unfortunately, our first detector has several limitations. For instance, it obeys a safe strategy, so it may warn about correct code in specific situations (*false positives*). In addition, testing the detector with large real Java applications has proved that the followed algorithm in charge of finding out the application’s hierarchy can be too inefficient. This limitation is especially serious when dealing with larger applications.

The second custom detector, which is discussed in Chapter 5, examines pieces of Java Card [21] code where the objects which deal with some Java Card API [22] methods must obey certain properties, such as being *transient*, *persistent*, etc [1]. In order to do so, we have used the Java annotations mechanism [19].

The Java annotations mechanism is a way of adding metadata to Java source code, available to the programmer at run-time. Java annotations can be applied to a wide range of elements, such as variables, methods, classes and even other annotations. It is available at JDK [23] since version 1.5.

To implement the second plugin, we have followed the same approach used to write the first. Our tool examines Java bytecodes and issues a warning for each found bug instance.

In regard to the second detector’s limitations, the main one is caused by the impossibility of knowing if a local variable has been annotated or not. Nowadays, annotations on local variables are not stored into the Java bytecodes, so neither FindBugs nor our detector can get this information. This defect means that, in most cases, our detector cannot be applied to local variables.

Despite this limitation, the detector successes in finding a broad range of bugs, even on real Java Card software. Actually, our detector finds several defects in real Java Card applications such as Pacap or Demoney, especially in the former.

In conclusion, automatic techniques, like the source code analyzers implemented for this thesis, do help making code reviews both cheaper and easier. That is the main reason why, in most cases, their use is so recommended.

[1]The definition of transient or persistent is only applicable within the Java Card scope, not within the Java scope.

2 Introduction

Who resists, wins.
(C J ´C)

Nowadays, one of the most serious problems found in software development processes consists in obtaining high-quality products. Using analysis techniques may ensure more error-free applications, although it needs some effort to be applied. For this reason, many different analysis techniques have been developed over the years. Depending on the technique approach, applying it would be more or less difficult. Frequently, it is possible to find bugs in software using not very sophisticated strategies.

Some of the most used finding bugs techniques are:

- **Code inspections**: as the name suggests, it involves reviewing the applications source code while looking for bug instances. Their main drawback is the high costs they imply, as well as other cons:
  - Human observers are easy to be influenced by what a piece of code is *intended* to do.
  - It is really a very tedious activity.
  - Normally, it is very difficult trying to understand another one's program. Sometimes it is even hard analyzing our own source code.
- **Automatic techniques**: analyze programs in a mechanical manner. Usually these strategies are more objective and easier to apply than code inspections. Two different categories can be distinguished:

– Dynamic techniques: are focused on the runtime behaviour of the analyzed program. Their primary drawback is that they require both writing down high-quality tests and applying them. So, these strategies can only find errors in those program parts executed in the test cases.

– Static techniques: can analyze every single program part, although it might be never be run. Unlike dynamic techniques, these approaches require neither writing nor applying any test.

Within source code analyzers, there is a vague distinction between *style checkers* from *bug checkers*:

– Style checkers: examine programs looking for pieces of code which violate particular style rules. This kind of tools is particularly useful during software development process, since they make each one understands each other's code easier. Sometimes, there are style rules which help preventing certain kinds of bugs, but a style violation does not always mean to be a bug. For instance, PMD [10] and Checkstyle [15] are widely used style checkers.

  Listing 2.1 depicts one violation of a style rule. Remember that, in Java, each class name should start with a capital letter.

```
public class classStartingWithSmallLetter {
...
}
```

Listing 2.1: Violation of a style rule in Java.

– Bug checkers: examine programs looking for pieces of code which may cause an application to misbehave at runtime. This kind of tools are useful when achieving high-quality software is a crucial issue. One relevant disadvantage shown by bug checkers is the high percentage of false warnings they tend to notify. To solve this problem, one possibility consists in tuning the detector implementation once it is almost finished.

  Some of these tool's pros are:

  ∗ They do find real bug occurrences effectively.
  ∗ Their output is pretty easy for programmers to understand, so many detected bugs can be fixed easily.
  ∗ For particular interests, writing custom detectors is quite easy.

  Jlint [12] and FindBugs [28] are two of the most famous bug checkers. Both of them analyze Java bytecode looking for bug occurrences.

Listing 2.2 illustrates one example of a bug caused by the use of a null pointer reference. If the variable *c* is null, then *c.isDisposed()* will throw a NullPointerException.

```
if (c == null && c.isDisposed()) {
  ...
}
```

Listing 2.2: Null pointer bug found in Eclipse 3.0

It is important to emphasize that the border between style checkers and bug checkers is pretty vague. For instance, the PMD tool inspects style rules, but it also checks some sorts of bugs.

One interesting possibility consists in using several of these tools to analyze an application and afterwards combining all their outputs together. This combination of checkers will both show how close their results are and provide a mechanism for detecting a wider range of bugs. For instance, a comparison of bug finding tools for Java is available at [14].

Even more, from my own experience, when I started to scan my own applications, a lot of bugs arose and showed me the way to improve my own code. Therefore, I encourage everyone to test his own programs and realize how well these techniques work.

For my research, I have been working with FindBugs. As was said before, FindBugs is a bug checker which looks for bug instances by analyzing Java bytecodes (a further description of FindBugs is available at Chapter 3). Moreover, I have written two custom detectors intended to be added to the FindBugs detectors set.

3
FindBugs

FindBugs [28] is a static analysis tool that examines Java class/JAR files looking for potential problems by matching their bytecodes against a list of bug patterns. A bug pattern is a code idiom that is likely to be an error.

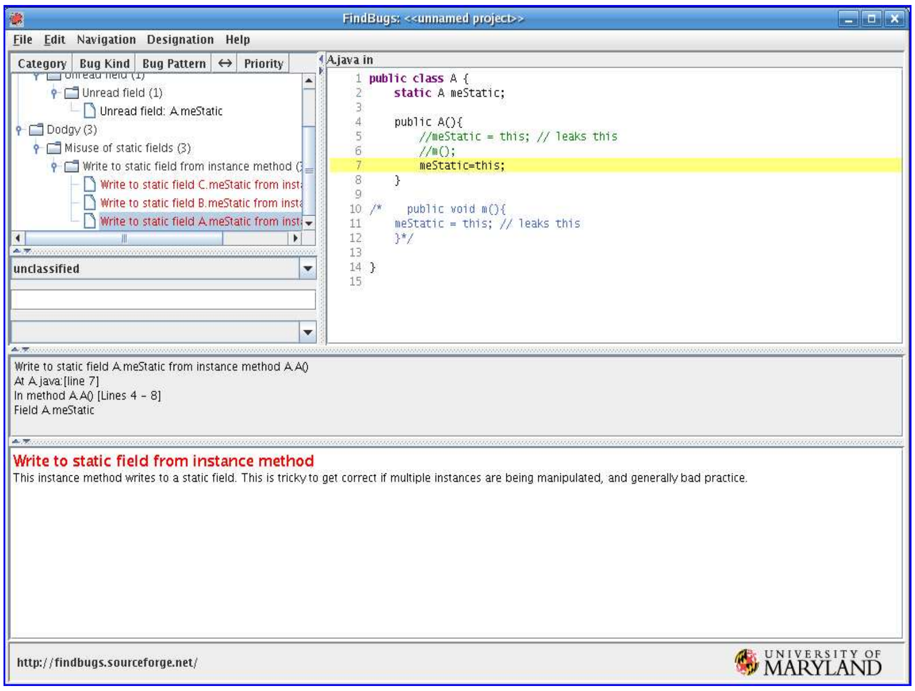
The tool is free software, distributed under the terms of the Lesser GNU Public License and has been developed by the University of Maryland. Since FindBugs is open source, all its source code, binaries and documentation can be downloaded from its web page. This is particularly useful when writing custom detectors, apart from using those already implemented.

Currently, FindBugs contains detectors for more than fifty bug patterns and more custom detectors can be written for specific purposes. All of the detectors are implemented using the FindBugs API [29] and/or the Byte Code Engineering Library [26].

The detectors are implemented according to the Visitor design pattern [36], so each detector visits each class of the analyzed application. There are several implementation strategies, such as *Linear code scan* and *Dataflow analysis*, although none uses very complicated analysis techniques.

Three different front ends may be used when dealing with FindBugs:

- A batch application that generates text reports, one line per warning.
- A batch application that generates XML reports.
- An interactive tool that can be used to browse warnings and its associated source code, as well as reading/writing XML reports. See Figure 3.


FindBugs: <<unnamed project>>
File Edit Navigation Designation Help
Category Bug Kind Bug Pattern ↔ Priority
Unread field (1)
Unread field: A.meStatic
Dodgy (3)
Misuse of static fields (3)
Write to static field from instance method (3
Write to static field C.meStatic from inst
Write to static field B.meStatic from inst
Write to static field A.meStatic from inst
unclassified
A.java in
http://findbugs.sourceforge.net/
UNIVERSITY OF MARYLAND

```
public class A {
    static A meStatic;

    public A(){
        //meStatic = this; // leaks this
        //m();
        meStatic=this;
    }

/*    public void m(){
    meStatic = this; // leaks this
    }*/

}
```

```
Write to static field A.meStatic from instance method A.A()
At A.java:[line 7]
In method A.A() [Lines 4 - 8]
Field A.meStatic
```

Write to static field from instance method
This instance method writes to a static field. This is tricky to get correct if multiple instances are being manipulated, and generally bad practice.

Figure 3.1: Screenshot of FindBugs

Within source code analyzers, FindBugs is classified as a *bug checker* (see definition in Chapter 2), so it is focused on trying to find pieces of code which may cause a program to misbehave at runtime. However, the tool does not search any violation of specific style rules.

4 Don't let the "this" reference escape during construction

This chapter describes the implementation of a custom detector intended to look for bug instances where "this" escapes during construction. This chapter has the following outline:

- Section 4.1 describes in detail the bug our detector tries to find. For this purpose, Goetz's article [1] has been pretty helpful.
- Section 4.2 is focused on explaining, using a high-level overview, how the detector is implemented and the way it works.
- Section 4.3 shows several tests our detector has been applied to. These tests include both custom programs and real Java applications.
- Section 4.4 describes the main limitations we have found while dealing with our detector. Most of these limitations have been inferred from section 4.3.
- And finally, section 4.5 summarizes the previous sections and emphasizes the most relevant issues/limitations.

4.1 Bug Description

4.1.1 Introduction

One hazardous defect found in Java source code is *data racing*. A data race appears when several threads/processes are reading/writing a shared data unit concurrently and without any kind of synchronization. Consequently, the final result of the program will depend on the order in which threads/processes are scheduled.

Listing 4.1 (taken from [1]) shows an example of a simple data race. Depending on the threads scheduling, the program will print either 0 or 1. If the second thread is scheduled before the first one writes on *a*, it will print 0, otherwise it will print 1.

```java
public class DataRace {
  static int a = 0;

  public static void main(){
    new MyThread().start();
      a = 1;
  }

  public static class MyThread extends Thread {
    public void run() {
      System.out.println(a);
    }
  }
}
```

Listing 4.1: Simple data race.

Running this program several times might produce different results (either 0 or 1), which is not a desirable situation at all.

4.1.2 Don't let the "this" reference escape during construction

In his article, Goetz [1] points out a bug which may introduce a data race within a Java program. This section explains this bug in detail.

One of the hazards which can produce a data race into a program is publishing the "this" reference to another thread before the "this" object is completely created.

Leaking a reference to an object before it is completely created involves a risky situation since another thread(s) may use that reference to access the object's fields and methods, even before initializing every data structure. In Listing 4.2, the class *A* leaks "this" within its constructor (line 7):

```java
1 public class A {
2   static A meStatic;
3   int number = 0;
4 
5   public A(){
6     ...
```

```
      meStatic = this; // Leaks ‘‘this’’
      ...
      number = 1;      // Initializes number
      ...
   }

   public static void printNumber(){
      System.out.println("Number: "+number);
   }
}
```

Listing 4.2: Class A leaks “this”.

Listing 4.3 depicts another thread which uses the published “this” reference to access the fields or methods of “this”. Notice that depending on the threads scheduling, the used data structure might not have been initialized yet.

```
1 System.out.print(A.meStatic.number); //It will print either 0 or 1
2 A.meStatic.printNumber();            //It will print either 0 or 1
```

Listing 4.3: Another thread using the leaked “this” reference.

So, lines 1 and 2 in Listing 4.3 will print either 0 or 1 since *number* might not have been initialized before accessing it. As was said before, the final result will depend on the threads scheduling, which is clearly undesirable.

However, not all the “this” escapes during construction are harmful, only those that publish the reference where other threads can see it. For instance, Listing 4.4 contains one statement in which a “this” reference is stored in a field (line 4). As *me* is not visible from any other threads, this assignment does not mean a bug.

```
1 public class A{
2    private A me;
3    public A(){
4       me = this; //Safe since me is invisible from other threads
5    }
6 }
```

Listing 4.4: Safe “this” reference escape during construction.

There are two different ways of publishing the “this” reference:

- Using an **explicit reference**. For instance, when “this” is stored in a static field or passed as parameter within a method call. Listing 4.5 shows an example for each case: both the field *meStatic* and the method *registerObj* may be used to access “this” before it is completely created.

```
public class A {
  static A meStatic;

  public A(AnotherClass ac){
    ...
    meStatic = this;       // Leaks ‘‘this’’ explicitly
    ...
    ac.registerObj(this); //Leaks ‘‘this’’ explicitly
    ...
  }
}
```

Listing 4.5: Class A leaks “this” explicitly.

- Using an **implicit reference**. For instance, when a reference to an instance of a non-static inner class is published within a constructor. Listing 4.6 shows an example for this case: the new inner class instance maintains an implicit reference to an outer class instance: “this”. So it is possible for another thread to access “this” before completing its creation.

```
public class OuterClass {

  public OuterClass(AnotherClass ac){
    ...
    ac.regObj(new InnerClass()); //Leaks ‘‘this’’ implicitly
    ...
  }

  class InnerClass{}
}
```

Listing 4.6: Class OuterClass leaks “this” implicitly.

In addition to the previous cases, there is another relevant situation which must be taken into account: A “this” reference can not only escape from a constructor, but also from methods called from within a constructor. Listing 4.7 shows one example which illustrates this situation. Notice that both method *m1* and method *m2* contain a statement with a “this” reference escape. As *m1* is a method that is reachable from constructor, its defect means a hazard. On the other hand, *m2* is not reachable from constructor, so its defect does not involve any risk [2].

```
public class A {
  static A meStatic;

  public A(){
    ...
    m1();
    ...
  }

  //m1() is reachable from constructor
  public m1 (){
    ...
    meStatic = this; //Leaks ''this''
    ...
  }

  //m2() is not reachable from constructor
  public m2 (){
    ...
    meStatic = this; //Leaks ''this''
    ...
  }
}
```

Listing 4.7: “this” reference escaping from non-constructor methods.

Henceforth a method which is reachable from constructor will be referred to as a RFC method.

Our detector has to notify only those bug instances found in constructors and/or RFC methods. These bug occurrences are classified as *real* bugs. The rest of bug instances must not be issued because they do not mean any real risk.

[2]In the example showed in Listing 4.7, if *m1* invokes another method, called *m3* for example, then *m3* will be reachable from constructor as well. Consequently, every bug occurrence found in *m3* will mean a real bug.

4.1.3 Conclusion

To conclude, our detector is intended to:

1. Detect all those pieces of code where a "this" reference escapes. Remember that the harmful escapes are those which can be seen from other threads.
2. Notify all the detected bug occurrences, except those which were not found in RFC methods.
3. Minimize as much as possible the amount of both *false positives* and *false negatives*.
   - A *false positive* is defined as the error of warning about correct code.
   - On the other hand, a *false negative* is defined as the error of failing to warn about incorrect code.
4. Once warnings are issued, the user will be able to fix them in order to get a higher-quality software, whenever those issued warnings involve real bug instances.

4.2 Detector Implementation

4.2.1 Introduction

This section is focused on describing, using a high-level overview, how the detector is implemented and the way it works. To create a FindBugs detector, the next steps have to be followed:

1. Write down some Java [20] programs including the bugs intended to be detected.
2. Using the output of the `javap` command [17], examine the disassembled JVM code for those programs in order to infer the bug pattern(s).
3. Using the bugpattern(s) inferred in 2, the next step is writing a sketch of the strategy our detector will follow.
4. Implement the detector in Java using the Bytecode Engineering Library (BCEL) [26] and the FindBugs API [29]. The detector has to be able to find the bug pattern(s) inferred in step 2, according to the strategy written in step 3.
5. Package the detector in a JAR file, so that FindBugs can recognize it as a plugin. This JAR file has to contain the class file(s) used to implement the detector, as well as two XML files: `messages.xml` and `build.xml`.

