UNIVERSITY OF CALIFORNIA
RIVERSIDE

Pushing the Boundaries of JavaScript Static Analysis

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Madhurima Chakraborty

June 2025

Dissertation Committee:

Professor Manu Sridharan, Chairperson
Professor Rajiv Gupta
Professor Zhijia Zhao
Professor Chengyu Song

Copyright by
Madhurima Chakraborty
2025

The Dissertation of Madhurima Chakraborty is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I want to begin by thanking my advisor, Manu Sridharan, for being a constant source of support and guidance over the years. His calm presence, insightful feedback, and endless patience made this difficult journey a little easier and a lot more meaningful. I've learned so much from him—lessons I will carry with me wherever I go next. I also appreciate the time and insightful feedback provided by my committee members—Rajiv Gupta, Chengyu Song, and Zhijia Zhao—which helped to shape this dissertation.

To my collaborators in academia and industry, your engaging conversations and shared curiosity have shaped my growth as a researcher. I'm especially grateful to Anders Møller. His way of thinking has had a big impact on how I approach problems and do research, and he continues to inspire me to think differently.

Baba and Maa, you ever asked how far along I was or what I had accomplished — only whether I had eaten, if I was sleeping well, and whether I was happy. It used to frustrate me at times, but over the years, I've come to understand that this was your way of loving me, of making sure I was okay in the ways that truly mattered. Thank you for your steady love and for always being there.

Didi, I couldn't have done this without you. Your strength, kindness, and quiet confidence have carried me through more than you know. You've always been there with the right words, the right silence, and a kind of support that asks for nothing in return. Thank you for being my rock. Bhai, knowing that you were there for Didi while I was away gave me comfort and peace of mind.

Leo, thank you for always standing by me, especially when things felt overwhelming. Your belief in me, even when I couldn't believe in myself, gave me strength. I'm truly grateful for your constant support, patience, and kindness. Aunty, your warmth and prayers have meant a lot.

To my friends, thank you for the check-ins, the laughter, for celebrating the small victories, and for listening to my endless ramblings. You kept me grounded and reminded me there was more to life than deadlines and drafts.

This work carries the fingerprints of everyone who walked alongside me on this journey. Thank you for believing in me and for helping me reach this point.

ABSTRACT OF THE DISSERTATION

Pushing the Boundaries of JavaScript Static Analysis

by

Madhurima Chakraborty

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2025
Professor Manu Sridharan, Chairperson

Static analysis of JavaScript remains a notoriously difficult problem due to the language's highly dynamic and reflective nature. Features such as dynamic property accesses, higher-order functions, prototype-based inheritance, and event-driven control flow introduce fundamental obstacles to sound and scalable analysis. A central challenge is the construction of precise call graphs, which serve as a critical prerequisite for most interprocedural analyses. Imprecise call graphs often result in high false-positive rates, undermining the effectiveness of downstream analyses and increasing the manual burden on developers. This thesis addresses core limitations in JavaScript static analysis. The first work presents an empirical study of call graph unsoundness in real-world JavaScript applications. We systematically identify and categorize the root causes of missing call graph edges in state-of-the-art call graph construction techniques, such as dynamic property accesses, higher-order functions, and indirect call patterns. This analysis reveals fundamental weaknesses in existing techniques and provides actionable insights for improving call graph recall. The second work introduces an indirection-bounded call graph construction technique. Motivated by the observation that real-world JavaScript code typically exhibits shallow levels of indirection in data flows, the technique bounds the depth of object and function dereferencing to reduce analysis overhead. This approach achieves up to a $2X$ speedup on large Node.js applications while maintaining nearly identical precision and incurring a modest reduction in recall, making whole-program analysis more tractable at scale. The final work focuses on statically detecting promise misuses, a class of bugs that arise from JavaScript's asynchronous programming model through PromisePatrol–an abstract interpretation framework that models promise creation, state transitions, and handler attachment interprocedurally. The

analysis identifies missing fulfillment and rejection handlers that may result in silent loss of asynchronous results or unhandled rejections. Applied to 100 popular *npm* packages, PromisePatrol uncovers 15 previously undocumented instances of missing handlers across 7 projects, demonstrating the practical utility of the approach in identifying real-world asynchronous defects. Together, these techniques advance the state of JavaScript static analysis by improving the soundness of call graph construction, enabling scalable interprocedural reasoning, and providing effective tools for detecting asynchronous programming errors in complex, real-world codebases.

Contents

1 Introduction **1**

2 Automatic root cause quantification for missing edges in JS call graphs **4**

2.1 Introduction 5

2.2 Background 8

2.2.1 JavaScript analysis challenges 8

2.2.2 Call graph construction 10

2.3 Dynamic Analyses 12

2.4 Missing Flow Detection 13

2.4.1 Finding Relevant Dynamic Copies for a Call 14

2.4.2 Flow Graph Matching 17

2.5 Implementation 20

2.6 Study Setup 22

2.6.1 The ACG algorithm 22

2.6.2 Root Cause Labeling 23

2.6.3 Benchmarks and Harness 25

2.7 Results 27

2.7.1 Recall Measurements 27

2.7.2 Root Cause Quantification 30

2.7.3 Name Flow for Dynamic Property Accesses 35

2.7.4 Threats to Validity 36

2.8 Related Work 36

2.9 Conclusions 39

3 Indirection-Bounded Call Graph Analysis **40**

3.1 Introduction . . . 41

3.2 Motivating Examples . . . 44

3.3 Dynamic Indirection Bound Estimation . . . 47

3.3.1 Language . . . 47

3.3.2 Dynamic Analysis . . . 48

3.3.3 Study Results . . . 52

3.4 Indirection-Bounded Call Graph Construction . . . 55

3.4.1 Analysis Formulation . . . 55

3.4.2 Simplified Wave Propagation . . . 56

3.4.3 Adding Bounds . . . 58

3.4.4 Bounded ACG . . . 60

3.5 Evaluation . . . 61

3.5.1 Benchmarks . . . 61

3.5.2 Experimental Configuration . . . 64

3.5.3 Results . . . 64

3.5.4 Threats to Validity . . . 67

3.6 Related Work . . . 68

3.7 Conclusion . . . 69

4 Promises Unkept: Detecting Mismanaged Promises in JS **71**

4.1 Introduction . . . 72

4.2 Promise Mismanagement Defects . . . 74

4.3 Approach . . . 78

4.3.1 Call Graph Construction . . . 79

4.3.2 Identification of Promise-Relevant Functions . . . 79

4.3.3 Entry Anchor Determination . . . 81

4.3.4 Promise-Aware Abstract Interpretation . . . 81

4.4 Evaluation . . . 91

4.4.1 RQ1: Precision of Handler Detection . . . 92

4.4.2 RQ2: Verifiability of Absence of Handler Bugs . . . 93

4.4.3 RQ3: Scalability and Performance 94
4.4.4 RQ4: Impact of handler necessity heuristics 95
4.4.5 Illustrative Case Studies 95
4.4.6 Discussion 97
4.5 Related Work 99
4.6 Conclusion 101

5 Conclusions **103**

Bibliography **105**

List of Figures

2.1 Overview of our methodology. 6
2.2 Small example to illustrate our techniques. 11
2.3 Flow graph for Figure 2.2. The red dashed edge is missing from the graph. 12
2.4 Detailed recall results for our three metrics across the benchmarks. 29
2.5 Average recall across benchmarks for original WALA ACG implementation. 30
2.6 Original root causes 31
2.7 Improved root causes 31
2.8 Root causes for three TodoMVC benchmarks for optimistic ACG. 31
2.9 Root causes for pessimistic ACG for Juice Shop. 32
2.10 Finer-grained dynamic property access root causes 34

3.1 Example programs that illustrate indirection levels for function calls. 45
3.2 Example programs that illustrate indirection levels for object field accesses. 46
3.3 Analysis time and recall for JELLY 66
3.4 Analysis time and recall for WALA 66
3.5 Relative analysis time and recall for JELLY 67

4.1 Lifecycle of a JavaScript Promise and points of handler attachment. 75
4.2 Relationship between code size and analysis time for `npm-100` benchmark. . 94

List of Tables

2.1 Benchmark Statistics. 25

3.1 Statement types 48
3.2 Number of call edges with each minimum bound 53
3.3 Node.js benchmarks used for for experiments with JELLY 62
3.4 Web and Mobile benchmarks used for experiments with WALA. 63

4.1 Per-project classification of warnings produced by PromisePatrol 92
4.2 Summary of projects with Promise usage 93

Chapter 1

Introduction

JavaScript plays a central role in the modern software ecosystem, powering both interactive front-end applications in the browser and large-scale server-side systems through platforms like Node.js. As these applications evolve in scale and complexity, ensuring correctness, security, and maintainability becomes increasingly important. Static analysis tools offer a scalable solution for detecting bugs, enforcing coding standards, and supporting tasks such as auto-completion, type inference, and refactoring. However, applying static analysis to JavaScript remains an enduring challenge.

Unlike statically typed languages, JavaScript exhibits highly dynamic behavior: objects can be modified at runtime, functions can be assigned to variables or object properties and invoked dynamically, and asynchronous constructs like Promises and event-driven callbacks introduce non-deterministic and non-local control flow. These features obscure the program's structure and make the control and data flows highly implicit. As a result, static analyses struggle to maintain soundness—capturing all possible behaviors—without sacrificing precision or scalability.

