Static program analysis of multi-applet JavaCard applications

Alexandros Loizidis[1] Vasilios Almaliotis[1] Panagiotis Katsaros[1]

[1] Department of Informatics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
{aloizidi, valmalio, katsaros}@csd.auth.gr

Abstract

Java Card provides a framework of classes and interfaces that hides the details of the underlying smart card interface and makes it possible to load and run on the same card several applets, from different application providers with complex trust relationships. This fact opens prospects for new business applications, but the card issuer has to secure absence of malicious or faulty card applets. He has to be able to check that (i) applets do not cause illicit method invocations that violate temporal restrictions of inter-applet communication, (ii) applets protect themselves from unwanted information flow to third parties and (iii) it is not possible for an unhandled Java Card API exception to leave an applet in an unpredictable state that is potentially dangerous for the application's security. We explore recent advances in theory and tool support of static program analysis and we present an approach for automatic verification of smart card applications that by definition are security critical.

Keywords
smart card, static program analysis, software security, Java Card, verification

Introduction

In current work, we propose program analysis techniques that have been implemented in the FindBugs open source framework (Hovemeyer & Pugh, 2004; The FindBugs project site, 2009), for statically verifying important security properties of interacting Java Card applets.
Static program analysis has the potential to become a credible means for automatic verification of smart card applications that by definition are security critical. There is an extended base of well established analysis techniques and recent research developments, as well as versatile analysis frameworks like FindBugs that open excellent prospects to exploit the provided analysis support and the already implemented error detectors. Current article introduces error detectors adapted to the security

requirements of the Java Card multi-applet environment, in order to highlight the perspectives and the limitations of the described alternative.
The most significant virtue is the use of a single verification technique, in place of diverse verification approaches that require highly specialized formal analysis skills, for the different Java Card security verification tasks. In our case, analysis assumes only basic Java programming skills, but with precision levels that are restricted by existing limitations of the analysis support in FindBugs. We highlight these limitations and we discuss recent research works that aim in higher precision, based on requirements that may be fulfilled in future versions of FindBugs.
Our analyses address the security concerns caused by the fact that Java Card allows loading and running on the same card several applets, from different application providers with complex trust relationships and partnerships. The Java Card platform controls cooperation of interacting applets through a firewall mechanism that enforces applet isolation and allows communication only through explicitly declared shareable interfaces including the explicitly permitted method invocations. This sort of checks is static in nature, i.e. one method call is either allowed in all cases or it is never allowed. Thus, the provided protection cannot impose temporal restrictions on inter-applet communications.
In a scenario, where several independent application providers have applets on a single card, the aforementioned weakness incurs a serious security risk for illicit method invocations between the interacting applets. Let us consider the typical case of a multi-applet smart card with one purse applet and two loyalty applets that are notified when card transactions occur, in order to award bonus points. Loyalty applets are communicated by calls to methods declared in shareable interfaces. Temporal restrictions for secure applet interaction include the requirement of recursion freeness for the methods of the shareable interfaces and the absence of transitive communications that span the contexts of the two loyalty applets. In the more complex case, where a loyalty applet shares bonus points through some agreed loyalty applet to loyalty applet communication, secure interaction requires additional temporal restrictions besides those mentioned.
Apart from temporal safety, it is also important to assure that the allowed communication between the three applets does not imply unwanted information flow from one loyalty applet to the other. This is the only way to guarantee that secret and potentially commercial data produced in one applet cannot be leaked to another applet, while at the same time applets of one application cannot be crashed by other applet's corrupt data.
Another source of security risk are the potentially misused Java Card API calls, the multiple-entry-point program structure and the possibility a potentially unhandled exception to reach the invoked entry point. This contingency opens a possibility to leave a Java Card applet in an unpredictable state that may be dangerous for the application's security.
Next section presents the current state of the art in security verification of Java Card multi-applet applications. We highlight the problems faced and we

examine the available alternatives for static program analyses that focus on the described Java Card security problems. In the subsequent section, we introduce basic concepts of static program analysis with the FindBugs framework. The provided analysis support is exploited in the developed FindBugs security violation detectors that are presented in the following section. The developed bug detectors belong to a category of static program analyses, which are commonly referred to as typestate tracking. These techniques are appropriate for the first and the third mentioned security problems. In separate sections, we discuss state of the art program analysis techniques that improve analysis precision, as well as techniques for the second mentioned security problem. Finally, we conclude with a critical view of the shown security verification approach and we discuss its anticipated impact.

Background

Static Analysis

Static analysis can be effective in verifying the behavior of a program against a partial specification that represents the absence of a security error. By the term static analysis we refer to any approach for assessing code without executing it. This broad definition includes fully-automated model checking techniques, semi-automated formal analyses that involve logical inference and program analyses that are based on abstract interpretation or alternatively on dataflow facts over the control-flow graph of the source program.

In the Java Card multi-applet environment, security errors may be attributed to illegal applet interactions that are not caught by the Java Card firewall or to misused and therefore dangerous calls of Java Card API methods. The static analyses found in the related bibliography are based on established formal techniques that aim in precise program verification (Burdy et. al., 2003; Beckert & Mostowski, 2003; Marché et. al., 2004; Meyer & Poetzsch-Heffter, 2000; Jacobs et. al., 2004; Van den Berg & Jacobs, 2001; Breunesse et. al., 2005), but they are not fully automated. It is important to note that there is no single technique that can cope with all kinds of security errors in Java Card applications.

For detecting unwanted information flows to third parties we refer to the following alternatives:

- The type inference approach proposed in (Akdemir, 1998), which essentially introduces changes in the original type inference algorithm of the Java Card platform.
- The assume-guarantee model checking approach of (Bieber et. al., 2002), which is based on abstracting the byte codes of the methods of a Java Card application to interconnected SMV modules.

In overall, complete static verification may have to be based on a combination of techniques that will cover all sources of security violations in the Java Card

multi-applet environment. Efficient use of these techniques requires highly specialized formal analysis skills that cannot be found in most software engineers.
The mentioned limitations make evident the need for easier to apply fully automated static analysis techniques, perhaps at the cost of affordable lower precision in the provided verification results. Static program analyses are neither sound nor complete verification approaches meaning that in the general case there is no guarantee that they will detect all security violations, if any (yielding false negatives). Moreover, there is also no guarantee for the absence of false positives. In most cases, we can usually afford a relatively small number of false positives, but we require the analysis to exclude all possibilities of false negatives.