4.2.2 Bug patterns

Even before writing the detector implementation, the first step is finding out exactly what the detector has to look for, i.e. knowing the bug pattern(s) it has to find. A bug pattern is a code idiom that is likely to be an error. The easiest way to infer bug patterns is writing down some Java programs containing the defects intended to be detected. Next, we have to study the disassembled JVM code for those programs and finally infer the bug pattern(s) [3].

[3]The Java Virtual Machine instruction set is available at [31].

In this section we discuss the different bug patterns our detector has to look for. These bug patterns are classified into two main categories, depending on how the reference escapes: **explicitly** or **implicitly** (see section 4.1.2).

1. Cases where an **explicit** “this” reference escapes:

   - Cases where “this” is written to a static field. For the source code listed in 4.8:

```java
public class A {
  static A staticA;
  public A(){
    staticA = this; //Leaks ``this''
  } }
```

Listing 4.8: “this” is written to a static field.

Its disassembled code for the Java Virtual Machine is shown in 4.9:

```
 ...
ALOAD_0    //Loads ``this''
PUTSTATIC  //Writes ``this'' on staticA
 ...
```

Listing 4.9: Dissasembled code for Listing 4.8.

   - Cases in which “this” is passed as parameter within a method call. For the source code depicted in 4.10:

```java
public class A {
  public A(AnotherClass obj){
    obj.registerObj(this); //Leaks ``this''
  }
}
```

Listing 4.10: “this” is passed as parameter within a method call.

Listing 4.11 shows its JVM disassembled code:

```
 ...
ALOAD_1         //Loads obj
ALOAD_0         //Loads ``this''
INVOKEVIRTUAL   //Calls obj.registerObj(this)
 ...
```

Listing 4.11: Dissasembled code for Listing 4.10.

- To sum up, Listing 4.12 contains the inferred bug patterns for the **explicit** “this” reference escapes:

```
 ...
 ALOAD_0           //Loads ``this''
 PUTSTATIC         //Writes ``this'' on a static field
 ...
and
 ...
 (ALOAD_<n>)+      //Loads one or more objects
 ALOAD_0           //Loads ``this''
 (ALOAD_<n>)*      //Optionally, more objects can be loaded
 INVOKE_Instr.     //Invokes a method of the 1st loaded object
 ...               //using one or more objects (including
                   //``this'') as parameters
```

Listing 4.12: Inferred bug patterns for the explicit escapes.

2. Cases where an **implicit** reference escapes:

- Cases where an inner object (stored in a field) is passed as parameter within a method call. For the source code listed in 4.13:

```java
public class OuterClass {
  InnerClass innerObj;
  public OuterClass(AnotherClass ac){
    innerObj = new InnerClass();
    ac.registerObj(innerObj); //Leaks ``this'' implicitly
  }
  class InnerClass{}
}
```

Listing 4.13: An inner object publishes “this” within a method call.

Its disassembled code for Java Virtual Machine is shown in 4.14:

```
 ...
 ALOAD_1           //Loads ac
 ALOAD_0           //Loads ``this''
 GETFIELD #no      //Pops ``this'' and pushes innerObj
                   //innerObj contains a reference to ``this''
 INVOKEVIRTUAL     //Invokes ac.registerObj(innerObj),
 ...               //leaking the innerObj's reference
                   //to ``this''
```

Listing 4.14: Dissasembled code for Listing 4.13.

- Cases in which a just created inner object is passed as parameter within a method call. For the source code listed in 4.15:

```
public class OuterClass {
  InnerClass innerObj;
  public OuterClass(AnotherClass ac){
    //Leaks ``this'' implicitly
    ac.registerObj(new InnerClass());
  }
  class InnerClass{}
}
```

Listing 4.15: Just created inner obj publishes "this"implicitly.

Listing 4.16 depicts its JVM disassembled code:

```
...
ALOAD_1           //Loads ac
NEW               //Creates a new inner class instance
ALOAD_0           //Loads ``this''
INVOKESPECIAL     //Stores the ``this''reference in the
                  //just created inner class instance
INVOKEVIRTUAL     //Invokes ac.registerObj(...)
...               //Notice the reference stored in the
                  //INVOKESPECIAL opcode is published
                  //here
```

Listing 4.16: Dissassembled code for Listing 4.15.

- To conclude, Listing 4.17 contains the inferred bug pattern for the **implicit** "this" reference escapes:

```
...
(ALOAD_<n>)+   //Loads one or more objects
ALOAD_0        //Pushes the ``this'' reference

``POP&PUSH''   //Instruction popping the ``this''
               //reference and storing it in an
               //object, which is pushed.
               //POP&PUSH = {GETFIELD, INVOKESPECIAL,...}

INVOKE_Instr   //Invokes a method of the 1st loaded object
...            //including the object that references
               //``this'' as parameter
```

Listing 4.17: Inferred bug pattern for the implicit escapes.

Based on the bugpatterns listed in 4.12 and 4.17, our detector will have to find the sequence of opcodes depicted in Listing 4.18:

```
...
(ALOAD_<n>)*          //Optionally, Loads one or more objects
ALOAD_0               //Loads ``this''

{``POP&PUSH'' |       //Instruction popping the ``this'' reference
                      //and storing it in an object, which is pushed.
                      //POP&PUSH = {GETFIELD, INVOKESPECIAL,...}
 (ALOAD_<n>)*}        //Loads more objects (parameters for INVOKEs)

{INVOKEInst |         //Publishes ``this'' by including it within a
 ``WRITEInst''}       //method call or by writing it on a static field.
...                   //WRITEInst = {PUTSTATIC, AASTORE}
```

Listing 4.18: Bugpattern intended to be found.

4.2.3 Strategy

Once the bugpattern is inferred, the next step is writing a sketch of the strategy our detector will follow. In order to do so, defining the concept of *scope* is needed.

Definition 1 *Informally, a scope is defined as a sequence of Java VM instructions where the last is either InvokeVirtual or InvokeSpecial or InvokeStatic or InvokeInterface or Putstatic or Aastore.*[4]

In a formal way, a scope is defined as

$$x_1,\ x_2,\ \dots,\ x_{n-1},\ x_n$$

where

every x_i is a Java VM instruction

and

$S = \{InvokeVirtual,\ InvokeSpecial,\ InvokeStatic,\ InvokeInterface,\ Putstatic,\ Aastore\}$

$x_1,\ x_2,\ \dots,\ x_{n-1}\ \notin S$

$x_n \in S$

Therefore, our detector will obey the following **strategy**:

- Every Java application is a list of Java VM instructions.
- Divide each analyzed application into different *scopes*, and analyze each one separately.
- For each *scope:*
  When the detector reaches its end, determine whether or not a bug has been seen.

Figure 4.2.3 shows the *scopes* structure for every analyzed application.

[4] Notice the inferred bugpattern (see listing 4.18) ends with one of these instructions.

$$Scope\ 1 \begin{cases} x_1 \\ x_2 \\ . \\ . \\ . \\ x_{a-1} \\ x_a \end{cases}$$

$$Scope\ 2 \begin{cases} y_1 \\ y_2 \\ . \\ . \\ . \\ y_{b-1} \\ y_b \end{cases}$$

$$\begin{array}{c} . \\ . \\ . \end{array}$$

$$Scope\ n \begin{cases} z_1 \\ z_2 \\ . \\ . \\ . \\ z_{c-1} \\ z_c \end{cases}$$

where

every x_i, y_j *and* z_k *is a Java VM instruction*

and

$S = \{InvokeVirtual, InvokeSpecial, InvokeStatic, InvokeInterface, Putstatic, Aastore\}$

$x_1,\ x_2,\ \ldots,\ x_{a-1}\ \notin S$

$y_1,\ y_2,\ \ldots,\ y_{b-1}\ \notin S$

$z_1,\ z_2,\ \ldots,\ z_{c-1}\ \notin S$

$x_a \in S$

$y_b \in S$

$z_c \in S$

Figure 4.1: *Scopes* structure for every analyzed application.

4.2.4 Detector Implementation

In FindBugs, bug detectors are classified into two categories:

- Visitor-based detectors.
- CFG-based detectors, where CFG stands for Control Flow Graph.

Our detector is a Visitor-based one. This sort of checkers is used for simpler cases, whereas the CFG-based type is used for more complex analysis. Our detector, called `FindThisReferenceEscape`, extends the `BytecodeScanningDetector` [5] class, so it is based on the Visitor Design Pattern [36], implemented by FindBugs.

The `BytecodeScanningDetector` class and its superclasses provide a set of fields and methods which are pretty useful when writing a custom detector. Moreover, there are certain methods which must be overridden. The list of these overridden methods, as well as their operations will differ from one detector to another. For our purposes, the next methods have been overridden:

- `public void visitClassContext(ClassContext classContext):` FindBugs invokes this method whenever a new class in the application is being analyzed. Specifically, our detector uses this method in order to gather information related to static fields, inner classes instances, class contexts, inheritance issues, etcetera.
- `public void visit(Method obj):` FindBugs invokes this method whenever a new method in the source code is being analyzed. Our detector uses this method to restart the currently analyzed *scope* (See Definition 1).
- `public void sawOpcode(int seen):` FindBugs calls this method for each opcode seen within a method body. Specifically, an opcode is an integer which identifies one Java VM instruction to be performed (each Java VM instruction has one and only one associated opcode). Depending on the current opcode sequence, our detector determines whether it finds a bug or not.
- `public void report():` FindBugs calls this method at the end of the analysis. It is in charge of notifying all the previously stored bug instances found in constructors and/or RFC methods.
- `public Object clone():` Creates and returns a copy of the custom detector object.

Apart from overriding the methods listed above, defining some auxiliary custom methods has been necessary as well. For instance:

- `private void findReachableMethods():` Finds RFC methods and stores them into a data structure. Remember that only those bug instances found in RFC methods mean real bugs.

[5]The `BytecodeScanningDetector` documentation is available at the FindBugs API [29].

- `private void storeBugInstance(String class, Method m, BugInstance bi)`: Stores a new bug instance into a data structure. Once the analysis has finished, the detector will issue only those stored bug instances which are in RFC methods.

High-level Algorithm

This section provides an abstract vision of the algorithm our detector follows (See listing 4.19). See the following section (Low-level Algorithm) for the concrete version.

```
L : list of bugs (Initially empty)

For each Class C in the Application Do
  Gather information related to C
  For each Method M in the class C Do
    Restart the current scope
    For each Opcode O in the method M Do
      Store information about the sequence of opcodes seen
      If (the end of a scope is reached) Then
        Using the information about the seen opcodes, determine whether
            or not a bug has been detected. If so, store a new bug
          instance in L
        Restart the current scope

For each BugInstance bi in L Do
  If (bi is in a RFC method) Then
    Notify bi
```

Listing 4.19: High-level algorithm.

Low-level Algorithm

This section explains in detail the algorithm followed by our detector. Based on the high-level algorithm, it will guide us to the detector implementation, which is explained in the next section.

So, the next steps compose the algorithm of our detector:

1. Whenever a new class is being analyzed, the detector invokes the `visitClassContext(ClassContext)` in order to gather some information related to the current class. For each class, this information includes the list of its superclasses, the class context[6], the name of its static fields and finally, the name of its inner classes objects.

[6] In the FindBugs API[29]: "A ClassContext catches all of the auxiliary objects used to analyze the methods of a class. That way, these objects don't need to be created over and over again." Basically, our detector stores every class context in order to access every class's methods whenever.

In order to store all this information, several data structures are needed. For each class, the detector needs:

- A list containing all its superclasses. This is useful when dealing with inheritance issues. Remember that if a superclass leaks "this", then its subclasses may leak "this" too.
- A list containing all its static fields. This is needed to control cases where "this" escapes explicitly. See Listing 4.8 for an example.
- A list containing all its inner classes instances. Storing them helps to control those cases where leaking "this" implicitly is possible by publishing an inner class instance. See Listing 4.13 for an example.

In addition, three global lists are need as well. These are:

- A list with all the class contexts for the current analyzed application.
- A list containing all the RFC methods for the current analyzed application.
- A list containing all the bugs detected within each method.

2. Whenever a new method is being analyzed, the detector invokes the method `visitMethod(Method)`. Remember that when a new method starts, a new *scope* starts as well. Therefore `visitMethod(Method)` is in charge of starting new *scopes*.

3. Within a method body, the method `sawOpcode(int)` is invoked for each seen opcode. This method is used in order to find pieces of code which follow a bug pattern, i.e. bug instances. For this purpose, the detector uses a set of boolean variables to determine whether a bug appears or not. Examples of these boolean variables are:

   - `seenALOAD_0`: true if an `ALOAD_0` opcode has been seen within the current *scope*, false otherwise.
   - `seenALOAD`: true if an `ALOAD[_n]` opcode has been seen within the current *scope*, false otherwise.
   - Etcetera.

   Obviously, every seen opcode may cause a change in one or more of these boolean variables. Now, when step 2 says "`visitMethod(Method)` is in charge of starting new *scopes*", it means restarting the whole set of booleans.

4. Whenever the end of a scope is reached, the detector determines if a bug occurrence has been found. If so, instead of reporting it immediately, it is stored in a data structure using the auxiliary method `storeBugInstance(...)`.

5. Once the source code is completely analyzed, the detector has to notify only those bug instances which appear in constructors and/or RFC methods. For this purpose the method `report()` is used.

As this method is invoked at the end of the analysis, by this time we know:

- Every class context.
- The whole inheritance structure: each class knows its superclasses.

So, even before reporting anything, we use this method to:

- Find the RFC methods (using the methods contained in the list of class contexts).
- Inherit methods with bugs: if a class contains methods with bugs, then its subclasses inherit those bugs (except those which override a method with bugs without introducing any defects). In this case, the found bugs are reported only in the superclass.

And finally report the bug instances: scan the list of reachable methods and the list of bugs and notify only those bug occurrences which appear in constructors and/or RFC methods.

Implementation

This section summarizes the detector implementation for the concrete algorithm just explained. The detector is composed of three classes:

- `FindThisReferenceEscape`: Main class, Listing A.1 (Appendix) shows its commented source code.
- `ClassMethodSignature`: Auxiliary class. Groups a class name, a method name and a method signature.
- `JavaClassMethod`: Auxiliary class. Groups a JavaClass [7] instance and a Method object.

Apart from the main class, whose commented source code is listed in A.1, the two auxiliary classes are:

- `ClassMethodSignature`: Groups a class name, a method name and a method signature. It is used by the main class in order to index the Hashtable `htMethodWithBugs`. Its main methods are:
  - `public ClassMethodSignature(String c, String m, String s)`.
  - `public void setClassName(String c)`.
  - Equivalent set for MethodName and Signature.
  - `public String getClassName()`.
  - Equivalent get for MethodName and Signature.
  - `public String toString()`.
- `JavaClassMethod`: Groups a JavaClass instance and a Method object. It is used by the main class when storing a reachable method into the ArrayList `reachableMethods`. Its main methods are:

[7]The definition of the JavaClass class is available at the FindBugs API [29]

  - `public JavaClassMethod(JavaClass javaClass, Method m)`.
  - `public void setJavaClass(JavaClass javaClass)`.
  - Equivalent set for Method.
  - `public JavaClass getJavaClass()`.
  - Equivalent get for Method.
  - `public String toString()`.

Detector JAR package

After implementing the detector, next step is packaging it as a JAR file so that FindBugs can recognize it as a plugin. For this purpose, Graat's master thesis [13] and Grindstaff's article [3] have been two helpful sources of information, especially Graat's research.

The JAR package contains the class files our detector has, these are:

- `FindThisReferenceEscape.class`
- `JavaClassMethod.class`
- `ClassMethodSignature.class`

as well as two XML files:

- `findbugs.xml`
- `messages.xml`

The most recommended way of constructing this JAR file is using Apache Ant [25], so a build script file is also needed to create the plugin.

Next, these XML files are explained in detail:

- `findbugs.xml`: Contains relevant information used by FindBugs. This information includes where the detector is and what sort of bugpatterns it detects. For each custom detector, a `Detector` and a `BugPattern` element must be added to the `findbugs.xml` file.
  - `Detector`: Indicates the class file that implements the detector, its speed attribute(fast, moderate or slow) and which bug it reports.
  - `BugPattern`: Specifies:
    * `Abbrev`: Abbreviation used to group several related bugs.
    * `Type`: An unique identifier which represents each bug detector.
    * `Category`: Indicates what kind of bugs the detector looks for. Six categories can be distinguished:

      · C : General correctness issues.
      · M _C : Multithreading correctness issues.
      · M _C : Potential vulnerabilities if exposed to malicious code.
      · P : Performance issues.
      · S : Style issues.
      · I18 : Internationalization issues.
    * In addition to the mentioned attributes, one can add `true` as fourth parameter to indicate the detector is experimental.

  Listing A.2 contains the `findbugs.xml` file defined for this detector.

- `messages.xml`: Contains details related to each detector and the bugpatterns it detects. For each custom detector, a `Detector`, a `BugPattern` and a `BugCode` element must be added to `messages.xml` file.
  - `Detector`: Specifies the class that implements the detector and describes it briefly using some HTML code. This description will appear in the FindBugs API (Configure Detectors dialog).
  - `BugPattern`[8]: Provides a short, a long an a detailed description of that kind of bugs the detector looks for. All these descriptions are used in the FindBugs UI (View Full Description / View Details).
  - `BugCode`: It is used by the UI when using the By Bug Type tab. The `abbrev` attribute (See `findbugs.xml`) is used to group different but related bugs.

