Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs

Madhurima Chakraborty ✉
University of California, Riverside

Renzo Olivares ✉
University of California, Riverside

Manu Sridharan ✉
University of California, Riverside

Behnaz Hassanshahi ✉
Oracle Labs Australia

Abstract

Building sound and precise static call graphs for real-world JavaScript applications poses an enormous challenge, due to many hard-to-analyze language features. Further, the relative importance of these features may vary depending on the call graph algorithm being used and the class of applications being analyzed. In this paper, we present a technique to *automatically* quantify the relative importance of different root causes of call graph unsoundness for a set of target applications. The technique works by identifying the dynamic function data flows relevant to each call edge missed by the static analysis, correctly handling cases with multiple root causes and inter-dependent calls. We apply our approach to perform a detailed study of the recall of a state-of-the-art call graph construction technique on a set of framework-based web applications. The study yielded a number of useful insights. We found that while dynamic property accesses were the most common root cause of missed edges across the benchmarks, other root causes varied in importance depending on the benchmark, potentially useful information for an analysis designer. Further, with our approach, we could quickly identify and fix a recall issue in the call graph builder we studied, and also quickly assess whether a recent analysis technique for Node.js-based applications would be helpful for browser-based code. All of our code and data is publicly available, and many components of our technique can be re-used to facilitate future studies.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases JavaScript, call graph construction, static program analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.3

Related Version *Extended Version*: https://arxiv.org/abs/2205.06780

Funding This research was supported in part by a gift from Oracle Labs and by the National Science Foundation under grant CCF-2007024. This research was partially sponsored by the OUSD(R&E)/RT&L and was accomplished under Cooperative Agreement Number W911NF-20-2-0267. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL and OUSD(R&E)/RT&L or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

1 Introduction

Effective call graph construction is critically important for JavaScript static analysis, as JavaScript analysis tools often need to reason about behaviors that span function boundaries (e.g., security vulnerabilities [26, 27] or correctness of library updates [40]). Unfortunately, call graph construction for real-world JavaScript programs poses significant challenges, particularly

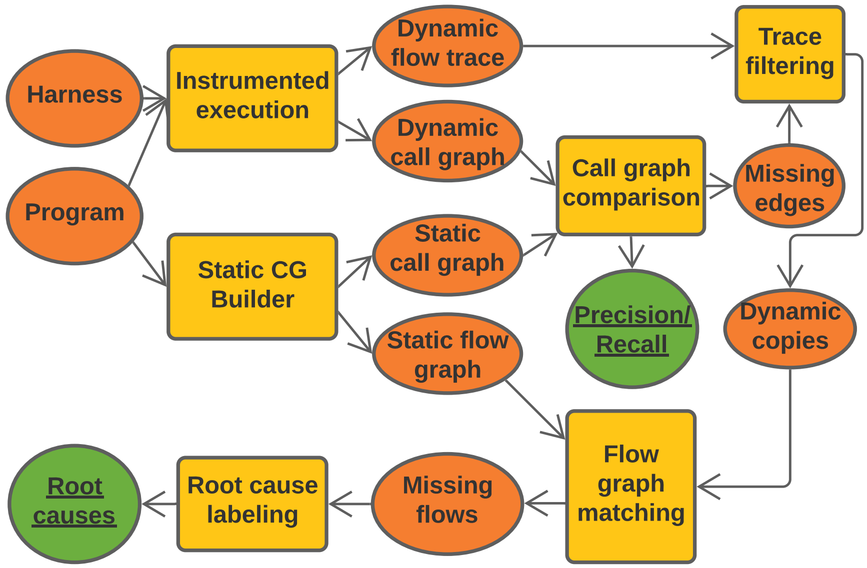


Figure 1 Overview of our methodology.

for client-side code in web applications. Modern web applications are increasingly built using sophisticated frameworks like React [4] and AngularJS [6].[1] Sophisticated recent JavaScript static analysis frameworks [32, 33, 36, 52] often focus on sound and precise handling of complex JavaScript constructs. While these systems have advanced significantly, they cannot yet scale to handle modern web frameworks. There are also a growing number of unsound but pragmatic call graph analyses designed primarily to give useful results for real-world code bases [8, 25, 40, 44]. While these techniques have been shown effective in certain domains, their unsoundness can lead to missing many edges when analyzing framework-based applications [27], i.e., the analyses can have low *recall*. For bug-finding and security analyses, these missing edges are of key concern as they can lead to false negatives like missed vulnerabilities.

To guide development of better call graph builders, it would be highly useful to know which language constructs are contributing most to reducing recall for a set of benchmarks of interest. JavaScript has many different constructs that are typically ignored or only partially handled by pragmatic static analyses, due to their dynamic nature [49]. Further, there are complex tradeoffs involved in adding support for these constructs, as a more complete handling may lead to scalability and precision problems. Analysis designers aiming to improve results for a set of benchmarks would be helped by quantitative guidance on the relative importance of different unhandled language features.

This paper presents a novel technique for *automatic root cause quantification* for missing edges in JavaScript call graphs. figure 1 gives an overview of our technique. Given a program, a static call graph builder enhanced to also export static flow graphs (see Section 2.2), and a harness for exercising the program, our technique automatically finds *missing flows*, data

[1] A recent Stack Overflow developer survey shows popularity of these frameworks is growing, with total usage surpassing older libraries like jQuery [56].

flows of function values that occur at runtime but are not modeled by the static analysis. Our technique associates a set of missing flows with each missed call graph edge, thereby indicating which data flows must be handled by the static analysis to discover the missed edge. The technique correctly accounts for *inter-dependent calls*, where a call graph edge is missing due to the absence of other call graph edges.

We further observe that given a missing flow, one can often automatically determine a *root cause label* for the flow, indicating which unhandled language construct(s) were responsible for the flow being missed. Such labeling can be performed at different levels of granularity, depending on what level of detail is desired by the analysis designer. Given logic to map missing flows to root cause labels, our technique automatically quantifies the prevalence of each root cause for the desired benchmarks.

We have implemented our techniques, and we used them to study the recall of two variants of the approximate call graphs (ACG) algorithm of Feldthaus et al. [25], as implemented in the WALA framework [58], on a suite of modern web applications. We found the root cause quantification to provide useful insights, in particular:

- To our surprise, a large initial cause of low recall was the lack of models in WALA for a variety of built-in library functions. By adding models, we were able to increase recall by up to 5 percentage points.
- After fixing the native models, dynamic property accesses were the largest root cause of low recall, at 70%. The second-largest root cause varied significantly across the benchmarks.
- We applied a finer-grained root cause labeling for dynamic property accesses, and found that their property names are computed in a wide variety of ways, with no single dominant pattern. We studied the potential of a recently-described recall-improving technique for dynamic property accesses in Node.js programs [44], and found that it would at best have a small impact for our web-based benchmarks.

Our dynamic call graph and flow trace analyses were challenging to implement due to JavaScript's hard-to-analyze language features. JavaScript includes many difficult-to-analyze features, including (but not limited to) reflective call mechanisms, "native" library methods, getter/setter methods, and dynamic code evaluation. Pragmatic static analyses often ignore most of these features, as they do not aim for sound results. However, since we aimed to study which calls were missed by such analyses and *why* those calls were missed, our dynamic analyses had to faithfully capture the behavior of these features, and thereby incurred significant additional complexity (see section 4.2).

All of our code and data is publicly available in an artifact [21]. Our infrastructure is reusable and could be applied to study other static analyses, other benchmarks, and other platforms (e.g., Node.js). Together, our infrastructure, methodology, and results can help guide the design of future analyses targeting real-world JavaScript code.

Contributions This paper makes the following contributions:

- We present a novel approach to quantifying the importance of language features causing low recall in JavaScript call graphs. The approach properly handles missing call graph edges with multiple root causes, and also inter-dependent calls, where an edge is missing due to the absence of another edge.
- We describe implementations of a dynamic call graph and dynamic flow trace analysis of function values for JavaScript, both of which handle several hard-to-analyze JavaScript features.

- We present results and key observations from applying our techniques for the ACG algorithm [25] and a suite of framework-based web applications.

The remainder of this paper is organized as follows. Section 2 provides background, and Section 3 describes our dynamic analyses. Section 4 presents our technique for automatically discovering root causes for missing edges. Section 5 gives details of our implementation. Section 6 describes the setup of our study, and Section 7 presents our results. Section 8 discusses related work, and Section 9 concludes.

2 Background

We first give some background on challenges for JavaScript static analysis and on call graph construction.