A cornerstone of interprocedural static analysis is the call graph, which encodes the set of potential function calls that may occur during execution. Call graphs form the basis for many downstream analyses, including alias tracking, taint propagation, and control-flow-sensitive optimizations. However, constructing accurate call graphs for JavaScript is notoriously difficult. Features such as dynamic property access, higher-order functions, prototype-based inheritance, and runtime code loading (e.g. `eval`) can cause static analyz-

ers to miss critical call edges or over-approximate the set of possible callees. In practice, incomplete call graphs result in unsound analyses that miss true bugs, while imprecise graphs yield excessive false positives, diminishing the utility of analysis results.

This dissertation addresses these foundational limitations by rethinking how we diagnose, improve, and apply call graph construction in the context of dynamic JavaScript programs. We propose three key contributions, each rooted in real-world challenges and designed for practical impact.

First, we introduce a systematic method for diagnosing the sources of call graph unsoundness. Our root cause analysis technique operates on real-world JavaScript applications and identifies concrete language features and coding patterns responsible for missing edges in the computed call graph. By attributing unsoundness to specific constructs—such as indirect calls via computed property accesses, we enable both developers and tool designers to understand where existing techniques fall short and how recall can be improved in a targeted manner.

Second, we present an optimization strategy that improves the scalability of call graph construction by bounding the depth of indirection explored during analysis. Empirical studies show that most useful call edges arise within a small number of dereference steps. By bounding the depth of indirection of object pointers and higher-order functions, we significantly reduce the computational overhead without sacrificing excessive precision or recall. Our indirection-bounded algorithm integrates with existing call graph construction techniques such as field-based and wave-propagation analyses, yielding on an average 2X speedups in large Node.js applications with nearly identical precision and acceptable impact on recall.

Third, we leverage these improvements to statically detect a class of asynchronous programming bugs involving Promises. Promises are a central abstraction for managing asynchronous control flow in modern JavaScript. While they help structure asynchronous logic more cleanly than callbacks, they also introduce new pitfalls: missing fulfillment handlers may result in ignored values, while missing rejection handlers can cause unhandled exceptions. We develop PromisePatrol, a static analysis framework based on abstract interpretation, which models the lifecycle of Promises—tracking their creation, resolution/re-

jection, and handler attachment across function boundaries. Our analysis identifies common misuse patterns that are difficult to expose via testing alone, and uncovers previously undocumented bugs in popular npm packages.

The rest of this dissertation is organized as follows. Chapter 2 is adapted from [15] and formalizes our root cause quantification technique and applies it to evaluate the recall limitations of state-of-the-art call graph algorithms. Chapter 3 is adapted from [14] describes our indirection-bounded call graph construction approach and demonstrates its impact on performance, precision and recall. Chapter 4 introduces PromisePatrol, our analysis framework for detecting Promise mismanagement bugs. Finally, Chapter 5 discusses the broader implications of our findings and outlines future research directions in building sound, scalable static analyses for dynamic languages like JavaScript.

Chapter 2

Automatic root cause quantification for missing edges in JS call graphs

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 chapter, 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 ap-

plications 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.

Our contribution lies in developing an open source framework quantifying the potential root causes of low recall of static call graphs. We posit that the findings of this study will guide future research in improving JavaScript based static analysis techniques.

2.1 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 [34, 36] or correctness of library updates [71]). Unfortunately, call graph construction for real-world JavaScript programs poses significant challenges, particularly for client-side code in web applications. Modern web applications are increasingly built using sophisticated frameworks like React [92] and AngularJS [33].[1] Sophisticated recent JavaScript static analysis frameworks [44, 46, 53, 99] 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 [26, 32, 71, 79]. While these techniques have been shown effective in certain domains, their unsoundness can lead to missing many edges when analyzing framework-based applications [36], 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 [96]. Further, there are complex tradeoffs involved in adding support for these constructs, as a

[1] A recent Stack Overflow developer survey shows popularity of these frameworks is growing, with total usage surpassing older libraries like jQuery [104].

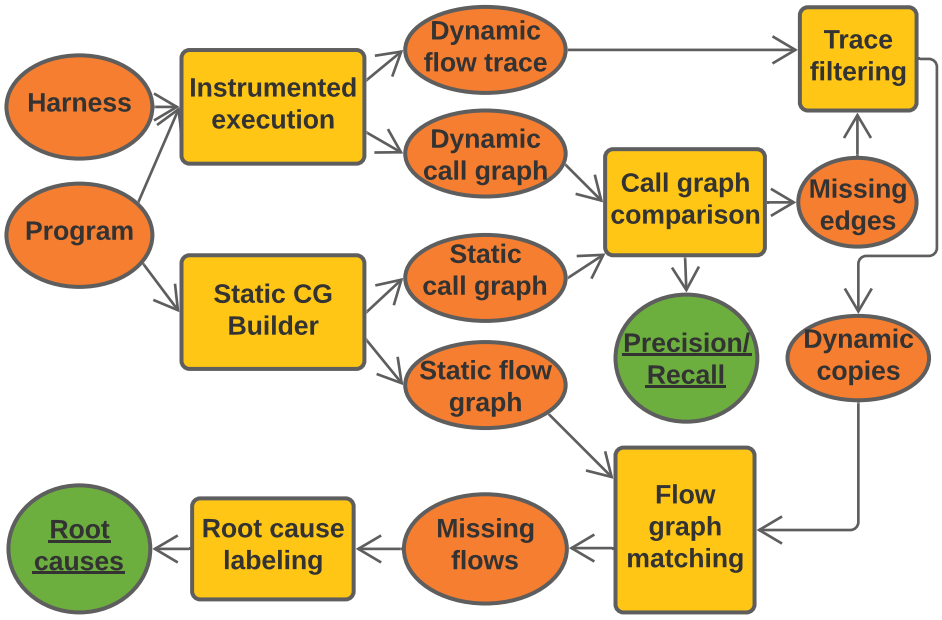


Figure 2.1: Overview of our methodology.

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 chapter presents a novel technique for *automatic root cause quantification* for missing edges in JavaScript call graphs. Figure 2.1 gives an overview of our technique. Given a program, a static call graph builder enhanced to also export static flow graphs (see Section 2.2.2), and a harness for exercising the program, our technique automatically finds *missing flows*, data 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. [26], as implemented in the WALA framework [27], 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 [79], 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 2.4.2).

All of our code and data is publicly available in an artifact [16]. 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 chapter 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 [26] and a suite of framework-based web applications.

The remainder of this chapter is organized as follows. Section 2.2 provides background, and Section 2.3 describes our dynamic analyses. Section 2.4 presents our technique for automatically discovering root causes for missing edges. Section 2.5 gives details of our implementation. Section 2.6 describes the setup of our study, and Section 2.7 presents our results. Section 2.8 discusses related work, and Section 2.9 concludes.

2.2 Background

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

2.2.1 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 [96]. We briefly present such features here; see previous work for detailed discussions (e.g., [42, 86, 96, 103]). 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 [42, 95].

- **With:** The `with` construct enables adding arbitrary variable bindings with a dynamically-constructed map [64]. As with `eval`, `with` usage complicates static analysis [86].

- **Getters and Setters:** A JavaScript property may be defined such that accessing the property actually invokes a *getter* or *setter* method with custom logic [65]. 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 [66]. 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.

2.2.2 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 2.1, 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 [26, 79, 98, 106]) 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 2.1 and further discussion in Section 2.4). 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 [102]. Further, any realistic JavaScript call graph construction algorithm must track function data flow, as

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

```
1 function main() {
2   var v1 = function f1() { return "John"; }
3   var v2 = function f2() { return "555-1234"; }
4   var obj = { MyName: v1, MyPhone: v2 };
5   obj.MyName();
6   obj["My" + "Phone"]();
7 }
8 main();
```

Figure 2.2: Small example to illustrate our techniques.

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 2.2 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 2.3 shows the flow graph constructed by a variant of the call graph builder we study [26] for the Figure 2.2 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 2.3, 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.

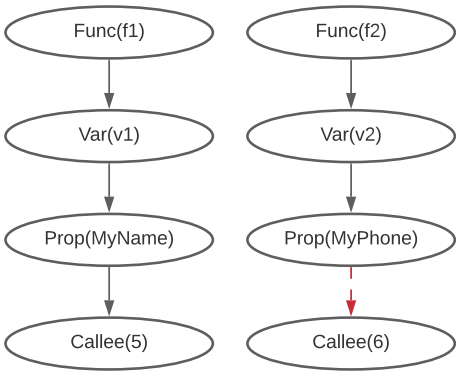


Figure 2.3: Flow graph for Figure 2.2.
The red dashed edge is missing from the graph.

2.3 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 2.4). 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 2.2.1) in Section 2.5.

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 [39]. However, our analysis goes further by exposing call-related behaviors of some of the tricky JavaScript constructs outlined in Section 2.2.1, 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 2.2 (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.

2.4 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 2.1 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 2.4.1). 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 2.4.2).

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 2.2.2). 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 sub-scripted 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.

