Differential Program Analysis with Fuzzing and Symbolic Execution

Yannic Noller
Humboldt-Universität zu Berlin
Germany
yannic.noller@hu-berlin.de

ABSTRACT

Differential program analysis means to identify the behavioral divergences in one or multiple programs, and it can be classified into two categories: identify the behavioral divergences (1) between two program versions for the same input (aka *regression analysis*), and (2) for the same program with two different inputs (e.g, *side-channel analysis*). Most of the existent approaches for both subproblems try to solve it with single techniques, which suffer from its weaknesses like scalability issues or imprecision. This research proposes to combine two very strong techniques, namely *fuzzing* and *symbolic execution* to tackle these problems and provide scalable solutions for real-world applications. The proposed approaches will be implemented on top of state-of-the-art tools like AFL and Symbolic PathFinder to evaluate them against existent work.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; • **Security and privacy** → **Systems security**;

KEYWORDS

Differential Program Analysis, Fuzzing, Symbolic Execution

ACM Reference Format:
Yannic Noller. 2018. Differential Program Analysis with Fuzzing and Symbolic Execution. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3238147.3241537

1 INTRODUCTION

Differential analysis aims to find different behaviors in programs. This includes the identification of divergences between two program versions for the same input (*regression analysis*), but also the identification of divergent behaviors for different inputs for the same program. Whereas regression analysis is interested in verifying software patches for unintended behavioral changes, the second problem can be used in security analysis to identify long running program paths (*worst-case complexity analysis* - WCA) that can be used by a potential attacker to conduct a denial of service attack, or to locate side-channel vulnerabilities that can be used to expose *secret* information (*side-channel analysis*). Therefore, having effective and scalable techniques to perform a differential analysis of programs is crucial for testing real-world software.

Fuzzing has become one of the most promising testing techniques for finding bugs and security vulnerabilities in software [10, 26]. Even though a large amount of invalid inputs get generated, fuzzing can be more effective in practice than more complex testing techniques due to its low computation overhead. Although, fuzzers are known to be good at identifying *shallow* bugs, they may fail to execute deep program paths [24], i.e. paths that are guarded by specific conditions. On the other hand, symbolic execution techniques [9, 23] are particularly well suited to explore various branches including paths that require these specific conditions. However, symbolic execution is usually much more expensive in terms of computational resources used during exploration.

Worst-case complexity analysis is mostly performed on symbolic execution based approaches [5, 14] or pure fuzzing approaches [13, 20], and both variants suffer from its weaknesses like scalability issues for symbolic execution based approaches and imprecision for fuzzing based approaches. Existent techniques for side-channel analysis, e.g., [2, 4, 7], are too inaccurate and too imprecise for real-world applications. State of the art approaches for regression analysis [17, 19, 27] aim at covering the changed behavior, but might miss important divergences due to, e.g., imprecise constraint handling. As Palikareva et al. [17] already mentioned, testing evolving software is a difficult problem, which is unlikely to be solved by a single technique.

There are approaches on how to combine fuzzing with symbolic execution for test case generation [6, 8, 11], above all Driller [24] that combines the AFL fuzzer with the angr symbolic execution engine. All these combinations try to combine the strengths of fuzzing and symbolic execution in order to overcome their weaknesses. However, neither of them focus on the problem of differential program analysis, but mainly on generating high-coverage test suites. Differential program analysis needs a multi-dimensional approach with more sophisticated cost functions. Especially the symbolic execution side needs to be designed to not only solve constraints for unexplored paths, but to also choose promising paths that likely lead to a measurable difference. Therefore, this research proposes to explore combinations of cost-guided fuzzing and dynamic symbolic execution driven by appropriate heuristics to tackle the problem of differential program analysis, namely worst-case complexity analysis, side-channel analysis and regression analysis.

In order to evaluate this approach, this work targets to build tools based on the fuzzer AFL (together with Kelinci [12], an interface for AFL that enables fuzzing of Java programs), and the symbolic

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3241537

execution engine Symbolic PathFinder (SPF) [22]. Such new tools can be evaluated against existing work in terms of effectiveness and efficiency. Already performed experiments with such a combination of fuzzing and symbolic for WCA show promising results in outperforming the single techniques (cf. the description of Badger in Section 3.1). Additionally, differential fuzzing (so far without symbolic execution) was successfully applied to detect side-channel vulnerabilities (cf. Section 3.2).

Efficient, scalable and automated techniques for differential program analysis can greatly help software developers in several applications. Identifying security vulnerabilities gets more and more important in today's ubiquitous computing environment. Fast and precise testing of software patches is crucial to make changes quickly in evolving systems. The proposed research aims at finding these efficient and scalable techniques that are applicable in a real-world environment. Furthermore, the goal of this research is to automate these techniques as much as possible, to increase their usability.

