# Static program analysis for Java Card applets

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

Panagiotis Louridas[2] Diomidis Spinellis[2]

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

[2] *Department of Management Science and Technology*
*Athens University of Economics and Business*
*Patision 76, 104 34 Athens, Greece*
*{louridas,dds}@aueb.gr*

**Abstract.** The Java Card API provides a framework of classes and interfaces that hides the details of the underlying smart card interface, thus relieving developers from going through the swamps of microcontroller programming. This allows application developers to concentrate most of their efforts on the details of application, assuming proper use of the Java Card API calls regarding (i) the *correctness of the methods' invocation targets and their arguments* and (ii) *temporal safety*, i.e. the requirement that certain method calls have to be used in certain orders. Several characteristics of the Java Card applets and their multiple-entry-point program structure make it possible for a potentially unhandled exception to reach the invoked entry point. This contingency opens a possibility of leaving the applet in an unpredictable state that is potentially dangerous for the application's security. Our work introduces automatic static program analysis as a means for the early detection of misused and therefore dangerous API calls. The shown analyses have been implemented within the FindBugs bug detector, an open source framework that applies static analysis functions on the applet bytecode.

**KEYWORDS:** Java Card, static program analysis, temporal safety

## 1 Introduction

Static analysis has the potential to become a credible means for automatic verification of smart card applications, which are security critical by definition. This work explores the adequacy of the FindBugs open source framework for the static verification of correctness properties concerning the API calls used in Java Card applications.

The Java Card API provides a framework of classes and interfaces that hides the details of the underlying smart card interface, thus allowing developers to create applications, called applets, at a higher level of abstraction. The Java Card applet life cycle defines the different phases that an applet can be in. These phases are (i) loading, (ii) installation, (iii) personalization, (iv) selectable, (v) blocked and (vi) dead. A

characteristic of Java Card applets is that many actions can be performed only when an applet is in a certain phase. Also, 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 mentioned lifetime phases.

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.

The present article introduces a static program analysis approach for the detection of misused Java Card method calls. We propose the use of appropriate bug detectors designed for the *FindBugs* static analysis framework. These bug detectors will be specific to the used API calls and will check (i) the correctness of the methods' invocation target and their arguments and (ii) temporal safety in their use.

We introduce the two bug detectors that we developed by applying interprocedural control flow based and dataflow analyses on the Java Card bytecode. Then, we discuss some recent advances in related theory that open new prospects to implement sufficiently precise property analyses.

Our proposal addresses the problem of unhandled exceptions based on *bug detectors that in the future may be supplied by Java Card technology providers*. Applet developers will check their products for correct use of the invoked API calls, without having to rely on advanced formal techniques that require highly specialized analysis skills and that are not fully automated.

Section 2 provides a more thorough view of the aims of this work and reviews the related work and the basic differences of the presented approach. Section 3 introduces static analysis with the FindBugs framework. Section 4 presents the work done on the static verification of Java Card API calls and section 5 reports the latest developments that open new prospects for implementation of efficient static analyses that do not compromise precision. The paper ends with a discussion on our work's potential impact and comments interesting future research prospects.

## 2 Related work on static verification of Java Card applets

Our work belongs to a family of static verification techniques which, in general, *do not guarantee sound and complete analysis*. This means that there is no guarantee that we will find all property violations and also we cannot guarantee the absence of

*false positives*. However, our bug detectors may implement advanced static analyses that eliminate false negatives and minimize false positives (Section 5).

In related works, this sort of analysis cannot be compared with established formal approaches that have been used successfully in static verification of Java Card applets: JACK [1], KeY [2], Krakatoa [3], Jive ([4], [5]) and LOOP ([6], [7]). These techniques aim in precise program verification and they are *not fully automated*. Moreover, they require highly specialized formal analysis skills for the applet developer.