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 2.6.1). For more precise call graph algorithms, the logic could

[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:     f ← function invoked by t_c
3:     t_r ← PrecedingReadOrCreate(T, t_c, f)
4:     C ← [(t_r --invoke--> t_c)]
5:     while t_r is not a Create operation do
6:         t_w ← MatchingWrite(T, t_r, f)
7:         t_r' ← PrecedingReadOrCreate(T, t_w, f)
8:         C ← (t_r' --t_w--> t_r) :: C
9:         t_r ← t_r'
10:    end while
11:    return C
12: end procedure
13: procedure MatchingWrite(T, t_r, f)
14:    if t_r reads variable x then
15:        return PrecedingVarWrite(T, t_r, f, x)
16:    else if t_r reads property prop then
17:        return PrecedingPropWrite(T, t_r, f, prop)
18:    else if t_r reads formal p of function f' then
19:        // preceding invoke of f' passing f to p
20:        return PrecedingInvoke(T, t_r, f', f, p)
21:    else if t_r is return value of call to f' then
22:        // preceding return of f from f'
23:        return PrecedingReturn(T, t_r, f', f)
24: end procedure
```

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 2.2. 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 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 2.2.1) and also implementation limitations; see Section 2.5 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.

2.4.2 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 2.2 described static flow graphs, and Figure 2.3 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 6–9).

If flow graph nodes *fgSrc* and *fgDst* are discovered, we next check for a *path* from *fgSrc* to *fgDst* in the flow graph (Line 10). We must check for a path, rather than just an edge, 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

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