2.1 JavaScript analysis challenges

JavaScript programs often pose particularly difficult challenges for static analysis. JavaScript includes numerous dynamic and reflective language features that are difficult to analyze, and unfortunately these features are used often in practice [49]. We briefly present such features here; see previous work for detailed discussions (e.g., [30, 46, 49, 55]). Tricky features include:

- **Dynamic Property Accesses:** JavaScript object fields, or *properties*, can be accessed using the syntactic form `x[e]`, where `e` is an arbitrary expression evaluating to a string property name. Determining what memory locations may be accessed by an expression `x[e]` (fundamental to tracking data flow) can be a significant analysis challenge. Further, if `e` evaluates to a property name that does not exist on `x`, a write to `x[e]` *creates* the property rather than failing, making precise analysis even more challenging.
- **Eval:** JavaScript allows for evaluating arbitrary strings as code at runtime, most commonly via its `eval` construct or the `Function` constructor. This dynamically-evaluated code is known to pose significant problems for static analysis [30, 48].
- **With:** The `with` construct enables adding arbitrary variable bindings with a dynamically-constructed map [2]. As with `eval`, `with` usage complicates static analysis [46].
- **Getters and Setters:** A JavaScript property may be defined such that accessing the property actually invokes a *getter* or *setter* method with custom logic [12]. This feature makes it difficult to precisely identify the program locations where a function call can occur.
- **Reflective Calls:** JavaScript provides reflective methods to pass function parameters in flexible ways, e.g., binding the `this` parameter explicitly or passing arguments in an array [13]. Also, any function may read its formal parameters via a special `arguments` array, enabling variadic functions. Finally, any function may be legally invoked with *any* number of parameters, independent of how many formal parameters it declares. Together, these features complicate tracking of inter-procedural data flow.
- **Native Methods:** JavaScript and the web platform provide a large standard library whose implementation is typically opaque to static analysis; hence, models must be constructed for a large number of these "native" methods.

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

2.2 Call graph construction

In a static call graph, nodes represent program methods, and an edge from a to b means that a may invoke b at runtime.[2] The utility of a computed call graph CG can be measured in terms of *precision* and *recall*. Precision measures the number of infeasible edges in CG (edges for calls that cannot occur in any execution), while recall measures the number of feasible call edges (those that *can* occur in some execution) missing from CG. Recall will be 100% for any sound call graph construction technique, but as noted in Section 1, many practical techniques sacrifice soundness for improved scalability and precision. It is undecidable to compute the "ground truth" of possible calls for an arbitrary program, required to measure precision and recall perfectly. Our evaluation (and previous work [25, 44, 51, 57]) proceeds by exercising benchmarks using a best-effort process and then studying recall using the measured dynamic behaviors.

Static Flow Graphs Our technique also relies on obtaining a *static flow graph* from the static call graph analysis, to determine what dynamic data flow of function values was missed by the static analysis (see Figure 1 and further discussion in Section 4). In a flow graph, each node represents either a memory location (variables, object properties, etc.), a function value, or a call sites. Edges in the flow graph are defined as follows: if the call graph analysis determines that a function value may be read from (abstract) memory location m_1 and then written to location m_2 (i.e., it may be directly copied from m_1 to m_2), the static flow graph should include an edge from m_1 to m_2. So, flow graph edges should capture observed assignments of function values into variables and object properties, and passing of function values as parameters or return values to capture inter-procedural data flow. Additionally, for a call $m_i(...)$, the flow graph should contain an edge from m_i to a "callee" node for the call site (see example below). With this construction, the static call graph should have an edge from call site s to function f iff there is a path from f to the callee node for s in the flow graph.

Graph representations are standard in analyses that track data flow [54]. Further, any realistic JavaScript call graph construction algorithm must track function data flow, as JavaScript provides no basis for a cheaper technique (functions cannot be coarsely matched to possible call sites using types or even function arity). Hence, we expect extraction of flow graphs from JavaScript call graph analyses will be straightforward.

Example Figure 2 gives a small running example for illustrative purposes. Line 4 creates an object with two fields `MyName` and `MyPhone`, respectively holding functions `f1` and `f2`. Line 5 reads and invokes `f1` using a *static* property access (the property name is syntactically evident), whereas line 6 reads and invokes `f2` using a dynamic property access.

Figure 3 shows the flow graph constructed by a variant of the call graph builder we study [25] for the Figure 2 example. Edges represent the possible flow of function `f1` to the variable `v1`, then the object property `MyName`, and finally the call at line 5. Given this path, the static call graph includes an edge from `main` to `f1`. In contrast, the edge from the `MyPhone` property node to the call on line 6 is missing in Figure 3, due to the dynamic property access. Our approach can determine that this missing flow graph edge leads to a missing `main`-to-`f2` edge in the call graph, and further reason that a dynamic property access is the root cause of the missed edge.

[2] The call graph also includes information on which instruction in a, or *call site*, may invoke b.

```javascript
function main() {
  var v1 = function f1() { return "John"; }
  var v2 = function f2() { return "555-1234"; }
  var obj = { MyName: v1, MyPhone: v2 };
  obj.MyName();
  obj["My" + "Phone"]();
}
main();
```

Figure 2 Small example to illustrate our techniques.

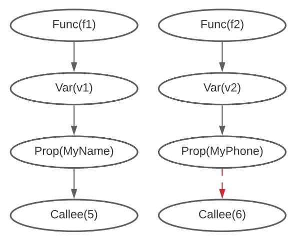


Figure 3 Flow graph for Figure 2. The red dashed edge is missing from the graph.

3 Dynamic Analyses

Our technique uses dynamic analyses to determine calls and data flows of function values occurring in executions of a program; this information is then compared with that in the static call graph and flow graph to detect missing flows (see Section 4). Here we describe the dynamic analyses at a high level; we discuss implementation challenges related to complex JavaScript language constructs (such as those listed in Section 2.1) in Section 5.

Dynamic Call Graphs A dynamic call graph captures the calls that occurred in an execution (or set of executions) of a program. As with static call graphs, nodes represent program methods and edges represent invocations between methods. At a high level, constructing dynamic call graphs only requires recording the actual functions invoked at each call instruction in some suitable data structure, and this type of analysis has been built many times before, including for JavaScript [29]. However, our analysis goes further by exposing call-related behaviors of some of the tricky JavaScript constructs outlined in Section 2.1, crucial for a more complete understanding of static call graph recall.

Dynamic Flow Traces Beyond dynamic call graphs, our technique requires *dynamic flow traces* to find gaps in the data flow reasoning of static call graph builders. A dynamic flow trace logs all data flow and invocation operations performed on function values. The trace includes an entry for each creation of a function value (e.g., an expression `function () { ... }`) and for each function call. It also includes an entry for each read or write of a function value to or from a variable or object property.

As an example, here is an excerpt of the dynamic flow trace for the code in Figure 2 (some details elided):

Create(`f1`,`2`); VarWrite(`v1`,`f1`,`2`);
Create(`f2`,`3`); VarWrite(`v2`,`f2`,`3`);
VarRead(`v1`,`f1`,`4`); PropWrite(`MyName`,`f1`,`4`);

VarRead(`v2,f2,4`); PropWrite(`MyPhone,f2,4`);
PropRead(`MyName,f1,5`); Invoke(`f1,5`);
PropRead(`MyPhone,f2,6`); Invoke(`f2,6`);

Each entry includes information on the function value being accessed and the location of the access (here, line numbers). For property accesses, our traces only record the accessed property name, as the call graph techniques we studied in our evaluation do not distinguish base objects of accesses. The trace could easily be extended to include base object identifiers if needed to study other analyses.

For handling of higher-order functions, the trace includes entries for parameter passing and returns of function values. A call passing a function as a parameter is treated as a "write" of a parameter variable, which can be read via the formal parameter in the callee. For returns, a **`return`** statement "writes" a special variable associated with the function's return value, which is "read" at the corresponding call site.

4 Missing Flow Detection

In this section, we describe our technique for discovering the *missing flows* explaining why a static call graph is missing an observed dynamic call graph edge. See Figure 1 for our overall architecture. Given a dynamic flow trace for a program, we first post-process the trace to discover the relevant *dynamic copies* for a function call (Section 4.1). Then, our technique matches these dynamic copies to the static flow graph, and automatically computes the missing flows relevant to each missing call edge (Section 4.2).

4.1 Finding Relevant Dynamic Copies for a Call

Given a dynamic flow trace and an invocation of function f at a call site, our technique computes the *dynamic copies* by which f was invoked at the site. Dynamic copies capture data flow of function values at runtime—they are the dynamic analogue of the possible data flow captured in a static flow graph (Section 2.2). A dynamic copy captures one of three operations on function values: (1) the value is *created* and then stored in some memory location; (2) the value is *copied* from one memory location to another; and (3) the value is read from a location and *invoked*. By computing the relevant dynamic copies for a particular call, our technique can expose which data flows may have been missed by the static analysis.

Pseudocode for finding relevant dynamic copies appears in Algorithm 1. We use subscripted t variables for trace entries. Given an entry t_c for a call invoking function f in trace T, FindDynamicCopies computes a list C of the relevant dynamic copies, starting at the creation of f and ending at the call. Each dynamic copy is represented in the form $t_{r'} \xrightarrow{t_w} t_r$, read as: the function was read from memory by $t_{r'}$, and then copied to the memory location read by t_r, via write t_w. The algorithm proceeds *backwards* through the trace, starting at t_c and reconstructing step-by-step how f was copied through memory to reach the call site.

Algorithm 1 first finds the read or create operation t_r for f most closely preceding t_c in the trace (line 3), corresponding to evaluation of e in an invocation $e(...)$.[3] C is then initialized with $t_r \xrightarrow{invoke} t_c$, with the *invoke* label indicating this is not a true copy, but instead the invocation of f.