Static analysis alternatives for Java Card program verification include abstract interpretation [8], i.e. a semantics-based description of all possible executions based on abstract values in place of the actual computed values. In [9], the authors introduce the use of ESC/Java (2), a tool for proving specifications at compile time, without requiring the analyst to interact with the back-end theorem prover (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 contrast with [9], our proposal for static program analysis is not based on source code annotations. 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). Instead of this, the static analyses of FindBugs are implemented in the form of tool plugins that may be distributed together with the used 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. Note that in ESC/Java (2), user-specified properties assume familiarization, (i) with the Java Modeling Language (JML), (ii) with the particular "design by contract" specification technique and (iii) with the corresponding JML based Java Card API specification [10]. On the other hand, the development of new FindBugs bug detectors assumes only Java programming skills that most software engineers already have.

User defined bug detectors may be based on an initial set of basic bug detectors concerned with (i) the correctness of the API calls invocation target and their arguments and (ii) the temporal safety in their use. This article is inspired by the ideas presented in [11]. However, the focus on the Java Card API is not the only difference between these two works. The static analysis of [11] is based on stateless calls to a library that reflects the API of interest. Violations of temporal safety for the analyzed API calls, however, can be detected only by a statefull property analysis that spans the whole applet or even multiple applets in the same or different contexts. As we will see in next sections, FindBugs static analyses are applied by default to individual class contexts and this is one of the restrictions we had to overcome.

## 3 Static analysis with the FindBugs framework

FindBugs [12] 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" [13]. 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), which are used in our property analyses and dataflow analyses.

The bug pattern detectors are implemented using the Byte Code Engineering Library (BCEL) [14], 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 containing detector classes and 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 developer'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.

## 4 Static verification of Java Card API calls

The test cases for the bug detectors shown here were derived from an electronic purse applet developed for the purposes of this work. The electronic purse applet adds or removes units of digital currency and stores the personal data of the card owner. Moreover, there is also a bonus applet that interacts with the electronic purse for crediting the bonus units corresponding to the performed transactions. The two applets lie in separate contexts and communicate data to each other through a shareable interface. Both applets are protected by their own PINs. They are accessed through the Java Card Runtime Environment (JCRE) that invokes the `process` method, which in turn invokes the methods corresponding to the inputted APDU commands.

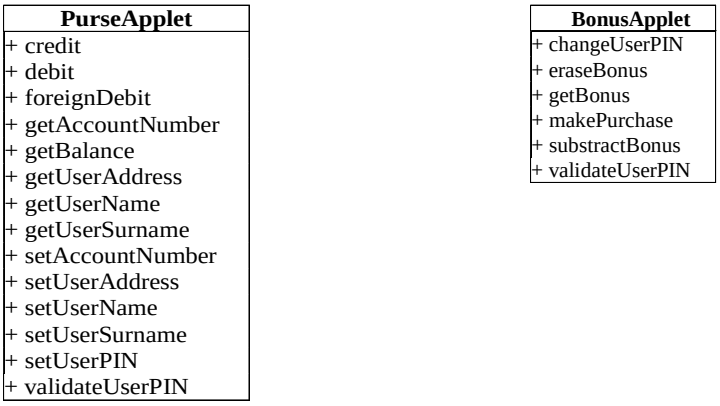



**Figure 1.** Public members of the PurseApplet and the BonusApplet

### 4.1 Bug detectors for the temporal safety of Java Card API calls

Bug detectors for the temporal safety of API calls use a control flow graph (CFG) representation of Java methods to perform static verification that either exploits the builtin dataflow analyses or is based on more sophisticated user-defined analyses. 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.

*Temporal safety of API calls concerns rules about their ordering that are possibly associated with constraints on the data values visible at the API boundary*. Temporal safety properties for the Java Card API are captured in appropriate state machines that recognize finite execution traces with improper use of the API calls. Figure 2 introduces the state machine for a Java Card applet bug raising an APDUException for improper use of the `setOutgoing()` call.

