A Study of Call Graph Effectiveness for Framework-Based Web Applications

Madhurima Chakraborty
University of California, Riverside
Riverside, CA, USA
madhurima.chakrborty@email.ucr.edu

Abstract

Modern web applications are continuously evolving and becoming increasingly reliant on web frameworks to support their ever-changing needs. This necessitates the realization of efficient static analysis methodologies for the purpose of bug finding and security auditing of such applications. Moreover, the majority of these frameworks are written in JavaScript, which is difficult to analyze due to its extremely dynamic nature. The primary goal of this work is to study the effectiveness of the present state-of-the-art call graph approaches for JavaScript and propose techniques to enhance them such that they discover more of the crucial functions and call edges in modern, framework-based JavaScript applications. Ideally, these new techniques must enhance function and call edge discovery without much impact on precision and scalability.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Keywords: JavaScript, static analysis, call graphs

ACM Reference Format:
Madhurima Chakraborty. 2021. A Study of Call Graph Effectiveness for Framework-Based Web Applications. In *Companion Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '21), October 17–22, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3484271.3484975

1 Motivation

Several frameworks [2, 6–9, 14] have been developed over the years, focusing on effective call graph generation of JS applications. While they have appeared to be effective in certain domains, a recent study [4] suggests that with the increasing complexity of JS applications along with the modern libraries and frameworks, many of the feasible edges may be missed, i.e., call graph recall can be low. For analysis tools searching for bugs and security vulnerabilities, missing edges are of key concern as they may lead to missed vulnerabilities. This motivated us to study which JS features render the static call graph approaches most ineffective and identify strategies to mitigate them. We expect that these findings would guide the community to certain focus areas for future call graph designs.

SPLASH Companion '21, October 17–22, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9088-0/21/10.
https://doi.org/10.1145/3484271.3484975

2 Background

JavaScript analysis challenges: The frequently used dynamic and reflective features [12] of JS often possess a serious threat to static analysis. Recent works [5, 10, 11, 16] have pointed out that some of these hard-to-analyze features of JS are dynamic property accesses, getter/setters, *eval* and *with*. Also, JS and the web platform include a myriad of native functions whose execution is mostly obscure to static analyses and call for manual modeling.

JavaScript Call Graph Construction: Static analysis of JS-based web applications has numerous significant use cases, including bug discovery, security analysis, and code understanding. In many of these cases, a call graph is essential to appropriately reason about the behavior of the function calls. Different approaches have been developed over the years to build call graphs [3, 6, 7, 13, 15, 18]. However, the highly dynamic nature of JS makes it difficult for them to scale and be practically useful for modern applications.

3 Initial Experiments

In order to develop enhanced static call graph approaches, we must first understand the primary causes of their unsoundness in practice. To this end, we implemented a general infrastructure to discover the root causes for which a static call graph may miss an observed dynamic call graph edge, enabling quantification of the relative importance of different causes of low recall.

Figure 1 gives an overview of our methodology to determine root causes. Given a program and a harness to exercise it, we generate the dynamic call graph and the dynamic flow trace to capture all the function calls and data flow of function values at run-time. At the same time, we use a static

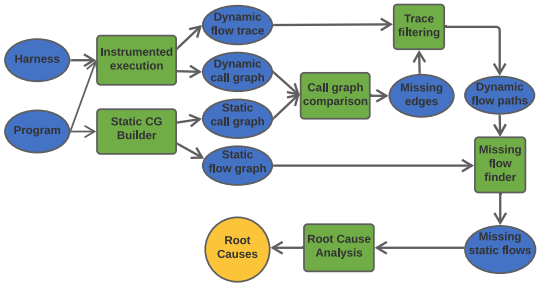

Figure 1. Methodology Overview

call graph generator to generate the static call graph and the associated flow graph for the code base. Next, we compare the dynamic call graph and the static call graph to find out the set of edges missing from the static call graph and compute all possible root causes for each of these missing edges. At this stage, we filter the dynamic trace to find the relevant trace elements (the dynamic copies of a function value that capture its flow from the point of creation to the calling location) for each of the missing edges. And then we compare those copies with the constraints in the static flow graph to determine which specific data flows were missed. Finally, a root cause is automatically assigned to each of the missing static flows.

4 Study Setup

Test Subject: We chose Approximate Call Graph (ACG) [2] as our test subject. To the best of our knowledge, ACG remains the state-of-the-art call graph construction technique for real-world JS web applications, and the WALA [17] implementation of ACG is well-maintained and widely used; which piqued our interest to study its drawbacks in details.

Benchmarks: We considered the TodoMVC suite [1], a collection of implementations of todo applications written in the most popular MVC frameworks (AngularJS, React, Vue) or Libraries (jQuery); often used as a reference to draw comparison between different JS frameworks.