[3] In certain corner cases, the closest preceding operation may not be the correct one; we discuss further under Limitations.

Algorithm 1 Finding dynamic copies for a call.

1: **procedure** FINDDYNAMICCOPIES(T, t_c)
2: $\quad f \leftarrow$ function invoked by t_c
3: $\quad t_r \leftarrow$ PRECEDINGREADORCREATE(T, t_c, f)
4: $\quad C \leftarrow [(t_r \xrightarrow{invoke} t_c)]$
5: $\quad$**while** t_r is not a CREATE operation **do**
6: $\quad\quad t_w \leftarrow$ MATCHINGWRITE(T, t_r, f)
7: $\quad\quad t_{r'} \leftarrow$ PRECEDINGREADORCREATE(T, t_w, f)
8: $\quad\quad C \leftarrow (t_{r'} \xrightarrow{t_w} t_r) :: C$
9: $\quad\quad t_r \leftarrow t_{r'}$
10: $\quad$**end while**
11: $\quad$**return** C
12: **end procedure**
13: **procedure** MATCHINGWRITE(T, t_r, f)
14: $\quad$**if** t_r reads variable x **then**
15: $\quad\quad$**return** PRECEDINGVARWRITE(T, t_r, f, x)
16: $\quad$**else if** t_r reads property *prop* **then**
17: $\quad\quad$**return** PRECEDINGPROPWRITE($T, t_r, f, prop$)
18: $\quad$**else if** t_r reads formal p of function f' **then**
19: $\quad\quad$*// preceding invoke of f' passing f to p*
20: $\quad\quad$**return** PRECEDINGINVOKE(T, t_r, f', f, p)
21: $\quad$**else if** t_r is return value of call to f' **then**
22: $\quad\quad$*// preceding return of f from f'*
23: $\quad\quad$**return** PRECEDINGRETURN(T, t_r, f', f)
24: $\quad$**end if**
25: **end procedure**

The loop at lines 5–10 discovers relevant dynamic copies by matching writes and reads backward in the trace. First, Line 6 finds the closest-preceding write operation t_w that updated t_r's location, using the MATCHINGWRITE procedure. MATCHINGWRITE's logic proceeds in cases, handing variables, object properties, formal parameters, and return values in turn. For a read of property *prop*, the pseudocode matches with the most recent write to *prop* on any object, matching the heap abstraction used by the call graph builder variants we study (see Section 6.1). For more precise call graph algorithms, the logic could easily be updated to also match the exact base object used in the property read operation. Once the matching write t_w is discovered, line 7 finds the closest-preceding read or create $t_{r'}$, which "produced" f for the write, and prepends a dynamic copy $t_{r'} \xrightarrow{t_w} t_r$ to C.

As an example, consider the call to `f2` on line 6 in Figure 2. Here are the relevant trace entries for that call visited by Algorithm 1:

CREATE(`f2,3`); VARWRITE(`v2,f2,3`);
VARREAD(`v2,f2,4`); PROPWRITE(`MyPhone,f2,4`);
PROPREAD(`MyPhone,f2,6`); INVOKE(`f2,6`);

Starting from the INVOKE entry, the closest preceding read of `f2` is the PROPREAD of `MyPhone` on line 6. So, C is initialized with PROPREAD(`MyPhone,f2,6`) $\xrightarrow{invoke}$ INVOKE(`f2,6`). The matching PROPWRITE for the read occurs on line 4, and its closest preceding read of `f2` is the VARREAD on line 4. Hence, we prepend a dynamic copy VARREAD(`v2,f2,4`) $\xrightarrow{t_{w_1}}$ PROPREAD(`MyPhone,f2,6`), where t_{w_1} = PROPWRITE(`MyPhone,f2,4`). Proceeding similarly, we reach the creation point of `f2` on line 3, prepend a dynamic copy CREATE(`f2,3`) $\xrightarrow{t_{w_2}}$ VARREAD(`v2,f2,4`), where t_{w_2} = VARWRITE(`v2,f2,3`), and terminate.

Limitations

Algorithm 1 assumes that the most-closely-preceding read of a function f in the trace matches the subsequent write or invocation involving f. In rare cases with parameter passing, this assumption may not hold, e.g.:

```
1 function foo(p, q) { p(); }
2 function bar() {}
3 var x = bar;
4 var y = bar;
5 foo(x, y);
```

Assume we are trying to discover the dynamic copies for the call to `bar` on line 1. Here is the relevant excerpt of the flow trace:

...; VarWrite(`x`,`bar`,`3`); VarWrite(`y`,`bar`,`4`); VarRead(`x`,`bar`,`5`);
VarRead(`y`,`bar`,`5`); Invoke(`foo`,`5`); VarRead(`p`,`bar`,`1`); Invoke(`bar`,`1`);

For the final Invoke of `bar`, the closest-preceding read is of formal parameter `p`. The matching "write" is the Invoke of `foo` on line 5. From here, the closest-preceding read of `bar` is from variable `y`, which is *not* the parameter that gets passed in `p`'s position. Hence, the analysis will discover an infeasible dynamic copy from the read of `y` to the read of `p`. This simple case could be handled by using source locations during matching, but in cases involving recursion, dynamic call stacks would also need to be tracked. We did not observe this behavior in any of our benchmarks, so we chose to employ the simpler technique of Algorithm 1.

In some cases, the dynamic flow trace may be missing entries relevant to dynamic copies, due to JavaScript features like native methods and `with` (Section 2.1) and also implementation limitations; see Section 5 for details. In such cases, our algorithm returns the subset of the relevant dynamic copies that it is able to reconstruct, and if possible notes a reason for its failure to find all copies.

4.2 Flow Graph Matching

Given relevant dynamic copies for a call c missed in the static call graph (discovered based on comparison with the dynamic call graph), we identify the missing flows for c by matching the dynamic copies to the static flow graph extracted from the call graph builder. (Section 2 described static flow graphs, and Figure 3 gave an example.) Algorithm 2 gives pseudocode for finding missing flows in a static flow graph. The routine FindMissingFlows takes as inputs a list of dynamic copies C produced by FindDynamicCopies in Algorithm 1, a static call graph CG, and the corresponding static flow graph FG. Its result is a set of missing flows R, where each missing flow is one of three types: (1) MissingFGNode, indicating a node is missing in the flow graph, (2) MissingFGPath, indicating a path is missing in the flow graph, and (3) DependentCall, for when the absence of a flow is due to the absence of another call in the call graph.

For a dynamic copy $t_{r'} \xrightarrow{t_w} t_r$, the algorithm first tries to identify corresponding flow graph nodes *fgSrc* and *fgDst* (lines 4 and 5). In most cases, this matching is straightforward, done either by matching code entities or matching an accessed memory location to the flow graph node that abstracts it (we elide the details). In some cases, the flow graph may not have a matching node, e.g., due to use of `eval` or due to an unmodelled property name from a dynamic property access. In such cases, we record an MissingFGNode entry in the result (lines 6–11).

If flow graph nodes *fgSrc* and *fgDst* are discovered, we next check for a *path* from *fgSrc* to *fgDst* in the flow graph (line 12). We must check for a path, rather than just an edge,

Algorithm 2 Finding missing flows in a flow graph for a call.

1: **procedure** FindMissingFlows(C,CG,FG)
2: $\quad R \leftarrow \emptyset$
3: $\quad$ **for** each dynamic copy $t_{r'} \xrightarrow{t_w} t_r \in C$ **do**
4: $\quad\quad$ $\mathit{fgSrc} \leftarrow$ FlowGraphNode($FG, t_{r'}$)
5: $\quad\quad$ $\mathit{fgDst} \leftarrow$ FlowGraphNode(FG, t_r)
6: $\quad\quad$ **if** $\mathit{fgSrc} =$ `null` **then**
7: $\quad\quad\quad R \leftarrow R \cup \mathsf{MissingFGNode}(t_{r'})$
8: $\quad\quad$ **end if**
9: $\quad\quad$ **if** $\mathit{fgDst} =$ `null` **then**
10: $\quad\quad\quad R \leftarrow R \cup \mathsf{MissingFGNode}(t_r)$
11: $\quad\quad$ **end if**
12: $\quad\quad$ **if** $\mathit{fgSrc} \neq$ `null` $\wedge$ $\mathit{fgDst} \neq$ `null` $\wedge$ NoPath($FG, \mathit{fgSrc}, \mathit{fgDst}$) **then**
13: $\quad\quad\quad R \leftarrow R \cup \mathsf{MissingFGPath}(\mathit{fgSrc}, \mathit{fgDst}, t_{r'}, t_w, t_r)$
14: $\quad\quad$ **end if**
15: $\quad\quad$ **if** t_w is a call **then**
16: $\quad\quad\quad f \leftarrow$ function invoked by t_w
17: $\quad\quad\quad$ **if** MissingFromCG(CG, t_w, f) **then**
18: $\quad\quad\quad\quad R \leftarrow R \cup \mathsf{DependentCall}(t_w, f)$
19: $\quad\quad\quad$ **end if**
20: $\quad\quad$ **end if**
21: $\quad$ **end for**
22: $\quad$ **return** R
23: **end procedure**

since the static analysis may use temporary variables and assignments not present in the source code. If no path is discovered, we note a MissingFGPath entry, retaining information about the dynamic copy to facilitate root cause labeling.