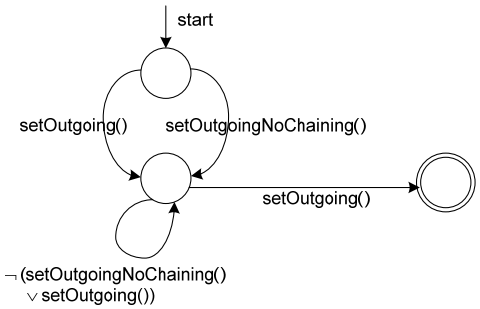



**Figure 2.** Illegal use of `short setOutgoing()` corresponding to a Java Card APDUException

Bug detectors for temporal safety of API calls track *the state of the property* and at the same time track the so-called *execution state*, i.e. the values of all program variables. 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.

For the property of Figure 2 we developed the *path-insensitive bug detector*, shown in this section, to explore the suitability of the FindBugs framework for the static verification of Java Card applets. The more precise *path-sensitive analyses* rely on the fact that for a particular property to be checked, it is likely that most branches in the code are not relevant to the property, even though they affect the execution state of the program. Detectors of this type may be based on heuristics that identify the relevant branches and in this way they reduce the number of potential false positives. Recent advances in path-sensitive static analyses and their applicability in the FindBugs framework are discussed in section 5.

In any applet, it is possible to access an APDU provided by the JCRE, but it is not possible to create new APDUs. This implies that all calls to `setOutgoing()` in a single applet are applied to the same APDU instance and this fact eliminates the need to check the implicit argument of the `setOutgoing()` calls. The developed detectors take into account two distinct cases of property violation:

1. *Intraprocedural property violations* are detected by simple *bytecode scanning* that follows the states of the property state machine (Figure 2)
2. *Interprocedural property violations* are detected by extending the CFG based and *call graph analysis* functions provided in the Findbugs framework.

More precisely, the `InterCallGraph` class we developed makes it possible to construct call graphs including calls that span different class contexts. This extension allowed the detection of nested method calls that trigger the state transitions of Figure 2 either by direct calls to `setOutgoing()` or by nested calls to methods causing reachability of the final state. The following is the pseudo-code of the path-insensitive interprocedural analysis.

```
request the call graph of the application classes
for each method in the call graph do     //mark methods with setOutgoing() call
        if method contains setOutgoing() then
                add method to the black list
        end if
end for
for each method in the class do          //mark methods with nested black method call(s)
        start a Depth First Search from the corresponding graph node:
            if method of the node is in the black list then
                add method to the gray list
                if final state of Fig. 2 is reached then
                        report the detected bug
                end if
            end if
end for
for each method in the class do          //detect property violation caused in a loop
        request a CFG for the method
        check if method has loop, enclosing call of setOutgoing() or a gray method
end for
```

Finally, the methods' CFGs are inspected for loops enclosing method calls that do not cause reachability of the final state by themselves, but they result in a property

violation when encountered in a loop. Figure 3 shows the bytecode patterns matching the use of a loop control flow in a CFG. Unhandled exception violations are detected by looking for an *exception thrower block* preceding the instruction by which we reach the final state (Figure 4). 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.