  Listing A.3 shows the `messages.xml` file defined for this detector.

- `build.xml`: As was said before, the best way of creating the JAR package is using Apache Ant, so a build script file is also needed. Based on the script Graat uses in his FindBugs-Java Card plugin [13], I have written my own `build.xml` file (See listing A.4). Basically it contains the following sections:
  - `Properties`: Define general properties/variables used in the rest of the file, such as the plugin source directory.
  - `Build`: It is the standard target when invoking the `ant` command. It calls the `Compile` and `Jar` sections.
  - `Compile`: Compiles the plugin. Takes the source files and creates the class files.
  - `Jar`: Creates the JAR file using the just compiled class files. Furthermore, it invokes the `Validate XML` section.
  - `Validate XML`: Validates the XML files (`findbugs.xml` and `messages.xml`) needed by Findbugs.
  - `Javadoc`: Generates API documentation.
  - `Version`: Shows version information.
  - `Help`: Prints a help message including all the available targets.

[8]Note that for each BugPattern, the type used in `findbugs.xml` and in `messages.xml` must fit.

In addition to the `Properties` section, there is another external file which defines some variables used in `build.xml`. Typically this external file, called `build.properties`, includes both the FindBugs and the plugin directory. Anyway the use of this file is only optional since the information it contains can be included in `build.xml` directly [9].

See Listings A.4 and A.5 for the contents of the used `build.xml` and `build.properties`, respectively.

4.3 Tests

Once the detector is written, its implementation must be tested. In order to do so, the following steps are recommended:

1. Write down some Java programs containing the bugs intended to be found, i.e. some custom tests.
2. Test our detector using those programs.
3. Check the detector's behaviour to determine if it works properly, if there are any *false positives* and/or *false negatives*, etc.

Furthermore, testing the detector with some real Java applications is strongly recommended. This will help us to find new *false positives* and/or *false negatives*, check the detector's speed when dealing with large projects, etc.

4.3.1 Custom Tests (1 of 2)

The first custom test is composed of two classes: CustomTest1.java (See Listing A.6) and ClassA.java (See Listing A.7). The main class is CustomTest1 since it contains all the relevant statements, whereas ClassA is only an auxiliary class.

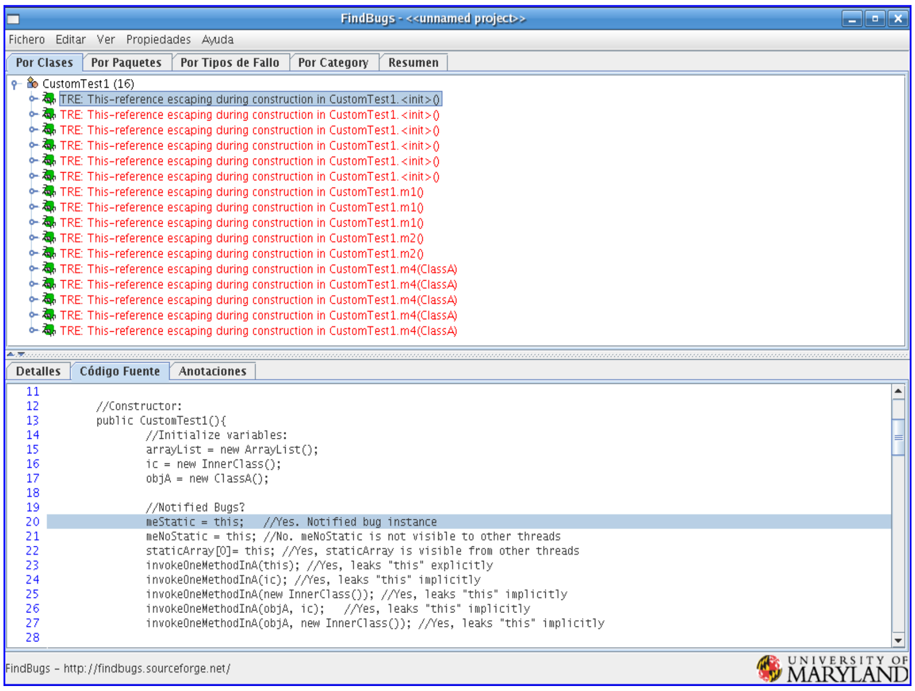
Table 4.1 summarizes the results obtained during the first test process. Notice the detector only issues those bugs found in constructors and in RFC methods: `m1()`, `m2()` and `m4()`. On the other hand, it does not notify any defect found in non-RFC methods: `m3()`, `overloaded Method(Object o)` and `CustomTest1.methSameName()`.

Notice that one *false positive* and one *false negative* appear in lines 40 and 45, respectively. These errors will be studied in detail in section 4.4 (Limitations).

[9]Obviously, if an external properties file is used, then it has to be referenced in the build script file.

Line	Source code	Real bug?	Notified?	Comments
22	meStatic = this;	Yes	Yes	Leaks “this” within a constructor
23	meNoStatic = this;	No	No	meNoStatic is invisible from other threads
24	staticArray[0] = this;	Yes	Yes	Leaks “this” within a constructor
25	staticIC = new IC();	Yes	Yes	Leaks ”this” implicitly
26	methCallingA(this);	Yes	Yes	Leaks ”this” explicitly
27	methCallingA(ic);	Yes	Yes	Leaks ”this” implicitly
28	methCallingA(new IC());	Yes	Yes	Leaks ”this” implicitly
29	methCallingA(objA, ic);	Yes	Yes	Leaks ”this” implicitly
30	methCallingA(objA, newIC());	Yes	Yes	Leaks ”this” implicitly
40	arrayList.add(this);	**No**	**Yes**	It thinks a method is leaking “this”
45	objA.method(arrayList);	**Yes**	**No**	It doesn’t detect arrayList leaks “this”
52	meStatic = this;	Yes	Yes	Leaks “this”and `m1()` is RFC
53	meNoStatic = this;	No	No	meNoStatic is invisible from other threads
54	objA.method(this, null);	Yes	Yes	Leaks “this” and `m1()` is RFC
55	ClassA.staticCustom1 = this;	Yes	Yes	Leaks “this” and `m1()` is RFC
65	meStatic = this;	Yes	Yes	Leaks “this” and `m2()` is RFC
66	objA.method(this, null);	Yes	Yes	Leaks “this” and `m2()` is RFC
76	meStatic = this;	No	No	Leaks “this” but `m3()` is not RFC
77	meNoStatic = this;	No	No	meNoStatic is invisible from other threads
78	objA.method(this, null);	No	No	Leaks “this” but `m3()` is not RFC
85	paramA.method(this);	Yes	Yes	Leaks “this”and `m4()` is RFC
86	paramA.method(ic);	Yes	Yes	Leaks “this” implicitly and `m4()` is RFC
88	paramA.method(new IC());	Yes	Yes	Leaks “this” implicitly and `m4()` is RFC
90	objA.method(ic);	Yes	Yes	Leaks “this” implicitly and `m4()` is RFC
91	objA.method(new IC());	Yes	Yes	Leaks “this” implicitly and `m4()` is RFC
110	meStatic = this;	No	No	`overloadedMethod (Object o)` is not RFC
111	objA.method(this, null);	No	No	`overloadedMethod (Object o)` is not RFC
118	meStatic = this;	No	No	`CustomTest1.methSameName()` is not RFC
119	objA.method(this, null);	No	No	`CustomTest1.methSameName()` is not RFC

Table 4.1: Obtained results in Custom Test 1.



```
CustomTest1 (16)
  TRE: This-reference escaping during construction in CustomTest1.<init>()
  TRE: This-reference escaping during construction in CustomTest1.<init>()
  TRE: This-reference escaping during construction in CustomTest1.<init>()
  TRE: This-reference escaping during construction in CustomTest1.<init>()
  TRE: This-reference escaping during construction in CustomTest1.<init>()
  TRE: This-reference escaping during construction in CustomTest1.<init>()
  TRE: This-reference escaping during construction in CustomTest1.m1()
  TRE: This-reference escaping during construction in CustomTest1.m1()
  TRE: This-reference escaping during construction in CustomTest1.m1()
  TRE: This-reference escaping during construction in CustomTest1.m2()
  TRE: This-reference escaping during construction in CustomTest1.m2()
  TRE: This-reference escaping during construction in CustomTest1.m4(ClassA)
  TRE: This-reference escaping during construction in CustomTest1.m4(ClassA)
  TRE: This-reference escaping during construction in CustomTest1.m4(ClassA)
  TRE: This-reference escaping during construction in CustomTest1.m4(ClassA)
  TRE: This-reference escaping during construction in CustomTest1.m4(ClassA)
```

```

        //Constructor:
        public CustomTest1(){
                //Initialize variables:
                arrayList = new ArrayList();
                ic = new InnerClass();
                objA = new ClassA();

                //Notified Bugs?
                meStatic = this;   //Yes. Notified bug instance
                meNoStatic = this; //No. meNoStatic is not visible to other threads
                staticArray[0]= this; //Yes, staticArray is visible from other threads
                invokeOneMethodInA(this); //Yes, leaks "this" explicitly
                invokeOneMethodInA(ic); //Yes, leaks "this" implicitly
                invokeOneMethodInA(new InnerClass()); //Yes, leaks "this" implicitly
                invokeOneMethodInA(objA, ic);   //Yes, leaks "this" implicitly
                invokeOneMethodInA(objA, new InnerClass()); //Yes, leaks "this" implicitly

```

FindBugs - http://findbugs.sourceforge.net/

Figure 4.2: Screenshot of FindBugs during custom test 1.

4.3.2 Custom Tests (2 of 2)

Since Java is an Object-Oriented programming language, it provides a quite useful tool: Inheritance. By using inheritance, it is possible to:

- Define a class as an extension of another one.
- Group different classes which have certain features in common.
- Re-use pieces of code, instead of copy-pasting.
- Etcetera.

In Java, each class is allowed to have one direct superclass, whereas each superclass may have an unlimited number of subclasses.

Since Inheritance is used so often, this section is focused on testing the detector's behaviour when dealing with it.

Listing 4.20 shows one program where inheritance is present. Notice the method `meth()` leaks "this" (line 3) and the class `SubClass` publishes it during construction (line 12).

```java
public class SuperClass{
   static SuperClass meStatic;
   public void meth (){
      meStatic = this; //Leaks ‘‘this’’
   }
}

...

public class SubClass extends SuperClass{
   public SubClass(){
      super.meth(); //or simply meth();
   }
}
```

Listing 4.20: Inheriting bug instances.

The second custom test is composed of three classes: A.java (See Listing A.8), B.java (See Listing A.9) and C.java (See Listing A.10). Notice that C extends B, which extends A.

Table 4.2 summarizes the results obtained during the second test process. Once again, only those bugs found in RFC methods are notified. In this case, neither *false positives* nor *false negatives* appear.

Class	Line	Source code	Real bug?	Notified?	Comments
A	6	meStatic = this	Yes	Yes	Leaks “this” & `m1()` is RFC
A	11	meStatic = this	No	No	Leaks “this” but `m2()` is not RFC
A	16	meStatic = this	No	No	Leaks “this” but `methodWithBugs()` is not RFC
A	21	meStatic = this	No	No	Leaks “this” but `methodWithBugs(i)` is not RFC
B	10	meStatic = this	No	No	Leaks “this” but `method()` is not RFC
B	14	meStatic = this	Yes	Yes	Leaks “this” & `method2()` is RFC
B	22	meStatic = this	Yes	Yes	Leaks “this” & `methodWithBugs()` is RFC
B	27	meStatic = this	Yes	Yes	Leaks “this” & `methodWithBugs(i)` is RFC

Table 4.2: Obtained results in Custom Test 2.

4.3.3 Testing the detector with real Java applications

Once we have checked the detector's behaviour when analyzing some custom tests, the following step is testing it with some real Java applications and finding out how well our detector really works.

Testing the detector with a custom real Java application

Once I had written the detector, I was quite interested in testing it with the largest Java application I ever wrote, so that I could find out if my own code contained "this" escapes.

I developed this application within an Object-Oriented Design and Programming course [24]. In addition, its main features are:

- Written in Java, of course.
- It contains 35 classes, including several non-static inner classes.
- It allows the user to draw small UML diagrams, as well as generating its associated source code automatically. In addition, it can also be used to both compile and run Java applications.

Table 4.3 summarizes the issued warnings during this test. Notice that two *false positives* are issued, as well as two real bug instances. Besides, by inspecting the analyzed source code I realized that there was no bug missing, so no *false negatives* appeared.

Class	Line	Source code	Real bug?	Notified?	Comments
Shared	32	this.srcClass=null;	No	Yes	*False positive*
Shared	33	this.tgtClass=null;	No	Yes	*False positive*
DrawClass	53	addMouseListener(this);	Yes	Yes	Leaks "this" explicitly
DrawClass	54	addMouseMotionListener(this);	Yes	Yes	Leaks "this" explicitly

Table 4.3: Obtained results in real test 1.

Now the problem is finding out why those two *false positives* appeared. Firstly, have a look at its disassembled bytecode at Listing 4.21:

```
...
ALOAD_0        //Loads ``this''
POP            //Pops ``this''
ACONST_NULL    //Pushes null
PUTSTATIC      //Stores null in a static field
...
```

Listing 4.21: Dissasembled code for a part of real test 1

Notice how similar it is to the disassembled bytecode for a `staticField=this` statement:

```
...
ALOAD_0       //Loads ``this''
PUTSTATIC     //Stores ``this'' in a static field
...
```

Listing 4.22: Writing this on a static field (Bytecodes)

So, the detector issued those two *false positives* because it supposed they were `staticField=this` statements. The solution for this problem is as easy as adding to our detector a piece of code obeying the algorithm contained in Listing 4.23:

```
For (every seen POP opcode):
 If (An ALOAD_0 opcode has just been seen) Then
   Reset the seenALOAD_0 boolean field since ``this'' is being popped
```

Listing 4.23: Algorithm for fixing the detector

After tuning the implementation, I re-tested the detector using the same application and it only warned about the two real bug instances, i.e. we got neither *false positives* nor *false negatives*.

This simple test points out clearly how important testing the detector with real applications is, especially for getting rid of *false positives*.

Testing the detector with a large real Java application

This section focuses on the detector's behaviour when analyzing large Java applications. For this test, I used an open-source Java application called MegaMek [30].

MegaMek is a networked Java clone of BattleTech, whose main features are:

- It is open source, distributed under the terms of the GPL license.
- It is programmed in Java and runs on any system with a compatible Java Virtual Machine.
- It is a real large application, containing almost 900 different classes.

Now the test, when I checked the detector's behaviour while analyzing such a large application, I was quite surprised because of the long time it spent to analyze it. Even more, I tried to analyze MegaMek twice and did not manage to end the test since it reached a point where it made no progress [10].

[10]After two days working!

Several reasons may explain this limitation:

- The most likely cause is that the algorithm used to discover the application's hierarchy is too inefficient when dealing with larger applications. This algorithm contains several nested loops which iterates over the whole classes set. Therefore, one possibility would be tuning its implementation and checking the detector's behaviour then.

- Another possibility is that the machine used during the tests does not have enough resources (mainly RAM memory) to handle such tests. If a machine does not have enough RAM memory to load a program, then it spends a lot of time moving data from RAM to SWAP, and vice versa.

Figure 4.3.3 illustrates the use, in percentages, of RAM and SWAP spaces during the beginning of the test. The graph is divided into two parts: one represents the machine's behaviour before executing the test (first 9 minutes), whereas the other depicts how the test is being carried out (next 51 minutes). Notice that the use of both memories starts to increase rapidly as soon as the test starts (minute 9 approximately).

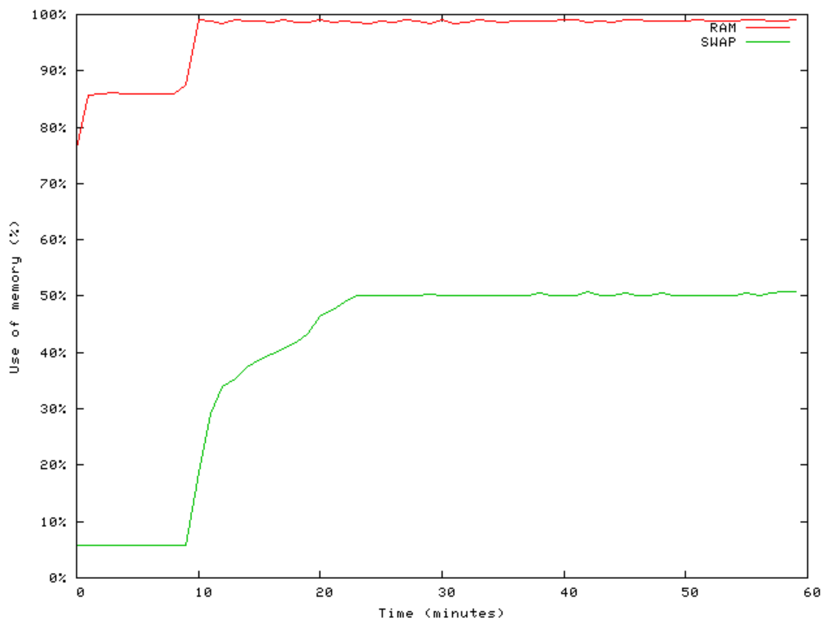


Figure 4.3: Use of RAM and SWAP (first 60 minutes of the test).