```
1: procedure FindMissingFlows(C, CG, FG)
2:     R ← ∅
3:     for each dynamic copy t_r' --t_w--> t_r ∈ C do
4:         fgSrc ← FlowGraphNode(FG, t_r')
5:         fgDst ← FlowGraphNode(FG, t_r)
6:         if fgSrc = null then
7:             R ← R ∪ MissingFGNode(t_r')
8:         if fgDst = null then
9:             R ← R ∪ MissingFGNode(t_r)
10:        if fgSrc ≠ null ∧ fgDst ≠ null ∧ NoPath(FG, fgSrc, fgDst) then
11:            R ← R ∪ MissingFGPath(fgSrc, fgDst, t_r', t_w, t_r)
12:        if t_w is a call then
13:            f ← function invoked by t_w
14:            if MissingFromCG(CG, t_w, f) then
15:                R ← R ∪ DependentCall(t_w, f)
16:    return R
17: end procedure
```

information about the dynamic copy to facilitate root cause labeling.

To illustrate this, consider the call to `f2` in Figure 2.2, and the corresponding dynamic copies described in Section 2.4.1. In Figure 2.3 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 2.6.2.

Dependent calls Lines 12–15 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 2.6.1), 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 12 if the "write" operation for the copy was a call. (Recall from Section 2.3 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 14), we add a DependentCall missing flow to the result (line 15).

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 2.7 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 2.6.2, we discuss the root cause labeling strategies used in our example study of the ACG call graph algorithm [26].

2.5 Implementation

Dynamic analyses We implemented our dynamic call graph (DCG) and dynamic flow trace analyses (Section 2.3) atop the Jalangi framework [100],[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 the programs with our analysis code because it is easy to maintain and can work across different 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 DCG will include an invocation of the `call` native method at line 2, and also an invocation of `foo` from `call`, as desired. [5]

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

[5] Our technique does not yet precisely handle cases with multiple levels of native calls, such as `Array`.

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 [63]. 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 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` [67]. 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.

`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`

To generate dynamic call graphs and flow traces, we exercised our benchmarks manually and recorded the actions as Puppeteer [112] automation scripts to allow for repeatable runs; Section 2.6.3 details the coverage obtained for benchmarks in our study.

Missing Flow Detection The missing flow detection algorithms of Section 2.4 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 [27] for JavaScript constructs (our use of WALA is detailed in Section 2.6.1) 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 [74].[7]

2.6 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 2.6.1), describe how we performed root cause labeling for this algorithm (Section 2.6.2), and then present our benchmarks and how they were exercised (Section 2.6.3).

Our study demonstrates the potential of our techniques to provide valuable insights into 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 into root causes across JavaScript applications is beyond the scope of this work.

2.6.1 The ACG algorithm

In our evaluation, we studied variants of the approximate call graph (ACG) algorithm of Feldthaus et al. [26]. The ACG algorithm was designed to entirely skip analysis

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

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 [103], 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 [38] (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. [26] 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 [27]. 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 [66]. 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`.

2.6.2 Root Cause Labeling

We implemented root cause labeling for missing flows based on the gaps we observed in the WALA implementation [27] of the ACG algorithm [26]. 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 2.7.2.

For MissingFGNode (see Section 2.4.2), 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 2.5). 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 2.4.1, 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 [66]. 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

Table 2.1: Benchmark Statistics.

	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%

identifying when an unmatched variable corresponds to a `with` object property.

As Section 2.7.2 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 2.7.3.

2.6.3 Benchmarks and Harness

For benchmarks, our study used several programs from the TodoMVC suite [1]. 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 [85], 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.

Table 2.1 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 [100]. Missing flow detection (Section 2.4) 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

[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.

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.

2.7 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 2.7.1. Then, we discuss the top root cause labels for missed call graph edges in Section 2.7.2 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 2.7.3.

2.7.1 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 work [17].

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 [26]. 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 [106], 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 [34].

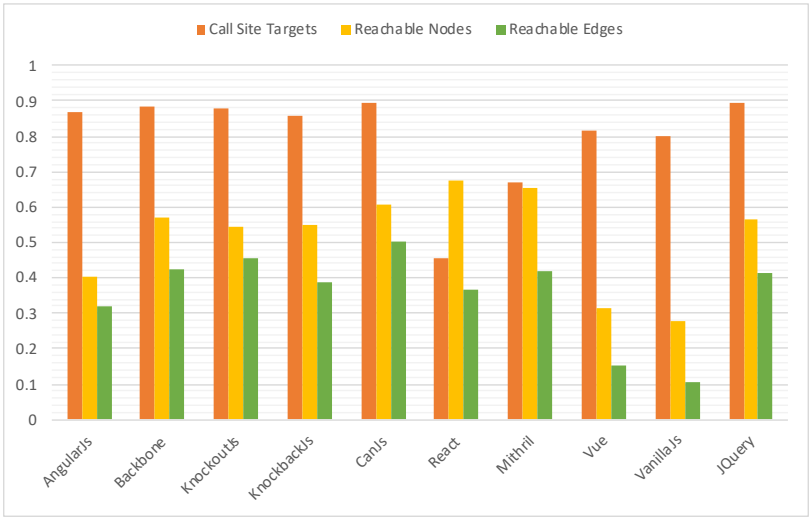
Given our collected data, we studied the following research questions:

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

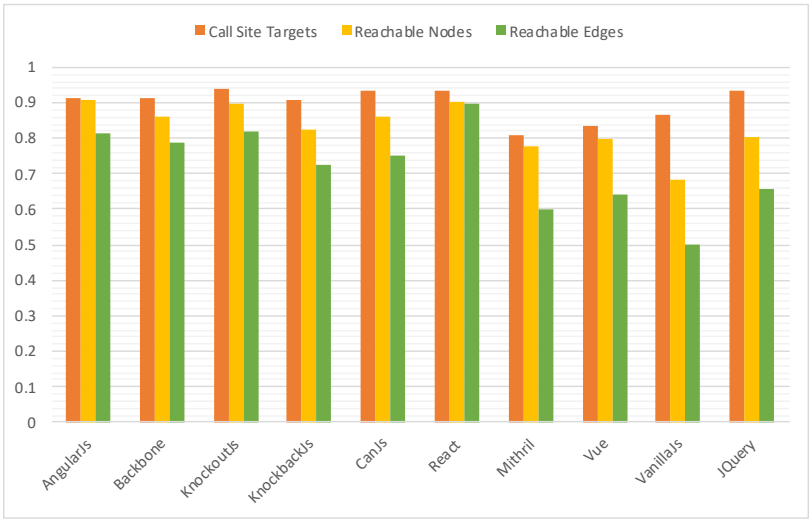
Results Figure 2.4 gives detailed recall results for WALA's original ACG implementation for each TodoMVC benchmark, with results for the pessimistic variant in Figure 2.4a and for optimistic in Figure 2.4b. Average recall across the TodoMVC benchmarks is shown in Figure 2.5.

For RQ1, the data show that recall of ACG tends to suffer with more exacting metrics. The ACG paper [26] used the call site targets metric, and showed that both precision and recall were typically above 80% for their benchmarks. Figure 2.5 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 2.4 shows that recall can vary widely across benchmarks. In Section 2.7.2 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 analysis comes at a cost of very low precision (less than 5% for reachable edges; see the extended version of the work [17] 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 [7, 52, 120]; 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.



(a) Pessimistic ACG.



(b) Optimistic ACG.

Figure 2.4: Detailed recall results for our three metrics across the benchmarks.

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 2.6.3), we do not quantify the precision and recall of pessimistic ACG for the benchmark, nor do we include it in aggregate statistics.

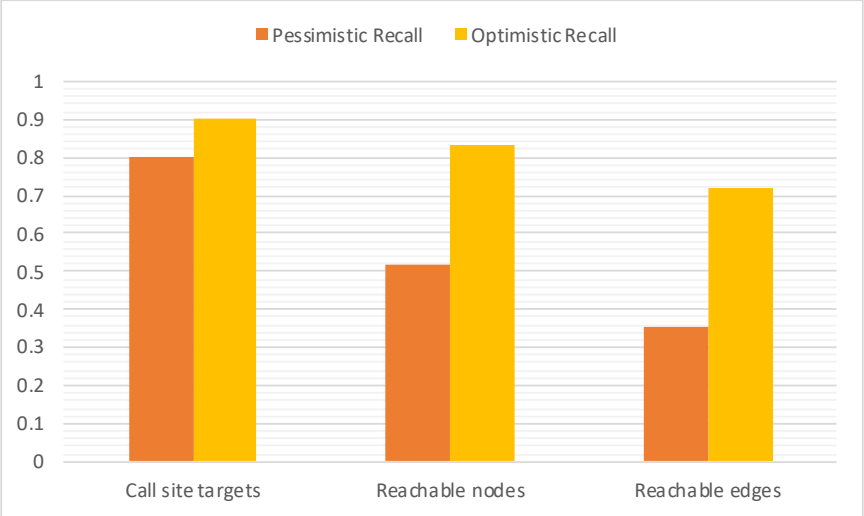


Figure 2.5: Average recall across benchmarks for original WALA ACG implementation.

2.7.2 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 of all results; all experimental data is available in our artifact [16]. 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 2.7.1. The color legend for the pie charts appears below Figure 2.8.

Using data to improve recall Figure 2.6 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

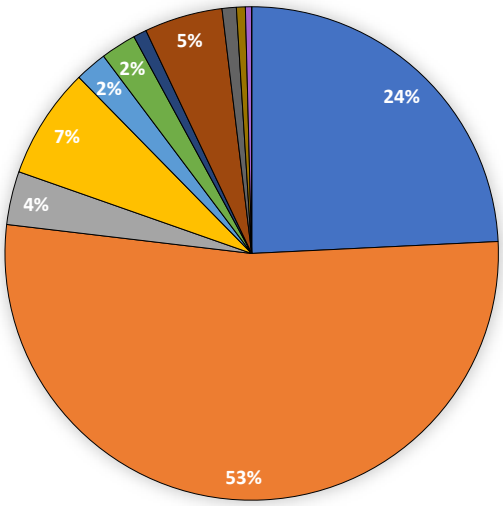
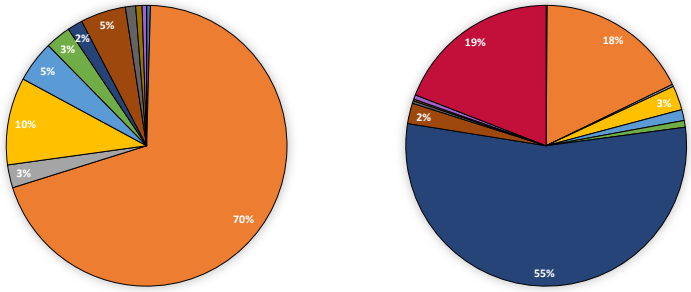
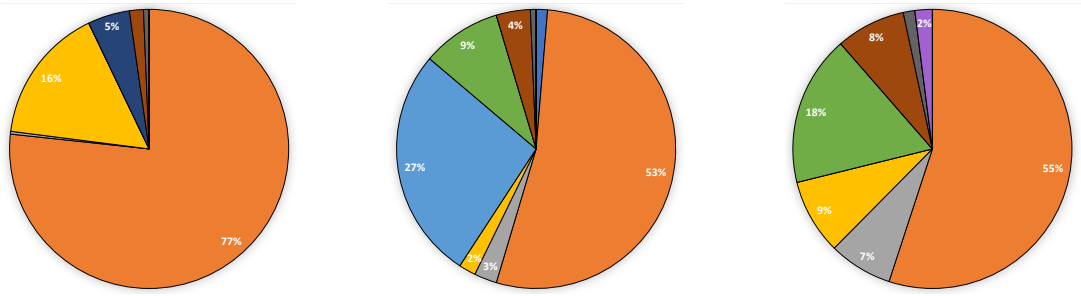


Figure 2.6: Original root causes for optimistic ACG across TodoMVC, before WALA improvements.



(a) Optimistic (b) Pessimistic

Figure 2.7: Improved root causes for ACG variants across TodoMVC, after WALA improvements.



(a) React (b) AngularJS (c) Vue

Figure 2.8: Root causes for three TodoMVC benchmarks for optimistic ACG.

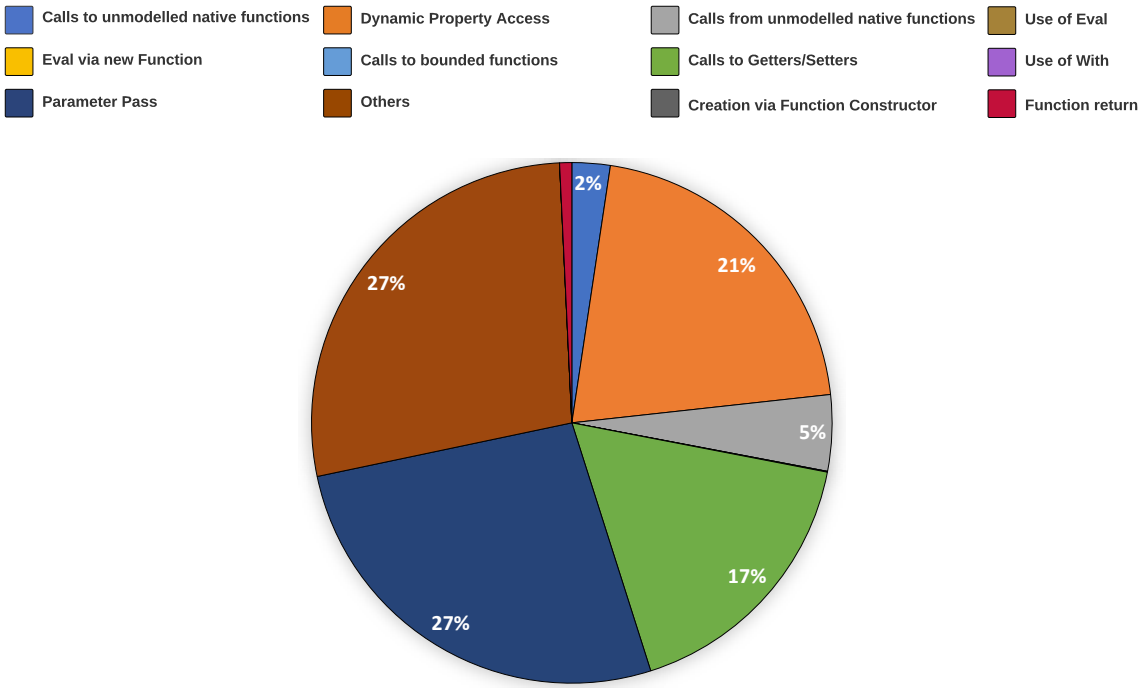


Figure 2.9: Root causes for pessimistic ACG for Juice Shop.

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 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 2.7a and 2.7b 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 2.5, 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 2.7a, 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 2.7.3. 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.

Variance across benchmarks For RQ4, we use illustrative examples to show the variance in root cause prevalence across benchmarks. Figures 2.8a–2.8c 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 2.9 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.

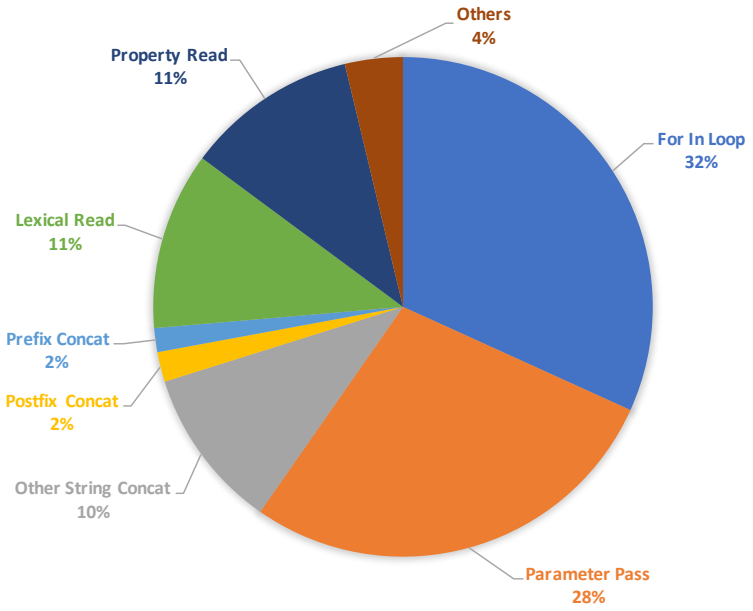


Figure 2.10: Finer-grained dynamic property access root causes for TodoMVC benchmarks.

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 2.7b. 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.

2.7.3 Name Flow for Dynamic Property Accesses

Given the importance of dynamic property accesses as a root cause in Section 2.7.2, 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. [79] 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 [27] 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 2.10; our artifact has the complete data [16]. As shown in Figure 2.10, 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., [6, 87]). 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. [79]. 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.

2.7.4 Threats to Validity

As noted in Section 2.6, 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 2.6.3), 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 2.5, our tooling still does not handle certain language features completely, which may have impacted our measurements.

2.8 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. [106]. 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 [4] 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 [7]. 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 [123] 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. [123] 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 [55] 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 [7, 120] used dynamic analysis to generate traces and find root causes of imprecision in JavaScript static analyses, and Wei et al. [120] also provides suggestions to fix the root causes of imprecision. Lee et al. [52] 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 [21] to handle the interdependent problem of scalability and precision in JavaScript [44, 46, 53]. These frameworks have been steadily enhanced with techniques to improve precision and scalability when analyzing libraries, particularly TAJS [6, 43, 44, 77] and SAFE [48, 52, 53, 86, 87, 97]. While these techniques have shown enormous improvement in analyzing libraries like jQuery [94] and Lodash [23], they do not yet scale to complex MVC frameworks like React [92].

Other techniques use dynamic information to improve static analysis. Wei and Ryder introduced blended analysis [117], which uses dynamic analysis to aid static anal-

ysis in handling JavaScript's dynamic features. The dynamic flow analysis by Naus and Thiemann [75] 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 [82] 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 [59]. 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 [59] focuses on Windows libraries. For Node.js, Madsen et al. [60] 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 [76]. 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 [31] includes an under-approximate call graph builder for JavaScript [32]. CodeQL's analysis is primarily intra-procedural, targeted toward taint analysis, and does not handle dynamic property accesses.[10] Møller et al. [71] 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. [79] present a pragmatic modular call-graph construction technique for Node.js programs; we discussed its specialized handling of property name expressions in Section 2.7.3. 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 [98]; it

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

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 [122] and type inference [18, 37].

2.9 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.

Chapter 3

Indirection-Bounded Call Graph Analysis

In this chapter, we propose a simple but effective technique that addresses performance issues encountered in call graph generation. We observe via a dynamic analysis that typical JavaScript program code exhibits small levels of indirection of object pointers and higher-order functions. We demonstrate that a widely used analysis algorithm, wave propagation, closely follows the levels of indirections, so that call edges discovered early are more likely to be true positives. By bounding the number of indirections covered by this analysis, in many cases it can find most true-positive call edges in less time. We also show that indirection-bounded analysis can similarly be incorporated into the field-based call graph analysis algorithm ACG.

We have experimentally evaluated the modified wave propagation algorithm on 25 large Node.js-based JavaScript programs. Indirection-bounded analysis on average yields close to a 2X speed-up with only 5% reduction in recall and almost identical precision relative to the baseline analysis, using dynamically generated call graphs for the recall and precision measurements. To demonstrate the robustness of the approach, we also evaluated the modified ACG algorithm on 10 web-based and 4 mobile-based medium sized benchmarks, with similar results.

3.1 Introduction

The construction of accurate call graphs is crucial for various static analysis tasks. Call graphs provide a comprehensive representation of the calling relationships between functions, enabling analysis techniques such as vulnerability and bug detection, program comprehension, and refactorings [8, 26, 71, 79, 110]. Static call graph analyzers aim to over-approximate, meaning that they may include false positives, i.e., unexecutable call edges. Analysis time tends to correlate with the number of call edges produced, so improving precision can also improve analysis time. Although soundness is desirable, all existing practical whole-program analyses by design sacrifice some amount of soundness to achieve useful precision and scalability [57]. This perhaps makes them unsuitable for code optimization purposes, but other use cases can tolerate false negatives (e.g., many bug finding and vulnerability detection tools aim to expose issues but not to prove their absence). Nevertheless, achieving high accuracy and low analysis time for large, real-world programs is challenging due to the inherent complexity of call graph generation. Analysis time is often prohibitively large, so it is important to explore new approaches that can substantially reduce analysis time even if the price is slightly more false negatives.