As an example, consider again the call to `f2` in Figure 2, and the corresponding dynamic copies described in Section 4.1. In the Figure 3 flow graph for the code, there are matching nodes for all the copy locations, but there is no path matching the final copy PropRead(`MyPhone,f2,6`) $\xrightarrow{invoke}$ Invoke(`f2,6`). So, the single missing flow computed for this case is a MissingFGPath entry with the details of this dynamic copy. Given this information, a root cause labeler can discover that the flow was missed due to the dynamic property access; see Section 6.2.

Dependent calls Lines 15–20 handle *dependent calls*, where a path corresponding to a parameter passing or return dynamic copy is missing from the flow graph due to *some other* missed call. Consider this example:

```
1 function f() { ... }
2 var x = { foo: function f2() { return f; } };
3 var y = x["fo"+"o"]();
4 y();
```

For the optimistic ACG call graph algorithm we use in our evaluation (see Section 6.1), the calls to `f2` at line 3 and to `f` at line 4 will be missing in the call graph. When finding missing flows for the line 4 call, a missing path for the function return dynamic copy at line 3 is discovered. However, the issue with the analysis is not that it does not model returns of function values; this flow was missed *because* the call target at line 3 was missed, so no flow could be discovered from the appropriate callee. Our discovery of missing flows must account for such cases, to enable accurate quantification of root causes.

To handle dependent calls, Algorithm 2 checks at line 15 if the "write" operation for the copy was a call. (Recall from Section 3 that calls are treated as the writes for parameter passing or function returns.) If so, and if the static call graph is missing the relevant target for the call (line 17), we add a DependentCall missing flow to the result (line 18).

When counting the frequency of root causes, for dependent calls, we *reuse* the root causes for one call as the root causes for the other. For the example above, the dynamic property access at line 3 is identified as the single root cause for the missing calls at lines 3 and 4. All results presented in Section 7 precisely account for dependent calls.

Root Cause Labeling Given a set of missing flows, quantification of root cause prevalence requires attributing a *root cause label* to each missing flow. The root cause labels may be specific to the call graph construction algorithm being studied, and must be developed with knowledge of the soundness gaps in the algorithm. Additionally, root cause labeling may be performed with different levels of granularity, depending on what information is required by the analysis developer. In Section 6.2, we discuss the root cause labeling strategies used in our example study of the ACG call graph algorithm [25].

5 Implementation

Dynamic analyses We implemented our dynamic call graph (DCG) and dynamic flow trace analyses (Section 3) atop the Jalangi framework [53],[4] which leverages source code instrumentation. While this instrumentation approach is more maintainable and portable than the alternatives, a downside is that the semantics of certain language constructs are not exposed in a straightforward way at the source level. In spite of source code instrumentation's limitations, one of its primary advantages is that it does not require modification of a JavaScript engine. Production JavaScript engines in browsers are challenging to modify, for two reasons: (1) they have complex implementations, so any change will require considerable engineering effort; and (2) they evolve rapidly, making it difficult to maintain an analysis. We use Jalangi2 to instrument JavaScript programs with our analysis code because it is easy to maintain and can work across different JavaScript engines. The tool allows us to perform analyses even when certain fragments of the source code are not instrumented. Our analyses contain significant extra logic to capture the behavior of several hard-to-analyze constructs as accurately as possible, despite the limitations of source instrumentation.

As an example, our DCG analysis exposes many callbacks from "native" library functions. Such callbacks occurred regularly in the benchmarks used in our study, e.g., using `Function.prototype.call`, as shown in this small example:

```
1    function foo() { }
2    foo.call(this);
```

Line 2 invokes `foo` via `call`, but Jalangi does not expose the invocation directly, as it cannot instrument `call`. Instead, Jalangi exposes the invocation of `call`, followed by the start of execution in `foo`, but with no explicit invocation of `foo`. To handle such cases, our DCG analysis maintains its own representation of the call stack. Upon invocation of a native method, a marker for the method is pushed on the call stack. Then, at the entry of a (non-native) method, if the top of our call stack is a native method marker, we record the fact that a native callback occurred. For the above case, the dynamic call graph will include

[4] We use version 2 of Jalangi, available at `https://github.com/Samsung/jalangi2`.

an invocation of the `call` native method at line 2, and also an invocation of `foo` from `call`, as desired.[5]

Our DCG analysis also exposes getter and setter calls, and calls to and from dynamically-evaluated code. For getters and setters, the analysis detects their presence via a library API [1]. If a getter or setter is detected at a property access, it is treated as a call site and the call edge is recorded. We leverage Jalangi's built-in support for dynamic code evaluation via `eval` or `new Function`; the relevant code string gets instrumented at runtime, so our analysis has visibility into calls into or out of such code.

Our dynamic flow trace analysis also includes special handling of some challenging JavaScript features. The analysis distinguishes getters and setter calls using specially-marked INVOKE entries, to enable tracking getter and setter use as a root cause. For uses of the `arguments` array to access parameters, we generate relevant property write entries at a function entry as "synthetic" entries (not corresponding to explicit source code). To handle `eval`-like constructs, any trace entry from the evaluated code includes a special source location marking it as from code executed via `eval`.

JavaScript has a very broad set of features and native methods requiring special handling, and our dynamic analyses still do not model all such features. For the flow trace analysis, in certain cases a property write or read occurs in an unmodelled native method, and hence is missed in the trace. The analysis generates special entries to model memory accesses performed by commonly-used library methods, such as `push` and `pop` on arrays. We have not fully modeled all reflective constructs like `Object.defineProperty` [14]. Also, use of the `with` construct can thwart our technique, as it is not fully supported by Jalangi. (We note that all relevant uses of `with` in our benchmarks appeared *within* an eval construct,[6] posing a severe challenge for static analysis.)

In terms of performance, we implemented some optimizations to reduce the size of the dynamic flow trace for larger benchmarks. First, we limited tracing to only those function values that could be involved in a missing edge in the static call graph, based on the creation site of the function. Second, we track a unique identifier for each function value using Jalangi's shadow memory functionality, and once the call site with the missing static call graph edge executes, we disable flow tracing for the corresponding value.

To generate dynamic call graphs and flow traces, we exercised our benchmarks manually and recorded the actions as Puppeteer [15] automation scripts to allow for repeatable runs; Section 6.3 details the coverage obtained for benchmarks in our study.

Missing Flow Detection The missing flow detection algorithms of Section 4 are implemented in `1154` lines of Python code. For the most part, detecting missing flows in the static flow graph given a dynamic flow trace was straightforward. Some effort was required to match source locations provided by WALA [58] for JavaScript constructs (our use of WALA is detailed in Section 6.1) with what was observed by the dynamic analyses. In the process of ensuring this matching was precise, we contributed a couple of fixes to WALA, and also found and fixed a longstanding issue with incorrect source locations in the Rhino JavaScript parser [5].[7]

---

[5] Our technique does not yet precisely handle cases with multiple levels of native calls, such as `Array.prototype.map.call(...)`; we plan to add further modeling for such cases in the future.

[6] For example, see this code from the Knockout framework: `https://tinyurl.com/1jxtrpz3`

[7] `https://github.com/mozilla/rhino/pull/809`

6 Study Setup

Here, we detail the setup of our study of root causes of missed call graph edges for framework-based web applications. We describe the ACG call graph algorithm used in our study (Section 6.1), describe how we performed root cause labeling for this algorithm (Section 6.2), and then present our benchmarks and how they were exercised (Section 6.3).

We note that the main purpose of our study was to show the potential of our techniques to give useful insights on the relative importance of different root causes for missed static call graph edges. We do *not* claim that the results for the benchmarks used in our study will generalize to any broad class of framework-based web applications. A study of a wider variety of benchmarks, to obtain generalizable insights on root causes across JavaScript applications, is beyond the scope of this work.

6.1 The ACG algorithm

In our evaluation, we studied variants of the approximate call graph (ACG) algorithm of Feldthaus et al. [25]. The ACG algorithm was designed to entirely skip analysis of many challenging JavaScript language features, while still providing good precision and recall for real-world programs. ACG leverages the insight that many dynamic property accesses in JavaScript are correlated [55], with a paired dynamic read and write used to copy a property from one object to another. By using a *field-based* handling of object properties [28] (treating each property as a global variable), ACG could ignore dynamic property accesses entirely and still provide good recall, assuming most accesses are correlated.

Feldthaus et al. [25] describe *pessimistic* and *optimistic* variants of ACG, differing in their handling of inter-procedural flow. Pessimistic ACG only tracks data flow across procedure boundaries in limited cases, whereas optimistic ACG performs full inter-procedural tracking. We performed root cause quantification for both variants in our study.

Our study uses the open-source implementation of ACG in WALA [58]. This implementation directly builds a flow graph during call graph building, which we serialize alongside the computed call graph. The WALA implementation also includes partial handling of the `call` and `apply` reflective constructs for parameter passing [13]. In the optimistic variant, interprocedural flow is handled fully for `call`, but only return values are handled for `apply` (as it passes parameters via arrays, which is hard to analyze). We confirmed via inspection that the WALA implementation of ACG has no handling of getters and setters, `eval`, and `with`.