The RAM memory is so busy during the test, that a lot of time is spent allocating/ deallocating information to/from SWAP. Therefore, another possibility will be applying the same test on a machine with larger RAM space (current RAM size plus 50 % of the current SWAP size), and checking the detector's behaviour then.

4.4 Limitations

This section describes in detail the limitations we have found when testing the detector. Basically these are one *false positive* and one *false negative* case.

As Rutar, Almazan and Foster say in their paper [14], “It is a basic undecidability result that no bug finding tools can always report correct results”. Therefore, our goal will be developing a detector able to notify every true bug, as well as generating not many *false positives*. Notice that if the detector can find every bug occurrence, then there will be no *false negatives*.

Remember

- A *false positive* is defined as the error of warning about correct code.
- On the other hand, a *false negative* is defined as the error of failing to warn about incorrect code.

When a bug detector issues too many *false positives* and/or fails to warn about incorrect code (*false negative*) too many times, then the most recommended thing is making some tradeoff to get better results. Depending on the nature of the problem, choices will follow one approach or another. Specifically, our detector will obey the following strategy:

Be always as safe as possible

Obviously, this strategy has its pros and cons. On the one hand, *false negatives* rarely occur since the detector issues every piece of code that follows a bug pattern. On the other hand, the detector may notify some *false positives* in specific situations. If this happens, then the user may ignore our detector’s output, which would be a very serious hazard. Anyway, getting a few *false positives* is usually safer than missing some bug instances.

4.4.1 First limitation: *false positive*

As we saw in section 4.2.2, it is possible to publish the “this” reference during construction explicitly by using either a static field or a concrete method. For instance, Listing 4.24 shows one piece of code where a method call is used to publish the “this” reference during construction.

```
public class A{
  public A (AnotherClass ac){
     ...
     ac.regObj(this);
     ...
  }
}
```

Listing 4.24: Explicit bug escape.

This kind of bugs obeys the following pattern:

{Field | Parameter}.methodName({Parameter$_i$, }* **this** {, Parameter$_j$}*)

As we already know, this is one of the bug patterns our detector has to look for.

Now think of the example showed in Listing 4.25: a non-static ArrayList field, which invokes one of its method passing "this" as parameter.

```
public class A{
  ArrayList arrayList;
  public A (){
    ...
    arrayList.add(this);
    ...
  }
}
```

Listing 4.25: Bug-free piece of code.

In this example, it is clear that the ArrayList object is not static and that its invoking one of its methods passing "this" as parameter. Therefore, this code does not involve any risk with leaking "this", unless arrayList is leaked.

Now notice how the `add()` call follows the bug pattern listed above:

arrayList.add(this);

which matches with:

Field.methodName(this)

To sum up, if the detector finds the previous `add()` call while analysis, then it will warn about correct code, i.e. it will issue a *false positive*.

For instance, if the detector analyzes the program depicted in Listing 4.26, then it will think there is bug instance in line 6. However, although line 6 follows one of the bug patterns, it does involve any risk at all. So here a *false positive* arises clearly.

```
import java.util.*;
public class A{
  ArrayList arrayList;
  public A (){
    arrayList = new ArrayList();
    arrayList.add(this);  //Notified bug instance. False positive!
  }
}
```

Listing 4.26: Example of *false positive*.

For this limitation, I have not found any strategy to prevent it yet.

4.4.2 Second limitation: *false negative*

This section describes one case in which our detector fails to warn about incorrect code, i.e. a *false negative*.

In section 4.2.2 we saw the “this” reference can be published during construction both explicitly and implicitly.

Within implicit escapes, as well as using non-static inner classes, there is another way of leaking “this”. It consists in publishing an auxiliary variable which maintains a reference to “this”. Afterwards, that reference may be used to access “this” before completing its construction.

For instance, if the detector analyzes the program shown in Listing 4.27, then it will ignore the leaked ArrayList variable, and consequently will ignore the published “this” reference. Obviously this means a defect in our implementation which should be fixed as soon as possible.

```
import java.util.*;
public class A{
  static ArrayList list;
  public A (AnotherClass ac){
    list = new ArrayList();
    list.add(this);
    ac.regObj(list); //False negative!
                     //arrayList maintains a reference to ‘‘this’’
                     //The detector doesn’t notify any warn
  }
}
```

Listing 4.27: Example of *false negative*.

One possibility for solving this problem would be using a list containing the objects which reference “this”. If one of these objects, or a reference to it, is leaked, then the detector will find a new bug instance.

4.5 Conclusions

This section summarizes the previous research and emphasizes the most relevant issues/limitations.

Throughout the previous sections, we have:

- Explained in detail a Java bug consisting of letting the “this” reference escape during construction. We saw several examples which showed how hazardous this defect can be. Remember that this bug enables another threads to access the “this” information even before initializing it. Furthermore, we saw that the “this” reference can be published either explicitly or implicitly.
- Examined the Java bytecodes for all the possible bug occurrences, as well as inferred a bug-pattern which groups them all.
- Designed an algorithm whose goal is finding instances of the inferred bug pattern.
- Implemented the previous algorithm in Java using the Bytecode Engineering Library (BCEL) and the FindBugs API.
- Tested the detector implementation using both custom tests and real Java applications.
- Using the tests, found out the following limitations:
  - The detector follows a safe strategy, so it may warn about correct code (*false positive*) in very specific situations.
  - In addition, it is possible to get some *false negatives* in those cases in which a “this” reference is published in two steps.
  - The detector is too inefficient when it analyzes larger applications. It can be due to the algorithm used to discover the inheritance hierarchy and/or the lack of resources (RAM memory) of the testing machine.

Finally, several enhancements for this detector are discussed in section 6.2 (chapter 6: Conclusions and future work).

5 Detector for Java Card Applications

This chapter describes a custom detector which analyzes Java Card programs where the objects which deal with some Java Card API [22] functions must obey certain properties, such as being *transient*, *persistent*, etc [11]. The outline of the chapter is as follows:

- Section 5.1 describes what a smart card is, its different categories, hardware components, main applications, etc.
- Section 5.2 gives an overview of the Java Card technology, its memory model, its atomicity and transaction mechanism, etc. Furthermore, it explains some defects found in the Java Card specification.
- Section 5.3 explains in detail the implementation of the detector discussed in this chapter.
- Section 5.4 describes the results we have obtained when testing the detector with custom and real applications. It is important to mention that our detector finds serious bugs in the latter.
- Section 5.5 is about two limitations we have found in our implementation. One is a false *positive* and the other is caused by a Java constraint.
- Finally, section 5.6 summarizes and emphasizes the most relevant issues.

[11]The definition of both *transient* or *persistent* applies only to the Java Card scope, not to the general Java scope

5.1 Smart cards

5.1.1 Definition

A smart card, chip card, or Integrated Circuit(s) Card (ICC) is defined as "a plastic card, usually similar in size and shape to a credit card, containing a microprocessor and memory (which allows it to store and process data) and complying with ISO 7816 standard." [16]

In addition, "a smart card has various tamper-resistant properties (e.g. a secure crypto-processor, secure file system, human-readable features) and is capable of providing security services (e.g. confidentiality of information in the memory)." [35]

Informally, a smart card is a miniature computer with a very restrictive I/O (no screen, keyboard,...) and a very restrictive hardware (tiny memory components, very low-frequence CPU, ...).

5.1.2 Classifications

Smart cards can be classified on the basis of various parameters, the most important are *card components*, *card interface* and *smart card OS*.

Component-Based Classification

According to its components, a smart card can be placed into two categories:

- Memory cards. They are not really smart since they do not contain any microprocessor. However, they contain some non-volatile memory and some specific security logic. This is the simplest kind of smart cards and it is used in segments such as transport, ticketing or prepaid phone cards.

- Microprocessor cards. As the name implies, they contain a microprocessor, as well as several memory components. Technically these are the ones that can be named smart cards. They offer a higher security level and multifunctional capabilities, although they are not as simple as memory cards. Microprocessor cards are used in segments where security and privacy are major concern, such as banking applications, electronic purses, etc.

Interface-Based Classification

Depending on how the communication with the outside world is carried out, two different types of smart cards can be distinguished:

- Contact cards. They must be inserted in a reader named *Card Acceptance Device*, or simply *CAD*, in order to communicate with the outside world. This communication is carried out using a serial interface via eight contact points. Besides, contact cards does not have batteries, so power is supplied by the reader.

- Concactless cards. They do not require insertion into a reader to communicate with the outside world, they use a wound antenna instead. In this case, power can be supplied by an internal battery or can be collected by the antenna. Contactless cards and readers use electromagnetic fields to communicate with each other.

Contact cards have certain limitations (e.g. their contacts can become worn from excessive use). Contactless cards overcome such limitations, although they have their own drawbacks (e.g. they require a certain distance to communicate with the reader).

In this thesis, we will consider only contact cards since they are the most common. Nevertheless, many considerations can be applied equally well to contactless cards.

OS-Based Classification

There are many smart card operating systems, commonly called SCOS, available on the market. The main are JavaCard, MultOS, Cyberflex, etc.

The SCOS are placed on the ROM(See section 5.1.3) and are in charge of memory management, file handling, data exchange, etc.

5.1.3 Smart card hardware

The most important hardware of a chip card is:

- CPU (Central Process Unit). It is the heart of the smart card. It is usually an 8-bit microprocessor based on CISC with clock speeds up to 5-10 MHz. However, CPUs based on RISC and/or {16,32}-bit architecture are also available.

- ROM (Read Only Memory). It is used for storing fixed programs of the card, such as the operating system, permanent data and user applications, etc. As the name implies, it is written only once (usually during the chip fabrication process), so no write is allowed after the card production. Its size usually varies from a few kilobytes to 32 kB.

- EEPROM (Electrical Erasable Programmable Read-Only Memory). It holds the card's application programs and the application data. Like ROM, it preserves data content when power is turned off. By contrast, its data is not permanent and is often erased and rewritten. Typical EEPROM sizes range from 2 kB to 32 kB.

- RAM (Random Access Memory). It is used as temporary working space for storing and modifying data while running certain functions. RAM is non persistent memory, so its content is erased whenever the power is switched off. Typically, the average size of RAM memory in smart cards is between 512 bytes and 2KB.

ROM is the cheapest of these three kinds of memory, while RAM is the fastest and the scarcest.

In addition to this hardware, a chip card can contain more components: a cryptographic co-processor (used in security applications for math computation), etc.

5.1.4 Applications

Nowadays, smart cards are used in a very diverse range of applications, such as:

- Credit and debit cards.
- GSM SIM cards used in mobile phones. Notice not every smart card is necesseraly credit card sized.
- Electronic wallets (e.g. the Dutch *chipknip*).
- ATM cards.
- ID cards (e.g. the recently created Dutch E-passport).
- Loyalty cards.
- Health care cards (e.g. the French *Sesam Vitale* and the German *Versichertenkarte*).
- Public phone and public transport payment cards.
- Etcetera.

The main advantages of smart cards are:

- In contrast to magnetic stripe cards or memory-only chip cards, smart cards actually do processing. Furthermore, smart cards protect the programs/information they contain, so they cannot (easily) be read, altered, etc.
- One smart card can house multiple applications, so just one card can be used as credit card, driving license, ID card, etc.

5.2 Java Card technology

5.2.1 Definition

Java Card technology defines "a platform on which applications written in the Java programming language can run in smart cards and other memory-constrained devices"[37]. In a few words, Java Card is a dialect of Java for programming smart cards.

The Java Card platform consists of three parts:

- The Java Card Virtual Machine (JCVM).
- The Java Card Runtime Environment (JCRE).
- The Java Card Application Programming Interface (API) [22].

5.2.2 Java Card language subset

The Java Card programs, commonly named *applets*, are written in the Java programming language. However, due to smart card's resources constraints, the Java Card platform supports only a subset of the features of the Java language. Some of the unsupported features are dynamic class loading, threads, cloning, multidimensional arrays, some API classes from the Java core or types such as `double`, `float` or `long`. See [37] for a detailed description of the unsupported features.

5.2.3 Java Card Memory Model

As said in section 5.1.3, smart cards have three main kinds of memory: ROM, RAM and EEPROM. Permanent programs and data (SCOS, JCVM, API classes, etc.) are burned into ROM during card manufacture. Both RAM and EEPROM contents are not permanent and are often erased and rewritten. However, RAM and EEPROM differ in many characteristics, the main are:

- When power is switched off, EEPROM preserves its data, but RAM does not.
- Writes to EEPROM are approximately a thousand times slower than writes to RAM.

So, on one hand **EEPROM** its used to store **persistent** programs/data (downloaded applets, objects with their fields, etc.). On the other hand, **RAM** is used for **transient** storage of data (stack, "scratchpad memory", etc.).

5.2.4 Persistent and transient objects

By default, Java Card objects are persistent and are created in persistent memory (EEPROM). That is, whenever the `new` operand is used, a new persistent object is automatically allocated in EEPROM.

By contrast, there exists some objects which are accessed frequently and whose contents need not to be persistent. This kind of transient objects is allocated in RAM and is created by invoking a Java Card API method.

Listing 5.1 depicts a piece of code where both a persistent and a transient object are created using their respective appropriate method.

```java
package example;
import javacard.framework.*;
public class ExampleApplet extends Applet{
  // Class fields
  Object [] persitentArray, transientArray;

  // Constructor
  public ExampleApplet (){
    /* persistentArray is persistent since the new operand is used to
        create it */
    persistentArray = new Object[10];

    /* transientArray is transient since the appropriate API method is
        used to create it */
    transientArray = JCSystem.makeTransientObjectArray((short)10,
                                                      JCSystem.CLEAR_ON_RESET);
  }

  ...

}
```

Listing 5.1: Examples of persistent and transient objects creation.

Properties of persistent objects

The most important properties of persistent objects are [37]:

- A persistent object is always created by the `new` operand.
- The memory and data of persistent objects are preserved across CAD sessions [12].
- A persistent object can reference persistent and/or transient objects.
- A persistent object can be referenced by persistent and/or transient objects.

[12]A CAD session is defined as the period from the time the card is inserted into the CAD and is switched on until the time the card is removed from the CAD.

Properties of transient objects

The most important properties of transient objects are [37]:

- A transient object is always an array with elements of either the type `Object` or a primitive type, where the primitive types are `byte`, `short` and `boolean`.
- A transient object is always created by the appropriate Java Card API method available at the class `JCSystem` (see [22] for more details). Table 5.1 summarizes these methods.
- The memory and data of transient objects are **not** preserved across CAD sessions.
- A transient object can reference persistent and/or transient objects.
- A transient object can be referenced by persistent and/or transient objects.
- Compared to persistent objects, writes to the fields of transient objects are much faster.

Method	**Returns**
`public static boolean[]` `makeTransientBooleanArray(short length, byte event)`	A transient `boolean` array
`public static byte[]` `makeTransientByteArray(short length, byte event)`	A transient `byte` array
`public static short[]` `makeTransientShortArray(short length, byte event)`	A transient `short` array
`public static Object[]` `makeTransientObjectArray(short length, byte event)`	A transient `Object` array

Table 5.1: Java Card API methods for creating transient arrays.

5.2.5 Atomic operations and transactions

A write to a persistent object (e.g. a class field, an array element, etc.) is **atomic** when it is ensured either completing it successfully or else restoring its previous value. That is, if a sudden loss of power, the so-called *card tear*, happens during an assignment to a persistent object, then its original value will be restored.

The method `Util.arrayCopy` is an example of an atomic operation, while `Util.arrayCopyNonAtomic` and `Util.arrayFillNonAtomic` are non-atomic functions [13]. In section 5.2.6, we will analyze why these three methods can imply certain hazards.

[13]See the Java Card API [22] for a detailed description of these methods.

In addition to atomicity, which is only applicable to single data elements, Java Card provides the **transaction** mechanism. This mechanism ensures that either all writes in a block complete successfully or none of them proceeds.

The API methods `JCSystem.beginTransaction()`, `JCSystem.commitTransaction` and `JCSystem.abortTransaction` make possible joining several assignments into one atomic action.

5.2.6 Problem description

Unfortunately, some features of the Java Card platform, if incorrectly used, can lead to security holes. One of these holes consists in the possibility of updating a persistent object, typically an array, by using a non-atomic method. If a sudden loss of power, i.e. a card tear, happens while updating the persistent array, then its actual value will be unknown. This defect can mean a really serious hazard if that object is a crucial data structure, such as the PIN code of a credit card.