2 RELATED WORK

State of the art techniques for worst-case complexity analysis focus on either fuzzing techniques or symbolic execution based techniques. SlowFuzz [20] is a fuzzer that prioritizes inputs that lead to increased execution times, and hence, it aims at finding the worst-case input in terms of program execution time. PerFuzz [13] is another recent fuzzing approach that uses multidimensional feedback and maximizes the execution counts for each reached program location, in order to identify distinct performance hot spots. Approaches like WISE [5] and SPF-WCA [14] use concolic execution to learn a path policy that likely leads to a worst-case execution of the program. Both perform exhaustive symbolic execution for large enough, user-defined input sizes to obtain good policies, which may not feasible in practice. A combination of fuzzing and symbolic execution would avoid an exhaustive exploration and could be fully automatic.

Typically a side-channel analysis accepts programs as secure if the secret data can not be inferred by the side-channel measurements that an attacker can make of the systems. This intuitive property is called *non-interference*, which can be checked with, e.g., *self-composition* [4]. Instead of checking non-interference, which might not hold for most realistic applications, the very recent approach Themis [7] checks a notion of ϵ-bounded non-interference, which accepts a program as secure as long as the cost differences stays within the specified threshold. With the focus on verifying the absence of timing channels, Blazer [2] departs from the composition-based strategies and instead establish a novel decomposition methodology. Both approaches, Blazer and Themis, are based on a static analysis, which might lead to false alarms and miss to generate concrete values that are crucial in reproducing and fixing the found vulnerabilities. Other approaches [3, 18, 21] use symbolic execution and constraint solving (in addition with model counting) for quantifying side-channel leakage and for synthesis of attacks. Although, they address the analysis of Java programs they can not yet scale to large applications, due to the expensive constraint manipulation. An efficient technique based on dynamic analysis could provide concrete inputs to reproduce bugs and overcome the imprecision, which is necessary for the application on complex, real-world problems.

State-of-the-art regression analysis techniques aim at covering the changed program statements by applying dynamic symbolic execution. Directed Incremental Symbolic Execution (DiSE) [19, 27] leverages static analysis to guide symbolic execution to changed program locations only. Due to the fact that it executes only the new version of the program, DiSE might lead to imprecise path conditions, which can miss divergences between the old and the new version. Shadow symbolic execution by Palikareva et al. [17] applies on a changed-annotated program version, which combines the old and the new program. Thus it can use the information from both versions. They introduce a dynamic analysis technique, which needs concrete test inputs to drive the symbolic execution. They assume to have a test suite created by developers, from which they can retrieve tests that touch the changed portion of the code. Shadow symbolic execution might miss divergences that could expose regression errors if the concrete inputs lead to path conditions that eliminate certain future program paths.

Several existent approaches try to combine fuzzing with symbolic execution for test case generation. EvoSuite [8] is a test-case generation tool for Java, based on evolutionary algorithms and dynamic symbolic execution. SAGE (Scalable Automated Guided Execution) [10] extends dynamic symbolic execution with a *generational search* that, instead of negating only the final condition of a complete symbolic execution, negates all conditions on the path. Mayhem [6], a symbolic execution engine with special focus on security vulnerabilities in binaries, was combined with the Murphy fuzzer and won the 2016 DARPA Cyber Grand Challenge [25]. Driller [24] is another promising tool that combines the AFL fuzzer with the angr symbolic execution engine and that has achieved similar results to Mayhem.

3 PROPOSED SOLUTIONS

3.1 Worst-Case Complexity Analysis

Figure 1 shows the overview of the technique Badger [15], which was presented at the ISSTA'2018. Badger uses the combination of the cost-guided fuzzer KelinicWCA and concolic execution based on SPF (named *SymExe*). By running both techniques in parallel while they exchange their results with eachother, it is possible to leverage both strengths and overcome their single limitations. KelinicWCA prioritizes costly paths by allowing AFL to not only use inputs that increase coverage, but also inputs that lead to an increased cost value (in time, memory or user-defined cost). *SymExe* imports the generated inputs from the KelinicWCA and builds the symbolic execution tree driven by the concrete values. Based on heuristics it picks the n most promising nodes for further exploration and generates inputs that then can be exported to the fuzzer to further push it into deeper paths.