As JavaScript has both objects and functions as first-class values, and it has no static type system, constructing call graphs for JavaScript programs generally requires reasoning about the possible flow of objects and functions through the program being analyzed. Functions frequently appear both as arguments and return values of other functions and as values of object fields. State-of-the-art call graph analyzers for JavaScript are based on subset-based flow-insensitive analysis techniques [5]. Objects are typically modeled using allocation-site abstraction [19, 79], or more coarsely using field-based analysis [26, 27]. Functions are tracked using variations of control-flow analysis [69]. In general, the analyses can be expressed using conditional subset constraint systems, which are solved using cubic-time algorithms [89, 102].

Such algorithms build a call graph for a given program iteratively until a fixpoint is reached. Iteration is necessary because of indirections that may occur. For example, if a higher-order function `f` contains a call `g(...)` to a function that is provided via a parameter `g` of `f`, then the call edges for that call site cannot be resolved until the analysis

has inferred the calls to `f`. Similarly, the possible values at an object field read operation `x.a` generally cannot be obtained until the analysis has inferred which objects `x` may reference. Different analysis algorithms solve the analysis constraints in different orders, but there is one algorithm, the wave propagation algorithm by Pereira and Berlin [89], that directly follows the levels of indirections, as we explain in detail in Section 3.4. That algorithm was designed for points-to analysis but is also well suited for call graph analysis.

Interestingly, in real-world code, we observe that function values typically do not flow through many levels of indirection, which means that call graph analysis only needs few iterations to infer most of the possible call edges. In Section 3.2 we show that creating call graphs for higher-order functions (if not involving objects) requires only as many iterations as the maximum order of the functions, and a similar property holds when objects are involved. Therefore, when wave propagation-style analysis has reached a certain number of iterations, all call edges that are discovered after that point must be false positives that arise only due to analysis imprecision (assuming an idealized sound analysis). Thus, simply by terminating the wave propagation algorithm after a fixed number of iterations, we can reduce analysis time while only missing the relatively few call edges that involve high levels of indirection. This also works when objects are involved; however, due to the asymmetric nature of field read and field write operations (see Section 3.2) it is beneficial to leave analysis constraints for field read operations and method calls unbounded. This is the first work to utilize observations about low levels of indirection in data flows to achieve a more scalable static analysis.

Another program analysis technique that has proven effective for JavaScript programs is the approximate call graphs (ACG) algorithm of Feldthaus et al. [26]. This algorithm applies field-based analysis, meaning that objects are modeled more abstractly, which generally leads to faster but also less accurate analysis compared to techniques that use allocation-site abstraction of objects. We demonstrate that the ACG algorithm can also easily be adapted to indirection-bounded analysis.

The proposed approach is inspired from Mathiasen and Pavlogiannis [61] on the complexity of Andersen's pointer analysis. One of their key results is that a version of pointer analysis where the number of store operations (corresponding to field write opera-

tions in our setting) in witnesses of points-to relations is bounded can be solved in sub-cubic time. They conjecture that the level of indirection of store operations is typically small in practice, but they have left the practical realizations and an experimental evaluation for future work, which we explore here in the context of call graph analysis for JavaScript.

Compared to ad hoc approaches to reduce analysis time, for example, stopping analysis after a time-out, the proposed indirection-bounded approach gives more predictable and interpretable outcomes because of the connection to the program semantics. That is, the results of indirection bounding under a particular bound are deterministic, and one can give a precise semantic characterization of the types of data flows that will be missed due to the bound. Also, instead of tuning time-outs for individual programs, indirection-bound analysis gives good results with a fixed bound applied uniformly for all programs.

In summary, the contributions of this chapter are:

- We propose the use of *indirection-bounded* analysis (Section 3.4) for achieving faster call graph analysis while with little sacrifice in recall, which can be a useful compromise when analyzing large, real-world programs.

- We demonstrate via a dynamic analysis (Section 3.3) that typical JavaScript programs tend to exhibit small levels of indirections of object pointers and higher-order functions, thereby semantically justifying the use of indirection-bounded analysis.

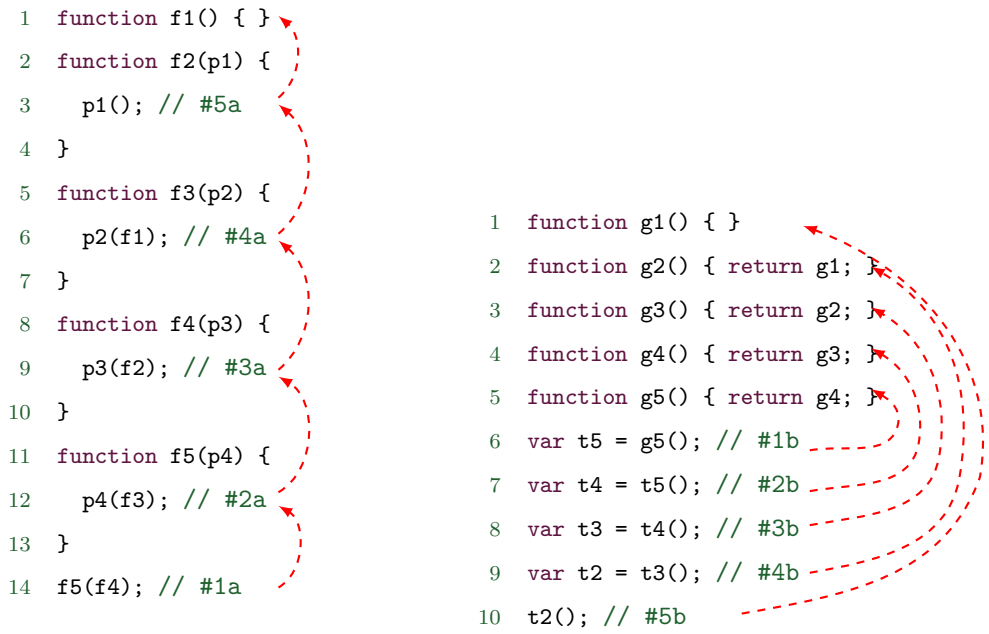
- By incorporating indirection-bounded analysis into an existing state-of-the-art call graph analyzer for JavaScript that uses the wave propagation algorithm, we present experimental results (Section 3.5) on 25 large open source programs, showing that the approach on average (geometric mean) results in a 2X speed-up of the analysis with only 5% reduction in recall (and nearly identical precision) relative to the baseline analysis when using dynamically generated call graphs for measuring recall and precision. For many use cases, this can be a valuable trade-off between analysis time and recall. Applying the technique on 10 web-based and 4 mobile-based medium-sized benchmarks using the modified ACG algorithm similarly resulted in an approximately 2X speed-up in analysis time, with only a 1% reduction in recall (again with nearly identical precision).

3.2 Motivating Examples

In a call graph, call sites and functions are represented as nodes, and a directed edge from node a to node b indicates that call site a may invoke function b at runtime. Sometimes, individual call sites are abstracted by their enclosing function, so that the call edges are from functions to functions.

The accuracy of a call graph construction technique can be measured by its precision and recall relative to the (noncomputable) semantically correct call graph for a given program, or using a call graph produced via one or more dynamic executions of the program as an approximation. (A sound analysis will have perfect recall, but as noted in Section 3.1, practical analyses are not fully sound.) Different use cases have motivated different metrics in the literature [17]. With the *call site targets* metric [26], precision for a specific call site is computed as the percentage of true (i.e., semantically possible) call targets among those predicted by the static analysis, and recall is the percentage of predicted targets among the true targets. The precision and recall for an entire program are then computed as the averages over all call sites that are semantically reachable. A variant is the *call edge sets* metric [114], which computes precision and recall by comparing the sets of call edges produced by the static analysis and the dynamic analysis. The *reachable functions* metric [106] and the *reachable edges* metric [34] instead compare the sets of functions or call edges, respectively, that are reachable from the program entry points (e.g., application modules).

Figures 3.1 and 3.2 contain five small example JavaScript programs that illustrate the indirections that can arise when computing call graphs. The red edges show the call edges, pointing from call sites to functions. In Figure 3.1a, function `f5` is a higher-order function, which is called at line 14 with `f4` as argument. The function `f4` is itself a higher-order function that is then called at line 12 with `f3` as argument, etc., until finally line 3 calls `f1`. In other words, `f5` is a 5'th-order function, `f4` is a 4'th-order function, etc. This means that the call edge from the call site marked `#1a` must be discovered before the call edge for `#2a`, etc., until after a total of 5 indirections have been resolved, the call edge for `#5a` can be found. Figure 3.1b shows a similar example of a 5'th-order function where also 5 levels of indirections arise, but this time due to return values rather than arguments. As we explain



(a) Function arguments

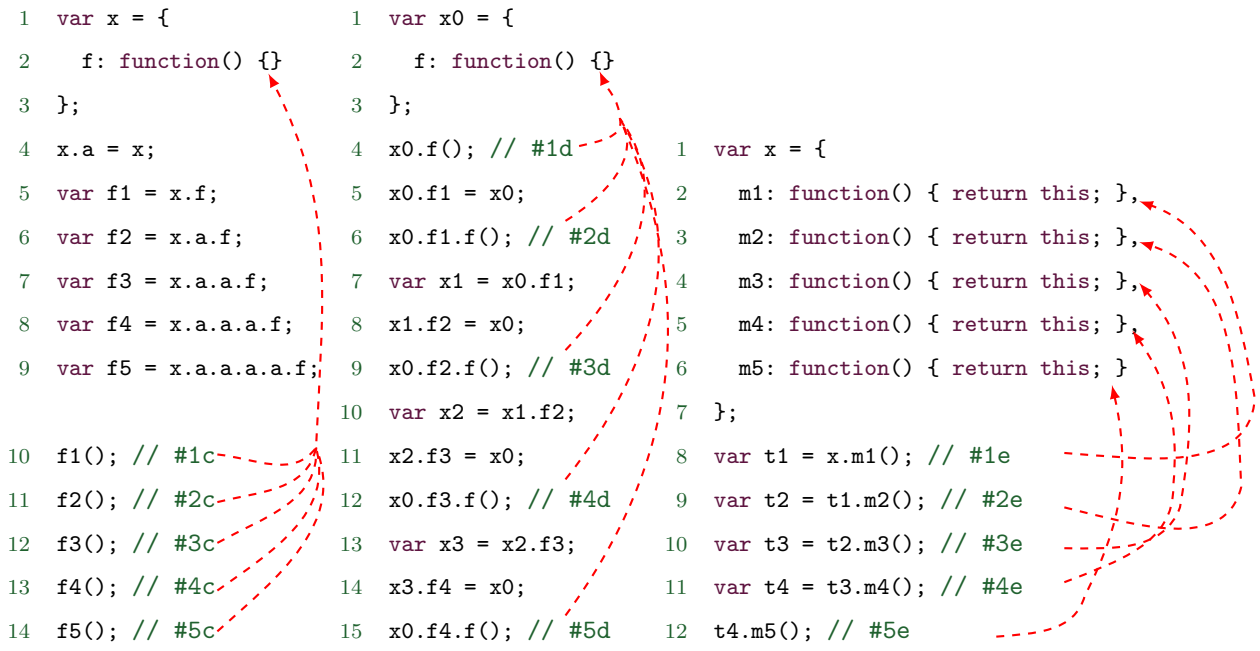
(b) Function return values

Figure 3.1: Example programs that illustrate indirection levels for function calls.

in Section 3.4, analysis algorithms like wave propagation [89], ACG [26], or 0-CFA [101] can compute the call graphs for these programs in 5 iterations. 5'th-order functions are not common in real-world code, which we can exploit to terminate analysis early and save time without risking too many missed call edges (i.e., false negatives).

A similar situation arises when objects and field access operations are involved. In Figure 3.2a, lines 1–4 set up a simple object structure. The variables `f1`–`f5` all refer to the same function in this case, but via different levels of indirections. Specifically, discovering the call edge for `f5` at call site `#5c` requires 5 levels of indirection because of the chain of field reads at line 9. Such long chains of field reads operations (sometimes split into smaller parts with variables holding intermediate results) are not uncommon in real-world code, which is why we give special treatment to these operations in the following sections.

On the other hand, it is perhaps less obvious how field write operations can lead to high levels of indirections. Figure 3.2b shows an example where call site `#5d` has 5 levels



(a) Field reads (b) Field writes (c) Method calls

Figure 3.2: Example programs that illustrate indirection levels for object field accesses.

of indirection. That call site cannot be resolved until the analysis has discovered that `x0.f4` is an alias of `x0`. This in turn requires discovering that the field write on the previous line updates `x0.f4`, since `x0` and `x3` are aliases, and so on through each of the aliasing relations established by the preceding lines.[1]

Finally, Figure 3.2c shows an example with chains of method calls, combining objects and functions. Since each method call consists of a pair of a field read and a function call, this example involves a total of 10 levels of indirection to resolve call site `#5e`. As with chains of field reads, this pattern is also common in real code (e.g., with fluent interfaces [28]), which suggests that method calls should be treated in the same way as field reads in indirection-bounded analysis.

[1]One might expect that nested object initialization would yield a high level of indirections via field writes, e.g.: `x = { f: ... }; y = {}; z = {}; y.b = x; z.a = y; z.a.b.f();` But, this code has only *one* level of indirection due to field writes, as objects are copied to the base variables for all field writes without indirection. The indirection level due to field reads is 3, due to the call.

3.3 Dynamic Indirection Bound Estimation

To confirm the intuitions from Section 3.2 regarding the depth of indirections encountered in real code, we designed a dynamic analysis to estimate the *minimum indirection bound* required for a static analysis to discover each function call observed during execution. With this dynamic analysis, we can observe the true bound under which function values flow to their invocations in an execution, independent of any static analysis limitation or approximation. If these minimum indirection bounds are observed to typically be low, that provides good evidence that using low indirection bounds in a static analysis will preserve most analysis recall. Here we present the design of the dynamic analysis and give results from a study across a large set of benchmarks.

3.3.1 Language

The first column of Section 3.3.1 defines the types of canonical statements in a core language. (The constraint rules in the second column will be explained in Section 3.4.1.) The statement types are standard for a flow- and context-insensitive Andersen-style points-to analysis [5] for a JavaScript-like language. A program is a set of functions, each of which contains statements of the types shown in the table (more complex assignments and expressions can be normalized to these forms via temporary variables). We elide details of standard language constructs like conditionals, loops, etc., as they are not relevant given our focus on flow-insensitive static analysis. We assume for simplicity that local variable names are unique across functions.

The values in the language are either object values, written `{}` (like a JavaScript object literal) or (first-class) function values, written `p => {...}` (using JavaScript arrow syntax). Without loss of generality we assume every function has one parameter and returns some value. The first two statement types respectively allocate a new object or new function value and assign it to a variable; each such statement has a unique label i. An `x = y` statement copies between variables. For object fields, `x = y.f` loads field `f` and `x.f = y` stores to field `f`. We assume a JavaScript-like semantics where writing to a non-existent object field `f` creates `f` on the object (obviating the need for field declarations). Finally,

Table 3.1: Statement types for our analysis and the corresponding static analysis constraint rules (discussed in Section 3.4).

Statement Type	Constraint Rule
`x = {}`$_i$	$\{o_i\} \subseteq pt(\texttt{x})$
`x = p =>`$_i$ $\{\dots\}$	$\{f_i\} \subseteq pt(\texttt{x})$
`x = y`	$pt(y) \subseteq pt(x)$
`x = y.f`	$\dfrac{o_i \in pt(\texttt{y})}{pt(o_i.\texttt{f}) \subseteq pt(\texttt{x})}$
`x.f = y`	$\dfrac{o_i \in pt(\texttt{x})}{pt(\texttt{y}) \subseteq pt(o_i.\texttt{f})}$
`x = y(z)`	$\dfrac{f_i \in pt(y)}{pt(\texttt{z}) \subseteq pt(p_i) \qquad pt(ret_i) \subseteq pt(\texttt{x})}$
`return`$_i$ `x`	$pt(\texttt{x}) \subseteq pt(ret_i)$

we have `x = y(z)` statements for calling functions, and `return x` statements for returning values. The label i on each return statement identifies the containing function.

3.3.2 Dynamic Analysis

Here we present our dynamic analysis to estimate minimum indirection bounds. The analysis does not provide an exact value for these bounds; it may under-estimate due to lack of input coverage or unhandled language features, and it may over-estimate due to an imperfect simulation of the static analysis. Still, we have found its results to be accurate in practice (via manual inspection) and useful for understanding indirection levels in real programs.

Algorithm 3 Dynamic bounds estimation.

