SPLASH: G: 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

1 PROBLEM AND MOTIVATION

Advances in technology and the internet have made web applications an integral part of businesses, enabling them to thrive in increasingly competitive environments. However, this emergence of web applications has also led to the development of sophisticated cyber-threats. As a result, insecure web applications are now highly vulnerable to cyber-threats, like malware and cyber-attacks. An attack may break the functionality of the application, prevent website visitors from accessing it, and even compromise customers' personal information. Based on a data generated from more than 15 million application security scans performed by different companies throughout 2021, researchers at NTT Application Security [6] found that half of all sites had at least one serious exploitable vulnerability throughout the year. Moreover, these web applications often use JavaScript (JS) to improve the interactivity and functionality of the websites.

While, JS has powerful capabilities to interact with webpage documents and browser windows, it is also susceptible to many security attacks. Insecure engineering practices of using JS has the potential to introduce security vulnerabilities and greatly increase the risks of browser-based attacks [35]. Furthermore, today's web applications routinely rely on sophisticated JS-based frameworks like React [5] and AngularJS [2] to take advantage of commonly-needed features like state management, synchronization, and navigation. Thus, a critical weakness of modern websites is that they are always at a risk of inheriting vulnerabilities from the third-party JS frameworks and libraries. For these kinds of applications, an effective defensive solution requires in-depth analysis of the codebase, such as static analysis.

Static analysis-based tools are particularly useful for finding bugs and improving security because they reveal patterns that are otherwise not obvious from manually-drafted test cases. However, in order to be effective, many of these tools need a call graph that captures the calling relationships between the various functions in a program. Unfortunately, static call graph construction is plagued by undecidability, and JS makes the problem even worse due to its extensive use of dynamic language features that are hard to analyze. Several sophisticated JS static analyses [14, 15, 17, 30] have been developed over the years to handle the complex JS constructs, but they do not yet scale well to modern web frameworks. There are also a growing number of unsound but pragmatic call graph analyses designed specifically to provide useful results for real-world code bases [3, 11, 20, 23]. Though these techniques have proven effective in some domains, they can be highly unsound when analyzing framework-based applications [13], thereby missing many edges, i.e., have low recall. These missing edges are of key concern for bug-finding and security analyses, as they can lead to false negatives like missed vulnerabilities. However, there has been no empirical study on this topic to date.

This work quantitatively assesses the effectiveness of state-of-the-art call graph algorithms on modern framework-based JS applications. Specifically, we aim to understand which JS features or constructs are most detrimental to the proposed call-graph approaches. Such data could be very useful in prioritizing research on how to effectively improve future analyzers for these applications.

2 BACKGROUND AND RELATED WORK

2.1 Call graph construction: In a static call graph, nodes represent functions, and an edge from a to b means that a may invoke b at runtime. *Precision* and *recall* are two measures of the utility of a computed call graph (CG). While precision measures the number of infeasible edges in the CG (call edges that cannot occur in any execution), recall measures the number of feasible call edges (those that *can* occur in some execution) missing from the CG.

2.2 JavaScript analysis challenges: JS developers frequently use dynamic and reflective language features that are difficult to analyze and pose a great challenge for static analysis [27]. Such features include:

- **Dynamic Property Accesses:** The syntactic form `x[e]` is often used in JS for accessing object fields or *properties*, where `e` represents an arbitrary expression evaluating to a property name. Identifying which memory locations may be accessed by an expression `x[e]` can be an extremely challenging for static analysis. Additionally, if `e` evaluates to a property name that does not exist on `x`, a write to `x[e]` *creates* the property rather than failing, complicating static analysis even further.
- **Getters and Setters:** It is possible to define properties in JS such that accessing a property actually invokes a *getter* or *setter* method with custom logic, making it

difficult to statically identify even the program locations where a function call can occur.
- **Eval:** In JS, arbitrary strings can be evaluated as code at runtime via `eval` construct or using `new Function`. Such dynamically-evaluated code can pose significant challenges for static analysis [26].
- **With:** The `with` construct extends the scope chain for variable lookups with arbitrary mappings. Like `eval`, `with` usage complicates static analysis [25].
- **Parameter Passing:** With JS, you can pass parameters to functions in a variety of ways, e.g., binding the `this` parameter explicitly or passing arguments in an array. A function can also read its formal parameters via a special `arguments` array. Finally, a function may be legally invoked with *any* number of parameters, irrespective of how many formal parameters it declares. The combination of these features makes tracking data flows between processes extremely difficult.
- **Native Methods:** A large portion of JS and the web platform's standard library is usually opaque to static analysis, so models must be built for many of these "native" methods.