This situation is well illustrated in Listing 5.2. Statement in line 19 tries to copy a new pin code, typed by the user, over the original. If a card tear occurs during the execution of the method `Util.arrayCopyNonAtomic`, then the user will ignore which his pin code is (the original?, the new?, a mix of both?). The reader will notice easily this kind of bugs is strongly not recommended.

```
 1 package myInsecureApp;
 2 import javacard.framework.*;
 3 public class MyInsecureApp extends Applet{
 4   // Class field: pin code
 5   byte [] pin;
 6
 7   // Constructor
 8   public CreditApp (){
 9     // pin is persistent (operand new used)
10     pin = new byte[4];
11   }
12
13   // Method that changes the pin code
14   public void changePinCode(){
15     // Read the new pin typed by the user
16     byte[] newPin = readNewPin();
17
18     // Establish the new pin code (insecurely)
19     Util.arrayCopyNonAtomic(newPin, (short)0, pin, (short)0,
20                                  (short)4);
21   }
22   ...
23 }
```

Listing 5.2: Use of Util.arrayCopyNonAtomic on a persistent object.

Similarly, the method `Util.arrayFillNonAtomic` non-atomically fills the elements of a byte array with a specified value. So, for the reasons just commented, this method should not write on persistent arrays either.

Apart from using non-atomic methods to update persistent arrays, there is another issue to take into account: using atomic operations (originally designed for EEPROM) to update transient arrays. For instance, suppose the method `Util.arrayCopy` is used to update a transient array: it means another defect, not in terms of security, but in terms of efficiency [14].

Obviously, the defects related to security issues are much more dangerous than those which have to do with efficiency.

5.2.7 Special methods

By *special methods* we mean those from the Java Card API whose use can cause bugs. These are:

- `Util.arrayCopyNonAtomic`, `Util.arrayFillNonAtomic` and `Util.arrayCopy`. These methods have just been explained in section 5.2.6.
- `JCSystem.makeTransient{Object,Boolean,Short,Byte}array`. See table 5.1, on page 47, for more details.
- Array declarations using the `new` operand.
- `APDU.getBuffer`. It returns a reference to an APDU buffer byte array, which must be stored only in a transient object. Furthermore, another issue must be taken into account: “References to the APDU buffer byte array cannot be stored in class variables or instance variables or array components” [22].
- `JCSytem.isTransient`. It checks whether a specified object is transient or not. However, if it is ensured that the argument is either transient or persistent, then it does not make sense asking about it.

[14] `Util.arrayCopy` is slow for transient arrays because of the overhead caused by its atomicity mechanism, which may involve the use of EEPROM.

5.3 Detector implementation

Goal:

Using static analysis techniques again, implement a custom detector in charge of analyzing Java Card programs and looking for pieces of code containing the following defects:

- *Related to security, calls to the methods Util.arrayCopyNonAtomic and/or Util.arrayFillNonAtomic using a persistent array as destination parameter. This defect is quite dangerous, so it has a high priority.*
- *Related to efficiency, calls to the method Util.arrayCopy using a transient array as destination parameter. This defect is not as hazardous as the first one, so it has a medium priority.*

5.3.1 Strategy

The strategy we propose in this thesis consists in *annotating* the source code somehow. The main reason for using such strategy is because it offers us the possibility of determining if a certain operation can be applied to a certain object, by examining its annotations.

For our purposes, we only need to annotate some arrays. Afterwards, whenever an array is referenced, our detector will determine whether a bug occurs or not, by checking its tags and the place where it appears.

In order to annotate arrays, the Java annotations mechanism [19] has been chosen because of its usefulness and simplicity.

Java annotations mechanism

Basically, “a Java annotation is a way of adding metadata to Java source code that is available to the programmer at run-time. (...) Java annotations can be added to program elements such as classes, methods, fields, parameters, local variables, packages and even other annotations.” [33] This feature is available since JDK version 1.5.

The declaration of an annotation looks like an interface declaration, but it is preceded by an at sign (@). Optionally, an annotation can be marked with other meta-annotations. Finally, custom annotations can include their own parameters [15].

Listing 5.3 shows the custom Java annotation we need to mark persistent arrays. Notice how simple this declaration is.

[15] Further information about Java annotations is available at [11] and [19].

```
package packageAnnotations;

public interface @Persistent{}
```

Listing 5.3: Persistent Java annotation (Persistent.java)

For our purposes, four custom Java annotations are defined [16]:

- First level annotations: used to annotate `Object` arrays and arrays of primitive types, where the primitive types are `boolean`, `short` and `byte`. Two first level annotations can be distinguished:

    1. Persistent: used to annotate persistent arrays.
    2. Transient: used to mark transient arrays.

- Second level annotations: used to annotate `Object` arrays whose elements are arrays. NB an `Object` array can be marked with a second level annotation if and only if it is already been marked with a first level annotation. Two second level annotations are defined:

    3. ElemsPersistent: used to annotate `Object` arrays whose elements are persistent arrays.
    4. ElemsTransient: used to mark `Object` arrays whose elements are transient arrays.

At this point, the reader may wonder why no more levels have been implemented, this question is quite easy to answer. Examining the source code of real Java Card applications, we have noticed that even arrays with two dimensions are rarely used. Thus, multidimensional arrays are so uncommon, that deeping into more levels is worthless.

Listing 5.5 illustrates the way of marking class fields. Similar annotations can be applied to method parameters, method results, etc.

```
package myApp;
import javacard.framework.*;
public class MyApp extends Applet{
  // 1st level annotated class fields:
  @Transient Object[] tObjArray;
  @Persistent byte[] pByteArray;
  @Transient short[] tShortArray;
  @Persistent boolean[] pBooleanArray;

  // 2nd level annotated class fields:
  // Persistent array with Persistent elements:
  @Persistent @ElemsPersistent Object[] pEpObjArray;

```

[16]They are all declared equally to the way shown in Listing 5.3

```
  // Persistent array with Transient elements:
  @Persistent @ElemsTransient Object[] pEtObjArray;

  // Transient array with Persistent elements:
  @Transient @ElemsPersistent Object[] tEpObjArray;

  // Transient array with Transient elements:
  @Transient @ElemsTransient Object[] tEtObjArray;

  ...
}
```

Listing 5.4: Custom annotations examples.

As was said before, our detector will use these annotations in order to determine whether a defect exists or not. Listing 5.5 contains several interesting examples.

```
package examples;
import javacard.framework.*;
public class ExampleApp extends Applet{
  // Annotated class fields:
  @Transient byte[] transientArray1, transientArray2;
  @Persistent byte[] perstArray1, perstArray2;

  // Constructor: initializes class fields:
  public ExampleApp (){
    // Ok, no bugs because transientArray1 is @Transient
    transientArray1 = JCSystem.makeTransientObjectArray((short)10,
                                              JCSystem.CLEAR_ON_RESET);

    // Ok, no bugs because perstArray1 is @Persistent
    perstArray1 = new byte[10];

    // Bug during initialization, transientArray2 must be @Persistent
    transientArray2 = new byte[10];

    // Bug during initialization, perstArray2 must be @Transient
    perstArray2 = JCSystem.makeTransientObjectArray((short)10,
                                          JCSystem.CLEAR_ON_RESET);
  }

  ...

  // Method containing several bugs:
  public void method(){
    byte[] src = new byte[10];
    short srcOff = 0;
    short dstOff = 0;
    short length = 10;

    //No bugs, destination is @Transient
    Util.arrayCopyNonAtomic(src, srcOff, transientArray1,dstOff,length);

    //High priority bug (security). Destination is @Persistent
    Util.arrayCopyNonAtomic(src, srcOff, perstArray1, dstOff, length);

    //Medium priority bug (efficiency). Destination is @Transient
    Util.arrayCopy(src, srcOff, transientArray1, dstOff, length);

    //No bugs, destination is @Persistent
    Util.arrayCopy(src, srcOff, perstArray1, dstOff, length);
  }
}
```

Listing 5.5: Custom annotations examples.

5.3.2 Detector implementation

As seen in section 4.2.4, FindBugs supports two sorts of bug detectors: Visitor-based and CFG-based. The second detector, as the first one, is based on the Visitor design pattern and extends the class `BytecodeScanningDetector`. In this case, the following methods are overridden:

- `public void visit(Method obj)`: As we already know, `visit` is called whenever a new method within the source code is found. Our detector uses this method to restart the stack.

- `public void visitAnnotation(String annotationClass, Map <String, Object> map, boolean runtimeVisible)`: FindBugs calls this method whenever it detects an annotation on either a class field or a method or a class. Our detector uses this method to collect which annotations have been done to which data.

- `public void visitParameterAnnotation(int p, String annotationClass, Map <String, Object> map, boolean runtimeVisible)`: this method is invoked whenever an annotation on a method parameter is detected. Therefore, this method compiles which annotations have been done to which method parameters.

- `public void sawOpcode(int seen)`: this method is called whenever a new opcode is being analyzed within a method body. In this case, it stores pieces of code which may contain bugs. For instance, whenever an `Util.arrayCopy` call is detected, this method stores it together with its destination parameter (taken from the stack). At the end of the analysis, the auxiliary methods `checkSpecialMethodsCalls` and `checkNonSpecialMethodsCalls`, which are explained in detail on page 54, determine if each method call contained in the source code has any bugs.

- `public void report()`: As seen in section 4.2.4, this method is called at the end of the analysis. It invokes the auxiliary methods `checkSpecialMethodsCalls` and `checkNonSpecialMethodsCalls`. Finally, `report` notifies all the previously stored bug instances.

Apart from the overridden methods, implementing our own auxiliary methods has been necessary as well. The most important are:

- `private void checkSpecialMethodsCalls():` analyzes each call to a special method together with its relevant argument [17] and determines whether a bug occurs or not. Every found bug is stored in a data structure and later notified by `report`.

- `private void checkNonSpecialMethodsCalls():` analyzes each call to an annotated method (see Listing 5.6 for some examples) together with its arguments and determines if these follow the marks made on the method parameters. Again, every bug occurrence is stored in a bugs list and later notified by `report`.

```java
import javacard.framework.*;
public class AppWithAnnotatedMethods extends Applet{

  // Annotations on method parameters:
  public void methWithAnnotatedParams1(@Transient byte[] transientParam,
                                       @Persistent byte[] perstParam){
    // Ok, no bugs because perstParam is @Persistent:
    perstParam = new byte[10];

    // High priority bug, transientParam must be @Persistent:
    transientParam = new byte[10];
  }

  // Annotations on method parameters:
  public void methWithAnnotatedParams2(@Transient byte[] transientDst,
                                       @Persistent byte[] perstDst,
                                       byte [] src, short len,
                                       short srcOff, short dstOff){
    // Ok, destination parameter is @Transient:
    Util.arrayCopyNonAtomic(src, srcOff, transientDst, dstOff, len);

    // High priority bug, perstDst must be @Transient:
    Util.arrayCopyNonAtomic(src, srcOff, perstDst, dstOff, len);
  }
  ...
}
```

Listing 5.6: Examples of annotations on method parameters.

[17] For instance, for the method `Util.arrayCopyNonAtomic`, its relevant argument is the destination parameter. Every special method has one and only one relevant argument.

High-level Algorithm

This section provides an abstract vision of the algorithm our detector follows (See listing 5.7). The following section (Low-level Algorithm) describes the concrete version.

In the high-level algorithm, there are several issues which must be taken into consideration. These are:

- For each analyzed application, a set of data structures is needed. Basically, each of these structures is a list containing different relevant information.
- For each analyzed application, steps A, B and C are executed in an interlaced way, depending on the order the different elements (e.g. annotations, opcodes, etc.) are seen.
- For each analyzed application, steps D, E and F are executed sequentially at the end of the analysis.

After reading the high-level algorithm, the reader may wonder why we check the method calls at the end of the analysis, instead of as soon as each one is seen. The reason is because this strategy ensures that all annotations on method parameters have been detected (and stored) before analyzing each method call.

By contrast, calls to special methods might be checked as soon as they are detected. However, we decided to treat them similarly in order to get an unchanging algorithm.

```
// For the whole analyzed application, the following data structures are needed
listBugs: list of bugs (Initially empty)
lAnnotData: list of annotated data (Initially empty)
lMethAnnots: list of annotations on method parameters (Initially empty)
lSpecialMethsCalls: list of calls to special methods (Initially empty)
lNonSpecialMethsCalls: list of calls to non-special methods (Initially empty)


// For the whole analyzed application: steps A, B and C are executed in an
// interlaced way, depending on the order the different elements are seen.

(A) For each seen annotation on either a class field or a method Do
      Store the element together with its annotation in lAnnotData

(B) For each seen annotation on a method parameter Do
      Store the method parameter together with its annotation in lAnnotData
      Store the method parameter together with its annotation and position in
         lMethAnnots

(C) For each seen Opcode O Do
      If (a special method is called) Then
        Store the current call together with its relevant argument in
           lSpecialMethsCalls
      Else If (a non-special method is called) Then
        Store the current call together with all its arguments in
           lNonSpecialMethsCalls


// For the whole analyzed application: steps D, E and F are executed
// sequentially at the end of the analysis

(D) For each special method call in lSpecialMeths Do
      argAnnots = relevant argument's annotations from lAnnotData
      If (argAnnots does not follow the special method's constraint) Then
        Store a new bug instance in listBugs

(E) For each non-special method call in lNonSpecialMethsCalls Do
      If the current non-special method has been annotated Then
        For each position P in the call Do
          argAnnots = current argument's annotations from lAnnotData
          methAnnotsP = method's annotations for position P from lMethAnnots
          If (argAnnots is not contained in methAnnotsP) Then
            Store a new bug instance in listBugs

(F) For each BugInstance bi in listBugs Do
      Notify bi
```

Listing 5.7: High-level algorithm.

Low-level Algorithm

This section explains the algorithm followed by our detector in detail. Based on the high-level algorithm, it will guide us to the detector implementation, which is explained in the next section.

1. Whenever an annotation on either a class field or a method is seen, the method `visitAnnotation` is invoked. This method is intended to:

    - Check if the current annotation is impossible (e.g. @Transient boolean a). If so, store a new bug occurrence in a global bugs list and end its execution.
    - Store the element (class field or method) together with the seen annotation in a global list of annotated data. For each element, a list with all its annotations is needed.

    This step corresponds to step A in the high-level algorithm (Listing 5.7).

2. Whenever an annotation on a method parameter is seen, the method `visitParameterAnnotation` is invoked. This method is intended to:

    - Check if the current annotation is impossible (e.g. void method(@Transient @Persistent byte[] b)). If so, store a new bug occurrence in the list of bugs and end its execution.
    - Store the element, in this case a method parameter, together with its annotation in the list of annotated data.
    - For the current method position, store its annotation in a global list of annotated method parameters. For each method, a list with all the annotations marking each position is needed.

    This step matches with step B in the high-level algorithm (Listing 5.7).

3. Whenever a new method is being analyzed, the detector invokes the method `visitMethod`. Our detector uses this method to restart the stack data structure.

4. Within a method body, the method `sawOpcode` is invoked for each seen opcode. Whenever a method call is found:

    If the invoked method is special, then its relevant argument is taken from the stack and stored together with the current call in a global list of calls to special methods. For this purpose, each special method needs a list containing all its calls and their corresponding relevant argument.

    If the invoked method is non-special, then all its arguments are taken from the stack and stored together with the current call in a global list of calls to non-special methods. For this purpose, each non-special method needs a list containing all its calls and their corresponding arguments.

    This step corresponds to step C in the high-level algorithm (Listing 5.7).

5. Once the source code is completely analyzed, the detector invokes the method `report`. However, before reporting anything, this method is used to:

   - Check if there are bugs related to special methods calls. In order to do so the private method `checkSpecialMethodsCalls` is invoked. This method scans the list of calls to special methods. For each special method and for each of its calls, if the relevant argument does not follow the method's constraint, then a new bug instance is stored in the global bugs list. This step matches with step D in the high-level algorithm (Listing 5.7).
   - Check if there are bugs related to non-special methods calls. In order to do so the private method `checkNonSpecialMethodsCalls` is invoked. This method scans the list of calls to non-special methods. For each non-special method and for each of its calls, every argument is checked: if it has not been marked with the annotation corresponding to its position, then a new bug is added to the bugs list. This step corresponds to step E in the high-level algorithm (Listing 5.7).

   As bugs can be stored from different parts of the detector, maybe they will be not stored in order. To solve this problem, bugs are ordered using an implementation of the Mergesort algorithm [34]. Finally, scan the list of ordered bugs and report them all (step F in the high-level algorithm, Listing 5.7).

Implementation

This section summarizes the detector implementation for the concrete algorithm just explained. The detector is composed of three classes:

- `JavaCardAppsDetector`: main class, Listing A.11, on page 85 in the Appendix, shows its commented source code.
- `CallerInfoAndACall`: auxiliary class. It groups:
  - Information (class name, method name and line number) related to the place where a method call appears.
  - List of passed arguments.
- `PositionAnnotation`: auxiliary class. It groups an annotation name together with a position (number) in a method call.