6.2 Root Cause Labeling

We implemented root cause labeling for missing flows based on the gaps we observed in the WALA implementation [58] of the ACG algorithm [25]. For a different algorithm or implementation, some different root causes may be required, but we expect significant overlap, as several root causes pertain to challenging language features that many techniques handle unsoundly (e.g., `eval`). The referenced root cause names are also used when discussing their prevalence in Section 7.2.

For MissingFGNode (see section 4.2), in some cases, there is no node representing the creation of a function value in the flow graph. If the function was from the standard library, we assigned the label "Call to unmodelled native function," as WALA was likely missing a model for the function. In cases where the function was created via a call to `new Function` (unhandled by the ACG implementation), we assigned the label "Creation via Function constructor."

In other MissingFGNode cases, the node representing the call site itself is missing. For this case, a common root cause label is "Call to getter/setter," as getters and setters are not modeled by ACG. Also, the "Calls from unmodelled native functions" label captures cases where an unmodeled native function calls back into application code. Finally, for a dynamic property access, if the property name is never used as part of a non-dynamic property access, the flow graph may not have a node for the property, in which case we use the label "Dynamic Property Access."

For MissingFGPath, one possible root cause is "Dynamic Property Access," which can be identified by the corresponding dynamic reads / writes. For the pessimistic ACG variant, paths may be missing since the algorithm does not model passing function values as parameters or returning function values; we use the labels "Parameter Pass" and "Function return" for these scenarios. For both ACG variants, the "Parameter Pass" label is also used to reflect passing of parameters in an array via `Function.prototype.apply`.

In the case of dynamically-evaluated code (the "Use of Eval" and "Eval via new Function" labels), many relevant nodes may be missing from the static flow graph. We assign an appropriate root cause in these cases by recording in the flow trace which events occurred in dynamically-evaluated code (Section 5). Note that we *prioritize* the `eval`-related root causes over others; e.g., if there is a relevant dynamic property access in `eval`'d code, we will assign the `eval`-related root cause, even though it is possible the analysis also could not handle the property access. We chose this labeling due to the high difficulty of handling `eval` constructs in static analysis; for an analysis with significant support for `eval` a different choice may be appropriate.

Finally, as noted in Section 4.1, in certain cases we cannot compute all dynamic copies for a call. For these cases, our technique makes a base-effort attempt to assign an appropriate root cause label. "Call to bounded function" captures missing handling of the `Function.prototype.bind` feature [13]. The "Multiple levels of native functionality" label captures cases where native methods are invoked reflectively (see Footnote 5). Finally, we identify the "Use of With" root cause by tracing objects used in `with` statements and identifying when an unmatched variable corresponds to a `with` object property.

As Section 7.2 will show, dynamic property accesses are the most frequent root cause of missing call graph edges for our benchmarks. To further understand these root-cause accesses, we also implemented a finer-grained labeling for them, based on the expression used for the property name. This more granular labeling is described in Section 7.3.

6.3 Benchmarks and Harness

For benchmarks, our study used several programs from the TodoMVC suite [17]. TodoMVC contains many implementations of a simple web-based todo list application, with each implementation using a different web framework or language. The suite is designed to help developers compare different model-view-controller (MVC) frameworks. Because the suite contains idiomatic implementations of the same functionality across frameworks, it provides an opportunity to compare sources of missing call graph edges across frameworks.

To test with a larger web application, we also included OWASP Juice Shop [3], an AngularJs-based program that is a standard benchmark for security analyses. Counting the size of framework / library code for Juice Shop is difficult, as the code base does not clearly separate third-party code used as part of the web site from libraries used only to deploy the site; we conservatively estimated the framework / library code to be greater than 50 kLoC.

	Total LoC	Application LoC	Framework/ Library LoC	Application Stmt. Coverage
AngularJs	12091	256	11835	81.08%
Backbone	9003	216	8787	99.74%
KnockoutJs	1044	129	915	98.98%
KnockbackJs	15836	199	15637	99.73%
CanJs	11371	129	11242	100%
React	24855	383	24472	99.21%
Mithril	1433	252	1181	99.61%
Vue	7667	124	7543	97.73%
VanillaJs	751	561	190	98.10%
jQuery	9526	171	9355	99.59%
Juice Shop	>65000	15092	>50000	36%

Table 1 Benchmark Statistics.

Table 1 gives statistics for our benchmarks. The TodoMVC benchmarks are named based on the web framework that they use. The TodoMVC applications range from 751–24,855 lines of code, with framework sizes varying widely. We chose all eight of the JavaScript-framework-based implementations that worked with our infrastructure.[8] We also chose VanillaJS, which does not use any framework,[9] and jQuery, for comparison purposes.

To exercise the TodoMVC applications, we wrote a harness to cover as much application code as possible, and in the end our script achieved application code statement coverage of 97% or higher for nearly all benchmarks. We studied all uncovered code manually, and found that it was either dead code or could not be exercised in a single run of the application (e.g., for the AngularJs version, a small amount of code would only run if the app were used and then restarted in offline mode).

For Juice Shop, we were unable to exercise the application beyond fully completing its initial loading, explaining the significantly lower code coverage. Our infrastructure ran into scalability issues for deeper runs of Juice Shop, which we hope to fully address in the near future. Still, simply loading Juice Shop exercised a large amount of code (its flow trace was nearly 5 times larger than any fully-exercised TodoMVC benchmark), making a study of missed call edges for the loading portion of the execution interesting on its own.

In terms of running times for our tools, dynamic call graph and flow trace collection each took between 30 and 60 seconds for each TodoMVC benchmark, varying based on the amount of code executed; this overhead is comparable to previous Jalangi-based dynamic analyses [53]. Missing flow detection (Section 4) took time proportional to the size of the flow trace, ranging from around half a second (for VanillaJS) to around 10 minutes (for React). Overall running time for Juice Shop was much longer (more than an hour total) due to its size and the aformentioned scalability bottlenecks it exposed. We expect the missing flow detection times could be reduced significantly with a more optimized implementation.

[8] Some implementations used newer JavaScript language features not yet supported by Jalangi.
[9] All implementations use a common base JavaScript library, accounting for the library code in VanillaJS.

7 Results

In this section, we present results from performing root cause quantification for our benchmarks. The results show that our quantification techniques can provide interesting insights into the relative prevalence of different root causes for missing call graph edges. We first give recall measurements for our benchmarks using multiple metrics in Section 7.1. Then, we discuss the top root cause labels for missed call graph edges in Section 7.2 and insights gained from this data. Finally, we discuss results from performing a finer-grained labeling of missing flows related to dynamic property accesses (the most prevalent root cause) in Section 7.3.

7.1 Recall Measurements

We measured call graph recall for our benchmarks by comparing the ACG static call graphs with our collected dynamic call graphs. We first describe our methodology, and then present results. We also measured call graph precision for all benchmarks, but as our new techniques focus on root causes for low recall, we do not discuss the precision results here; they are presented in an extended version of the paper [22].

Methodology We used three different metrics to measure recall, suited to different client scenarios:

- **Call site targets:** the set of targets at each call site present in the dynamic call graph. This metric was used in the original ACG paper [25]. Recall is computed for each call site, and then averaged across call sites to produce recall for a benchmark. This metric is most relevant to clients like code navigation in an IDE.
- **Reachable nodes:** the set of reachable methods, where roots are the entrypoints in the dynamic call graph. This metric has been used in previous work [57], and is relevant to clients like dead-code elimination.
- **Reachable edges:** the set of call graph edges whose source method is present in the dynamic call graph. This metric is most relevant to clients doing deep inter-procedural analysis like taint analysis [26].

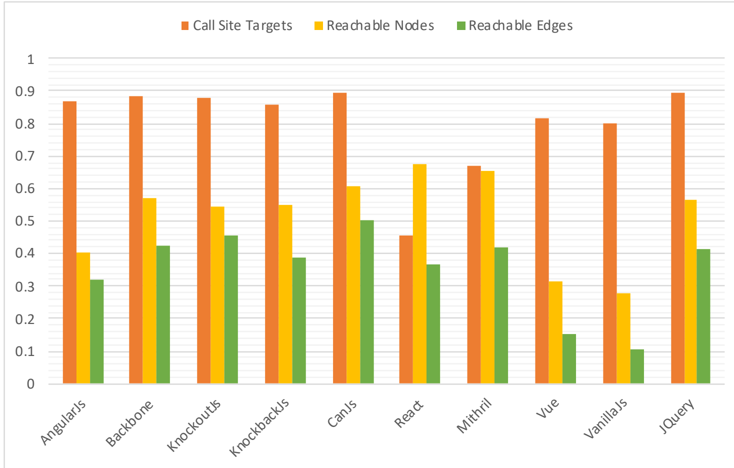
Given our collected data, we studied the following research questions:

- **RQ1:** How does recall vary across the three metrics?
- **RQ2:** How does recall vary across benchmarks?