Program analysis
Abstract interpretation is one of the proposed static program analyses for Java Card applets (The Java Verifier project, 2009). It lies on a semantics-based description of all possible executions by the use of abstract values in place of the actual computed values. Unfortunately, there are no published works with qualitative results on abstract interpretation of Java Card applets and there is no evidence that this technique is appropriate for analyzing security guarantees in multi-applet applications.
A well known static program analysis for Java Card applications (Catano & Huisman, 2002), introduces the use of ESC/Java (2), a static analysis tool for proving specifications, without requiring the analyst to interact with the back-end theorem prover (called Simplify). The provided analysis is neither sound nor complete, but has been found effective in proving absence of runtime exceptions and in verifying relatively simple correctness properties.
In (Almaliotis et. al., 2008) we introduced a static program analysis for temporal safety of Java Card API calls, which is based on computing dataflow facts over the control-flow graph of a Java Card applet. Our approach is implemented in bug detector plugins for the FindBugs tool (Hovemeyer & Pugh, 2004; The FindBugs project site, 2009) and in contrast to (Catano & Huisman, 2002) it does not require annotations in the applet source code. This reduces the verification cost to the applet developers, since they do not have to make explicit all implicit assumptions needed for correctness (e.g. the non-nullness of `buf` in many Java Card API calls). FindBugs bug detector plugins may be distributed together with the Java Card Development kit or by an independent third party. Applet developers use the bug detectors as they are, but they can also extend their open source code in order to develop bug detectors for custom properties. In ESC/Java (2), user-specified properties assume familiarization, (i) with the Java Modeling Language (JML), (ii) with the specificities of the "design by contract" specification technique and (iii) with the corresponding JML based Java Card API specification (Meijer & Poll, 2001). On the other hand, development of new FindBugs bug detectors assumes only Java programming skills that most software engineers already have.

We aim in a holistic fully-automatic verification approach with the only requirement of basic Java programming skills, perhaps at the cost of a few false positives in the provided results. The most important consequence is the use of a single verification technique, for the three types of security errors that we mentioned.
Recent developments in the theory of static program analysis open promising perspectives towards minimizing the number of false positives in the obtained results. However, as we will see in next sections the current version of FindBugs does not offer the necessary support for implementing the most advanced analyses published in the related work.

Static program analysis basics

Static program analyses, in general, explore the different execution paths that can take place, when the program is executed. In fact, they are based on a control flow graph (CFG) representation of the analyzed program, where the nodes of the graph are basic blocks, i.e. sequences of instructions that will always be executed without the possibility that any instruction(s) will be skipped. Edges in the control flow graph are directed and represent potential control flow paths between basic blocks. Back edges in a control flow graph represent potential loops.
A call graph represents potential control flow between methods. Nodes in the graph represent methods and directed edges represent the potential for one method to invoke another.
Dataflow analyses examine the way data move through a program by traversing a method's control flow graph, in order to estimate conservative approximations about facts that are true in each location of it. Facts are mutable, but they have to form a lattice.
An advanced static program analysis consists of at least two major pieces: an intraprocedural analysis component for analyzing an individual method and an interprocedural analysis component that operates across an entire program, flowing information from the caller to its callees and vice versa. FindBugs does not offer direct support for interprocedural analyses, which is a basic requirement for analyzing multi-applet applications. However, as we will see it was eventually possible to overcome this problem.
Interprocedural analysis in the FindBugs framework is still a challenging problem, if we want to reduce the number of false positives we currently get. The reason is that the behavior of each method is dependent upon the context in which it is called and the current version of FindBugs does not provide support for determining the circumstances and conditions under which a method runs. Another aspect related to context sensitivity is the fact that the number of paths through the code grows exponentially with the number of conditionals and for this reason when explicitly gathering facts along each path this may result in an unacceptably slow analysis. Advanced static program analyses try to alleviate this problem by allowing paths to share information about common subpaths, as well as by techniques that allow for implicit enumeration of paths.

Typestate tracking & tainted object propagation

The analyses presented here focus on the use of FindBugs for the three types of security errors that are needed to check in a Java Card multi-applet application. Also, we discuss recent developments towards advanced context-sensitive analyses with high precision, which can be potentially implemented in future versions of FindBugs that will potentially provide appropriate support. Static program analyses for security errors with a particular interest for the Java Card environment lie into two broad categories, namely:

- typestate tracking (Strom & Yemini, 1986), for illicit method invocations spanning the contexts of different applets, as well as for violations of temporal safety in Java Card API calls and
- taint propagation (Livshits & Lam, 2005; Sabelfeld & Myers, 2003; Haldar et. al. 2005) for unwanted information flows to third parties.

Temporal safety of API calls and applet interactions concern rules about their ordering that are possibly associated with constraints on the data values visible at the boundary of the interacting parties. Temporal safety violations are typically detected by typestate tracking. The typestate is a refinement of the concept of type: whereas the type of a data object determines the set of operations ever permitted on the object, typestate determines the subset of these operations, which are performed in a particular context. Typestate tracking aims to statically detect syntactically legal, but semantically undefined execution sequences.

On the other hand, a tainted object propagation problem consists of a set of source descriptors and sink descriptors. To represent the fact that data can be trusted for some purposes but not for others, different varieties of tainted data can be modeled as carriers of different taint flags. Source descriptors define program locations, where tainted data enter the applet and the different source descriptors introduce data with different taint flags. Sink descriptors define program locations that should not receive tainted data or data carrying a certain type of taint. Taint propagation analyses aim in detecting information flows, such that a variation of "confidential" or "high" input to some applet causes a variation of "public" or "low" output. This occurs when the flow of taint terminates in a sink point, as a consequence of not having stopped the propagation between objects by some sanitization method. Typically, sanitization may be performed by creating a new fresh sanitized object.

<u>Static analysis with the FindBugs framework</u>

FindBugs is a tool and framework that applies static analyses on the Java (Java Card) bytecode in order to detect bug patterns, i.e. to detect "places where code does not follow correct practice in the use of a language feature or library API" (Hovemeyer & Pugh, 2004). In general, FindBugs bug detectors behave according to the Visitor design pattern: each detector visits each class and each method in the application under analysis. The framework comes

with many analyses built-in and classes and interfaces that can be extended to build new analyses. In our work, we exploit the already provided intra-procedural control flow analysis that transforms the analyzed bytecode into control flow graphs (CFGs), in order to support the property analyses that we present in next sections.
The bug pattern detectors are implemented using the Byte Code Engineering Library - BCEL (Dahm, 2001), which provides infrastructure for analyzing and manipulating Java class files. In essence, BCEL offers to the framework data types for inspection of binary Java (Java Card) classes. One can obtain methods, fields, etc. from the main data types, `JavaClass` and `Method`. The project source directories are used to map the reported warnings back to the Java source code.
Bug pattern detectors are packaged into FindBugs plugins that can use any of the built-in FindBugs analyses and in effect extend the provided FindBugs functionality without any changes to its code. A plugin is a `jar` file that contains detector classes, analysis classes and the following meta-information: (i) the plugin descriptor (`findbugs.xml`) declaring the bug patterns, the detector classes, the detector ordering constraints and the analysis engine registrar, (ii) the human-readable messages (in `messages.xml`), which are the localized messages generated by the detector. Plugins are easily activated in the analyst's FindBugs installation by copying the jar file into the proper location of the user's file system.
FindBugs applies the loaded detectors in a series of `AnalysisPasses`. Each pass executes a set of detectors selected according to declared detector ordering constraints. In this way, FindBugs distributes the detectors into `AnalysisPasses` and forms a complete `ExecutionPlan`, i.e., a list of `AnalysisPasses` specifying how to apply the loaded detectors to the analyzed application classes. When a project is analyzed, FindBugs runs through the following steps:

1. Reads the project
2. Finds all application classes in the project
3. Loads the available plugins containing the detectors
4. Creates an execution plan
5. Runs the FindBugs algorithm to apply detectors to all application classes

The basic FindBugs algorithm in pseudo-code is:

```
for each analysis pass in the execution plan do
    for each application class do
        for each detector in the analysis pass do
            apply the detector to the class
        end for
    end for
end for
```

All detectors use a global cache of analysis objects and databases. An analysis object (accessed by using a `ClassDescriptor` or a `MethodDescriptor`) stores facts about a class or method, for example the results of a null-pointer dataflow analysis on a method. On the other hand, a database stores facts about the entire program, e.g. which methods unconditionally dereference parameters. All detectors implement the `Detector` interface, which includes the `visitClassContext()` method that is invoked on each application class. Detector classes (i) request one or more analysis objects from the global cache for the analyzed class and its methods, (ii) inspect the gathered analysis objects and (iii) report warnings for suspicious situations in code. When a `Detector` is instantiated its constructor gets a reference to a `BugReporter`. The `Detector` object uses the associated `BugReporter`, in order to emit warnings for the potential bugs and to save the detected bug instances in `BugCollection` objects for further processing.

Static verification of Java Card applications by typestate tracking

Static verification of interacting applets is illustrated with an example application that includes a purse applet and two loyalty applets that award bonus points. The purse applet keeps a `balance` that is updated upon requests from the environment that allow the card owner to purchase goods.

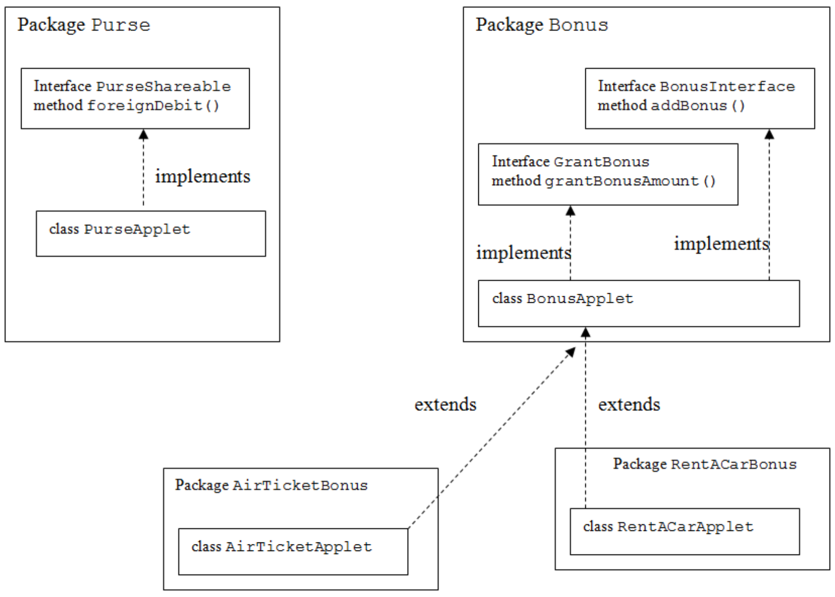


Figure 1. A purse applet and two loyalty applets that award bonus points

The interface method `foreignDebit()` is invoked by the loyalty applets that reside in separate packages, in order to transfer according to some fixed rate part of the bonus points back to the purse. We consider two loyalty applets, namely the `AirTicketBonus` and the `RentACarBonus` that basically implement the same interfaces. The interface method `addBonus()` is invoked by the purse applet, whenever there is a need to notify a loyalty applet for an occurred balance update. Finally, we consider the possibility for a loyalty applet to have an agreement with other loyalty applets, in order to share bonus points. This is achieved by a direct loyalty applet to loyalty applet communication using the interface method `grantBonusAmount()`. Figure 2 introduces the class diagram for the discussed application case.

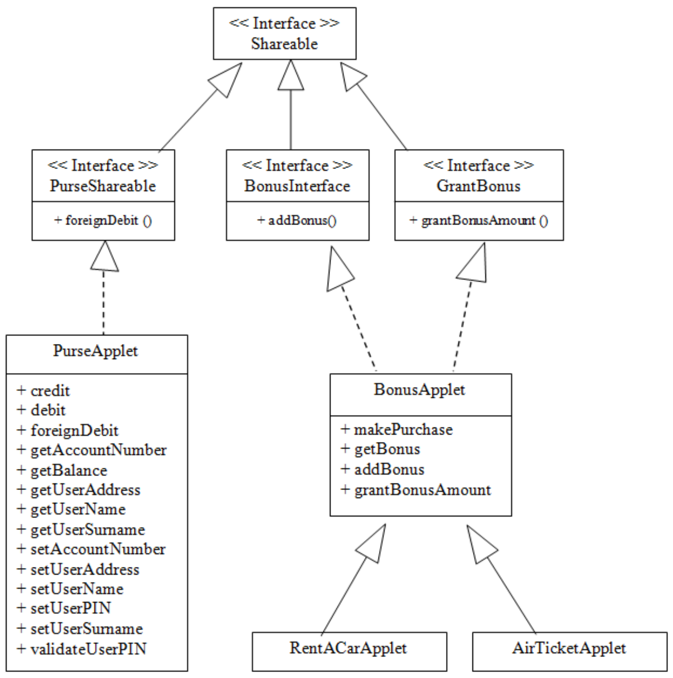


Figure 2. Class diagram for the application of the purse applet and the two loyalty applets

Typestate tracking for verifying temporal restrictions of inter-applet communication
Temporal restrictions of inter-applet communication concern rules about the ordering of method invocations, which span the contexts of different applets. Typestate tracking makes it possible to detect such illicit method calls that – as we already noted - cannot be caught by the Java Card firewall.
Bug detectors for verifying inter-applet communication track the state of the property of interest and at the same time track the so called execution state, i.e. the values of all program variables. In the considered application, example properties are the following:

- Methods declared in shareable interfaces are neither transitive nor recursive. Thus, the method `addBonus()` is not allowed to cause a direct or indirect call to `addBonus()` for the same or for a different loyalty applet.
- Method `foreignDebit()` is called at most once within a transaction.
- The method `grantBonusAmount()` is invoked only through a call to `addBonus()` for some loyalty applet and it is never called directly from the purse applet.

Correctness properties of inter-applet communication are captured in appropriate state machines that recognize finite execution traces with improper use of the methods declared in shareable interfaces. Figure 3 introduces the state machine for the first mentioned property. Accurate tracking of the execution state can be very expensive, because this implies tracking every branch in the control-flow, in which the values of the examined variables differ along the branch paths. The resulted search space may grow exponentially or even become infinite.