Apart from the main class, whose commented source code is listed in A.11, the two auxiliary classes are:

- `CallerInfoAndACall`: groups a class name, a method name and a line number together with a list of arguments (call). The main class uses it to store both where a method call appears and the list of passed arguments. Its main methods are:
  - `public CallerInfoAndACall (ArrayList ci, ArrayList c).`
  - `public void setCallerInfo (ArrayList ci).`

  - Equivalent set method for Call.
  - `public ArrayList getCallerInfo()`.
  - Equivalent get method for Call.
  - `public String toString()`.

- `PositionAnnotation`: groups an annotation name together with a position (number) in a method call. The main class uses it when an annotation on a method parameter has been detected. Its main methods are:
  - `public PositionAnnotation(int p, String annotClass)`.
  - `public void setPosition (int p)`.
  - Equivalent set for Annotation.
  - `public int getPosition()`.
  - Equivalent get for Annotation.
  - `public String toString()`.

Detector JAR package

As seen in 4.2.4, after implementing a detector, the next step is packaging it as a JAR file so that FindBugs can recognize it as a plugin. The only relevant differences between the package for the second detector and the one for the first plugin are:

- Obviously the class files which implement the detector have changed. Now these are:
  - `JavaCardAppsDetector.class`
  - `CallerInfoAndACall.class`
  - `PositionAnnotation.class`
- The description which explains the use of the detector has changed too.

The rest of the considerations made in Section 4.2.4 can be applied equally well to this detector, so we will skip over them.

Design decisions: classifying bugs according to their priority

As said in Section 5.2.6, not all the bugs are equally dangerous. Remember that those related to security issues are more hazardous than those which deal with efficiency. Therefore, a classification according to how dangerous each bug is must be made. This classification consists of three categories:

- **High-priority bugs:** these are really dangerous defects that must be fixed immediately since they can cause an application to misbehave. Within this category, the most serious bugs are those which have to do with security issues. Some examples of high-priority bugs:

  - Use of non-atomic operations to write on persistent data structures:
    * Use of `Util.arrayCopyNonAtomic` to write on a persistent byte array.
    * Use of `Util.arrayFillNonAtomic` to write on a persistent byte array.
  - Incorrect data assignment: assigning a persistent reference to a @Transient object, or vice versa. Possibilities:
    * Use of `JCSystem.makeTransient*Array` to assign an array marked as @Persistent.
    * Use of the `new` operand to assign an array marked as @Transient.
    * Use of `APDU.getBuffer` to assign an array marked as @Persistent.
  - Incorrect use of some Java Card API functions:
    * Use of `APDU.getBuffer` to write on a class field or an array element, since it throws a runtime exception.
  - Incorrect or contradictory use of our annotations:
    * Use of our custom annotations to mark elements which are not arrays.
    * Use of the second level annotations to mark arrays which have not been marked with a first level annotation.
    * Contradictory annotations (e.g. `@Transient @Persistent Object[] o`).
  - Missing annotations:
    * Use of `Util.arrayCopyNonAtomic` to write on a non-annotated byte array.
    * Use of `Util.arrayCopy` to write on a non-annotated byte array.
    * Etcetera.
  - Calls to non-special methods which do not meet the annotations made on the method parameters.

- **Normal-priority bugs:** these are defects that should be fixed, but which do not put an application at risk. That is, this sort of bugs can make an application to be slower, but not to misbehave. For that reason, they have to do more with efficiency than with security. For instance:
  - Use of atomic (slow) operations to write on transient data structures:
    * Use of `Util.arrayCopy` to write on a transient byte array.

- **Low-priority bugs:** these are slight defects that should be fixed in order to meet some recommended programming techniques. For instance:
  - In certain cases, not recommended API method call:
    * Use of `JCSystem.isTransient` to ask whether an already annotated object is transient or not. If the object has already been annotated as either @Persistent or @Transient, then it does not make sense asking about it.

- In certain cases, not recommended way of using memory:
  - Declare an `Object` array as @Transient @ElemsPersistent. Remember that transient objects are not preserved across CAD sessions. Therefore, this transient array is deleted whenever the power is switched off, but its elements are not. This situation may cause having several persistent elements in EEPROM which are not referenced by any other object. As garbage collection is only optional in Java Card, this sort of declaration may lead to a waste of EEPROM space.

5.4 Tests

As seen in 4.3, once the detector is implemented, the next step is testing it, so that we can check if it works properly, throws any *false positives* or *false negatives*, etc. In order to do so, we will analyze some Java Card programs (custom and real applications) and examine the obtained results.

5.4.1 Custom test

The custom application we developed for testing the detector contains a wide range of possibilities. Listing A.12, on page 88 in the Appendix, contains some of these possibilities. However, for simplicity reasons, many other combinations have been omitted.

5.4.2 Testing the detector with real Java Card applications

As we know, the most interesting tests are those which deal with real applications, in this case, with real Java Card applications. In this section we discuss about the results our detector obtains when it analyzes two real Java Card programs.

Testing the electronic purse included in PACAP

The PACAP Project [9] provides techniques and tools that enable a smart card issuer to verify that a new applet securely interacts with already downloaded applets. For our purposes, we will analyze an electronic purse implemented within this project. This Java Card application is quite large since it consists of forty-two classes (some of them longer than two thousand lines). For that reason, this test is quite interesting.

Before annotating the source code, our detector finds 214 bugs in it. Obviously, this is only an anecdotal result since without annotations our detector does not make sense.

After annotating the source code [18], our detector finds 9 bugs in it. These are:

- **High-priority bugs (7):** use of non-atomic operations to write on persistent arrays. Listing 5.8 depicts some examples.

[18]Annotating such a large application only took me one hour.

```
...
//Declarations:
@Persistent private byte[] terminalTC = new byte[TTC_LEN];
@Persistent private byte[] terminalSN = new byte[TSN_LEN];
@Persistent private byte[] id = new byte[ID_LEN];
...
//High-priority Bugs:
Util.arrayCopyNonAtomic(bArray,off, terminalTC, (short)0, TTC_LEN);
...
Util.arrayCopyNonAtomic(bArray,off, terminalSN, (short)0, TSN_LEN);
...
Util.arrayCopyNonAtomic(bArray, off, id, (short)0, ID_LEN);
...
Util.arrayFillNonAtomic(terminalTC, (short)0, TTC_LEN, (byte)0);
...
Util.arrayFillNonAtomic(terminalSN, (short)0, TSN_LEN, (byte)0);
...
```

Listing 5.8: High-priority bugs found in ExchangeSession.java

- **Normal-priority bugs (2):** use of atomic (slow) operations to write on transient arrays. Listing 5.9 shows one example.

```
...
//Declaration:
@Transient private byte temp[] =
  JCSystem.makeTransientByteArray((short)80,
                                  JCSystem.CLEAR_ON_DESELECT);
...
//Normal-priority Bug:
offset =
  Util.arrayCopy(bankCert, (short)0, temp, (short)0, (short)15);
...
```

Listing 5.9: Normal-priority bug found in Security.java

As we have seen throughout this chapter, these sorts of bugs do mean serious hazards, specially those with high-priority. Therefore, they should be fixed immediately. Otherwise, the application will not meet the desired security requirements.

Actually, PACAP is low priority code written by beginning Java Card programmers, so finding such dangerous defects is not as surprising as could be thought. In addition, it is important to emphasize that this test illustrates the usefulness of our plugin.

Testing Demoney

Demoney [32], as the PACAP application, implements an electronic purse for smart cards. This application is not large (only 9 classes), although its main class is longer than three thousand lines. For that reason, Demoney is another interesting program to test.

Before annotating the application, our detector finds only thirty-six bugs in it. However, as was said before, this is not a very relevant result because our annotations have not been used yet.

After annotating the source code, our plugin finds only three bugs in it. These are:

- **Normal-priority bugs (2):** use of atomic (slow) operations to write on transient arrays. Two cases can be distinguished:
  - A justified bug: `Util.arrayCopy` updates a transient array within a transaction block. In this specific situation, the use of `Util.arrayCopyNonAtomic` is insecure. See Listing 5.10 for more details.

```
...
public void process(APDU apdu){
  apduBuffer = adpu.getBuffer(); // Transient by default
  ...
  performTransaction(transAmount, apduBuffer, TRANSACTION_OFF);
  ...
}

...

private void performTransaction(short amount,
                                @Transient byte[] apduBuffer,
                                short offsetTransCtx){
  // Begin transaction block
  JCSystem.beginTransaction();
  ...
  // Util.arrayCopyNonAtomic is insecure here
  Util.arrayCopy(apduBuffer, offsetTransCtx, apduBuffer,
                 (short)0, LEN_TRANS_CTX);
  ...
  // End transaction block
  JCSystem.commitTransaction();
}
...
```

Listing 5.10: Justified efficiency bug found in Demoney.java

  - A non-justified bug: `Util.arrayCopy` updates a transient array, but no transaction block appears here. Listing 5.11 depicts this defect.

```
...
public void process(APDU apdu){
  apduBuffer = adpu.getBuffer(); // Transient by default
  ...
  if(!readRecordData(recordNumber, OFF_0, apduBuffer) ){
    ...
  }
}

...

public boolean readRecordData(short recNum, short offset,
                              @Transient byte[] buffer){
  ...
  Util.arrayCopy((byte[]) records[index], (short)0, buffer,
     offset, recordLength);
  ...
}
...
```

Listing 5.11: Non-justified efficiency bug found in CyclicFile.java

- ***False positive* (1):** Listing 5.12 depicts a method call which includes an element of a second level annotated array, followed by some loaded variables.

```
...
//Declaration
@Persistent @ElemsPersistent private Object[] records;
...
//False positive:
  /*The detector issues a warning here, but records is
     @ElemsPersistent, so this means a false positive*/
Util.arrayCopy(buffer, offset, (byte[]) records[nextRecordIndex],
   (short)0, recordLength);
...
```

Listing 5.12: False *positive* issued in CyclicFile.java

Section 5.5.2 (in Limitations) explains this defect in detail.

In contrast to PACAP, the sort of bugs detected in Demoney is related to efficiency, not to security. This shows the higher quality of Demoney, which was written by experienced Java Card programmers.

Finally, it is important to emphasize that no false *negatives* were found in any of the tested applications.

5.5 Limitations

In contrast to the first detector, see Section 4.4, only two drawbacks have been found in this plug-in.

5.5.1 First limitation: impossibility of accessing annotations on local variables

Nowadays, annotations on local variables are not stored into the Java bytecodes. Therefore FindBugs, and consequently our detector, cannot get this information. For our purposes, this means that our detector cannot take advantage of the annotations made on local variables.

Exception

Examining the source code of some real Java Card applications, we noticed that the method `APDU.getBuffer` often stores its result into one or more local variables. For that reason, we decided to implement a special treatment for such writes. Specifically, the following convention was chosen:

> *If the reference returned by* `APDU.getBuffer` *is stored into a local variable, then that local variable is marked implicitly as @Transient.*

Listing 5.13 illustrates one example.

```
...
public void process (APDU apdu){
  //Local variable: apduBuffer is @Transient implicitly
  byte[] apduBuffer = apdu.getBuffer();
  short sh = 0;
  byte b = 0;

  // Bug: apduBuffer is @Transient implicitly
  arrayCopy(new byte[0], sh, apduBuffer, sh, sh);

  // No Bugs: apduBuffer is @Transient implicitly
  arrayCopyNonAtomic(new byte[0], sh, apduBuffer, sh, sh);
  arrayFillNonAtomic(apduBuffer, sh, sh, b);
}
...
```

Listing 5.13: Marking the result of `APDU.getBuffer` implicitly as @Transient

5.5.2 Second limitation: false *positive*

As seen in Listing 5.12, our detector issues a *false positive* when it finds certain method calls. Specifically, our detector will issue a *false positive* when:

- A method parameter has been marked with one of our custom annotations.

- An element of a two-level annotated array occupies the marked parameter's position within a method call.
- After it, one or more variables are loaded within the same method call.

This defect appears because the loaded variables impede our detector to see the array element. If the method call does not include any variables after loading the array element, then no *false positive* is issued. Listing 5.14 illustrates one example.

```java
package limitation;
import javacard.framework;
public class AppWithLimitation extends Applet{
  @Persistent @ElemsTransient Object array[];
  short sh, len;
  byte by;

  public AppWithLimitation(){
    sh = 0;
    len = 2;
    by = JCSystem.CLEAR_ON_DESELECT;
    //array is @Persistent @ElemsTransient
    array = new Object[2];
    array[0] = JCSystem.makeTransientByteArray(sh, by);
  }

  public void method(){
    // No false positive here (no loaded variables after array[0])
    Util.arrayCopyNonAtomic(new byte[2], sh, array[0], (short)0, (short)
        2);

    // False positive here (loaded variables after array[0])
    Util.arrayCopyNonAtomic(new byte[2], sh, array[0], sh, (short)len);
  }
  ...
}
```

Listing 5.14: Limitation 2 (false *positive*)

5.6 Conclusions

This section summarizes this chapter and emphasizes the most relevant issues. These are:

- Smart cards are miniature computers with very restricted I/O and very limited resources.
- The Java Card technology is a restrictive dialect of Java for programming smart cards.

- In the Java Card platform, memory consists of ROM, EEPROM (persistent data which must be preserved across CAD sessions) and RAM (transient data which can be cleared when power is switched off).
- By default, fields of Java Card objects are stored in EEPROM. However, the API provides methods to create fields, which are arrays, and allocate them in RAM.
- In order to write on arrays, the API includes several methods whose atomicity is not guaranteed. That is, if a power loss (*card tear*) occurs during the update, then the actual data stored in the array is unknown.
- This defect means a serious hazard when non-atomic operations are used to update persistent arrays.
- To solve this problem, we have developed a FindBugs plugin which, using a set of custom Java annotations, analyzes Java Card applications statically and looks for such defects.
- In regard to our custom Java annotations, they do record important design decisions/properties as well as helping programmers in order not to introduce defects in their applications.
- To check our plugin's behaviour, we have used two real Java Card applications [19]:
  - In PACAP, our detector finds several dangerous bugs (e.g. use of `Util.arrayCopyNonAtomic` to update persistent arrays), as well as other defects related to efficiency (e.g. use of `Util.arrayCopy` to write on transient arrays). As PACAP is low quality code written by beginning Java Card programmers, finding such defects is not surprising. However, it does illustrate the usefulness of our plugin.
  - As Demoney is high quality code written by experienced Java Card programmers, only two efficiency bugs (one of them justified for security reasons) are found in it. Such defects are hidden behind nested method calls. This shows:
    - The reason why annotations on method parameters are needed.
    - The usefulness of using these detectors since they easily keep track of what kind of object is used at any time.
- Finally, we have found only two limitations:
  - Our detector cannot take advantage of the annotations made on local variables since they are not stored into the Java bytecodes.
  - Our detector issues a *false positive* when it finds an uncommon kind of method call: an element of a two-level annotated array occupying an annotated position, followed by one or more loaded variables.

Some possible enhancements for this plugin are discussed in section 6.2 (chapter 6: Conclusions and future work).

[19] Before testing an application, its source code must be marked using our custom annotations. Annotating two large applications such as PACAP and Demoney only took me one hour for the former and forty-five minutes for the latter.

6 Conclusions and future work

6.1 Conclusions

The main conclusions of our research are:

- In order to get higher quality software, many techniques have been developed over the years. One of the most used, the so-called code reviews, consists in analyzing the source code of an application by hand. However, this technique has a very serious disadvantage: its costs are usually too high (in terms of time and/or money).

- To make the costs of code reviews lower, several automatic techniques have been proposed. Specifically, we have dealt with static analysis techniques, which analyze a whole application without actually executing it. Within these static strategies, we have worked with a source code analyzer called FindBugs. It is important to emphasize that automatic techniques are designed to help code reviews, but not to replace them.

- As we have seen, FindBugs is quite easy to extend because of its plugin oriented architecture. However, it has several drawbacks:

  - There is a serious lack of documentation about how to write custom detectors. It does not exist any updated step by step manual even.
  - The FindBugs API is not well documented either.
  - The only possibility to learn how FindBugs and its detectors work is by browsing their respective source code and applying trial and error.
  - For my research, Graat's master thesis [13] has been the most useful source of information, even more than official documentation.

- Before implementing any custom detector, one has to overcome the difficulty of learning how FindBugs and its plugins work, which is not a simple task. As Graat says in his research, “FindBugs is quite easy to extend (...) but the learning curve of detector writing is very steep”. In my case, I spent two weeks browsing the source code of existing detectors, experimenting with very simple programs, etc.

- As to the question of how much time does it take to write a custom detector in FindBugs, it cannot be answered easily since it depends enormously on a wide range of factors, such as the problem complexity, the programmer’s experience with detector writing, etc. To give an idea, it took me one month to write the first detector, while I spent one month and a half writing the second.