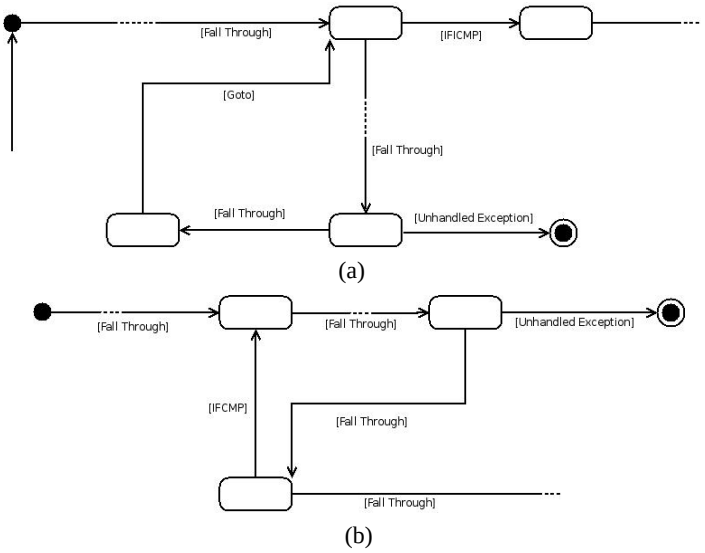



**Figure 3.** CFG patterns with basic blocks corresponding to (a) for/while and (b) do . . . while loop

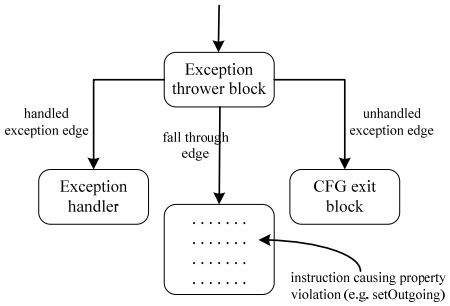
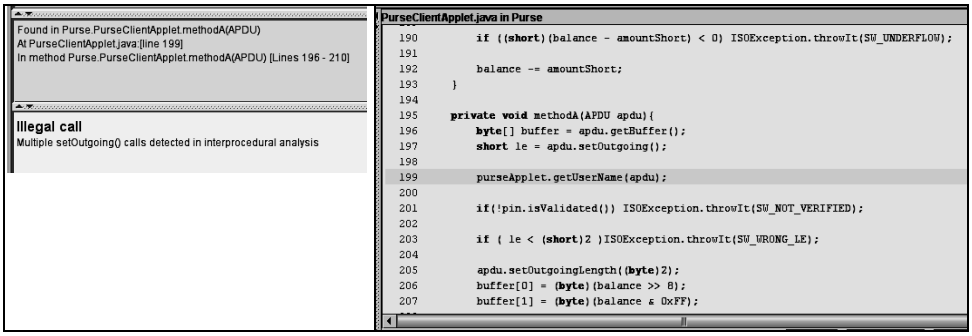
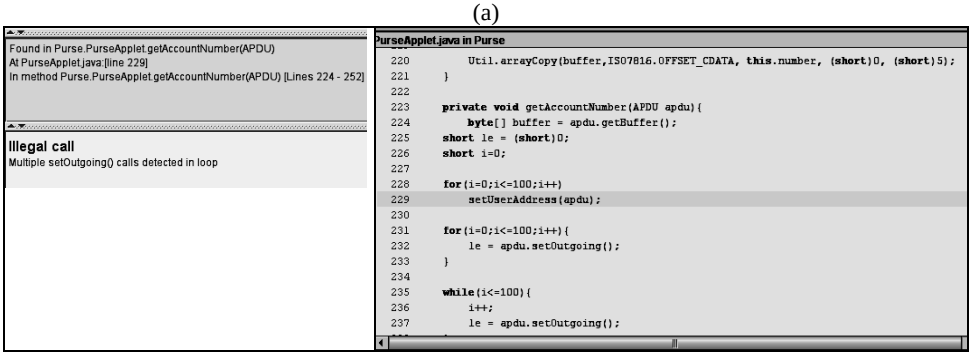



**Figure 4.** CFG pattern to find unhandled exception edges

Figure 5 demonstrates how the detector responds in two different property violation cases. In the first case, the client applet named *PurseClientApplet* calls `setOutgoing()` and subsequently invokes the method `getUserName()` of the *PurseApplet* thus causing the detected property violation. The second case concerns a property violation caused by a call to `setUserAddress()` in a for loop.




(a)




(b)

**Figure 5.** Illegal use of `setOutgoing()` detected (a) in interprocedural analysis and (b) within a loop via call to another method

### 4.2 Bug detectors for the correctness of the called methods' arguments

Dataflow analysis is the basic means to statically verify the correctness of the called methods' arguments. Its basic function is to estimate conservative approximations about *facts* that are true in each location of a CFG. Facts are mutable, but they have to form a lattice. The `DataflowAnalysis` interface shown in Figure 6 is the supertype for 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 that model values in local variables and 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.