While these root causes of difficult analysis are well known, our technique estimates their *relative* impact on call graph recall for a set of target benchmarks.

2.3 JavaScript Analyses: Several analysis frameworks [7, 16, 22, 28] use abstract interpretation [10] to handle the interdependent problem of scalability and precision in JS [14, 15], particularly when analyzing libraries. Though these techniques have shown impressive results when analyzing libraries like jQuery [4], they still cannot handle complex MVC frameworks like React [5].

Another approach is to use dynamic information to improve static analyses. Wei and Ryder [33] introduced blended analysis, which uses dynamic analysis to aid static analysis in handling JS's dynamic features. Lacuna [24] utilizes static and dynamic analysis to detect dead code in JS-based applications. Although dynamic information can be of great help to static analysis, improving pure static analysis is still desirable, since it can be done without instrumenting and running the code and without inputs.

For analyzing JS applications that use the Windows runtime and other libraries, Madsen et al. [19] proposed an automated use analysis that infers points-to specifications. In framework-based applications, where control flow is largely determined by the framework, not the application, it is unclear what impact their analysis may have. For server-side JS applications, Nielsen et al. [21] present a feedback-driven static analysis to identify the third-party modules that must be analyzed. We, however, focus on client-side MVC applications. The CodeQL system includes an under-approximate call graph builder [3] which makes use of pragmatic static analysis. CodeQL's analysis is primarily intra-procedural, targeted at taint analysis, and does not consider dynamic access to properties. A system for detecting breaking library changes in Node.js programs is described in [20], which uses an under-approximate analysis for high recall at the cost of precision. Nielsen et al. [23] presents a pragmatic modular call-graph construction technique for Node.js programs. Recently, Salis et al. [29] presented a pragmatic call graph builder for Python programs. It would be interesting to adapt our techniques to Python programs in the future.

2.4 Root cause analysis: Our work was partly inspired by a study of call graph recall for Java programs by Sui et al. [31], which used calling-context trees and runtime tagging of reflective operations to determine the language features impacting recall. As functions are first-class values in JS, we can trace function data flow directly to make this determination. JS's dynamic nature, however, makes the potential causes of missing edges and their usage patterns significantly different from Java's problematic constructs.

Andreasen et al. [8] presents techniques for isolating soundness and precision issues in the TAJS static analyzer for JS. For finding analysis unsoundness, their technique creates logs of expression values while executing target programs, and then checks that the static analysis abstractions account for all such values. Using delta debugging, they identify a reduced version of the program that exhibits the same unsoundness and determine the root cause. Contrary to their approach, which aims to achieve full soundness of an analysis, our approach is more targeted at analyses with deliberate unsoundness (for practicality), and aims to quantify the impact of different unsoundness root causes.

Lhoták [18] also presents a comparison of static and dynamic call graphs for Java, aimed at finding sources of imprecision in the static call graph. Other works [9, 34] have used dynamic analysis to generate traces and found root causes of imprecision in JS static analyses. Lee et al. [16] produced a tracing graph by tracking information flow from imprecise program points backwards, thereby aiding the user to identify main causes of the imprecision. Our work is different from all these studies as we focus on recall rather than precision, which necessitates different techniques.

3 APPROACH AND UNIQUENESS

This work presents an approach for quantifying the effectiveness of JS static call graphs, including a novel technique for *automatic root cause quantification* of low recall.

Figure 1 gives an overview of our methodology. Given a program and a harness to exercise it, we use dynamic analysis to collect the dynamic call graph and the dynamic flow trace. While the dynamic call graph captures all the function calls, the dynamic flow trace captures all the data flow of function values at runtime. Next, the dynamic call graph is compared with the static call graph to compute recall under various metrics. Given the set of dynamic call edges

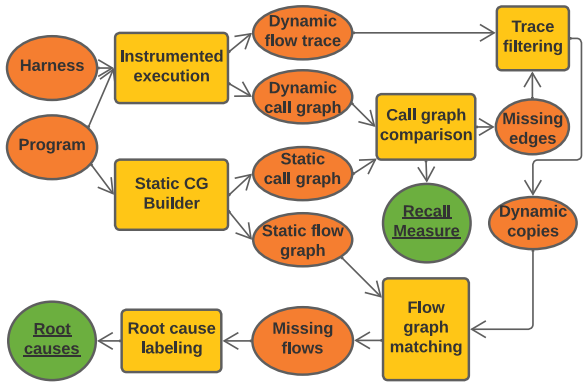

Figure 1. Overview of our methodology.