- Despite the implementation of the second plugin was more time-consuming (basically due to the effort needed to learn the Java Card platform and its constraints, the Java annotations mechanism, etc.), I did appreciate a clear improvement in my experience with detector writing. That may be the reason why the second detector has less limitations. Thus, it can be said that the more one writes custom detectors, the better/quicker he does.

- In regard to the second detector, it is important to emphasize that Java Card applications can normally be annotated effortlessly. To give an idea, annotating such a large application as Pacap took me one hour, while I spent only forty-five minutes marking a medium-sized program as Demoney.

- Testing our plugins with real applications has enabled us to know how good/useful our detectors actually are. On the one hand, the first detector, which looks for “this” leaks in Java, has several limitations. In addition, it is not efficient when applied to large Java applications. On the other hand, the second plugin, which looks for certain defects in Java Card applications, obtains much better results. For instance, remember the security holes found in Pacap, a low quality application written by beginning Java Card programmers, and the efficiency holes found in Demoney, a high quality electronic purse implemented by experienced Java Card programmers.

- Finally, the reader may wonder if applying static analysis techniques is worthwhile. This question is not easy to answer since it depends strongly on each situation. In general, if a reviewer faces a specific kind of application very often, then developing and applying such strategies is definately a good idea. By contrast, if a reviewer hardly ever analyzes a certain sort of application, then using such techniques is worthless since he will spend more time learning and implementing them than reviewing their source code by hand.

6.2 Future work

Related to future work, there are several interesting enhancements:

- First detector:
  - To avoid the *false negatives* we get when the “this” reference is published in two steps, one possibility will be maintaining a list with the objects which reference “this” and controlling if one of them is leaked.
  - To improve the plugin’s speed when analyzing larger applications, one possibility will be tuning the method which discovers the inheritance hierarchy of the analyzed application. Another possibility will be re-writing this method again, following a different strategy, and testing the detector’s behaviour with the same large applications.
- Second detector:
  - As soon as annotations on local variables are stored into the Java bytecodes and the FindBugs API includes a method to visit them, our implementation will be able to be improved to take advantage of such marks. To collect them, our plugin will use the same list it currently uses to compile those made on method parameters.
  - To avoid warning about justified efficiency bugs like the one found in Demoney (See Listing 5.10 on page 63), one possibility could be ignoring those calls to `Util.arrayCopy` found within a transaction block. Listing 6.1 depicts one abstract algorithm for this purpose.

```
// inTransaction: true if we are inside a transaction block,
//                false otherwise
boolean inTransaction = false;

If (JCSystem.beginTransaction() is called) Then
  inTransaction = true;

Else If (JCSystem.commitTransaction() is called) Then
  inTransaction = false;

Else If (JCSystem.abortTransaction() is called) Then
  inTransaction = false;

Else If (Util.arrayCopy() is called) Then
  If (not(inTransaction)) Then
    Analyze the current call
```

Listing 6.1: Algorithm to avoid warning about justified efficiency bugs.

A Appendix

A.1 Commented source code (Chapter 4)

This section contains the commented source code of the program that implements the first custom detector.

```
package nl.ru.cs.fbjc.detect;

import ...; //Import classes, packages, etc.

public class FindThisReferenceEscape extends BytecodeScanningDetector{
  //Fields: All of them are private (keyword private ommitted for simplicity)
  BugReporter bugReporter;
  boolean seenALOAD_0, seenALOAD, ...; //boolean variables set

  //Local information (for each class)
  Hashtable htInherit;      //For each class, 1 ArrayList with its superclasses
  Hashtable htStaticFields; //For each class, 1 ArrayList with its static fields
  Hashtable htInnerFields;  //For each class, 1 ArrayList with its inner
                            //classes instances

  //Global information (for each application)
  ArrayList listClassContext;  //List of class contexts
  ArrayList reachableMethods;  //List of RFC methods
  Hashtable htMethodsWithBugs; //For each method, 1 ArrayList with its bugs
                               //Indexed by ClassMethodSignature objects


  //Constructor: Initialities variables
  public FindThisReference(BugReporter bugReporter){
    this.bugReporter = bugReporter;
    resetBooleans(); //Initialize the whole booleans set
    htInherit = new Hashtable();
    htStaticFields = new Hashtable();
    //Initialize the rest of lists and tables
    ...
  }


  //Overriden. Visits the class context
  public void visitClassContext(ClassContext classContext){
    //Adds the current context to the list of class contexts
    //Adds a new ArrayList for the current class in the static fields table
    //Adds a new ArrayList for the current class in the inner fields table
    //Adds a new ArrayList for the current class in the inheritance table

    //Gets a list of fields for the current class and iterates over it:
    for (f in fields){
      if (f.isStatic())
        //Store f in htStaticFields, using current class name as key
      if (f.isInnerField())
        //Store f in htInnerFields, using current class name as key
    }

    //Stores current class' superclasses in the htInheritance table
    storeSuperClasses(classContext.getJavaClass());
  }


  //Overriden. Visits a method
  public void visit(Method obj){
    resetBooleans(); //For each method, a new scope starts
    super.visit();
  }


  //Overriden. Executed whenever a new opcode is found within a method body.
  //Used in order to find pieces of code which mean bug instances.
  public void sawOpcode(int seen){
    //Whenever a new opcode is seen, one or more of the boolean variables
    //may change.

    //Whenever a scope's end is reached, it uses the booleans set to determine
    //wheter a bug has been found or not. If so, it's stored in the table
    //htMethodWithBugs
    if (endOfScope){
      if (thereIsBug(...))
        storeBugInstance(...);
      resetBooleans(...);    //Starts a new scope anyway
    }
  }


  //Overriden. Executed at the end of the analysis
  //By this time we know:
  //  - All the class contexts
  //  - The inheritance structure
  //So, this method is used in order to:
  //  - Find the reachable from constructor methods (using the class contexts)
  //  - Inherit methods with bugs, when possible
  //Finally report the bug instances found in RFC methds.
  public void report(){
    findReachableMethods();
    addMethodsBugsInheritance();
    //Scans the reachableMethods and the htMethodWithBugs data structures and
    //notifies only those bug instance which appear in constructors and/or in
    //RFC methods.
  }


  //Auxiliary method: private access
  private boolean thereIsBug(int opcode, booleans ...){
    //Returns true if there is a bug for the given opcode and booleans set,
    //false otherwise
  }


  //Auxiliary method: private access
  private void storeBugInstance(String className, Method m, BugInstance bi){
    //Adds a new bug instance into one of the ArrayLists contained in
    //htMethodWithBug. In the table, className+m.name+m.signature is used as key
  }


  //Auxiliary method: private access
  private void findReachableMethods(){
    //Finds the RFCs methods and stores them into the
    //ArrayList reachableMethods
  }


  //Auxiliary method: private access
  private void addMethodsWithBugInheritance(){
    //For each class, look for its bugs and the list of subclasses
    //For each subclass and For each bug:
      //Add a new bug into the table htMethodWithBugs, except for those
      //subclasses which override a method with bugs without introducing
      //any defect.
  }

  //Other auxiliary methods:
  private void storeSuperClasses(JavaClass javaClass){...}
  private ArrayList subClasses(JavaClass javaClass){...}
  private void resetBooleans(){...}
}
```

Listing A.1: FindThisReferenceEscape commented source code.

A.2 XML files (Chapter 4)

This section contains the XML files needed to create the FindBugs plugin discussed in chapter 4. Furthermore, it contains the Apache Ant [25] build script files: build.xml and build.properties.

A.2.1 findbugs.xml

```xml
<FindbugsPlugin>
  <Detector class="nl.ru.cs.fbjc.detect.FindThisReferenceEscape" speed="fast" reports="THIS_REF_ESC"/>
  <BugPattern abbrev="TRE" type="THIS_REF_ESC" category="CORRECTNESS"/>
</FindbugsPlugin>
```

Listing A.2: findbugs.xml

A.2.2 messages.xml

```xml
<MessageCollection>
  <Detector class="nl.ru.cs.fbjc.detect.FindThisReferenceEscape">
    <Details>
      <![CDATA[
       <p>This detector finds "this" references escaping during construction. It's a fast detector.</p>
      ]]>
    </Details>
  </Detector>

  <BugPattern type="THIS_REF_ESC">
    <ShortDescription>This-reference escaping during construction</ShortDescription>
    <LongDescription>This-reference escaping during construction in {1}</LongDescription>
    <Details>
      <![CDATA[
       <p>This detector finds "this" references escaping during construction. For instance:
       <pre>
        public class Incorrect {
         static Object param;

         public Incorrect (){
         ..//..
         param=this;
         //or
         AnotherClass.methodRegister(this);
         ..//..
         }
        }
       </pre>
            It is incorrect because other threads could access the "this" object before it is completely created.
        </p>

        ... more text omitted...

      ]]>
    </Details>
  </BugPattern>

  <BugCode abbrev="TRE">
    "This" Reference Escaping during construction
  </BugCode>
</MessageCollection>
```

Listing A.3: messages.xml

A.2.3 build.xml

```
<!-- FindBugs Plugin. Apache Ant build script file -->

<project name="fbjc" default="build" basedir=".">
  <property file="build.properties"/>

  <property name="api.dir"      value="${plugin.home}/api" />
  <property name="build.dir"    value="${plugin.home}/build" />
  <property name="etc.dir"      value="${plugin.home}/etc" />
  <property name="lib.dir"      value="${findbugs.home}/lib" />
  <property name="plugin.src"   value="${plugin.home}/src/nl/ru/cs/fbjc" />

  <path id="plugin.classpath">
    <pathelement location="${lib.dir}/annotations.jar"/>
    <pathelement location="${lib.dir}/bcel.jar"/>
    <pathelement location="${lib.dir}/dom4j-full.jar"/>
    <pathelement location="${lib.dir}/findbugs-ant.jar"/>
    <pathelement location="${lib.dir}/findbugsGUI.jar"/>
    <pathelement location="${lib.dir}/findbugs.jar"/>
  </path>

  <!-- Standard build target -->
   <target name="build" depends="compile, jar" />


  <!-- Compiles the plugin -->
    <target name="compile">
      <mkdir dir="${build.dir}"/>
      <javac srcdir="${plugin.src}"
         destdir="${build.dir}"
         deprecation="on"
         debug="on">
        <classpath refid="plugin.classpath"/>
      </javac>
    </target>

```

```
<!-- Creates the jar file -->
  <target name="jar" depends="compile, validate">
    <jar destfile="${findbugs.home}/plugin/fbjc.jar">
       <fileset dir="build"/>
       <zipfileset dir="etc" includes="*.xml" prefix=""></zipfileset>
    </jar>
  </target>

<!-- Validates the .xml files in the etc subdirectory -->
  <target name="validate">
    <xmlvalidate lenient="false" failonerror="yes">
        <attribute name="http://apache.org/xml/features/validation/schema"
            value="true"/>
        <attribute name="http://xml.org/sax/features/namespaces" value="true
            "/>
        <fileset dir="${etc.dir}" includes="*.xml"/>
    </xmlvalidate>
  </target>

<!-- Generates API documentation -->
  <target name="javadoc">
    <delete quiet="on" dir="${api.dir}"/>
              <mkdir dir="${api.dir}"/>
    <javadoc access="protected"
       author="true"
       packagenames="*"
       classpathref="plugin.classpath"
       destdir="${api.dir}"
       doctitle="FindBugs Plugin API documentation"
       nodeprecated="false"
       nodeprecatedlist="false"
       noindex="false"
       nonavbar= "false"
       notree="false"
       sourcepath="${plugin.home}/src"
       splitindex="true"
       use="true"
       version="true"/>
  </target>

<!-- Shows version information -->
  <target name="version">
   <echo>
   FindBugs Plugin: Don't let the "this" reference escape during construction
       v0.1

   Copyright (C) 2007 - Manuel Villalba Cifuentes

   Please refer to LICENSE for license information.
   </echo>
  </target>

<!-- Show a help message -->
```

```
    <target name="help">
     <echo>
      FindBugs Plugin. Available targets:

      - build    : builds the plugin and deploys it in the FindBugs plugin
         directory.
      - jar      : creates the plugin .jar file and copies it to the FindBugs
         plugin directory.
      - validate : validates the .xml files in the etc subdirectory.
      - javadoc  : generates API documentation.
      - version  : shows version information.
      - help     : shows this message.
     </echo>
    </target>
</project>
```

Listing A.4: Main Apache Ant build script file (build.xml)

A.2.4 build.properties

```
# User configuration. This section must be modified to reflect your system!
# findbugs.home points out the FindBugs directory
  findbugs.home = ${user.home}/findBugs
# plugin.home points out the custom detector directory (where build.xml is)
  plugin.home = ${user.home}/mthesis/implementation/dontLetThisEscape/fbjc
```

Listing A.5: Auxiliar Apache Ant build script file (build.properties)

A.3 Custom Tests (Chapter 4)

This section lists the custom tests used in Chapter 4. See subsection 4.3.1 for the first custom test and 4.3.2 for the second.

A.3.1 Custom test 1

CustomTest1.java

```
import java.util.ArrayList;

public class CustomTest1{
  //Fields:
  static CustomTest1 meStatic;
  CustomTest1 meNoStatic;
  ArrayList arrayList;
  ClassA objA;
  IC ic;  //IC is an non static inner class
  static IC staticIC;
  static Object[] staticArray;

  //Constructor:
  public CustomTest1(){
    //Initialize variables:
    arrayList = new ArrayList();
    ic = new IC();
    objA = new ClassA();
    staticArray = new Object[1];

    //Notified Bugs?
    meStatic = this;   //Yes. Notified bug instance
    meNoStatic = this; //No. meNoStatic is not visible to other threads
    staticArray[0]= this; //Yes, staticArray is visible from other threads
    staticIC = new IC();//Yes, it leaks "this" implicitly
    methCallingA(this); //Yes, leaks "this" explicitly
    methCallingA(ic); //Yes, leaks "this" implicitly
    methCallingA(new IC()); //Yes, leaks "this" implicitly
    methCallingA(objA, ic); //Yes, leaks "this" implicitly
    methCallingA(objA, new IC()); //Yes, leaks "this" implicitly

    //Invoke other methods
    m1();
    overloadedMethod();
    objA.methSameName();

    //Limitation: False positive!
    //arrayList is not leaked anywhere
    //However, the add(this) call is treated as a general method
    arrayList.add(this);

    //Limitation: False negative!
    //arrayList contains a "this" reference
    //arrayList is leaked and none warning is issued
    objA.method(arrayList);
  }


  //m1() is reachable from constructor
  public void m1(){
    //Notified bugs?
    meStatic = this;   //Yes, leaks "this" explicitly
    meNoStatic = this; //No. meNoStatic is not visible to other threads
    objA.method(this, null); //Yes, leaks "this" explicitly
    ClassA.staticCustomTest1 = this; //Yes, leaks "this" explicitly

    //Invokes m2
    m2();
  }


  //m2() is reachable from constructor, invoked from within m1
  public void m2(){
    //Notified bugs?
    meStatic = this;  //Yes, leaks "this" explicitly
    objA.method(this, null); //Yes, leaks "this" explicitly

    //Invokes m4
    m4(objA);
  }


  //m3() is NOT reachable from constructor!
  public void m3(){
    //Notified bugs?
    meStatic = this;   //No, m3() is not reachable.
    meNoStatic = this; //No. meNoStatic is not visible to other threads
    objA.method(this, null); //No, m3() is not reachable.
  }


  //m4() is reachable from constructor, invoked from within m2
  public void m4 (ClassA paramA){
    //Notified bugs?
    paramA.method(this); //Yes, leaks "this" explicitly
    paramA.method(ic);   //Yes, leaks "this" implicitly using a inner
                         //class field
    paramA.method(new IC()); //Yes, leaks "this" implicitly using
                             //a just created new object
    objA.method(ic);         //Yes, leaks "this" implicitly
    objA.method(new IC());   //Yes, leaks "this" implicitly
  }

  public void methCallingA(Object o){
    objA.method(o);
  }

  public void methCallingA(ClassA ca, Object o){
    ca.method(o);
  }

  //overloadedMethod() is reachable from constructor
  public void overloadedMethod(){
    //No bugs
  }

  //overloadedMethod(Object o) is NOT reachable from constructor
  public void overloadedMethod(Object o){
    //Notified bugs?
    meStatic = this;   //No, this method is not RFC
    objA.method(this, null); //No, this method is not RFC
  }

  //This methSameName() is not RFC.
  //On the other hand, ClassA.methSamenName() is RFC
  public void methSameName(){
    //Notified bugs?
    meStatic = this;   //No, this method is not RFC
    objA.method(this, null); //No, this method is not RFC
  }


  class IC{}
}
```

Listing A.6: CustomTest1.java

ClassA.java (Auxiliary class used in A.6)

```
public class ClassA{
  static CustomTest1 staticCustomTest1;

  public void method(Object o){}

  public void method(Object o1, Object o2){}

  public void methSameName(){}

}
```

Listing A.7: ClassA.java (auxiliary class used in A.6)