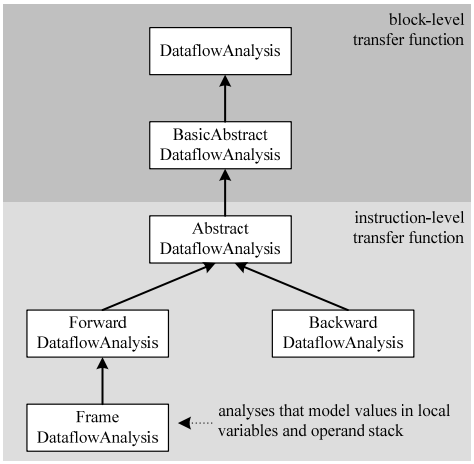



**Figure 6.** FindBugs base classes for dataflow analyses

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 shortcoming. 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 excludes the implementation of *context sensitive interprocedural analyses* like the ones explored in Section 5.

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

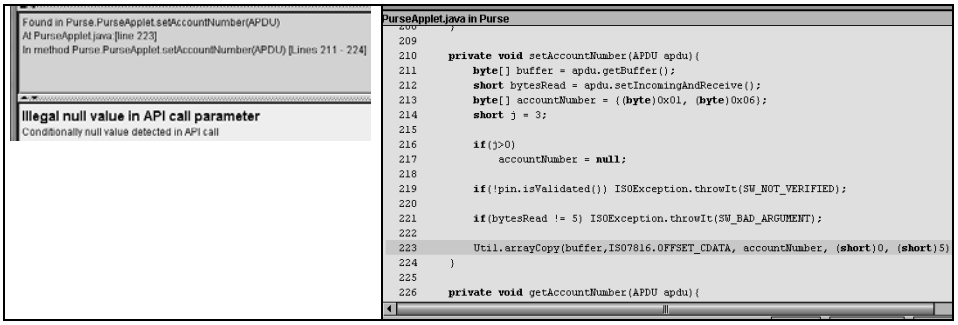
```
short arrayCopy(  byte[] src,  short srcOff,
                  byte[] dest, short destOff, short length)
```

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:

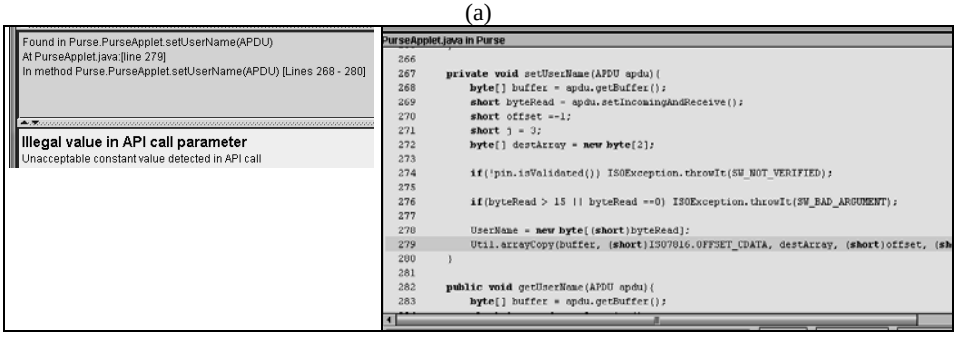
```
for each method in the class do
    request a CFG for the method
    get the method's ConstantDataflow from ClassContext
    get the method's ValueNumberDataflow from ClassContext
    get the method's IsNullValueDataflow from ClassContext
    for each location in the method do
        get instruction 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 also 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.




(a)




(b)

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

## 5 Precise and scalable analyses for the static verification of API calls

The static analysis case studies of Section 4 point out the merits as well as some shortcomings of the FindBugs open source framework, for the static verification of Java Card API calls. Although there is only limited documentation for the framework design and architecture, the source code is easy to read and self-documented. FindBugs is a live open source project and we will soon have new developments on shortcomings, like for example the lack of context-sensitive interprocedural dataflow analysis. Appropriate bug detectors can be supplied by the Java Card technology providers. Thus, Java Card applet providers will be able to use FindBugs in their development process with limited cost. This possibility opens new perspectives for automatically verifying the absence of unhandled security critical exceptions, as well as prospects for the development of bug detectors for application-specific correctness properties.

The static analysis techniques shown in the two case studies can be combined in bug detectors where either

- temporal safety includes constraints on the data values that are visible at the API boundary or
- we are interested in implementing sophisticated and precise analyses that reduce false positives and at the same time scale to real Java Card programs.

In the following paragraphs we review the latest developments in related bibliography that address the second aim and in effect designate static program analysis as a credible approach for the static verification of security critical applications.

A notable success story in temporal safety checking is the ESP tool for the static verification of C programs. ESP utilizes a successful heuristic called "*property simulation*" [15] and a path feasibility analysis called "*path simulation*" [16], in order to perform partial program verification based only on the control-flow branches that are relevant to the checked property. This results in a selective path-sensitive analysis that maintains precision only for those branches that appear to affect the property to be checked. For one particular instantiation of the approach, in which the domain of execution states is chosen to be the constant propagation lattice, the analysis executes in polynomial time and scales without problems in large C programs like the GNU C compiler with 140000 LOC.

It is still possible to construct programs for which property simulation generates false positives, but the authors claim that this happens only to a narrow class of programs that is described in their article. Property simulation is designed to match the behavior of a careful programmer. In order to avoid programming errors programmers maintain an implicit correlation between a given property state and the execution states under which the property state machine is in that state. Property simulation makes this correlation explicit as follows:

- For a given temporal safety property, ESP performs a first analysis pass where it instruments the source program with the state-changing events.
- For the second analysis pass, the property simulation algorithm implements a merge heuristic according to which if two execution states correspond to the same property state they are merged. In any other case, ESP explores the two paths independently as in a full path-sensitive analysis.

Interprocedural property simulation requires generation of context-sensitive function summaries, where context sensitivity is restricted to the property states. This happens in order to exclude the possibility of a non-terminated computation that exists if the domain of execution states is infinite (e.g. constant propagation). Thus, execution states are treated in a context-insensitive manner: at function entry nodes, all execution states from the different call sites are merged.

The proposed path simulation technique manages execution states and in effect acts as a theorem prover to answer queries about path feasibility. In general, path feasibility analysis is undecidable. To guarantee convergence and efficiency, ESP makes conservative assumptions when necessary. While such over approximation is sound (i.e. does not produce false negatives), it may introduce imprecision. More recent research efforts in cutting down spurious errors that are at the same time scalable enough for solving real world problems focus on applying iterative refinement to path-sensitive dataflow analysis [17].