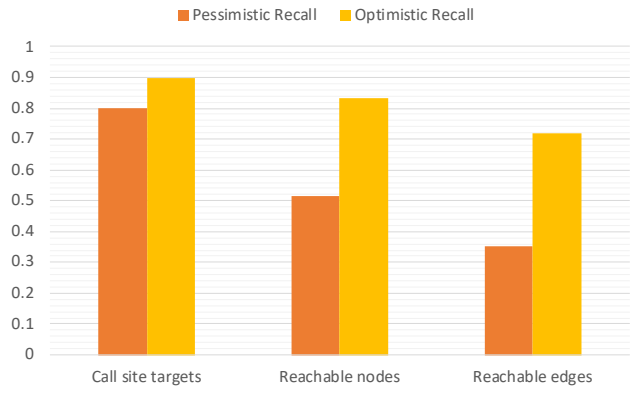

Figure 2. Average recall across TodoMVC benchmarks.

missing from the static call graph, we then compute the *root causes* for each missing edge. This involves computing the *dynamic copies* of a function value relevant to a particular missing edge, and comparing those copies to the constraints in *static flow graph* capturing data flow known to the static analysis. Dynamic copies represent the data flow of function values at runtime and given an invocation of function f at a call site, our technique processes the dynamic flow trace to compute the dynamic copies by which f was invoked at the site. Following that, our technique matches these dynamic copies to the static flow graph constraints and computes the missing flows relevant to each missing call edge. In the final step, the missing flows are automatically categorized to produce the set of root causes responsible for a missing edge, thereby highlighting the JS features which must be handled by the static analysis to discover the edge. Our approach is capable of handling missing edges in call graphs with multiple root causes as well as inter-dependent causes, such as an edge is missing due to the absence of another.

4 RESULTS AND CONTRIBUTIONS

The techniques outlined in fig. 1 have been used to study the recall and root causes of two variants (pessimistic and optimistic) of the approximate call graph (ACG) algorithm by Feldthaus et al. [11] , as implemented in the WALA framework [32], on a suite of modern web applications. 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 implementation of ACG is well-maintained and widely used; which piqued our interest to study its drawbacks in details. For benchmarks, we analyzed several programs from the TodoMVC suite [1], which contains idiomatic framework-based implementations of a "todo list" web application; the suite is often used as a reference for comparing frameworks. We built a harness to automatically and thoroughly exercise these applications, with coverage of application code above 97% in nearly all cases. This high coverage gave confidence that information collected via dynamic analysis would cover nearly all application behaviors.

4.1 Recall Measurements: In this section, we describe the recall measurements for the TodoMVC benchmarks, comparing the ACG static call graph variants with the dynamic call graphs as described in Section 3. We used three different metrics to measure recall, suited to different client scenarios:

- **Call site targets:** Under this metric, recall is computed for each call site present in the dynamic call graph, and then averaged across call sites. This metric matters most to clients like code navigation in an IDE.
- **Reachable nodes:** Recall is computed based on the set of reachable methods from the entrypoints in the dynamic call graph. This metric has been used in previous work studying call graph recall [31], and is relevant to clients, such as dead-code elimination.
- **Reachable edges:** This metric uses the set of call graph edges whose source method appears in the dynamic call graph to compute recall. A metric like this is most useful for clients doing deep inter-procedural analysis like taint analysis [12].

Figure 2 gives the average results of both variants of ACG under each metric and leads to the following observations:

> **Observation 1:** ACG recall decreases with more exacting metrics, particularly for pessimistic analysis reflecting the need for deeper static analyses.
> **Observation 2:** On our benchmarks, neither ACG variant provides a very accurate call graph in terms of the reachable edges metric.
> **Observation 3:** A detailed recall study of each benchmark in our suite reveals that recall can vary widely across benchmarks.