Results Figure 4 gives detailed recall results for WALA's original ACG implementation for each TodoMVC benchmark, with results for the pessimistic variant in Figure 4a and for optimistic in Figure 4b. Average recall across the TodoMVC benchmarks is shown in Figure 5.

For RQ1, the data show that recall of ACG tends to suffer with more exacting metrics. The ACG paper [25] used the call site targets metric, and showed that both precision and recall were typically above 80% for their benchmarks. Figure 5 shows that for our benchmarks, while recall is above 80% for this metric for both the optimistic and pessimistic variants, recall decreases for the more exacting metrics, particularly for pessimistic analysis.

For RQ2, Figure 4 shows that recall can vary widely across benchmarks. In Section 7.2 we dig further into these differences, showing that root causes for low recall can also vary across the benchmarks. For the TodoMVC React benchmark, recall is very high for the optimistic analysis but quite low for pessimistic. In this case, the high recall for optimistic



(a) Pessimistic ACG.

(b) Optimistic ACG.

Figure 4 Detailed recall results for our three metrics across the benchmarks.

analysis comes at a cost of very low precision (less than 5% for reachable edges; see the extended version of the paper [22] for full details). We suspect that some initial imprecision spirals out of control for optimistic analysis for React, leading to poor precision. Previous work studied diagnosing imprecision root causes [20, 35, 60]; such a study is out of scope here. However, improving recall can lead to reduced precision, and this tradeoff must be minded when devising solutions to improving recall.

For Juice Shop, only the pessimistic ACG variant could run to completion; optimistic ACG could not complete within 64GB of memory. Pessimistic ACG missed 15,060 edges that were present in the dynamic call graph. Since our coverage for Juice Shop was significantly lower than the other benchmarks (see section 6.3), we do not quantify the precision and recall of pessimistic ACG for the benchmark, nor do we include it in aggregate statistics.

7.2 Root Cause Quantification

We present illustrative results from applying our techniques to quantify prevalence of root causes for missing call graph edges for our benchmarks. Space does not allow a full presentation

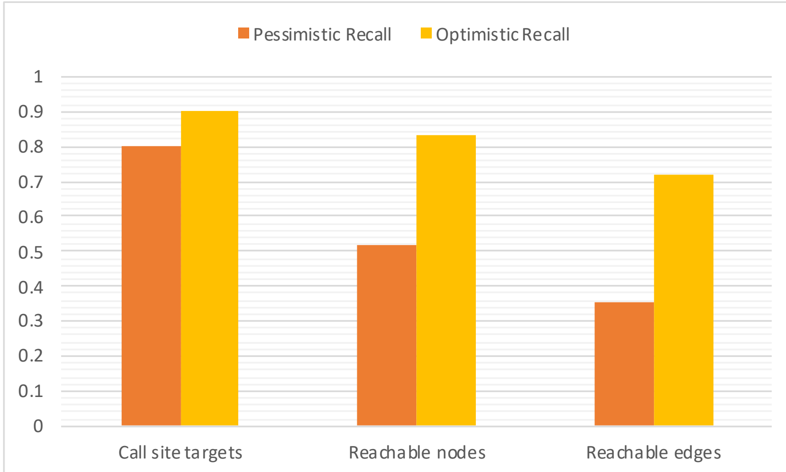


Figure 5 Average recall across benchmarks for original WALA ACG implementation.

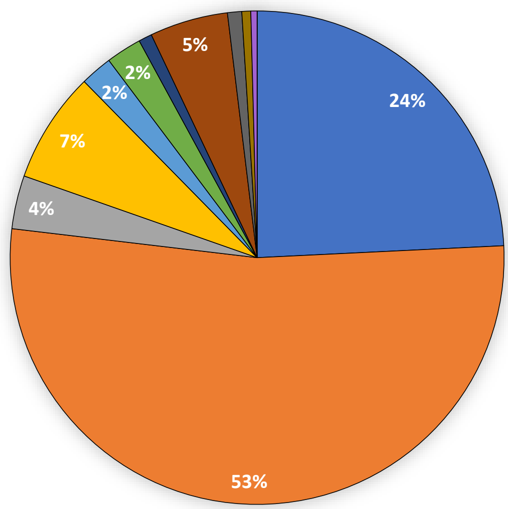


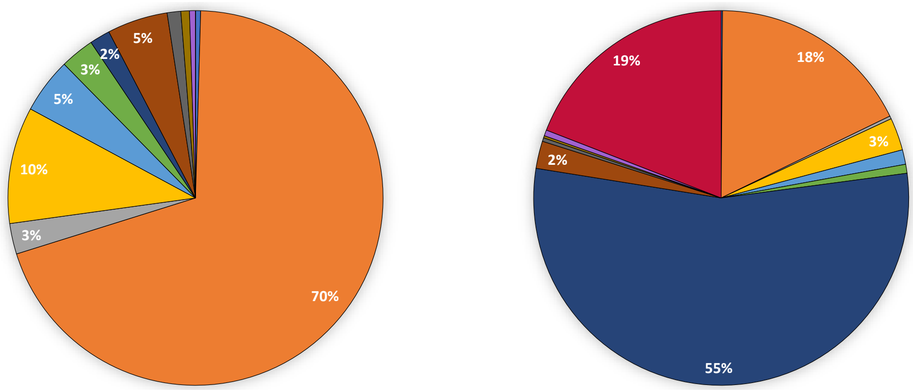
Figure 6 Original root causes for optimistic ACG across TodoMVC, before WALA improvements.

of all results; all experimental data is available in our artifact [21]. Here we focus on the following questions:

- **RQ3:** What are the most common root causes for missed call graph edges?
- **RQ4:** Does the relative importance of root causes vary across benchmarks?

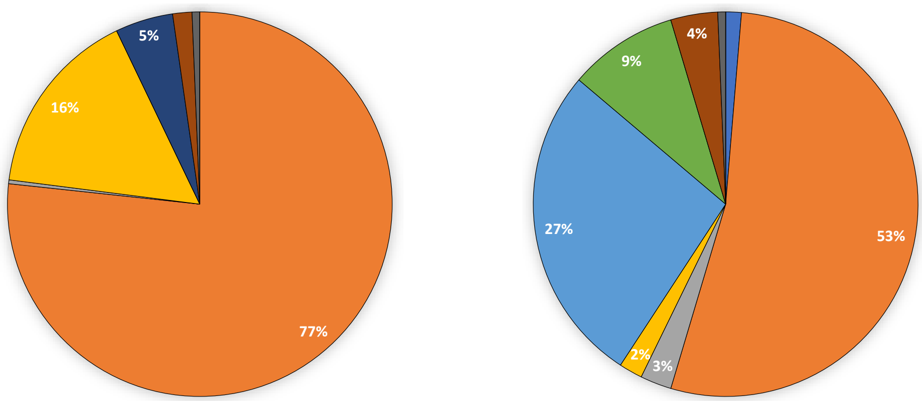
We compute root causes for each individual missed call edge in the static call graph, corresponding to the "Reachable edges" metric used to measure recall in Section 7.1. The color legend for the pie charts appears below Figure 8.

Using data to improve recall Figure 6 shows the prevalence of different root causes across the TodoMVC benchmarks for the optimistic variant of the original ACG implementation in WALA. When studying these root causes, we were surprised to see that 24% of missed call edges were due to calls to unmodeled standard library functions. Based on this data, we modified WALA to include basic models of many of these native functions. This change

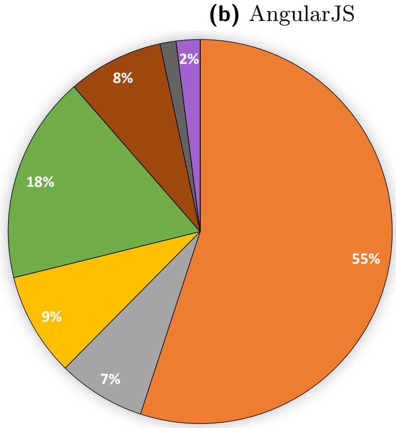


(a) Optimistic **(b)** Pessimistic

Figure 7 Improved root causes for ACG variants across TodoMVC, after WALA improvements.

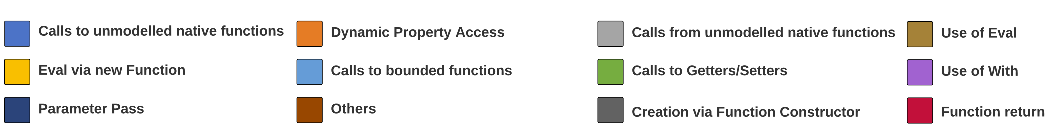


(a) React **(b)** AngularJS



(c) Vue

Figure 8 Root causes for three TodoMVC benchmarks for optimistic ACG.