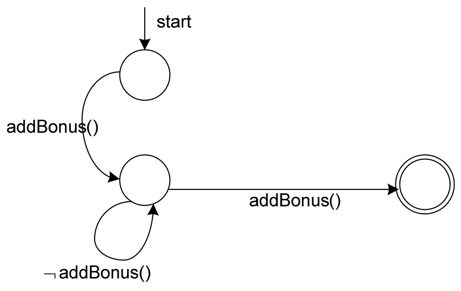


Figure 3. Transitive or recursive invocation of a loyalty applet method

Bug detectors have to take into account two distinct cases of property violations:

1. Intraprocedural property violations that may be detected by simple bytecode scanning or CFG-based analyses that basically follow the states of the property state machine.
2. Interprocedural property violations, which are detected by extending the CFG-based and call graph analysis functions provided in FindBugs.

As we saw in previous section, FindBugs static analyses are applied by default to individual class contexts. The following pseudo-code reflects the functionality of the `visitClassContext()` method of a typical CFG-based detector.

```
for each method in the class do
request a CFG for the method from the ClassContext
request one or more analysis objects on the method from the ClassContext
    for each location in the method do
        get the dataflow facts at the location
        inspect the dataflow facts
        if a dataflow fact indicates an error then
            report a warning
        end if
    end for
end for
```

The basic idea is to visit each method of the analyzed class in turn, requesting some number of analysis objects. After getting the required analyses, the detector iterates through each location in the CFG. A location is the point in execution just before a particular instruction is executed (or after the instruction, for backwards analyses). At each location, the detector checks the dataflow facts to see if anything suspicious is going on. If suspicious facts are detected at a location the detector issues a warning.

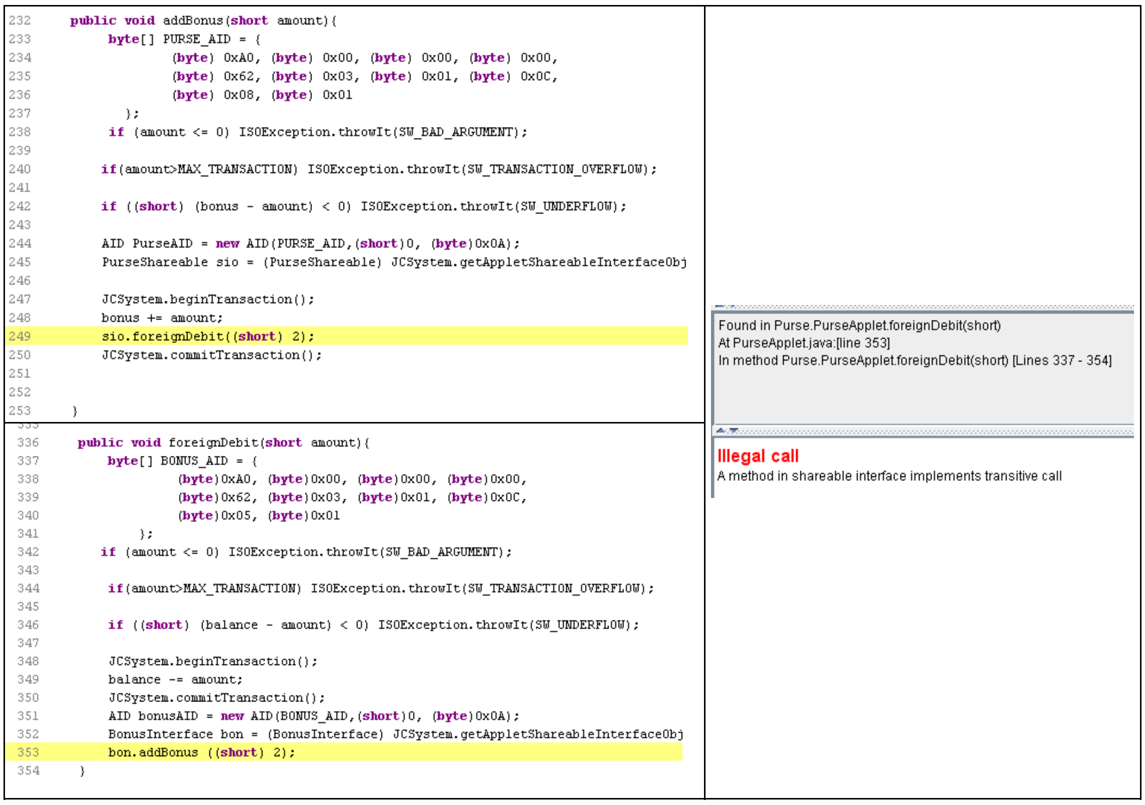
For interprocedural typestate tracking, we have to bypass the restriction that FindBugs analyses are normally applied to individual class contexts. We developed the class `InterCallGraph,` which implements a single call graph structure for the whole set of application classes and thus allows

detecting property violations that may be caused by nested method invocations.
For the property of Figure 3 we introduce the path-insensitive bug detector shown in the following pseudocode.

```
//populate black list
request all implemented interfaces for the current class
for each implemented interface do
    request all extended interfaces (parent interfaces)
    for each parent interface do
        if parent interface is "Shareable" then
            add all methods of implemented interface in the blacklist
        end if
    end for
end for
//costruction of the call graph
request all methods of all classes
make nodes for these methods
for each class in the program do
    for each method in class do
        scan for calls and make a link for each call between methods-nodes
    end for
end for

for each method in the blacklist do
for each method in the class do
    start a Depth First Search from the corresponding graph node:
    if method of the node is in the current node from blacklist then
        if the same method is visited again then
            report the detected bug
        end if
    end if
end for
end for
```

The bug detector first creates a black list, which is used to record all methods declared in implemented interfaces that inherit from “ShareableInterface”. This allows to statically verifying the property of Figure 3, for all methods declared in shareable interfaces. The bug detector discovers both intraprocedural and interprocedural property violations. Repeated calls of methods that bypass the application firewall may also occur as a method invocation enclosed in a basic block of a for/while or a do . . . while loop. Method CFGs are inspected for this particular CFG pattern and when detected, this causes a transition to the final state of the property violation automaton. For an occurred state transition from the initial state, the bug detector starts a depth first search from the current node of the instantiated `InterCallGraph`, in order to detect potential recursive or transitive calls to the black listed method. Figure 4 shows the FindBugs response for a transitive call to the method `addBonus()`.



```
232     public void addBonus(short amount){
233          byte[] PURSE_AID = {
234                   (byte) 0xA0, (byte) 0x00, (byte) 0x00, (byte) 0x00,
235                   (byte) 0x62, (byte) 0x03, (byte) 0x01, (byte) 0x0C,
236                   (byte) 0x08, (byte) 0x01
237            };
238          if (amount <= 0) ISOException.throwIt(SW_BAD_ARGUMENT);
239
240         if(amount>MAX_TRANSACTION) ISOException.throwIt(SW_TRANSACTION_OVERFLOW);
241
242         if ((short) (bonus - amount) < 0) ISOException.throwIt(SW_UNDERFLOW);
243
244         AID PurseAID = new AID(PURSE_AID,(short)0, (byte)0x0A);
245         PurseShareable sio = (PurseShareable) JCSystem.getAppletShareableInterfaceObj
246
247         JCSystem.beginTransaction();
248         bonus += amount;
249         sio.foreignDebit((short) 2);
250         JCSystem.commitTransaction();
251
252
253     }
```

```
336      public void foreignDebit(short amount){
337          byte[] BONUS_AID = {
338                   (byte)0xA0, (byte)0x00, (byte)0x00, (byte)0x00,
339                   (byte)0x62, (byte)0x03, (byte)0x01, (byte)0x0C,
340                   (byte)0x05, (byte)0x01
341              };
342         if (amount <= 0) ISOException.throwIt(SW_BAD_ARGUMENT);
343
344          if(amount>MAX_TRANSACTION) ISOException.throwIt(SW_TRANSACTION_OVERFLOW);
345
346          if ((short) (balance - amount) < 0) ISOException.throwIt(SW_UNDERFLOW);
347
348          JCSystem.beginTransaction();
349          balance -= amount;
350          JCSystem.commitTransaction();
351          AID bonusAID = new AID(BONUS_AID,(short)0, (byte)0x0A);
352          BonusInterface bon = (BonusInterface) JCSystem.getAppletShareableInterfaceObj
353          bon.addBonus ((short) 2);
354      }
```

```
Found in Purse.PurseApplet.foreignDebit(short)
At PurseApplet.java:[line 353]
In method Purse.PurseApplet.foreignDebit(short) [Lines 337 - 354]
```

```
Illegal call
A method in shareable interface implements transitive call
```

