Badger: Complexity Analysis with Fuzzing and Symbolic Execution

Yannic Noller[1], Rody Kersten[2], Corina S. Păsăreanu[3]

Abstract: In this work, we report on our recent research results on "Badger: Complexity Analysis with Fuzzing and Symbolic Execution" which was published in the proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis [NKP18]. Badger employs a hybrid software analysis technique that combines fuzzing and symbolic execution for finding performance bottlenecks in software. Our primary goal is to use Badger to discover vulnerabilities which are related to worst-case time or space complexity of an application. To this end, we use a cost-guided fuzzing approach, which produces inputs to increase the code coverage, but also to maximize a resource-related cost function, such as execution time or memory usage. We combine this fuzzing technique with a customized symbolic execution, which is also guided by heuristics that aim to increase the same cost. Experimental evaluation shows that this hybrid approach enables us to use the strengths of both techniques and overcome their individual weaknesses.

Keywords: Software Testing; Complexity Analysis; Fuzzing; Symbolic Execution

Summary

We explore here the application of fuzzing and symbolic execution to algorithmic complexity analysis, which can be applied to reason about programs, understand performance bottlenecks and find opportunities for compiler optimizations. Algorithmic complexity analysis can also reveal worst-case complexity vulnerabilities, which occur when the worst-case time or space complexity of an application is significantly higher than the average case. In such situations, an attacker can mount Denial-of-Service attacks by providing inputs that trigger the worst-case behavior.

Fuzzing is currently the state-of-the-art testing technique to discover security-related vulnerabilities in software. Companies like Microsoft [Mi16] and Google [OS17] are applying fuzzing techniques on a regular basis. Although fuzzing applies random mutation operations to produce inputs, and hence, a large fraction of them might be invalid, it still can be more effective in practice than more sophisticated testing techniques, such as symbolic execution based approaches, due to its low computation overhead. Most fuzzers are designed to generate inputs that increase code coverage, while for complexity analysis one is interested in generating inputs that trigger worst case execution behavior of the programs. Furthermore,

[1] Humboldt-Universität zu Berlin, Germany, yannic.noller@hu-berlin.de
[2] Synopsys, Inc., San Francisco, USA, rody@synopsys.com
[3] Carnegie Mellon University Silicon Valley, NASA Ames Research Center, Moffett Field, USA, corina.s.pasareanu@nasa.gov

doi:10.18420/se2019-16

fuzzers are known to be good at finding so called *shallow* bugs but they may fail to execute deep program paths [St16], i.e. paths that are guarded by specific conditions in the code. On the other hand, symbolic execution techniques are particularly well suited to find such cases, but usually are much more expensive in terms of computational resources required.

We therefore present Badger: a framework that combines fuzzing and symbolic execution for automatically finding algorithmic complexity vulnerabilities in Java applications. It extends the Kelinci fuzzer with a worst-case analysis (aka KelinciWCA) by adding a new heuristic to account for resource-usage *costs* of program executions. It further uses a modified version of Symbolic PathFinder (SPF) to import inputs from the fuzzer, analyze them and generate new inputs that increase both coverage and execution cost. We modified SPF by adding a mixed concrete-symbolic execution mode, similar to concolic execution which allows us to *import* the inputs generated on the fuzzing side and quickly reconstruct the symbolic paths along the executions triggered by the concrete inputs. These symbolic paths are then organized in a *tree*, which is analyzed with the goal of generating new inputs that expand the tree. The analysis is guided by *heuristics* on the SPF side that favor new branches that increase resource-costs. The newly generated inputs are passed back to the fuzzing side. Badger can use various cost models, such as number of conditions executed, actual execution time as well as user-defined costs that allow us to keep track of memory and disk usage as well as other resources of interest particular to an application.

Our evaluation of Badger demonstrates that its performance and quality benefits over fuzzing and symbolic execution by themselves. We plan to explore more heuristics for worst-case analysis on both the fuzzing and symbolic execution side. Furthermore, we plan to extend our approach not only for complexity analysis, but also for a differential side-channel analysis of security-relevant applications.

References

[Mi16] Microsoft Security Risk Detection. `https://www.microsoft.com/en-us/security-risk-detection/`, 2016. Accessed: Dec 14, 2018.

[NKP18] Noller, Yannic; Kersten, Rody; Păsăreanu, Corina S.: Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2018, ACM, New York, NY, USA, pp. 322–332, 2018.

[OS17] OSS-Fuzz: Five months later, and rewarding projects. `https://testing.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html`, 2017. Accessed: Dec 14, 2018.

[St16] Stephens, Nick; Grosen, John; Salls, Christopher; Dutcher, Andrew; Wang, Ruoyu; Corbetta, Jacopo; Shoshitaishvili, Yan; Kruegel, Christopher; Vigna, Giovanni: Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In: 23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA. 2016.