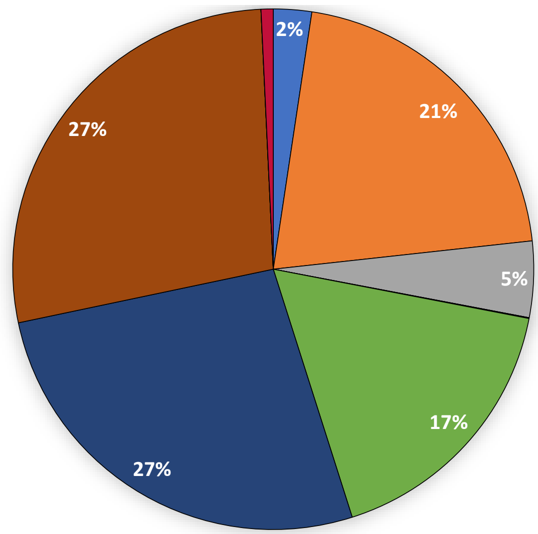


Figure 9 Root causes for pessimistic ACG for Juice Shop.

improved average recall for the pessimistic analysis by 2 percentage points to 37% (by the Reachable Edges metric); improvement for optimistic analysis was 5 percentage points, to 76%. These improvements show that quantifying root cause prevalence can guide an analysis developer to "quick wins" for improving analysis recall. The data in the remainder of this section were computed using the improved version of WALA ACG.

Top root causes Turning to RQ3, Figures 7a and 7b respectively show top root causes for pessimistic and optimistic ACG across the TodoMVC benchmarks (after improving WALA's native models). Comparing the two, we see a key difference is 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 optimistic and 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.

The "Others" label covers a small number of cases (5% overall) where our current scripts cannot yet find a root cause. In addition to the unhandled constructs and cases described in Section 5, our automated reasoning failed in rare cases due to a bug in WALA ACG's handling of `finally` blocks. During our work, we identified two other WALA ACG bugs that were fixed by the maintainers. Overall, our techniques successfully handle more than 95% of the missing call edges for our benchmarks, and we will continue to improve our tools to reduce the number of unhandled cases.

Focusing in on figure 7a, 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 Section 7.3. 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.

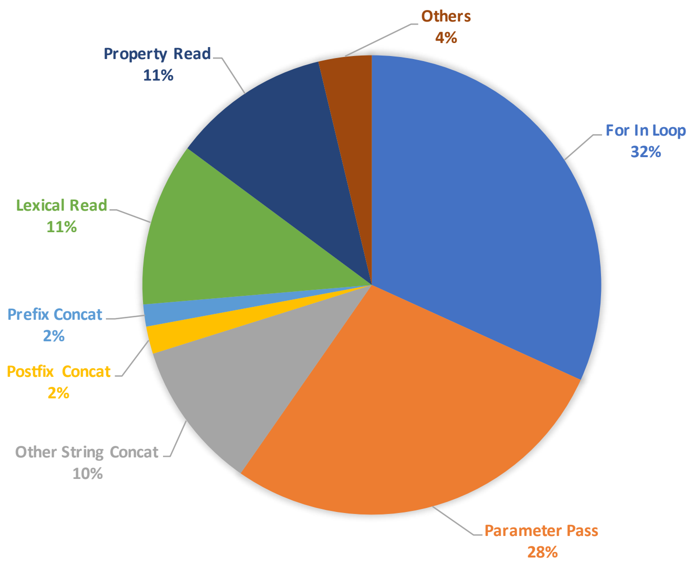


Figure 10 Finer-grained dynamic property access root causes for TodoMVC benchmarks.

Variance across benchmarks For RQ4, we use illustrative examples to show the variance in root cause prevalence across benchmarks. Figures 8a–8c respectively show root causes for the React, Angular, and Vue.js TodoMVC benchmarks, analyzed with optimistic ACG. While the most-prevalent root cause for each of these benchmarks was dynamic property accesses, the second-place root cause varies by benchmark: "Eval via new Function" is second for React, "Call to bounded functions" for AngularJS, and "Call to getter / setter" 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, it may be more worthwhile to improve handling of getters and setters than if the applications were more similar to the React benchmark.

Figure 9 shows root causes for the larger Juice Shop benchmark (analyzed with pessimistic ACG). Unfortunately, Juice Shop exercised gaps in our infrastructure's handling of tricky JavaScript constructs more heavily, particularly in the dynamic flow trace analysis. So, we could not compute proper root causes for 27% of missing call graph edges for Juice Shop. Still, the remaining data is interesting, particularly when compared to the pessimistic results for the TodoMVC benchmarks shown in Figure 7b. We see that handling returns of functions seems to be relatively less important than for the TodoMVC benchmarks, whereas handling of getters and setters is more important. Though making strong conclusions is difficult given the number of uncategorized edges in this case, these preliminary data again show the ability of our technique to expose benchmark-specific insights about causes of low recall.

To summarize, we have shown that our technique for quantifying root causes works across several benchmarks and can expose the most important root causes in aggregate and the differences between benchmarks. Since improving recall for JavaScript static analysis 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 in how to do so.

7.3 Name Flow for Dynamic Property Accesses

Given the importance of dynamic property accesses as a root cause in Section 7.2, we performed a finer-grained root cause labeling of these accesses. Our goal was to understand better how property names are computed for these accesses, to see if some targeted handling of the property name expressions could be useful. Recent work by Nielsen et al. [44] proposes just such a technique for analysis of Node.js code, via special handling of property name expressions that concatenate a string constant prefix or suffix to some other expression. We hoped to use root cause labeling to see if a similar technique could be effective for our web-based benchmarks.

We implemented a simple intra-procedural analysis using WALA [58] to label each root-cause dynamic property access based on how data flows into its property name expression (for an access `x[e]`, `e` is the property name expression). Aggregate results appear in Figure 10; our artifact has the complete data [21]. As shown in Figure 10, property names for root-cause dynamic accesses have a diverse set of sources. The largest single source are JavaScript's `for-in` loops for iterating over object properties, studied frequently in the literature as a challenge for static analysis (e.g., [19, 47]). However, they account for only 31% of cases in total, and many other sources exist. Property names are often passed in from outside the function containing the access, whether by parameter passing (28%) or variables in enclosing lexical scopes (12%); handling these cases may require inter-procedural tracking of property name value flow. Another major source is property reads (12%) (i.e., the property name is read from another object property), whose handling may again require deep tracking of value flow.

String concatenation cases comprise 14% of root-cause property name expressions. Only 4% of such expressions in our benchmarks had a string constant prefix or suffix, the type of expression targeted by Nielsen et al. [44]. Hence, the data show that their technique would likely have at most a small impact on recall for our benchmarks.

A deeper study of inter-procedural property name value flow could provide further insights on how these names are computed; this remains as future work. Still, our data show it is likely that a variety of challenges would need to be addressed to significantly improve ACG's recall with respect to dynamic property accesses.

7.4 Threats to Validity

As noted in Section 6, we do not claim generalizability of the results for our benchmarks to a broader set of JavaScript applications. In our benchmark suite, each individual framework is primarily exercised by a single TodoMVC benchmark, which may not be representative of other applications using that framework. Also, though our harness achieves high statement coverage for the TodoMVC benchmarks (Section 6.3), it is possible that certain application behaviors in those apps remain unexercised. Our dynamic coverage of Juice Shop was relatively low due to scalability limitations; more complete coverage is required to make strong conclusions about relative importance of root causes for that application. Finally, as noted in Section 5, our tooling still does not handle certain language features completely, which may have impacted our measurements.

8 Related Work

Here, we briefly discuss related studies of analysis effectiveness, and also other analysis frameworks and their applicability to framework-based web applications.

Root cause analysis Our work was partly inspired by a study of call graph recall for Java programs by Sui et al. [57]. As in that work, we measure recall with respect to dynamic analysis measurements, and we aim to determine which constructs are responsible for missing edges. Sui et al.'s approach used calling-context trees [18] and runtime tagging of reflective operations to determine language features impacting recall. Since functions are first-class values in JavaScript, we can trace function data flow directly to make this determination. Also, due to JavaScript's dynamic nature, the potential causes of missing edges and their usage patterns differ significantly from Java's problematic constructs.

Andreasen et al. present techniques for isolating soundness and precision issues in the TAJS static analyzer for JavaScript [20]. 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. When unsoundness is discovered for a program, delta debugging [61] is employed to find a reduced version of the program with the same unsoundness. From this reduced program, determining a root cause is often much simpler. In contrast to their work, which is focused on an analysis that strives for full soundness, our approach is targeted at analyses with deliberate unsoundness (for practicality), and aims to quantify the impact of different unsoundness root causes.

Reif et al. [61] present a system that provides methods for exposing sources of unsoundness in different Java call graph builders and also for measuring how frequently hard-to-analyze constructs appear in a set of benchmarks, yielding many useful practical insights. A difference with our work is that our technique can automatically connect specific uses of hard-to-analyze constructs to the corresponding missed call graph edges. This provides important additional information for JavaScript, since hard-to-analyze constructs can appear pervasively in JavaScript code, and not all occurrences cause call graph unsoundness.

Lhoták [37] also presents a comparison of static and dynamic call graphs for Java, aimed at finding sources of imprecision in the static call graph. Other work [20, 60] used dynamic analysis to generate traces and find root causes of imprecision in JavaScript static analyses, and Wei et al. [60] also provides suggestions to fix the root causes of imprecision. Lee et al. [35] produce 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 differs from all of these studies in its focus on recall rather than precision, which necessitates different techniques.