4.2 Root Cause Analysis: Here, we present selected, illustrative results from applying the analysis to discover root causes of missing call graph edges on our benchmarks. We compute root causes for each individual missed call edge

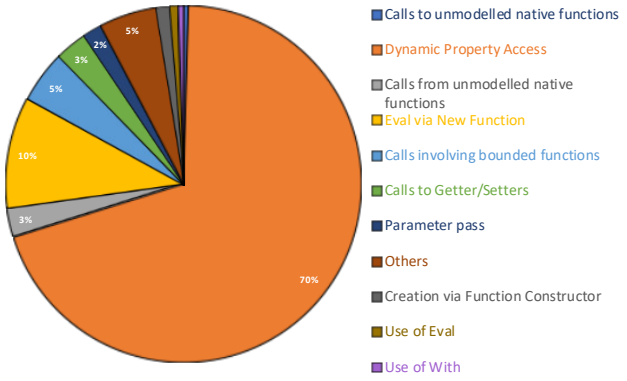

Figure 3. Root Cause Results (Optimistic ACG)

in the static call graph, corresponding to the "Reachable edges" metric used to measure recall in 4.1. Space does not allow a full presentation of all results.

Using data to improve recall: When studying the prevalence of different root causes across the TodoMVC benchmarks for the original implementation of the optimistic ACG variant in WALA, we were surprised to see that 24% of missed call edges were due to calls to unmodelled standard library functions. Based on this data, we modified WALA to include basic models of many of these native functions. This change improved the average recall by up to 5 percentage points. This improvement shows that quantifying root cause prevalence can guide analysis developers to "quick wins" for improving analysis recall. The figures and data presented in this paper are based on the improved version of WALA ACG.

Top root causes: Figure 3 shows the average results of all the ToDoMVC benchmarks that we tested using the optimistic variant of ACG (each benchmark is weighted equally). For root causes of missing edges, we see that dynamic property accesses are by far the most prevalent root cause for optimistic analysis of TodoMVC benchmarks at 70%. We dig further into these property accesses with a finer-grained labeling in 4.3. The second-most prevalent root cause on average is "Eval via new Function" at 10%, but as we shall see next, the second-highest root cause varies significantly across benchmarks.

Comparing the overall optimistic and pessimistic variant (not included in the paper) root cause results, we see that missed calls due to functions being passed as parameters or returned (the "Parameter Pass" and "Function return" labels) are significant root causes (totaling 74%) for pessimistic analysis but not optimistic. This result makes sense, as the key difference between the optimistic and the pessimistic ACG is that optimistic analysis tracks interprocedural flow of function values. Given that 74% of missed edges for pessimistic analysis are due to such interprocedural flows, it seems the best approach to improving pessimistic recall for these benchmarks would be to model some of these flows, rather than attacking other root causes.

Variance across benchmarks: If we try to understand the variance in root cause prevalence across benchmarks (say optimistic ACG), we will see while the most-prevalent root cause for each of the benchmark is dynamic property accesses, the second-place root cause varies by benchmark like "Eval via new Function" for React, "Calls to bounded functions" for AngularJS, and "Calls to getters / setters" for Vue. This benchmark-specific data could provide valuable information to an analysis developer. E.g., if the developer were primarily trying to improve recall for applications like the Vue benchmark, improving handling of getters and setters might be more worthwhile than if the applications were more similar to the React benchmark.

To summarize, we have shown that our technique for quantifying root causes is effective across several benchmarks, and can expose the most important root causes in aggregate and highlight the differences between benchmarks. Since improving recall for JS static analyses on real-world programs poses so many challenges, we expect improvements for specific types of benchmarks to prove worthwhile, and the data from our techniques can provide valuable guidance.

4.3 Property Name Flow Analysis: Given the importance of dynamic property accesses as a root cause in 4.2, we performed a finer-grained root cause labeling of these accesses. Our aim was to understand better how the property names were computed for these accesses, to see if some targeted handling of the property name expressions could be useful. With that goal, we 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 flow as a result, with no single type being dominant. JS's `for-in` loops for iterating over object properties appeared to be the biggest single source (32%). Other major sources included passing property names from outside the function containing the access (27%) or variables in enclosing lexical scopes (13%) and property reads (12%) (i.e., the property name is read from another object property); handling these cases may require inter-procedural tracking of property name value flow. Lastly, the string concatenation cases (14%) suggesting the need for string analysis.

