Hidden Truths in Dead Software Paths

Michael Eichberg[1] Ben Hermann[2] Mira Mezini[3] and Leonid Glanz[4]

Abstract: Approaches and techniques for statically finding a multitude of issues in source code have been developed in the past. A core property of these approaches is that they are usually targeted towards finding only a very specific kind of issue and that the effort to develop such an analysis is significant. This strictly limits the number of kinds of issues that can be detected.

In this paper, we discuss a generic approach – based on the detection of infeasible paths in code – that can discover a wide range of code smells ranging from useless code that hinders comprehension to real bugs. The issues are identified by computing the difference between the control-flow graph that contains all technically possible edges and the corresponding graph recorded while performing a more precise analysis using abstract interpretation.

The approach was evaluated using the Java Development Kit as well as the Qualitas Corpus (a collection of over 100 Java Applications) and enabled us to find thousands of issues.

1 Overview

Since the 1970s many approaches have been developed that use static analyses to identify a multitude of different types of issues in source code [Co06, CA01, GYF06]. The techniques used by these approaches range from pattern matching [Co06] to using formal methods [Co09] and vary widely w.r.t. their precision and scalability. But, they have in common that each one only targets a very specific kind of issues. Those tools (e.g., FindBugs [Co06]) that can identify issues across a wide(r) range of issues are typically just suits of relatively independent analyses. In all cases, the issues that can be found are limited to those that are identified by some tool developer.

We present a generic approach that detects control- and data-flow dependent issues in Java Bytecode without targeting any specific kind of issues per se. The approach applies abstract interpretation based techniques to analyze the code and while doing so records the paths that are taken. Afterwards, the analysis compares the recorded paths with the set of all paths that could be taken according to a naïve control-flow analysis that does not consider any data-flows. The paths computed by the latter analysis, but not found in the former graph, are then reported along with a justification why they were not taken.

The rationale underlying this approach is that many issues such as null dereferences or array index out of bounds accesses lead to executions that leave infeasible paths behind.

[1] Technische Universität Darmstadt, Fachbereich Informatik Fachgebiet Softwaretechnik, Hochschulstraße 10, 64289 Darmstadt, eichberg@cs.tu-darmstadt.de
[2] hermann@cs.tu-darmstadt.de
[3] mezini@cs.tu-darmstadt.de
[4] glanz@cs.tu-darmstadt.de

Hence, the hypothesis underlying the approach is threefold. First, in well-written code every path between an instruction and all it's direct successors is eventually taken, and, second, a path that will never be taken indicates an issue. Third, a large class of relevant issues manifests itself sooner or later in infeasible paths.

Though we opted for analyzing the code as precisely as possible, we deliberately limited the scope of the analysis to make it scalable. We start with each method of a project and then perform a context-sensitive analysis with a very small maximum call chain size. This makes the analysis unsound – i.e. we may miss certain issues – but it enables us to use it for large industrial sized libraries and applications.

To validate our approach we analyzed the Java Development Kit (JDK 1.8.0_25) and also the applications of the Qualitas Corpus [Te10]. The issues that we found range from seemingly benign issues to serious bugs that will lead to exceptions at runtime or to dead features. However, even at first sight benign issues, such as unnecessary checks that test what is already guaranteed, can have, e.g., an impact in code reviews such code generally hinders comprehension.[5]

2 Conclusion

The proposed approach is based on the idea that infeasible paths in software are a good indication of code issues and that a large class of relevant issues manifest themself sooner or later in infeasible paths. The implementation relies on a new static analysis technique that exploits abstract interpretation and is parametrized over abstract domains as well as the depth of call chains to follow inter-procedurally. This enables us to make informed reasonable trade-offs between scalability and soundness. The validity of the claims is evaluated by doing a case study of industrial size software; the issues revealed during the case study constitute themselves a valuable contribution of the paper and are publicly available.

References

[CA01] Cyrille, A.; Armin, B.: Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In: Proceedings of ASWEC '01. IEEE Computer Society, 2001.

[Co06] Cole, B.; Hakim, D.; Hovemeyer, D.; Lazarus, R.; Pugh, W.; Stephens, K.: Improving Your Software Using Static Analysis to Find Bugs. In: Companion to OOPSLA '06. ACM, 2006.

[Co09] Cousot, P.; Cousot, R.; Feret, J.; Mauborgne, L.; Miné, A.; Rival, X.: Why Does Astrée Scale Up? Form. Methods Syst. Des., 35(3):229–264, December 2009.

[GYF06] Geay, E.; Yahav, E.; Fink, S.: Continuous Code-quality Assurance with SAFE. In: Proceedings of PEPM '06. ACM, 2006.

[Te10] Tempero, E.; Anslow, E.; Dietrich, J.; Han, T.; Li, J.; Lumpe, M.; Melton, H.; Noble, J.: Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In: APSEC2010. 2010.

[5] The tool and the data set are available for download at `www.opal-project.de/tools/bugpicker`.