JavaScript Analyses Several analysis frameworks use abstract interpretation [24] to handle the interdependent problem of scalability and precision in JavaScript [32, 33, 36]. These frameworks have been steadily enhanced with techniques to improve precision and scalability when analyzing libraries, particularly TAJS [19, 31, 32, 43] and SAFE [34, 35, 36, 46, 47, 50]. While these techniques have shown enormous improvement in analyzing libraries like jQuery [10] and Lodash [11], they do not yet scale to complex MVC frameworks like React [4].

Other techniques use dynamic information to improve static analysis. Wei and Ryder introduced blended analysis [59], which uses dynamic analysis to aid static analysis in handling JavaScript's dynamic features. The dynamic flow analysis by Naus and Thiemann [41] generates flow constraints from a training run to infer types in JavaScript applications. (Their technique finds constraints by tracking operations on values; we determine how values are copied through memory, an orthogonal problem.) Lacuna [45] utilizes static and dynamic analysis to detect dead code in JavaScript applications; this work uses ACG and also uses TodoMVC applications for evaluation. While dynamic information can be very helpful in static analysis, improving pure static analysis is still desirable, as it can compute results without instrumenting and running the code and without inputs.

To analyze JavaScript applications that use the Windows runtime and other libraries, Madsen et al. proposed a use analysis that infers points-to specifications automatically [38]. It is unclear if their analysis will be effective for framework-based applications, where control flow is mainly driven by the framework, not the application. Also, we study applications using diverse frameworks from by many different developers, whereas [38] focuses on Windows libraries. For Node.js, Madsen et al. [39] presented a static analysis using call graphs augmented to represent event-driven control flow. To scale static analysis in server-side JavaScript applications in Node.js, Nielsen et al. present a feedback-driven static analysis to automatically identify the third-party modules that need to be analyzed [42]. Our focus, however, is on client-side MVC applications that often do not have clean module interfaces.

Other recent systems make use of pragmatic JavaScript static analyzers. The CodeQL system [7] includes an under-approximate call graph builder for JavaScript [8]. CodeQL's analysis is primarily intra-procedural, targeted toward taint analysis, and does not handle dynamic property accesses.[10] Møller et al. [40] describe a system for detecting breaking library changes in Node.js programs, based on an under-approximate analysis designed for high recall at the cost of some precision. Nielsen et al. [44] present a pragmatic modular call-graph construction technique for Node.js programs; we discussed its specialized handling of property name expressions in Section 7.3. For these approaches, our methodology could be used to quantify the importance of different causes of reduced recall. Salis et al. recently presented a pragmatic call graph builder for Python programs [51]; it would be interesting future work to extend our techniques to Python. Beyond dataflow-based reasoning about call graphs, other approaches to JavaScript static analysis include AST-based linting [9] and type inference [16, 23].

9 Conclusions

We have presented novel techniques for quantifying the relative importance of different root causes of missed edges in JavaScript static call graphs. We instantiated our approach to perform a detailed study of the results of the ACG algorithm 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. All of our code and data is publicly available. In future work, we 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. We also plan to use the techniques to further develop improved call graph builders and other JavaScript static analyses.

References

1 MDN Web Docs: Object.getOwnPropertyDescriptor(). `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor`, 2021. Accessed: 2021-01-11.

2 MDN Web Docs: with. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with`, 2021. Accessed: 2021-01-11.

3 OWASP Juice Shop. `https://owasp.org/www-project-juice-shop/`, 2021. Accessed: 2021-12-01.

[10] These details are based on personal communication with Max Schäfer in January 2021.

4 React – a JavaScript library for building user interfaces. `https://reactjs.org`, 2021. Accessed: 2021-01-11.

5 Rhino: JavaScript in Java. `https://github.com/mozilla/rhino`, 2021. Accessed: 2021-01-11.

6 Angular. `https://angular.io`, 2022. Accessed: 2022-05-13.

7 CodeQL for research. `https://securitylab.github.com/tools/codeql/`, 2022. Accessed: 2022-05-13.

8 CodeQL library for JavaScript: Call graph. `https://codeql.github.com/docs/codeql-language-guides/codeql-library-for-javascript/#call-graph`, 2022. Accessed: 2022-05-13.

9 ESLint. `https://eslint.org`, 2022. Accessed: 2022-02-25.

10 jquery. `https://jquery.com/`, 2022. Accessed: 2022-05-13.

11 Lodash. `https://lodash.com/`, 2022. Accessed: 2022-05-13.

12 MDN Web Docs: Defining Getters and Setters. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#defining_getters_and_setters`, 2022. Accessed: 2022-05-13.

13 MDN Web Docs: Function. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function`, 2022. Accessed: 2022-05-13.

14 MDN Web Docs: Object.defineProperty(). `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty`, 2022. Accessed: 2022-05-13.

15 Puppeteer. `https://pptr.dev/`, 2022. Accessed: 2022-05-13.

16 Tern: Intelligent JavaScript Tooling. `https://ternjs.net`, 2022. Accessed: 2022-02-25.

17 TodoMVC. `https://todomvc.com/`, 2022. Accessed: 2022-05-13.

18 Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, pages 85–96, 1997.

19 Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, part of SPLASH*, OOPSLA, pages 17–31, 2014.

20 Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the International Workshop on State Of the Art in Program Analysis*, SOAP, pages 31–36, 2017.

21 Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Artifact for "Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs", May 2022. `doi:10.5281/zenodo.6541325`.

22 Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs (Extended Version). 2022. URL: `https://arxiv.org/abs/2205.06780`.

23 Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-Il Choi. Type inference for static compilation of JavaScript. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2016.

24 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, POPL, pages 238–252, 1977.

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

26 Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–187, 2011.

27 Behnaz Hassanshahi, Hyunjun Lee, and Paddy Krishnan. Gelato: Feedback-driven and guided security analysis of client-side web applications. In *29th edition of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

28 Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 254–263, 2001.

29 Zoltán Herczeg and Gábor Lóki. Evaluation and comparison of dynamic call graph generators for JavaScript. In Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek, editors, *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2019*, pages 472–479, 2019.

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

31 Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 59–69, 2011.

32 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Static Analysis, 16th International Symposium*, SAS, pages 238–255, 2009.

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

34 Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. Weakly sensitive analysis for JavaScript object-manipulating programs. *Softw. Pract. Exp.*, 49(5):840–884, 2019.

35 Hongki Lee, Changhee Park, and Sukyoung Ryu. Automatically tracing imprecision causes in JavaScript static analysis. *Art Sci. Eng. Program.*, 4(2), 2020.

36 Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 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*, FOOL, 2012.

37 Ondrej Lhoták. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE, pages 37–42, 2007.

38 Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 499–509, 2013.

39 Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices*, 50(10):505–519, 2015.

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

41 Nico Naus and Peter Thiemann. Dynamic flow analysis for JavaScript. In *Trends in Functional Programming - 17th International Conference*, TFP, pages 75–93, 2016.

42 Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: feedback-driven static analysis of Node.js applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 455–465, 2019.

43 Benjamin Barslev Nielsen and Anders Møller. Value partitioning: A lightweight approach to relational static analysis for JavaScript. In *34th European Conference on Object-Oriented Programming*, ECOOP, pages 16:1–16:28, 2020.

44 Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of Node.js applications. In Cristian Cadar and Xiangyu Zhang, editors, *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, pages 29–41, 2021. `doi:10.1145/3460319.3464836`.

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

46 Changhee Park, Hongki Lee, and Sukyoung Ryu. 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, pages 73–84, 2013.

47 Changhee Park, Hongki Lee, and Sukyoung Ryu. Static analysis of JavaScript libraries in a scalable and precise way using loop sensitivity. *Softw. Pract. Exp.*, 48(4):911–944, 2018.

48 Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - A large-scale study of the use of eval in JavaScript applications. In *Object-Oriented Programming - 25th European Conference*, ECOOP, pages 52–78, 2011.

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

50 Sukyoung Ryu, Jihyeok Park, and Joonyoung Park. Toward analysis and bug finding in JavaScript web applications in the wild. *IEEE Softw.*, 36(3):74–82, 2019.

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

52 Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 165–174, 2013.

53 Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 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, pages 488–498. ACM, 2013.

54 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In David Clarke, Tobias Wrigstad, and James Noble, editors, *Aliasing in Object-Oriented Programming*. Springer, 2013. `doi:10.1007/978-3-642-36946-9_8`.

55 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *Object-Oriented Programming - 26th European Conference*, ECOOP, pages 435–458, 2012.

56 Stack Overflow 2020 Developer Survey: Web Frameworks. `https://insights.stackoverflow.com/survey/2020#technology-web-frameworks`, 2020. Accessed: 2022-05-13.

57 Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *International Conference on Software Engineering*, ICSE, pages 1049–1060, 2020.

58 T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net`.

59 Shiyi Wei and Barbara G. Ryder. A practical blended analysis for dynamic features in JavaScript. Technical Report TR-12-18, Virginia Tech, 2012. URL: `https://vtechworks.lib.vt.edu/handle/10919/19421`.

60 Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. Revamping JavaScript static analysis via localization and remediation of root causes of imprecision. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 487–498, 2016.

61 Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.