5 CONCLUSIONS AND FUTURE WORK

A primary objective of the work was to better portray the significance of the current gaps in static call graph algorithms. To this end, we developed a novel, fully-automated technique to quantify how unsound the best approaches of static call graph generation are for real-world applications and experimentally verified it on the results of state-of-the-art call graph algorithms on modern, framework-based web applications. The study's results provided numerous insights

on the variety and relative impact of root causes for missed edges. We believe that understanding the nature of these root causes will help in developing automated techniques to discover them.

We instantiated our approach to perform a detailed study of the effectiveness of the ACG algorithm on a set of web applications. We performed the study by developing dynamic analyses to expose occurrences of hard-to-analyze behaviors, and leveraging a suite of framework-based applications that we exercised with high coverage. We showed how recall of ACG suffers under exacting metrics, and we detailed the variety and relative impact of root causes responsible for missed edges in ACG call graphs.

In the future, we plan to study more static call graphs and draw our analysis on more complicated benchmarks. We also plan to extend the study to other domains; we expect that analyses for any dynamic language with extensive use of higher-order functions could benefit from our techniques. Our infrastructure and scripts will be made open source so that 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 an application without excessively compromising the performance or precision.

References

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

[2] 2021. Angular. https://angular.io. Accessed: 2021-01-11.

[3] 2021. CodeQL library for JavaScript: Call graph. https://codeql.github.com/docs/codeql-language-guides/codeql-library-for-javascript/#call-graph. Accessed: 2021-01-11.

[4] 2021. jQuery. https://jquery.com/. Accessed: 2021-1-24.

[5] 2021. React – A JavaScript library for building user interfaces. https://reactjs.org. Accessed: 2021-01-11.

[6] 2022. AppSec stats flash: 2021 Year in Review. https://www.whitehatsec.com/blog/appsec-stats-flash-2021-year-in-review/ Accessed: 2021-03-17.

[7] Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. In *SPLASH*. 17–31.

[8] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 2017. Systematic Approaches for Increasing Soundness and Precision of Static Analyzers. In *Proc. 6th International Workshop on the State Of the Art in Program Analysis (SOAP)*.

[9] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 2017. Systematic approaches for increasing soundness and precision of static analyzers. In *SOAP*. 31–36.

[10] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. 238–252.

[11] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *ICSE*. 752–761.

[12] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the world wide web from vulnerable JavaScript. In *ISSTA*. 177–187.

[13] 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]

[14] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *SAS*. 238–255.

[15] 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 *FSE*. 121–132.

[16] Hongki Lee, Changhee Park, and Sukyoung Ryu. 2020. Automatically Tracing Imprecision Causes in JavaScript Static Analysis. *Art Sci. Eng. Program.* 4, 2 (2020).

[17] 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*.

[18] Ondrej Lhoták. 2007. Comparing call graphs. In *PASTE*. 37–42.

[19] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ESEC/FSE*. 499–509.

[20] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting locations in JavaScript programs affected by breaking library changes. 4, OOPSLA (2020), 187:1–187:25.

[21] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. Nodest: feedback-driven static analysis of Node.js applications. In *ESEC/FSE*. 455–465.

[22] Benjamin Barslev Nielsen and Anders Møller. 2020. Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript. In *ECOOP*. 16:1–16:28.

[23] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node.js applications. In *ISSTA*. 29–41.

[24] Niels Groot Obbink, Ivano Malavolta, Gian Luca Scoccia, and Patricia Lago. 2018. An extensible approach for taming the challenges of JavaScript dead code elimination. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 391–401.

[25] Changhee Park, Hongki Lee, and Sukyoung Ryu. 2013. All about the with statement in JavaScript: removing with statements in JavaScript applications. In *SPLASH*. 73–84.

[26] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications. In *ECOOP*. 52–78.

[27] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*. 1–12.

[28] 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.

[29] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *ICSE*.

[30] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic determinacy analysis. In *PLDI*. 165–174.

[31] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the recall of static call graph construction in practice. In *ICSE*. 1049–1060.

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

[33] Shiyi Wei and Barbara G. Ryder. 2012. *A Practical Blended Analysis for Dynamic Features in JavaScript*. Technical Report TR-12-18. Virginia Tech.

[34] Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. 2016. Revamping JavaScript static analysis via localization and remediation of root causes of imprecision. In *FSE*. 487–498.

[35] Chuan Yue and Haining Wang. 2013. A measurement study of insecure javascript practices on the web. *ACM TWEB* 7, 2 (2013), 1–39.