Another notable success story in temporal safety checking is the SAFE project [18] at the IBM Research Labs. Both ESP and SAFE build on the theoretical underpinning of a *typestate* as a refinement of the concept of type [19]. 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. The heuristics applied in SAFE are reported in [20]. 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.

The most serious restriction in the current version of FindBugs regarding the perspectives to implement sophisticated analyses like those described is the lack of support for interprocedural context-sensitive dataflow analysis. However, we expect that this restriction will soon be removed.

## 6 Conclusion

This work explored the adequacy of static program analysis for the automatic verification of Java Card applets. We utilized the FindBugs open source framework in developing two bug detectors that check the absence of unhandled security critical exceptions, concerned with temporal safety and correctness of the arguments of Java Card API calls. The developed detectors are sound, but they are not precise. We explored the latest developments that open new prospects for improving the precision of static analysis, thus making it a credible approach for the automatic verification of security critical applications. The results of our work and the bug detectors source code are publicly available online http://mathind.csd.auth.gr/smart/.

A future research goal is the static verification of multi-applet Java Card applications (like the one in our case studies), in terms of temporal restrictions of inter-applet communications through shareable interfaces [21]. Also, we will continue to seek ways to overcome the experienced shortcomings in the current FindBugs version.

### Acknowledgments

This work was supported by the funds of the bilateral research programme between Greece and Cyprus, Greek General Research Secretariat, 2006-2008.

## References

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

2. Beckert, B. and Mostowski, W. A program logic for handling Java Card's transaction mechanism. In Proc. of 6th Int. Conference on Fundamental Approaches to Software Engineering (FASE'03), LNCS 2621 Springer, 2003, pp. 246-260.
3. Marché, C. Paulin-Mohring, C. and Urbain, X. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. Journal of Logic and Algebraic Programming 58 (1-2), 2004, pp. 89-106.
4. Meyer, J., Poetzsch-Heffter, A. An architecture for interactive program provers. In Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 1785 Springer, 2000, pp. 63-77.
5. Jacobs, B., Marche, C. and Rauch, N. Formal verification of a commercial smart card applet with multiple tools. In Proc. 10th Int. Conference on Algebraic Methodology and Software Technology (AMAST 2004), LNCS 3116 Springer, 2004, pp. 241-257.
6. Van den Berg, J. and Jacobs, B. The LOOP compiler for Java and JML. In Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 2031 Springer, 2001, pp. 299-312.
7. Breunesse, C. B., Catano, N., Huisman, M. and Jacobs, B. Formal methods for smart cards: an experience report. Science of Computer Programming 55, 2005, pp. 53-80.
8. The Java Verifier project, http://www.inria.fr/actualites/inedit/inedit36_partb.en.html
9. Catano, N. and Huisman, M. Formal specification and static checking of Gemplus's electronic purse using ESC/Java. In Proc. of Formal Methods Europe (FME'02), LNCS 2391 Springer, 2002, pp. 272-289.
10. Meijer, H. and Poll, E. Towards a full formal specification of the JavaCard API. In Proc. of the Int. Conf. on Research in Smart Cards: Smart Card Programming and Security, LNCS 2140 Springer, 2001, pp. 165-178.
11. Spinellis, D. and Louridas, P. A framework for the static verification of API calls. Journal of Systems and Software 80 (7), 2007, pp. 1156-1168.
12. The FindBugs project, http://findbugs.sourceforge.net/ (last access: 21st of Feb. 2008)
13. Hovemeyer, D., Pugh, W. Finding bugs is easy. SIGPLAN Notices 39 (12), 2004, pp. 92-106.
14. Dahm, M. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie University of Berlin, Institute of Informatics, 2001.
15. Das, M., Lerner, S. and Seigle, M. ESP: Path-sensitive program verification in polynomial time. In Proc. of the ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation (PLDI), 2002, pp. 57-68.
16. Hampapuram, H., Yang, Y. and Das, M. Symbolic path simulation in path-sensitive dataflow analysis. In Proc. of 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), 2005, pp. 52-58
17. Dhurjati, D., Das, M. and Yang, Y. Path-sensitive dataflow analysis with iterative refinemet. In Proc. of the Int. Symp. on Static Analysis (SAS), LNCS 4134 Springer-Verlag, 2006, pp. 425-442.
18. The SAFE (Scalable And Flexible Error detection) project, http://www.research.ibm.com/safe/ (last access: 21st of Feb. 2008)
19. Strom, R. E. and Yemini, S. Typestate: A programming language concept for enhancing software reliability. IEEE Trans. on Software Engineering 12 (1), 1986, pp. 157-171.
20. Fink, S., Yahav, E., Dor, N., Ramalingam G., Geay, E. Effective typestate verification in the presence of aliasing. In Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA), 2006, pp. 133-144.
21. Chugunov, G., Fredlund, L.-A., Gurov, D. Model checking of multi-applet Java Card Applications. In Proc. of the 5th Smart Card Research and Advanced Application Conf. (CARDIS), 2002.