Precise Analysis of Java Programs using JOANA

Dennis Giffhorn
Universität Karlsruhe (TH)
Karlsruhe, Germany
giffhorn@ipd.info.uni-karlsruhe.de

Christian Hammer
Universität Karlsruhe (TH)
Karlsruhe, Germany
hammer@ipd.info.uni-karlsruhe.de

Abstract

The JOANA project (Java Object-sensitive ANAlysis) is a program analysis infrastructure for the Java language. It contains a wide range of analysis techniques such as dependence graph computation, slicing and chopping for sequential and concurrent programs, computation of path conditions and algorithms for software security. This demonstration presents the JOANA plugin for the Eclipse framework. In the current version, a user can compute and navigate through dependence graphs for full Java bytecode, analyze Java programs with a broad range of slicing and chopping algorithms, and use precise algorithms for language-based security to check programs for information leaks.

1. Introduction

Dependence graphs are an established data structure for program analysis. A dependence-graph-based program analysis has basically a four-tiered architecture. The basic layer consists of the dependence graph computation with a broad range of issues such as points-to analysis, object-sensitivity and concurrency analysis, to just name a few. The second layer contains various algorithms to compute reachability properties in dependence graphs, where slicing is perhaps the best known of these algorithms. The applications of these algorithms form the third layer. There are numerous applications such as debugging, testing, complexity measurement, model-checking, and information flow control. The fourth layer addresses user interfaces, and concerns with visualising data of the underlying layers.

Since dependence graphs have been used for program analysis for more than 20 years, there exists a vast amount of work on those topics. But unfortunately only a few complete implementations came into existence – tools that contain everything from dependence graph construction over analysis algorithms and their applications to user-friendly interfaces like the *Code Surfer* for C/C++ [1] or the *Indus* Java slicer [7]. The JOANA plugin for the Eclipse framework[1] is such a tool for the Java language. The plugin focuses on *information flow control* (IFC), a method to ensure a program's integrity and confidentiality [9]. Whereas most current approaches for IFC are based on type systems, JOANA's slicing-based IFC is considerably more precise due to flow-sensitivity [3, 4].

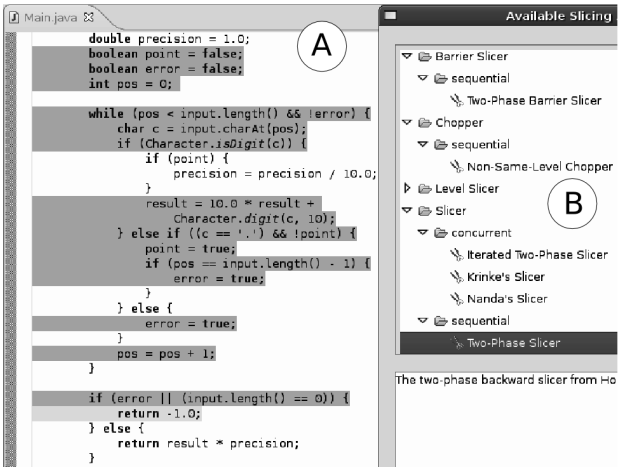

Figure 1. Slicing algorithms and visualisation

2. The JOANA Plugin for Eclipse

Our plugin currently offers the following features:

- A dependence-graph generator that can handle full Java bytecode, including object-sensitivity, concurrency and exception handling [5]. In previous case studies we were able to analyze programs of about 10kloc [2, 5].
- A broad range of slicing- and chopping algorithms for both sequential and concurrent Java programs, from standard algorithms over precise slicing algorithms for concurrent programs [2] to special algorithms for barrier- or level slicing and -chopping [8].