Figure 4. A method declared in a shareable interface triggers an indirect call to itself

Typestate tracking for verifying temporal safety of Java Card API calls
Contrary to ordinary Java programs that have a single `main()` entry point, Java Card applets have several entry points, which are called when the card receives various application (APDU) commands. These entry points roughly match the different phases that an applet can be in: (i) loading, (ii) installation, (iii) personalization, (iv) selectable, (v) blocked and (vi) dead.
In a Java Card, any exception can reach the top level, i.e. the applet entry point invoked by the Java Card Runtime Environment (JCRE). In this case, the currently executing command is aborted and the command, which in general is not completed yet, is terminated by an appropriate status word: if the exception is an `ISOException`, the status word is assigned the value of the reason code for the raised exception, whereas in all other cases the reason code is `0x6f00` corresponding to “no precise diagnosis”.
An exception in an applet’s entry point can reveal information about the behavior of the application and in principle it should be forbidden. In practice, whereas an `ISOException` is usually explicitly thrown by the applet code using throw, a potentially unhandled exception is implicitly raised when executing an API method call that causes an unexpected error. This may result in leaving the applet in an unpredicted and ill state that can possibly violate the application’s security properties.
Unhandled exceptions are detected by looking for an exception thrower block preceding the instruction by which typestate tracking reaches the final state (Figure 5). Access to an exception handler block (if any) is possible through a

handled exception edge. In FindBugs, method `isExceptionThrower()` detects an exception thrower block and method `isExceptionEdge()` determines whether a CFG edge is a handled exception edge.

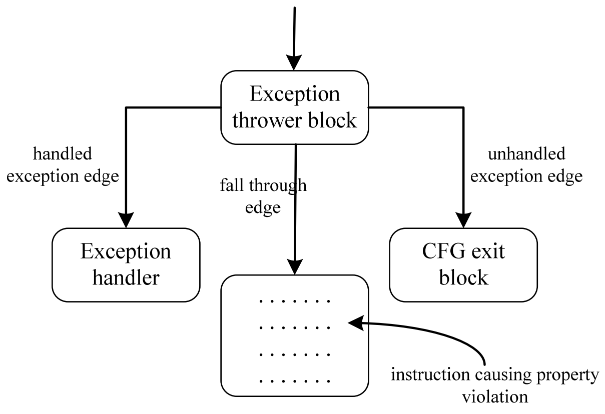


Figure 5. CFG pattern to find unhandled exception edges

Potentially unhandled exceptions are usually caused by violations of temporal restrictions in the use of Java Card API calls. Temporal safety violations are captured by typestate tracking based on an appropriate state machine, which recognizes finite execution traces with improper calls. In (Almaliotis et. al., 2008) we introduced a FindBugs bug detector that detects unhandled instances of `APDUException`, for improper use of the `setOutgoing()` call.

Constraints on data values visible at the boundary of the interacting parties
Dataflow analysis is the basic means to statically verifying the correctness of the called methods' arguments and it is necessary, when temporal safety involves constraints on the data values accessed at the boundary of the interacting parties (communicating applets or an applet interacting with the Java Card API). In this case, the described static program analyses are combined with dataflow analyses like the one shown here.
A dataflow analysis estimates conservative approximations about facts that are true in each location of a CFG. Facts are mutable, but they have to form a lattice. In FindBugs, the `DataflowAnalysis` interface, which is shown in Figure 6 is the super-type of all concrete dataflow analysis classes. It defines methods for creating, copying, merging and transferring dataflow facts. Transfer functions take dataflow facts and model the effects of either a basic block or a single instruction depending on the implemented dataflow analysis. Merge functions combine dataflow facts when control paths merge. The `Dataflow` class and its subclasses implement: (i) a dataflow analysis algorithm based on a CFG and an instance of `DataflowAnalysis`, (ii) methods providing access to the analysis results.
We are particularly interested for the `FrameDataflowAnalysis` class that forms the base for analyses, which model values in local variables and the operand stack. Dataflow facts for derived analyses are subclasses of the class `Frame`, whose instances represent the Java stack frame at a single CFG

location. In a Java stack frame, both stack operands and local variables are considered to be "slots" that contain a single symbolic value.

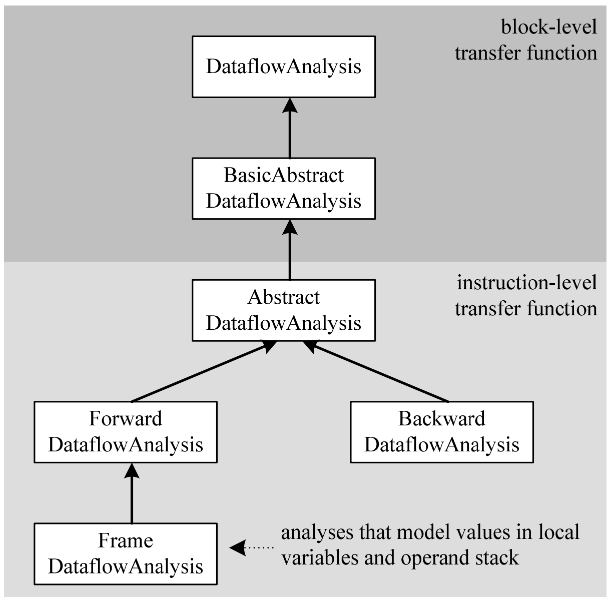

block-level
transfer function
DataflowAnalysis
BasicAbstract DataflowAnalysis
Abstract DataflowAnalysis
instruction-level
transfer function
Forward DataflowAnalysis
Backward DataflowAnalysis
Frame DataflowAnalysis
analyses that model values in local variables and operand stack

Figure 6. FindBugs base classes for dataflow analyses

The built-in frame dataflow analyses used in static verification of the called methods arguments are:

- The `TypeAnalysis` that performs type inference for all local variables and stack operands.
- The `ConstantAnalysis` that computes constant values in CFG locations.
- The `IsNullValueAnalysis` that determines which frame slots contain definitely-null values, definitely non-null values and various kinds of conditionally-null or uncertain values.
- The `ValueNumberAnalysis` that tracks the production and flow of values in the Java stack frame.