A.3.2 Custom test 2

A.java

```java
public class A{
  static A meStatic;

  //m1 leaks "this" explicitly
  public void m1(){
    meStatic = this; //Notified, m1 is RFC
  }

  //m2 is not reachable from constructor
  public void m2(){
    meStatic = this; //Not Notified, m2 is not RFC
  }

  //Method with a bug, but not RFC
  public void methodWithBugs(){
    meStatic = this; //Not Notified, methodWithBugs is not RFC
  }

  //Method with a bug, but not RFC
  public void methodWithBugs(int i){
    meStatic = this; //Not notified, methodWithBugs(int) is not RFC
  }
}
```

Listing A.8: A.java

B.java

```java
public class B extends A{
  public B(){
    meStatic = this;  //Leaks "this" explicitly
    m1();             //invokes m1 inherited from A
    super.m1();       //invokes m1 inherited from A
    method2();
  }

  public void method(){ //Not RFC
    meStatic = this;    //Not Notified, method is not RFC
  }

  public void method2(){
    meStatic = this;     //Notified, method2 is RFC
  }

  //A.m2() contains a bug, but B.m2() doesn't
  public void m2(){}

  //Overrides methodWithBugs, introducing new bugs
  public void methodWithBugs(){
    meStatic = this; // Notified, it is called from within C constructor
  }

  //Overrides methodWithBugs, introducing new bugs
  public void methodWithBugs(int i){
    meStatic = this; // Notified, it is called from within C constructor
  }
}
```

Listing A.9: B.java

C.java

```java
public class C extends B{
  public C(){
    m2(); //A.m2() contains a bug, but B.m2() doesn't
          //C inherites m2() from B, not from A.
          //So, this call doesn't involve any bug.
    methodWithBugs();
    methodWithBugs(1);
  }
}
```

Listing A.10: C.java

A.4 Commented source code (Chapter 5)

This section contains the commented source code of the program that implements the second custom detector.

```java
package nl.ru.cs.fbjc.detect;

import ...; //Import classes, packages, etc.

public class JavaCardAppsDetector extends BytecodeScanningDetector{

  //Fields: All of them are private (keyword private ommitted for simplicity)
  BugReporter bugReporter;
  // Lists:
  ArrayList listBugInstances; //List of instances
  Hashtable htAnnotatedData; //For each element 1 ArrayList with its annots.
  Hashtable htMethodAnnotations; /*For each method, a list of PositionAnnotation
       instances: (Position int, Annotation str)*/
  Hashtable htSpecialMethodsCalls; //For each special method, a list of calls
  Hashtable htNonSpecialMethodsCalls; // For each non special method,
                                      // a list of calls

  // Other fields:
  String lastInvokedStr; // Last invoked method (name and signature)
  String penultInvokedStr; // Penultimate invoked method (name and signature)
  OpcodeStack stack; //Stack
  ...

  //Constructor: Initialities variables
  public JavaCardAppsDetector(BugReporter bugReporter){
    this.bugReporter = bugReporter;

    listBugInstances = new ArrayList();
    htAnnotatedData = new Hashtable();
    htMethodAnnotations = new Hashtable();
    htSpecialMethodsCalls = new Hashtable();
    htNonSpecialMethodsCalls = new Hashtable();

    lastInvokedStr = "";
    penultInvokedStr = "";
    stack = new OpcodeStack();

    ...

  }


  //Overriden. Visits a method
   public void visit(Method obj) {
    stack.resetForMethodEntry(this); //Resets the stack
    super.visit(obj);
  }
  //Overriden. Visits annotations on fields, methods and classes
  public void visitAnnotation(String annotationClass, Map<String,Object> map,
      boolean runtimeVisible){
    if (visitingField()){
      /* Check if the current annotation is impossible, contradictory, etc:
          - @Transient Object simpleObj;
          - @Transient @Persistent Object[] cont;

          If (the annotation is possible) Then
            Include the current annotation to the field's annotations list:
              ArrayList al = ((ArrayList)(htAnnotatedData.get(Field)));
              al.add(Annotation);
      */
    }

    else if (visitingMethod()){
      /* Check if the current annotation is impossible, contradictory, etc:
          - @Transient Object methSimpleObj(){...}
          - @ElemsTransient Object[] methNoFirstLevelAnnot(){...}

          If (the annotation is possible) Then
            Include the current annotation to the method's annotations list:
              ArrayList al = ((ArrayList)(htAnnotatedData.get(Method)));
              al.add(Annotation);
      */
    }

    else{ //Annotating a class
      //Our annots. cannot be applied to classes, so here a new bug is stored
    }
  }


  //Overriden. Visits annotations on method parameters
  public void visitParameterAnnotation(int p, String annotationClass, Map<String
      ,Object> map, boolean runtimeVisible){
    /* Check if the current annotation is impossible, contradictory, etc:
        - void meth (@Transient Object param){...}
        - void meth (@Transient @Persistent Object[] cont){...}

        If (the annotation is possible) Then
          Include the current annotation to the method's list of annotated
          parameters. For this purpose, the class PositionAnnotation is used:
            ArrayList al = (ArrayList)(htMethodAnnotations.get(Method)));
            al.add(new PositionAnnotation(p, Annotation));

          Include the current annotation to the annotated data list:
            ArrayList al = ((ArrayList)(htAnnotatedData.get(Parameter)));
            al.add(Annotation);
    */
  }


```

```
//Overriden. Executed whenever a new opcode is found within a method body.
//Used in order to collect calls (and passed arguments) to methods.
public void sawOpcode(int seen){
  /*
  if (seenMethodCall){
    if (isSpecial(Method)){
      Take its relevant argument from the stack and store it with the call:
        ArrayList listArgs = new ArrayList();
        listArgs.add(stack.getItem(specificOffset));
      Add the current call and arg. to the list of special method calls:
        ArrayList al = ((ArrayList)(htSpecialMethodsCalls.get(Method)));
        al.add(listArgs);
    }
    else{ //NonSpecial
      Take all its arguments from the stack and store them with the call:
        ArrayList listArgs = new ArrayList();
        for (int off=0; off<Method.numberArgs; off++)
          listArgs.add(stack.getItem(off));

      Add the current call and args to the list of nonspecial method calls:
        ArrayList al = ((ArrayList)(htNonSpecialMethodsCalls.get(Method)));
        al.add(listArgs);
    }
  }
  */
}


//Overriden. Executed at the end of the analysis. So, this method is used to:
//  - Invoke the method that checks calls to special methods.
//  - Invoke the method that checks calls to nonspecial methods.
//  - Order the previously stored bugs
//  - Report all the bugs
public void report(){
  checkSpecialMethodsCalls();
  checkNonSpecialMethodsCalls();
  mergeSort(listBugInstances);
  for (int i=0; i<listBugInstances.size(); i++)
    bugReporter.reportBug(listBugInstances.get(i));
}


// Auxiliary method: private access.
// Checks calls to special calls.
private void checkSpecialMethodsCalls(){
  /*
  For each special method:
    Take all its calls and passed relevant argument
    If (the argument does not follow the special method's constraint) Then
      Store a new bug instance in the bugs list
  */
}

```

```
  // Auxiliary method: private access.
  // Checks calls to  nonspecial calls.
  private void checkNonSpecialMethodsCalls(){
    /*
    For each nonspecial method:
      Take all its calls and passed arguments
      For each argument:
        If (the current arg. does not follow the method annotation for its
            position) Then
          Store a new bug instance in the bugs list
    */
  }

  //Other auxiliary methods:
  private void mergeSort(ArrayList list){...}
  private boolean isContained (String annot, ArrayList listOfAnnots){...}

  ...

}
```

Listing A.11: JavaCardAppsDetector commented source code.

A.5 Custom Test (Chapter 5)

This section lists the custom test used in Chapter 5 in short.

```
package summary;
import javacard.framework.*;

public class CustomTest extends Applet {
  //Annotations on class fields:
  @Persistent byte[] perstByteArray; // Ok
  @Transient byte[] transByteArray; // Ok
  byte[] noAnnotsByteArray; // Ok
  @Persistent @ElemsTransient Object[] perstElemsTransArray; // Ok
  @Persistent @ElemsPersistent Object[] perstElemsPerstArray; // Ok
  @Transient int a; //Bug, it is not an array
  @Persistent @ElemsTransient short[] arrayShort; // Bug
  @Transient @ElemsPersistent Object[] transElemsPerstArray; // Bug

  //Annotations on methods:
  public @Transient byte method1(){ return 0;} // Bug
  public void method2(@ElemsTransient Object[] a){} // Bug

  //JavaCard API Tests on Fields
  public void testAPIOnFields(APDU apdu1){
    short sh = 0;
    byte b = 0;
    //make new transient byte arrays
    perstByteArray = JCSystem.makeTransientByteArray(sh, b); // Bug
    noAnnotsByteArray = JCSystem.makeTransientByteArray(sh, b); // Bug
    transByteArray = JCSystem.makeTransientByteArray(sh, b);  // Ok

    //arrayCopyNonAtomic writes only on Transient parameters.
    Util.arrayCopyNonAtomic(new byte[0], sh, perstByteArray, sh, sh); // Bug
    Util.arrayCopyNonAtomic(new byte[0], sh, transByteArray, sh, sh); // Ok

    //arrayFillNonAtomic writes only on Transient parameters.
    Util.arrayFillNonAtomic(noAnnotsByteArray, sh, sh, b); // Bug
    Util.arrayFillNonAtomic(transByteArray, sh, sh, b); // Ok

    //arrayCopy writes only on Persistent parameters.
    Util.arrayCopy(new byte[0], sh, transByteArray, sh, sh); // Bug
    Util.arrayCopy(new byte[0], sh, perstByteArray, sh, sh); // Ok

    //isTransient, it should work only with no annotated data
    JCSystem.isTransient(perstByteArray); // Bug
    JCSystem.isTransient(transByteArray); // Bug
    JCSystem.isTransient(noAnnotsByteArray); // Ok

    //getBuffer writes only on Transient data (no class fields nor array elems)
    transByteArray = apdu1.getBuffer(); // Bug, it's a class field

    //new byte[] writes only on persistent data
    transByteArray = new byte [2];    // Bug
    noAnnotsByteArray = new byte [2]; // Bug
    perstByteArray = new byte [2]; // Ok
  }

  //Test 2nd level annotations
  public void test2LevelsAnnots(){
    short sh = 0;
    byte b = 0;
    perstElemsTransArray = new Object[2];
    perstElemsPerstArray = new Object[2];

    //Bug, it's @ElemsPersistent:
    perstElemsPerstArray[0] = JCSystem.makeTransientBooleanArray(sh, b);

    //Bug, it'S @ElemsTransient:
    perstElemsTransArray[1] = new byte[3];

    //Bug, it's @ElemsTransient:
    Util.arrayCopy(new byte[0], sh, (byte[]) perstElemsTransArray[0], (short)0,
        (short)0);

    //Bug, it's @ElemsPersistent:
    Util.arrayCopyNonAtomic(new byte[0], sh, (byte[]) perstElemsPerstArray[0] ,
        (short)0, (short)0);
  }
  ...
}
```

Listing A.12: CustomTest.java

Bibliography

[1] **Brian Goetz**. *Don't let the "this" reference escape during construction*. IBM developerWorks (column Java Theory and Practice), June 2002.
http://www-128.ibm.com/developerworks/java/library/j-jtp0618.html.

[2] **Chris Grindstaff**. *FindBugs, Part 1: Improve the quality of your code*. IBM developerWorks, May 2004.
http://www-128.ibm.com/developerworks/java/library/j-findbug1/.

[3] **Chris Grindstaff**. *FindBugs, Part 2: Improve the quality of your code*. IBM developerWorks, May 2004.
http://www-128.ibm.com/developerworks/java/library/j-findbug2/.

[4] **David Hovemeyer and William Pugh**. *Finding bugs is easy*. OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages and applications, December 2004.
http://findbugs.sourceforge.net/docs/oopsla2004.pdf.

[5] **David Hovemeyer and William Pugh**. The FindBugs(TM) Manual, October 2006.
http://findbugs.sourceforge.net/manual/index.html.

[6] **Erik Poll**. *Java Card*. OOTI course: Formal Methods in the Software Life Cycle, May 2006.
http://www.cs.ru.nl/˜martijno/ooti/slides/javacard.pdf.

[7] **Erik Poll**. *What are smart cards and why would you use them?* OOTI course: Formal Methods in the Software Life Cycle, April 2006.
http://www.cs.ru.nl/˜martijno/ooti/slides/smartcards.pdf.

[8] **Free Software Foundation Inc.** *GNU Lesser General Public License*.
http://www.gnu.org/copyleft/lesser.html.

[9] **Gemplus and ONERA**. *PACAP*, 1999.
http://www.cert.fr/francais/deri/wiels/Pacap/.

[10] **InfoEther**. *PMD*, 2006.
http://pmd.sourceforge.net.

[11] **Jason Hunter**. *Making the most of Java's metadata*. ORACLE Technology Network, 2005.
http://www.oracle.com/technology/pub/articles/hunter_meta.html.

[12] **Knizhnik and Artho**. *Jlint*, 2006.
http://www.ispras.ru/˜knizhnik/jlint/.

[13] **Michiel Graat**. *Static analysis of Java Card applications*, August 2006. SOS Department: Nijmeegs Instituut voor Informatica en Informatiekunde (NIII) - Radboud University Nijmegen
http://www.cs.ru.nl/˜erikpoll/afstudeerders/MichielGraatScriptie.pdf.

[14] **Nick Rutar, Christian Almazan and Jeff Foster**. *A Comparison of Bug Finding Tools for Java*. The 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04), November 2004.
http://www.cs.umd.edu/˜jfoster/papers/issre04.pdf.

[15] **Oliver Burn**. *Checkstyle*, 2006.
http://checkstyle.sourceforge.net.

[16] **Sumit Dhar**. *Introduction to Smart Cards.* November 2004.
http://www.rootshell.be/˜isb/ISC.pdf.

[17] **Sun Microsystems, Inc.** *javap - The Java Class File Disassembler*, 2001.
http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/javap.html.

[18] **Sun Microsystems, Inc.** *Java(TM) 2 Platform SE 5.0 - API Specification*, 2004.
http://java.sun.com/j2se/1.5.0/docs/api/index.html.

[19] **Sun Microsystems, Inc.** *JSR 175: A Metadata Facility for the Java(TM) Programming Language*, September 2004.
http://jcp.org/en/jsr/detail?id=175.

[20] **Sun Microsystems, Inc.** *Java*, 2006.
http://www.java.com/en/.

[21] **Sun Microsystems, Inc.** *JavaCard*, 2006.
http://java.sun.com/products/javacard/.

[22] **Sun Microsystems, Inc.** *Java Card 2.1 Platform API Specification*, 2006.
http://java.sun.com/products/javacard/htmldoc/.

[23] **Sun Microsystems, Inc.** *The Java SE Development Kit (JDK 5.0 Update 9)*, 2006.
http://java.sun.com/javase/downloads/index.jsp.

[24] **Superior Polytechnic School of Albacete. University of Castilla-La Mancha.** *Object-Oriented Design and Programming*, 2006.
http://www.info-ab.uclm.es/asignaturas/42579/index.htm.

[25] **The Apache Software Foundation**. *Apache Ant*, 2006.
http://ant.apache.org.

[26] **The Apache Software Foundation**. *The Byte Code Engineering Library*, 2006.
http://jakarta.apache.org/bcel/.

[27] **The Eclipse Foundation**. *Eclipse*, 2006.
http://www.eclipse.org.

[28] **The FindBugs Development Team**. *FindBugs*, 2006.
http://findbugs.sourceforge.net.

[29] **The FindBugs Development Team**. *FindBugs API*, 2006.
http://findbugs.sourceforge.net/api/index.html.

[30] **The MegaMek development team.** *MegaMek - An unofficial, online version of the Classic BattleTech board game.*, 2006.
http://megamek.sourceforge.net/.

[31] **Tim Lindholm and Frank Yellin**. The Java(TM) Virtual Machine Specification (2nd Edition), chapter 6, pages 171–361. Addison Wesley Professional, April 1999.
http://java.sun.com/docs/books/vmspec/.

[32] **Trusted Logic**. *Demoney*, 2001-2002.

[33] **Wikipedia: The Free Encyclopedia**. *Java Annotation*, 2006.
http://en.wikipedia.org/wiki/Java_annotation.

[34] **Wikipedia: The Free Encyclopedia**. *Mergesort: A O(nlog(n)) sorting algorithm.*, 2006.
http://en.wikipedia.org/wiki/Mergesort.

[35] **Wikipedia: The Free Encyclopedia**. *Smart Card*, 2006.
http://en.wikipedia.org/wiki/Smart_card.

[36] **Wikipedia: The Free Encyclopedia**. *The Visitor design pattern*, 2006.
http://en.wikipedia.org/wiki/Visitor_pattern.

[37] **Zhiqun Chen**. Java Card(TM) Technology for Smart cards. Architecture and Programmer's Guide, chapter 2-5, pages 11–64. Addison Wesley Professional, September 2000.