```
1: SELECTIVE: boolean
2: V: map from variable and value to indirection level bound
3: F: map from object field and value to indirection level bound
4: procedure HANDLESTMT(s)
5:     match s
6:         case x = {}_i  or  x = p =>_i {...}:
7:             V[x, i] ← 0
8:         case x = y:
9:             v ← α(y)
10:            V[x, v] ← min(V[x, v], V[y, v])
11:        case return_i x:
12:            v ← α(x)
13:            V[ret_i, v] ← min(V[ret_i, v], V[x, v])
14:        case before x = y(z):
15:            f_i ← α(y), v ← α(z)
16:            t ← max(V[y, f_i] + 1, V[z, v])
17:            V[p_i, v] ← min(V[p_i, v], t)
18:        case after x = y(z):
19:            f_i ← α(y), v ← α(ret_i)
20:            t ← max(V[y, f_i] + 1, V[ret_i, v])
21:            V[x, v] ← min(V[x, v], t)
22:        case x = y.f:
23:            b ← α(y), v ← α(y.f)
24:            if SELECTIVE then
25:                t ← max(V[y, b], F[b.f, v])
26:            else
27:                t ← max(V[y, b] + 1, F[b.f, v])
28:            V[x, v] ← min(V[x, v], t)
29:        case x.f = y:
30:            b ← α(x), v ← α(y)
31:            t ← max(V[x, b] + 1, V[y, v])
32:            F[b.f, v] ← min(F[b.f, v], t)
33:    end match
34: end procedure
```

Algorithm 3 provides our dynamic analysis. We assume the analysis is implemented via an interface similar to that provided by frameworks like Jalangi [100], where a callback provided by the analysis is invoked before and possibly after the execution of each program statement. In Algorithm 3, the callback is the HandleStmt procedure. For all statement types, we assume HandleStmt is invoked before the statement s executes, except for calls `x = y(z)`, where we require the callback both before and after (to respectively handle parameter passing and returns).

The dynamic analysis relies on a function α that given an expression e, first evaluates e to a value v and then returns the allocation site for v (i.e., the label of the statement that allocated v). The dynamic analysis tracks bounds for allocation sites instead of individual dynamic values to match the finite abstraction of values typically used by static call graph builders. For readability, in the remainder of this section we refer to values and their allocation sites interchangeably.

Algorithm 3 computes two maps, V and F. The V map records for each variable `x` and value v the minimum observed indirection bound under which v flowed to `x` in the execution. Retaining only the minimum bound for each variable `x` and value v makes sense for our use case, as a sound static analysis models all possible data flows, and hence would discover the flow of v to `x` under that minimum bound. F is similar but is keyed on object fields b.`f`, where b is a value and `f` is a field name. After the analysis completes, the minimum observed indirection bound for discovering that call `x = y(z)` invokes function f is simply $V[\texttt{y}, f]$.

We now describe the handling of each type of statement in turn. For a creation of an object or function at allocation site i (line 6), $V[\texttt{x}, i]$ is set to 0, as the flow does not involve any indirections. For an assignment `x = y` (line 8), where v is the value of `y`, $V[\texttt{x}, v]$ is set to be the minimum of its current value and $V[\texttt{y}, v]$ (since we aim to find minimum observed indirection bounds). Return statements (line 11) are handled just like assignments, updating the bound for the synthetic ret_i variable for the enclosing function.

The next case (line 14), handling parameter passing, is the first involving an indirection, here via a call. Here, f_i is the function value being invoked, and v is the value of the parameter. Recall from the discussion of Figure 3.1a in Section 3.2 that to discover data

flow into a formal parameter from a call site, the analysis must first discover the data flow of the invoked function to the call; the parameter flow occurs at an increased indirection level. Hence, to discover the parameter data flow from this call, the indirection bound must be at least $V[\mathtt{y}, f_i] + 1$. Note that finding the flow also requires discovering that v flows to actual parameter `z`, so the true bound for this flow is the maximum of these two flows (line 16). Finally, line 17 updates the bound for formal parameter p_i to be the minimum observed thus far. Handling of the return value after a call completes (line 18) is analogous to handling of parameters.

Handling of field reads (line 22) and field writes (line 29) is also similar to that for parameter passing. Here, as discussed in Section 3.2, the increase in indirection level occurs because the static analysis must first observe the data flow of the relevant object into the base variable of the dereference. For both reads and writes, the base object is named b in the pseudocode, and we add 1 to the bound for the flow of b to the statement in each case (line 27 for reads, line 31 for writes). Recall from Figures 3.2a and 3.2b that writing code with a high indirection level for field reads is more natural than doing the same for field writes. Accordingly, the pseudocode has a flag SELECTIVE to control whether reads should be treated as bounded when computing estimates. If SELECTIVE is true, reads are not treated as bounded, and 1 is *not* added to the bound for the flow of b to the base variable `y` (see line 25). In our implementation, SELECTIVE also controls bounding of method calls; this is not shown in Algorithm 3 since our core language (Section 3.3.1) contains only function calls (with no receiver argument), not method calls.

Algorithm 3 may over-estimate the bounds required of a static analysis since it does not propagate information from later assignments to previous calls. Consider this JavaScript example:

```
1  x = function f1() { ... };
2  y = /* some flow with minimum bound 2 yielding f1 */
3  z = y;
4  z();
5  y = x;
```

After Algorithm 3 completes, $V[\mathtt{z}, f_1]$ will be 2, since f_1 was initially copied to `y` via a flow of bound 2 (line 2) and then copied to `z`. However, due to line 5, there exists a flow of f_1 to

y with bound 0. The algorithm updates $V[\mathtt{y}, f_1]$ accordingly, but does not propagate this update to $V[\mathtt{z}, f_1]$. This issue could be addressed by tracing the execution operations and computing a fixed point over that trace; we used the single-pass approach as we did not observe this over-estimation to occur in practice.

3.3.3 Study Results

Here we present results of applying our dynamic analysis to a suite of Node.js benchmarks to measure minimum indirection bounds in practice.

Implementation We implemented the analysis atop the NodeProf framework for the dynamic analysis of Node.js [108]. For scalability, we separated the analysis into a trace generation phase that runs during program execution, followed by a post-processing phase to compute the bounds. For trace generation, NodeProf does not invoke a single callback for assignments, but instead invokes separate callbacks for reads and writes of both variables and object fields. To adapt Algorithm 3 to this structure, we maintain an additional map S from each value v to the bound t for v corresponding to the location (variable or field) from which it was most recently read. Then, t is used when updating the bound at the next variable or field write. So, for a statement `x = y`, the analysis first sees a read of v from `y`, and it sets $S[v]$ to $V[\mathtt{y}, v]$. Then, when handling the subsequent write of v to `x`, it uses $S[v]$ instead of $V[\mathtt{y}, v]$ for the bound update (line 10 in Algorithm 3).

Benchmarks and methodology We created a suite of 74 Node.js benchmarks for our study due to the lack of a standard benchmark suite. These were randomly selected from the top 1,000 highest-ranked JavaScript projects on GitHub that both had unit tests available and worked with our dynamic analysis infrastructure and implementation. We exercised each benchmark by running its unit test suite. In total, these runs executed calls at 60,601 distinct call sites.

Results Table 3.2 gives the results from our study. We present results for two configurations. The first is our preferred configuration, with SELECTIVE enabled, so reads

Table 3.2: Number of call edges with each minimum bound across all benchmarks, for configurations with SELECTIVE enabled and disabled, respectively.

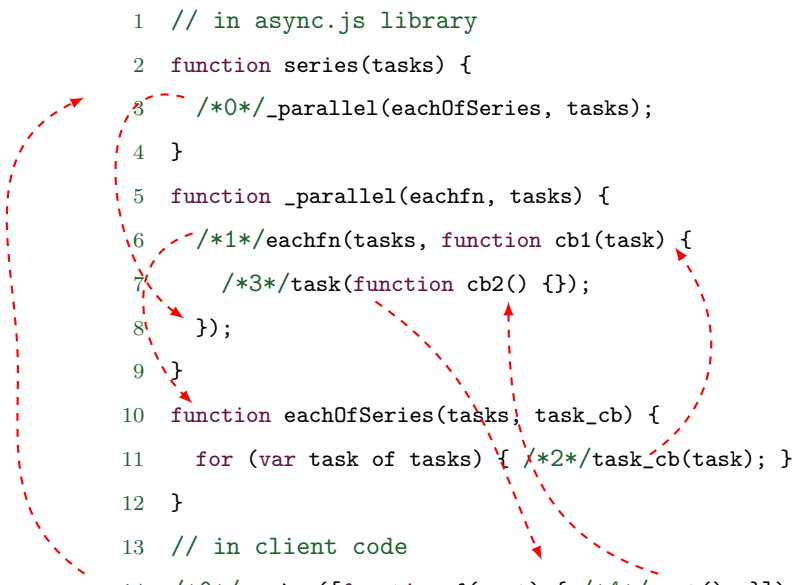
Configuration	0	1	2	3	4	5	6	7	8	9	10-19	20+
SELECTIVE enabled	40,087	16,560	5,958	2,207	356	29	16	-	-	-	-	-
SELECTIVE disabled	37,589	10,064	5,825	3,610	2,214	1,705	992	442	644	676	1,311	141

and method calls are treated as unbounded. The second row gives results when all indirections are bounded. Each column shows how many dynamic call graph edges (from call site to callee function) could be discovered within that bound. The numbers are aggregated across all the benchmarks, for a total of 65,213 call edges (greater than the number of distinct call sites, since some sites invoke different functions on different execution paths).

The results show that in both configurations, most call graph edges can be found within a small bound; more than 57% of edges are discoverable within a bound of 0, i.e., the function data flow involves no indirections. Note, however, that with SELECTIVE enabled, significantly more edges are discoverable within bound 1 (6,496 more than with SELECTIVE disabled), and the long tail of call edges with minimum bound 7 or higher is eliminated. In fact, with SELECTIVE disabled, we discovered calls with a bound as high as 75. This result confirms the intuition from Section 3.2 that long chains of field reads and method calls can occur regularly in real-world programs, justifying special handling.

Overall, the data from our study provide promising evidence that an indirection-bounded static analysis could discover most true call graph edges within a small bound. For the SELECTIVE configuration, roughly 96% (62,605 / 65,213) of calls are reached within bound 2. These insights guided our static analysis design, described in Section 3.4.

Examples For the configuration with SELECTIVE enabled, below is an example of a call that involves 4 levels of indirections, from the `express-react-views` benchmark (heavily simplified for readability). Calls that require even higher indirection bounds are rare, as depicted in Table 3.2, and are challenging to extract due to their complexity.