The most related work to this approach is SlowFuzz [20], which is a fuzzer similar to KelinicWCA. Unfortunately, the approaches cannot be directly compared because of different target program languages. Therefore, the evaluation for Badger uses similar subjects as SlowFuzz and shows how all components (symbolic execution, fuzzing, and the combination) perform in terms of the quality of

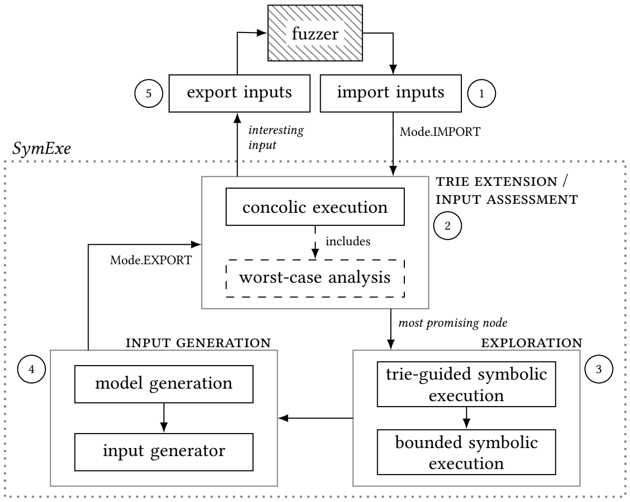

Figure 1: BADGER workflow representing how to combine fuzzing and symbolic execution for WCA [15].

the obtained worst-case and the speed in finding it. As an excerpt from the evaluation, Figure 2 shows the results for Insertion Sort (N=64), for which the experiment was conducted for 5 hours and the cost metric was the number of jumps in the JAVA bytecode.

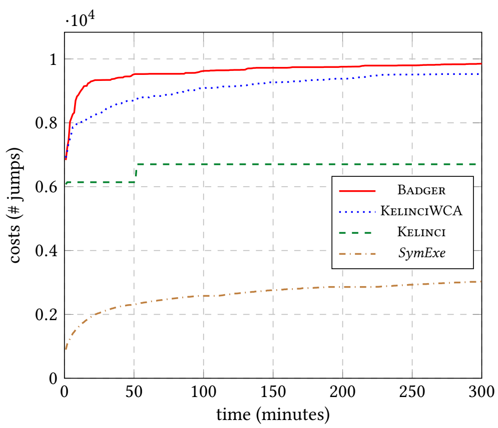

Figure 2: Results for Insertion Sort (N=64) [15].

3.2 Side-Channel Analysis

Side-channel vulnerabilities can be detected by maximizing the difference in observation between two program executions, for which only the *secret* values are different and the *public* value remains the same. A cost-guided fuzzer that can handle user-defined costs, similar to KELINICWCA [15], can be used to implement this idea by using the difference in observed cost between two program executions as user-defined value. As a first step, this idea was implemented in a prototype and the evaluation, which includes comparison with BLAZER [2] and THEMIS [7], showed that it effectively can identify side-channel vulnerabilities in real-world applications like Apache FtpServer [1]. As next step this research plans to combine this fuzzing approach with a similar symbolic execution approach used in BADGER. In contrast to BADGER, which uses fixed input sizes, differential fuzzing handles arbitrary input sizes up to a specified limit because it depends on the fuzzing step and the dynamic fuzzing driver to determine the current input size, which makes it necessary to apply a more sophisticated symbolic execution approach. The constructed symbolic execution tree has to handle multiple input sizes at the same time, which can be realized by using a virtual decision on the top, which determines the current size of the input.

3.3 Regression Analysis

For the worst-case complexity analysis it seems to be logical to follow or search next to costly paths to find even more costly paths. Unfortunately, the approach which works for WCA and side-channel analysis does not need to work as well for regression analysis. Regression analysis tries to find divergences in terms of taking different branches in two program versions for the same input. Quantifying these divergences is not as straightforward as quantifying cost differences like execution time.

As fuzzing cost metrics this research proposes to investigate two alternatives: (a) using cost defined by number of executed statements (similar to the previous work on worst-case complexity analysis and side-channel detection), for which a cost difference is clear indicator for a regression, although it might miss regression cases, and (b) using the difference in decision sequences, i.e. how similar have been the decision made in both program versions.

As symbolic execution counterpart this work identified two options: (i) applying shadow symbolic execution [17] driven by the concrete inputs imported from the fuzzer side, or (ii) applying (standard) symbolic execution similar to the side-channel analysis where the difference is measured by applying metric (b), the difference in decision sequences. Variant (i) would be based on a recent implementation of shadow symbolic execution on top of SPF, called $Shadow_{JPF}$ [16]. Shadow symbolic execution [17] needs concrete values, which could be obtained from the fuzzing side. In addition to be driven by concrete values, the symbolic execution in variant (i) should also apply a full four-way forking to find the most promising nodes for exploration and guide the fuzzing in deeper paths.

The overall workflow looks similar to the one by BADGER [15] (cf. Figure 1), although the analysis part needs to be replaced by appropriate metrics to identify regressions.