[1] www.eclipse.org

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.17

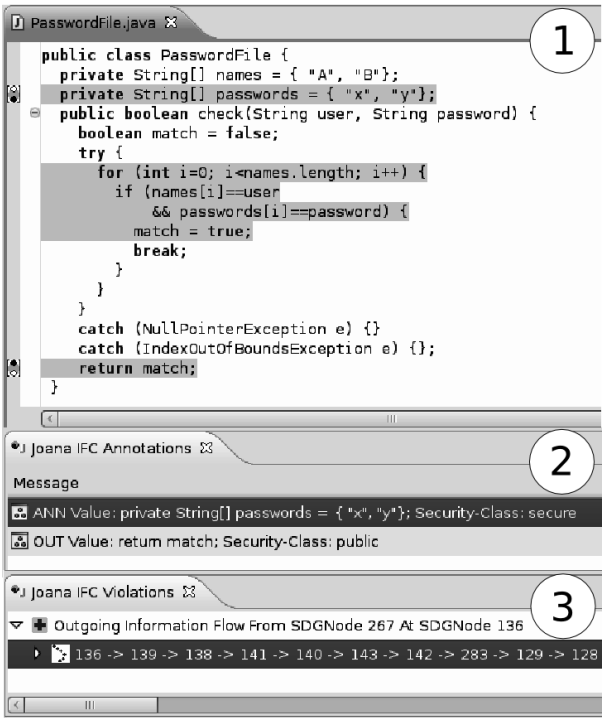

Figure 2. A possible information leak in a password manager

- Algorithms for IFC that can check Java programs for information leaks [3, 4].
- A user interface where the user can run all these algorithms directly on the Java source code.

Furthermore, we are currently integrating a generator of *path conditions*, which are precise correct conditions for information flow [6, 10].

Figures 1, 2 and 3 show parts of JOANA's user interface. In Figure 1, we see how slices are visualized in Java source code (A) and a list of currently available slicing and chopping algorithms (B). Part (A) shows a method which converts input strings to floating-point numbers. The highlighted slice was computed by the standard two-phase slicer for statement `return -1.0` (this statement denotes an invalid input).

Figure 2 shows an excerpt of the user interface for our IFC algorithms. The code in part (1) is a password manager that offers a `check` method to verify a given user name and password. A user of our IFC analysis has to identify secret information – the `passwords` array – and at which point there is information visible for unauthorized users – the return value of `check`. Part (2) shows that the user has classified `passwords` as 'secure' and the return value of `check` as 'public'. A combination of slicing and dataflow analysis [3, 4] reveals that `check` might leak secret information: The return value is 'true' if the given password was correct. Part (3) shows a summary of all detected infomation leaks. If the user selects one of the leaks, the path over which the infomation is leaking is highlighted in the source code. Part (1) shows that the password information flows to the return statement via the `if`-structure.

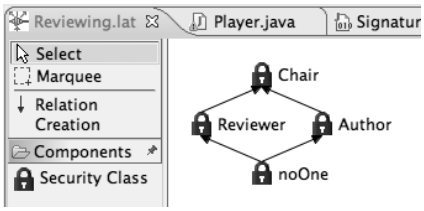

Figure 3. Defining a custom security lattice

Security classes for IFC are commonly arranged in a lattice. Figure 3 shows an example of our lattice editor for double-blinded reviewing: The chair may see all information in the system, but reviewers and authors may not know each other. The actual assignment of papers and notification of authors requires declassification from the chair. If a graph is defined that does not represent a valid lattice, the checking routine will highlight the corresponding part(s) in the graph representation.

References

[1] The Codesurfer Code Browser for C/C++. http://www.grammatech.com/.

[2] D. Giffhorn and C. Hammer. An evaluation of slicing algorithms for concurrent programs. In *7th IEEE SCAM*, 2006.

[3] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *ISOLA'06*, pp. 136–145, 2006.

[4] C. Hammer, J. Krinke, and G. Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE ISSSE*, pages 87–96, 2006.

[5] C. Hammer and G. Snelting. An improved slicer for Java. In *5th ACM PASTE*, pages 17–22, 2004.

[6] C. Hammer, R. Schaade and G. Snelting. Static Path Conditions for Java. In *3rd ACM PLAS*, 2008.

[7] The Indus Slicer for Java. http://indus.projects.cis.ksu.edu/.

[8] J. Krinke. Barrier slicing and chopping. In *IEEE SCAM*, 2003.

[9] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[10] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.