5 Preliminary Results

Root Cause Analysis: For root causes of missing edges, we found that dynamic property accesses were the most common issue for these benchmarks, followed by calls to unmodelled library functions. Further, a significant percentage of the missing edges stemmed from other root cause issues, with their relative importance varying across benchmarks. Figure 2 shows the mean results of all the benchmarks that we tested.

Property Name Flow Analysis: We further developed a light-weight intra-procedural analysis using WALA to study how property names flow to the dynamic property accesses causing missed call edges. We found out a variety of types of

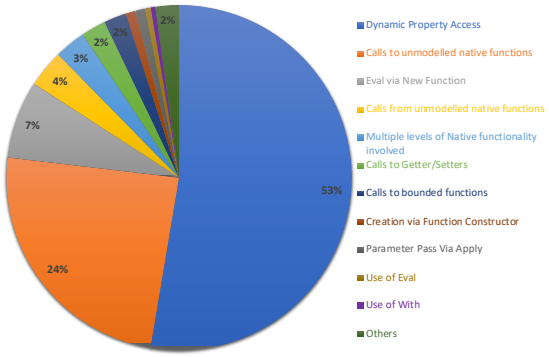

Figure 2. Root Cause Results

flow as a result, with no single type being dominant. Figure 3 summarizes our findings.

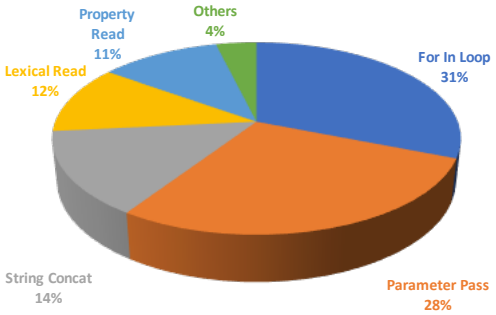

Figure 3. Property Name Flow Results

6 Conclusions and Future Work

We have developed a novel, fully automated technique to quantify how unsound the best approaches of static call graph generation are for real-world applications at this point and experimentally verified it on a state-of-the-art call graph algorithm. The preliminary results show that our technique was able to effectively uncover several interesting facts about the low recall of ACG. The evaluation shows that the biggest cause of low recall of ACG is dynamic property access, which accounted for 53%. For the remaining 47%, a myriad of JS constructs is responsible for bringing down the recall. We also notice that for dynamic property accesses, the property names flow from a variety of sources, and a range of remedies is needed to overcome them.

We plan to study more static call graphs and draw our analysis on more complicated benchmarks in the future. We also plan to make our infrastructure and scripts open source so the effectiveness of any future technique can be easily evaluated. Finally, we would like to use the results to assess the importance of the different constructs that need to be handled to improve the overall quality of JS static analysis tools. Our eventual goal is to develop an improved static call graph algorithm that captures the edges most critical to applications without excessively compromising the performance or precision.

References

[1] 2020. TodoMVC. https://todomvc.com/. Accessed: 2020-12-1.

[2] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *International Conference on Software Engineering (ICSE)*. 752–761.

[3] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. 2018. AFFOGATO: runtime detection of injection attacks for Node.js. In *Proceedings of the International Workshop on State Of the Art in Program Analysis (SOAP)*. 94–99.

[4] Behnaz Hassanshahi, Hyunjun Lee, Paddy Krishnan, and Jörn Güy Suß. 2020. Gelato: Feedback-driven and Guided Security Analysis of Client-side Web Applications. arXiv:2004.06292 [cs.SE]

[5] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the eval that men do. In *International Symposium on Software Testing and Analysis (ISSTA)*. 34–44.

[6] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium (SAS)*. 238–255.

[7] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. 121–132.

[8] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *In Proceedings of the International Workshop on Foundations of Object Oriented Languages*.

[9] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting locations in JavaScript programs affected by breaking library changes. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 187:1–187:25. https://doi.org/10.1145/3428255

[10] Benjamin Barslev Nielsen and Anders Møller. 2020. Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript. In *34th European Conference on Object-Oriented Programming (ECOOP)*. 16:1–16:28.

[11] Changhee Park, Hongki Lee, and Sukyoung Ryu. 2013. All about the with statement in JavaScript: removing with statements in JavaScript applications. In *Proceedings of the 9th Symposium on Dynamic Languages, part of SPLASH (DLS)*. 73–84.

[12] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 1–12.

[13] Sukyoung Ryu, Jihyeok Park, and Joonyoung Park. 2019. Toward Analysis and Bug Finding in JavaScript Web Applications in the Wild. *IEEE Softw.* 36, 3 (2019), 74–82.

[14] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *Proceedings of the 43rd International Conference on Software Engineering*.

[15] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 488–498.

[16] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *Object-Oriented Programming - 26th European Conference (ECOOP)*. 435–458.

[17] wala [n.d.]. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net.

[18] Shiyi Wei and Barbara G. Ryder. 2012. *A Practical Blended Analysis for Dynamic Features in JavaScript*. Technical Report TR-12-18. Virginia Tech. https://vtechworks.lib.vt.edu/handle/10919/19421