The class hierarchy of Figure 6 and the mentioned built-in dataflow analyses form a generic dataflow analysis framework, since it is possible to create new kinds of dataflow analyses that will use as dataflow facts objects of user-defined classes.

A bug detector exploits the results of a particular dataflow analysis on a method by getting a reference to the `Dataflow` object that was used to execute the analysis. There is no direct support for interprocedural analysis, but there are ways to overcome this restriction. More precisely, analysis may be performed in multiple passes. A first pass detector will compute method summaries (e.g. method parameters that are unconditionally dereferenced, return values that are always non-null and so on), without reporting any warnings and a second pass detector will use the computed method summaries as needed. However, this approach is not convenient for implementing context-sensitive interprocedural dataflow analyses.

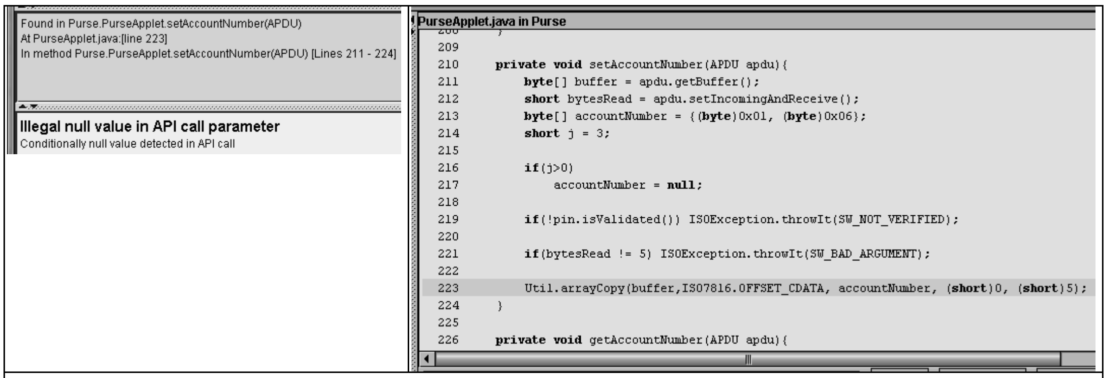
In the following paragraphs, we present a bug detector for unhandled API exceptions concerned with the correctness of arguments in method calls. Consider the following method:

```
short arrayCopy(    byte[] src, short srcOff,
                    byte[] dest, short destOff, short length)
```

A `NullPointerException` is raised when either `src` or `dest` is `null`. Also, when the copy operation accesses data outside the array bounds the `ArrayIndexOutOfBoundsException` is raised. This happens either when one of the parameters `srcOff`, `destOff` and `length` has a negative value or when `srcOff+length` is greater than `src.length` or when `destOff+ length` is greater than `dest.length`. We provide the pseudo-code of the `visitClassContext()` method for the detector of unhandled exceptions raised by invalid `arrayCopy` arguments:

```
for each method in the class do
    request a CFG for the method
    get the method's ConstantDataflow from ClassContext
    get the method's ValueNumberDataflow from ClassContext
    get the method's IsNullValueDataflow from ClassContext
    for each location in the method do
        get instruction handle from location
        get instruction from instruction handle
        if instruction is not instance of invoke static then
            continue
        end if
        get the invoked method's name from instruction
        get the invoked method's signature from instruction
        if invoked method is arrayCopy then
            get ConstantFrame (fact) at current location
            get ValueNumberFrame (fact) at current location
            get IsNullValueFrame (fact) at current location
            get the method's number of arguments
            for each argument do
                get argument as Constant, ValueNumber, IsNullValue
                if argument is constant then
                    if argument is negative then
                        report a bug
                    end if
                else
                if argument is not method return value
                  nor constant then
                    if argument is not definitely not null then
                        report a bug
                    end if
                end if
            end for
        end if
    end for
end for
```

Figure 7 demonstrates how the detector responds in two different property violation cases. In the first case, `PurseApplet` calls `arrayCopy` with null value for the parameter `accountNumber`. It is important to note that it is not possible to determine by static analysis the correctness of the method call for all of the mentioned criteria, because buffer gets its value at run time by the JCRE. However, a complete FindBugs bug detector could generate a warning for the absence of an appropriate exception handler. In the second test case, parameter `offset` is assigned an unacceptable value.



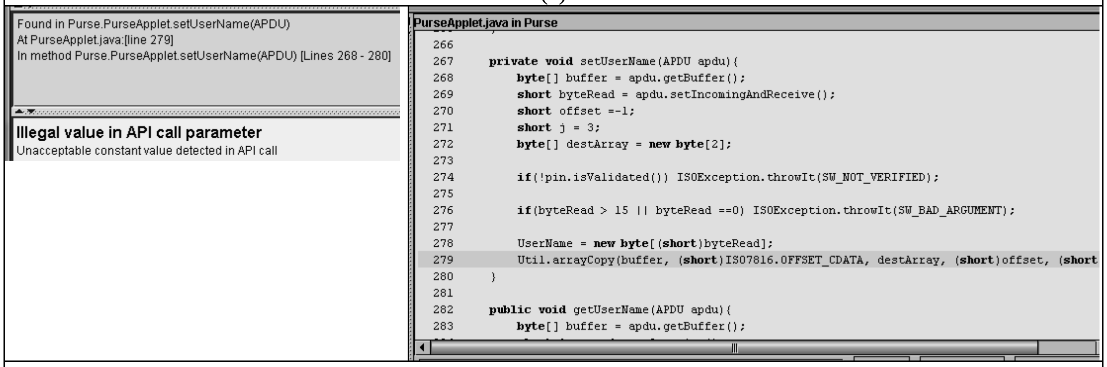
Found in Purse.PurseApplet.setAccountNumber(APDU)
At PurseApplet.java:[line 223]
In method Purse.PurseApplet.setAccountNumber(APDU) [Lines 211 - 224]

Illegal null value in API call parameter
Conditionally null value detected in API call

PurseApplet.java in Purse

```
209
210     private void setAccountNumber(APDU apdu){
211         byte[] buffer = apdu.getBuffer();
212         short bytesRead = apdu.setIncomingAndReceive();
213         byte[] accountNumber = {(byte)0x01, (byte)0x06};
214         short j = 3;
215
216         if(j>0)
217             accountNumber = null;
218
219         if(!pin.isValidated()) ISOException.throwIt(SW_NOT_VERIFIED);
220
221         if(bytesRead != 5) ISOException.throwIt(SW_BAD_ARGUMENT);
222
223         Util.arrayCopy(buffer,ISO7816.OFFSET_CDATA, accountNumber, (short)0, (short)5);
224     }
225
226     private void getAccountNumber(APDU apdu){
```