4 PROGRESS IN RESEARCH

The work on worst-case complexity analysis is finished with the publication of BADGER [15] at the ISSTA'2018. The work on side-channel analysis was started developing a fuzzing approach for the identification of side-channel vulnerabilities. Additional work is required to make usage of the full potential of fuzzing and symbolic execution in this area. Furthermore, it is necessary to work on the generation of concrete attacks based on a prior analysis based on fuzzing and symbolic execution. The work on regression analysis with fuzzing and symbolic execution was started by providing $Shadow_{JPF}$ [16], an extension for SPF to perform shadow symbolic execution at the JPF workshop 2017.

5 PLANNED EVALUATION

In order to evaluate the proposed approaches this research plans to quantitatively compare the resulting tools with existing approaches. For the side-channel analysis the evaluation will include micro benchmarks from existent work [2, 7] which consist of subjects from literature and smaller examples, but also popular real-world applications like Tomcat, Jetty, Spring-Security and Apache FtpServer. The evaluation will focus on comparing the identified cost differences as well as the needed analysis time. For the regression analysis the previous work includes the evaluation on smaller subjects for a preliminary assessment. It is necessary to identify more Java subjects applicable for regression testing.

6 CONCLUSION

The purpose of this research is to identify effective metrics for differential program analysis, which can be used as cost functions to drive a combination of fuzzing and symbolic execution. Additionally, this research aims to identify which types of dynamic symbolic execution should be combined with fuzzing for the specific subproblems. The resulting tools will be made publicly available, in order to support open science and to provide scalable solutions for real-world applications.

ACKNOWLEDGMENTS

This work is part of a PhD thesis supervised by Lars Grunske and Corina Păsăreanu. Thanks to my supervisors and my collaborators for their valuable inputs!

REFERENCES

[1] 2018. Apache FtpServer. https://mina.apache.org/ftpserver-project/. Accessed Jun 10, 2018.

[2] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 362–375.

[3] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. 2016. String Analysis for Side Channels with Segmented Oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/2950290.2950362

[4] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations (CSFW '04)*. IEEE Computer Society, Washington, DC, USA, 100–. https://doi.org/10.1109/CSFW.2004.17

[5] J. Burnim, S. Juvekar, and K. Sen. 2009. WISE: Automated test generation for worst-case complexity. In *2009 IEEE 31st International Conference on Software Engineering*. 463–473. https://doi.org/10.1109/ICSE.2009.5070545

[6] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 380–394. https://doi.org/10.1109/SP.2012.31

[7] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 875–890.

[8] J. P. Galeotti, G. Fraser, and A. Arcuri. 2013. Improving search-based test suite generation with dynamic symbolic execution. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 360–369. https://doi.org/10.1109/ISSRE.2013.6698889

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing *(PLDI '05)*. ACM, 213–223. https://doi.org/10.1145/1065010.1065036

[10] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[11] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1 (2012), 20:20–20:27.

[12] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2511–2513. https://doi.org/10.1145/3133956.3138820

[13] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[14] Kasper Luckow, Rody Kersten, and Corina Pasareanu. 2017. Symbolic Complexity Analysis using Context-preserving Histories. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*. 58–68. https://doi.org/10.1109/ICST.2017.13

[15] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. 2018. 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, 322–332. https://doi.org/10.1145/3213846.3213868

[16] Yannic Noller, Hoang Lam Nguyen, Minxing Tang, and Timo Kehrer. 2018. Shadow Symbolic Execution with Java PathFinder. *SIGSOFT Softw. Eng. Notes* 42, 4 (Jan. 2018), 1–5. https://doi.org/10.1145/3149485.3149492

[17] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a Doubt: Testing for Divergences Between Software Versions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1181–1192.

[18] Corina S Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run side-channel analysis using Symbolic Execution and Max-SMT. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*. IEEE, 387–400.

[19] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 504–515. https://doi.org/10.1145/1993498.1993558

[20] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2155–2168. https://doi.org/10.1145/3133956.3134073

[21] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. 328–342. https://doi.org/10.1109/CSF.2017.8

[22] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* (2013), 1–35. https://doi.org/10.1007/s10515-013-0122-2

[23] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[24] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. http://www.internetsociety.org/sites/default/files/blogs-media/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[25] Website. 2016. DARPA Cyber Grand Challenge. https://www.darpa.mil/news-events/2016-08-04.

[26] Website. 2016. Microsoft Security Risk Detection. https://www.microsoft.com/en-us/security-risk-detection/.

[27] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed Incremental Symbolic Execution. *ACM Trans. Softw. Eng. Methodol.* 24, 1 (2014), 3:1–3:42. https://doi.org/10.1145/2629536