(a)



Found in Purse.PurseApplet.setUserName(APDU)
At PurseApplet.java:[line 279]
In method Purse.PurseApplet.setUserName(APDU) [Lines 268 - 280]

Illegal value in API call parameter
Unacceptable constant value detected in API call

PurseApplet.java in Purse

```
266
267     private void setUserName(APDU apdu){
268         byte[] buffer = apdu.getBuffer();
269         short byteRead = apdu.setIncomingAndReceive();
270         short offset =-1;
271         short j = 3;
272         byte[] destArray = new byte[2];
273
274         if(!pin.isValidated()) ISOException.throwIt(SW_NOT_VERIFIED);
275
276         if(byteRead > 15 || byteRead ==0) ISOException.throwIt(SW_BAD_ARGUMENT);
277
278         UserName = new byte[(short)byteRead];
279         Util.arrayCopy(buffer, (short)ISO7816.OFFSET_CDATA, destArray, (short)offset, (short
280     }
281
282     public void getUserName(APDU apdu){
283         byte[] buffer = apdu.getBuffer();
```

(b)

Figure 7. Illegal use of `arrayCopy` detected with (a) null value parameter and (b) unacceptable constant value parameter

<u>Tainted object propagation</u>

In multi-applet Java Card applications, the leakage of "confidential" data to third parties is an information flow analysis problem. The basic idea behind using tainted object propagation for detecting information flow is to statically check that flow of information between variables is consistent with the trust relationships of the application providers. Although the accurate detection of information flow is undecidable, static analysis can over-approximate information flows, in order to ensure absence of "confidential" data leakage. Tainted object propagation can be formulated as a dataflow analysis problem and can be solved efficiently using an iterative algorithm. We consider only two possible lattice values, namely TAINTED and NOT TAINTED. On the lattice, the greatest lower bound of the two elements is defined such as anything that meets a tainted value becomes tainted. An appropriate static analysis will include the following steps:

1. Consider all variables as NOT TAINTED.
2. Annotate applet “confidential” source points as TAINTED.
3. Propagate the tainted values through the applet code. If a tainted value is used in an expression, mark the result of the expression as TAINTED.
4. Repeat step 3 until a fixed point is reached.
5. Find all TAINTED sink descriptors and report them.

It is important to note that the aforementioned static analysis approach detects only the explicit information flows. For any two variables, say `x` and `y`, an explicit information flow from `x` to `y` occurs, if the value of `x` is assigned to `y`, as in `y = x`. On the other hand, there is an implicit flow from `x` to `y`, if the value of `x` is used to evaluate the outcome of a conditional, which then controls an assignment to `y`, i.e.

```
if (x > 0) then
       y = 1
else
       y = 0;
```

For statically verifying explicit information flows towards sink descriptors, it is necessary to know what runtime objects these descriptors may refer to. Let us consider the following program fragment:

```
1 byte[] buffer = apdu.getBuffer();
2 byte[] amount = new byte[2];

3 Util.arrayCopy(   buffer,(short)ISO7816.OFFSET_CDATA,
                    amount,(short)0,(short)2);
4 byte[] credit;
  . . . . . . .
5 short creditS = (short)(credit[1] & 0x0F);
6 creditS += (short)(credit[1] & 0xF0);
7 creditS += (short)(credit[0]<<8);
8 BonusInterface
   sio=(BonusInterface)JCSystem.getAppletShareableInterfaceObject( bonusAID,
                                                               (byte)0x00);
9 sio.addBonus((short)creditS);
```

In the shown example, `apdu.getBuffer()` returns the reference `buffer` to some tainted data, i.e. the data contained in the binary array buffer with the APDU command. Also, `amount` becomes tainted, because it is derived from `buffer` by a call to `arrayCopy` (line 3). Finally, `creditS` is derived from `credit` and is subsequently passed to the sink method `addBonus` (line 9). Unless we know that variables `amount` and `credit` may never refer to the same object, we would have to conservatively assume that they may. Since `amount` is tainted, variable `credit` may also refer to a tainted object.
The general problem of determining what objects a given program variable may refer to is addressed by an appropriate pointer or points-to analysis. An unbounded number of objects may be allocated by the program at runtime and in order to compute a finite result points-to analysis statically approximates dynamic program objects with a finite set of values that in

FindBugs are instances of the class `ValueNumber`. Thus, points-to analyses are based on `ValueNumberAnalysis` (Hansen & Wahlgreen, 2007), which tracks the production and flow of values in the Java stack frame.

Future Research Directions

Static program analyses based on JSR-305 type qualifiers
Annotations play an important role in software defect detection especially in problems, where there is a need to state (implicit) design decisions about what the code is supposed to do. Java Specification Request 305 (Hovemeyer & Pugh, 2007) defines standard annotations that allow designers/developers to describe design intents in a way that will make them amenable to program analysis by FindBugs and other tools. JSR-305 annotations are being proposed for inclusion as standard in Java 7 (Harold, 2008).
A well known implicit design intent that cannot be documented without an annotation language like JSR-305 is the requirement, some method parameter to always be non null. FindBugs provides the annotation type @NonNull that may be applied into some field, method, parameter or local variable.
One of the most interesting aspects of JSR-305 is that it provides meta-annotations, which allow developers to define type qualifiers (Foster et. al., 1999). The added type qualifiers extend the language type rules, in order to model the flow of qualifiers through the program, where each qualifier or set of qualifiers comes with additional type constraints that capture its semantics. The latest version of FindBugs includes support for type qualifier dataflow analyses in the form of an abstract class. The functionality of abstract class `TypeQualifierDataflowAnalysis` is based on points-to information provided by a `ValueNumberAnalysis`, as well as on an appropriate structure that stores facts for source-to-sink mappings. There is also a CFG-based bug detector that exploits the provided support for checking user-defined type qualifiers.
For security properties that are formulated as information flow problems, JSR-305 introduces the following annotations:

- @Tainted
- @Untainted
- @Detainted

In a Java Card multi-applet application, data from outside of an applet will be marked as @Tainted, as opposed to data from inside the applet that will be marked @Untainted. Tainted data that has been sanitized e.g. by passing it to some sort of escaping function can be annotated as @Detainted. These annotations are sufficient for an appropriate tainted object propagation analysis to follow the path of data through the applet, in order to ensure that tainted data never reaches a method invoked into another applet through a shareable interface:

```
private void function(@Tainted APDU apdu){
. . . . . . .
BonusInterface
 sio=(BonusInterface)JCSystem.getAppletShareableInterfaceObject( bonusAID,
                                                                (byte)0x00);
sio.addBonus(@Untainted (short)creditS);
}
```

Precise and scalable static program analyses
Static program analysis has the potential to provide automated verification solutions for the whole range of security properties that arise in Java Card multi-applet applications. When using FindBugs, the real challenge towards efficient use of the described program analysis techniques is to exploit the benefits and the extensibility prospects of the FindBugs open source analysis support and at the same time to look for ways for implementing sophisticated and precise analyses that reduce false positives and at the same time scale to real Java Card programs.
For typestate tracking, an interesting source of inspiration is the SAFE project (The SAFE project, 2009) at the IBM Research Labs. The heuristics applied in SAFE are reported in (Fink et. al., 2006). In that work the authors propose a composite verifier built out of several composable verifiers of increasing precision and cost. In this setting, the composite verifier stages analyses, in order to improve efficiency without compromising precision. The early stages use the faster verifiers to reduce the workload for later, more precise, stages. Prior to any path-sensitive analysis, the first stage prunes the verification scope using an extremely efficient path-insensitive error path feasibility check.
As we already noted, in FindBugs, interprocedural dataflow analyses may be implemented by multiple-pass bug detectors, where the first pass computes method summaries. Since the precision of interprocedural analyses is improved by eliminating paths that are invalid, because of the method call and return structure, we believe that the possibility to implement truly context-sensitive analyses is a major concern for FindBugs program analyses.

Conclusion

We explored static program analyses with FindBugs, which can provide a credible automatic verification approach for the security concerns raised in Java Card multi-applet applications. When compared with existing security verification alternatives, our approach does not require specialized formal analysis skills to the application developer, but basic Java programming skills that most software engineers already have. Moreover, it provides a single fully-automatic verification prospect for all the three types of security problems discussed, in place of diverse costly approaches, for the different verification tasks.
FindBugs provides an open source analysis support that may be exploited in the development of new pluggable bug detectors, which can be easily installed in the static analysis tool suite. The presented bug detectors are available online in (The S-OMA SMART CARDS Project, 2009). Bug detector plugins may be distributed together with the Java Card Development kit or

by an independent third party. Applet developers are still capable to extend the bug detectors open source code, in order to develop bug detectors for custom properties.
All these attractive features open a new perspective for verifying Java Card multi-applet applications that by definition are security critical. We highlighted current restrictions in the proposed verification approach, but we believe that due to the high potential of an open source analysis support these restrictions will be eliminated in the foreseeable future.

References

Akdemir, I. O. (1998). *An implementation of secure flow type inference for a subset of Java.* Unpublished Master thesis, Naval Postgraduate School, Monterey, California.

Almaliotis, V., Loizidis, A., Katsaros, P., Louridas, P. & Spinellis, D. (2008). Static program analysis for Java Card applets. In G. Grimaud & F.-X. Standaert (Ed.), *Proc. of the 8th IFIP Smart Card Research and Advanced Application Conference (CARDIS)* (pp. 17-31), Springer LNCS 5189.

Beckert, B. & Mostowski, W. (2003). A program logic for handling Java Card's transaction mechanism. *Proc. of 6th Int. Conference on Fundamental Approaches to Software Engineering (FASE'03)* (pp. 246-260), Springer LNCS 2621.

Bieber, P., Cazin, J., Girard, P., Lanet, J.-L., Wiels, V. & Zanon, G. (2002). Checking secure interactions of smart card applets: extended version. *Journal of Computer Security, 10*, 369-398.

Breunesse, C. B., Catano, N., Huisman, M. & Jacobs, B. (2005). Formal methods for smart cards: an experience report. *Science of Computer Programming, 55*, 53-80.

Burdy, L., Requet, A. & Lanet, J. L. (2003). Java applet correctness: a developer-oriented approach. *Proc. of Formal Methods Europe (FME)*, Springer LNCS 2805.

Catano, N. & Huisman, M. (2002). Formal specification and static checking of Gemplus's electronic purse using ESC/Java. In G. Goos, J. Hartmanis & J. van Leeuwen (Ed.), *Proc. of Formal Methods Europe (FME'02)* (pp. 272-289), Springer LNCS 2391.

Dahm, M. (2001). *Byte code engineering with the BCEL API* (Tech. Rep. B-17-98). Freie University of Berlin, Institute of Informatics, Germany.

The FindBugs project site, Accessed February 18, 2009, in http://findbugs.sourceforge.net

Fink, S., Yahav, E., Dor, N., Ramalingam G., Geay, E. (2006). Effective typestate verification in the presence of aliasing. *Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA)* (pp. 133-144), ACM Press.

Foster, J. S., Fähndrich, M., Aiken, A. (1999). A theory of type qualifiers, *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation (PLDI)* (pp. 192-203), ACM Press.

Haldar, V., Chandra, D. & Franz, M. (2005). Dynamic taint propagation for Java. In D. Thomsen (Ed.), *Proc. of the 21st Annual Computer Security Applications Conference (ACSAC)* (pp. 303-311), IEEE Computer Society.

Hansen, T. J. & Wahlgreen, B. (2007). *Static analysis of concurrent Java programs* (Tech. Rep. IMM-B.Sc-2007-11). Technical University of Denmark, Denmark.

Harold, E. R. (2008, September). *The Open Road: javax.annotation*. java.net: The Source for Java Technology Collaboration, online: http://today.java.net/pub/a/today/2008/09/11/jsr-305-annotations.html

Hovemeyer, D. & Pugh, W. (2004). Finding bugs is easy. *SIGPLAN Notices, 39 (12)*, 92-106.

Hovemeyer, D. & Pugh, W. (2007). Status report on JSR-305: Annotations for software defect detection. *Proc. of Object Oriented Programming Systems Languages and Applications (OOPSLA)* (pp. 799-800), ACM Press.

Jacobs, B., Marche, C. & Rauch, N. (2004). Formal verification of a commercial smart card applet with multiple tools. *Proc. 10th Int. Conference on Algebraic Methodology and Software Technology (AMAST 2004)* (pp. 241-257), Springer LNCS 3116.

The Java Verifier project, Accessed February 28, 2009, in http://www.inria.fr/actualites/inedit/inedit36_partb.en.html

Livshits, V. B. & Lam, M. S. (2005). Finding security vulnerabilities in Java applications with static analysis. *Proc. of the 14th Conference on USENIX Security Symposium* (pp. 271-286).

Marché, C. Paulin-Mohring, C. & Urbain, X. (2004). The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming, 58 (1-2)*, 89-106.

Meijer, H. & Poll, E. (2001). Towards a full formal specification of the JavaCard API. In G. Goos, J. Hartmanis & J. van Leeuwen (Ed.), *Proc. of the Int. Conf. on Research in Smart Cards: Smart Card Programming and Security (E-smart)* (pp. 165-178), Springer LNCS 2140.

Meyer, J. & Poetzsch-Heffter, A. (2000). An architecture for interactive program provers. *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (pp. 63-77), Springer LNCS 1785.

Sabelfeld, A. & Myers, A. C. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, 21(1)*, 5-19.

The SAFE (Scalable And Flexible Error detection) project, Accessed February 28, 2009, http://www.research.ibm.com/safe/

The Security in Open Multi-Application Smart Cards (S-OMA SMART CARDS) Project, Accessed February 28, 2009, http://mathind.csd.auth.gr/smart/

Strom, R. E. & Yemini, S. (1986). Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering 12 (1)*, 157-171.

Van den Berg, J. & Jacobs, B. (2001). The LOOP compiler for Java and JML. *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (pp. 299-312), Springer LNCS 2031.