```
1  // in async.js library
2  function series(tasks) {
3    /*0*/_parallel(eachOfSeries, tasks);
4  }
5  function _parallel(eachfn, tasks) {
6    /*1*/eachfn(tasks, function cb1(task) {
7      /*3*/task(function cb2() {});
8    });
9  }
10 function eachOfSeries(tasks, task_cb) {
11   for (var task of tasks) { /*2*/task_cb(task); }
12 }
13 // in client code
14 /*0*/series([function f(next) { /*4*/next(); }]);
```

The `async.js` library provides a function `series` for running all task callbacks in a provided array. Internally, this functionality is implemented using layers of higher-order functions, leading to the high bound. The calls above are commented `/*0*/` through `/*4*/` to show their bounds. The layered implementation inside `async.js` does enable code reuse within the library, but it leads to convoluted and hard-to-understand control flow, as shown above. As the data in Table 3.2 show, calls like these requiring bound 4 or greater are quite rare across our benchmarks (only 0.6% of call edges).

In contrast, disabling SELECTIVE can cause natural code patterns to yield high bounds, as seen in Section 3.2. For instance, consider this code from the `express` benchmark:

```
1  block.paragraph = edit(block._paragraph)
2    .replace('hr', block.hr) /*2*/
3    .replace('heading', ' {0,3}#{1,6} +') /*4*/
4    .replace('|lheading', '') /*6*/
5    .replace('blockquote', ' {0,3}>') /*8*/
6    .replace('fences', ' {0,3}(?:`{3,}|~{3,})[^`\\n]*\\n') /*10*/
7    .replace('list', ' {0,3}(?:[*+-]|1[.)]) ') /*12*/
8    .replace('html', '</?(?:tag)(?: +|\\n|/?>)|<(?:script|pre|style|!--)') /*14*/
9    .replace('tag', block._tag) /*16*/
10   .getRegex(); /*18*/
```

This code uses a fluent interface [28]: the `edit` and `replace` methods both return `this`, allowing for chaining of method calls. Each step in the chain involves a field read (to access the method) followed by a call, thereby adding 2 to the minimum bound for this configuration. So, the final call to `getRegex` has 18 levels of indirection. We studied calls with higher bounds and found that they involved a complex mix of field reads and method calls, often spread across multiple functions and files.

3.4 Indirection-Bounded Call Graph Construction

In this section, we present static call graph construction algorithms that allow for indirection bounding. We first describe a constraint-based formulation of the wave propagation algorithm [89] (Section 3.4.1), and then present a simplified version for solving the constraints (Section 3.4.2). In Section 3.4.3 we extend the algorithm with indirection bounding. Finally, in Section 3.4.4 we show how further simplifications yield a bounded version of the ACG algorithm of Feldthaus et al. [26]. The static analyses presented in this section work independently of the dynamic analysis presented in Section 3.3. The purpose of the dynamic analysis was to provide evidence and insights that support the static analysis design by showing how indirection works in different scenarios. This semantic justification helps validate the conclusions drawn from static analysis and ensures the bounding approach is reliable.

3.4.1 Analysis Formulation

The second column of Section 3.3.1 gives constraint rules for computing an Andersen-style points-to analysis for our statement types. The rules are standard; see [102] for a more detailed description. The constraints define what values (objects or functions) must be present in the points-to set of each variable and object field. Given a solution to the constraints, a call graph can be extracted by adding an edge from each call site `x = y(z)` (or from the function containing the call site) to each function $f_i \in pt(\mathtt{y})$.

Concrete values are abstracted using allocation sites; we write o_i or f_i for an object or function, respectively, allocated at site i. As in the dynamic analysis formulation

(Section 3.3.2), we assume a formal parameter variable p_i and a return variable ret_i for each function f_i. The constraints for field read, field write, and call statements are *conditional* constraints [2] – they each impose new subset constraints based on the contents of another points-to set. For example, the constraint for `x = y(z)` checks if f_i is present in $pt(\texttt{y})$, which indicates that `y(z)` may invoke f_i. In this case, new subset constraints are imposed to capture the data flow from actual parameter `z` to formal p_i and from the returned value ret_i to `x`. These conditional constraints correspond directly to the notion of indirections discussed in Sections 3.2 and 3.3. Our approach implements indirection bounds by bounding the handling of these conditional constraints, leveraging the structure of the wave propagation algorithm, to be described next.

3.4.2 Simplified Wave Propagation

Algorithm 4 presents pseudocode for a simplified version of the wave propagation algorithm [89], the basis of our bounding technique. The pseudocode eschews many optimizations critical to the efficiency of the full wave propagation algorithm, including worklists, cycle elimination, and topological sorting. We simplify the pseudocode to clearly expose the two alternating phases of the algorithm, *propagation* and *edge addition*, the aspect of the algorithm most critical to bounding.

Algorithm 4 computes the points-to relation pt using a flow graph G. Each node in G represents a variable or an object field, and each edge $n \rightarrow n'$ in G represents a subset constraint $pt(n) \subseteq pt(n')$ from Section 3.3.1. The main entry point is the Analyze procedure at line 4.

The algorithm begins with the Init procedure (line 12), which initializes pt and G based on the simple (non-conditional) constraints for value creation, variable copy, and return statements in Section 3.3.1. Then, at lines 6–11, the algorithm alternates between calls to Propagate and AddEdges until pt and G reach a fixed point. Propagate (line 22) uses a fixed-point loop to ensure that for each edge $n \rightarrow n'$ currently in G, $pt(n) \subseteq pt(n')$. Then, AddEdges (line 29) processes each field read, field write, and call statement and updates G with new edges based on the current value of pt and the corresponding conditional constraints in Section 3.3.1.

Algorithm 4 Simplified wave propagation algorithm.

P: program to analyze
pt: points-to relation, initially empty
G: flow graph, initially empty
procedure ANALYZE()
    INIT()
    repeat
        $pt' \leftarrow pt$, $G' \leftarrow G$
        PROPAGATE()
        ADDEDGES()
    until $pt' = pt \wedge G' = G$
end procedure
procedure INIT()
    for each `x = {}`$_i$ $\in P$ **do**
        $pt(\texttt{x}) \leftarrow pt(\texttt{x}) \cup \{o_i\}$
    for each `x = p =>`$_i$ `{...}` $\in P$ **do**
        $pt(\texttt{x}) \leftarrow pt(\texttt{x}) \cup \{f_i\}$
    for each `x = y` $\in P$ **do**
        $G \leftarrow G \cup \{\texttt{y} \rightarrow \texttt{x}\}$
    for each `return`$_i$ `x` $\in P$ **do**
        $G \leftarrow G \cup \{\texttt{x} \rightarrow ret_i\}$
end procedure
procedure PROPAGATE()
    repeat
        $pt' \leftarrow pt$
        for each edge $n \rightarrow n'$ in G **do**
            $pt(n') \leftarrow pt(n') \cup pt(n)$
    until $pt' = pt$
end procedure
procedure ADDEDGES()
    for each `x = y.f` $\in P$, $o_i \in pt(\texttt{y})$ **do**
        $G \leftarrow G \cup \{o_i.f \rightarrow \texttt{x}\}$
    for each `x.f = y` $\in P$, $o_i \in pt(\texttt{x})$ **do**
        $G \leftarrow G \cup \{\texttt{y} \rightarrow o_i.f\}$
    for each `x = y(z)` $\in P$, $f_i \in pt(\texttt{y})$ **do**
        $G \leftarrow G \cup \{\texttt{z} \rightarrow p_i, ret_i \rightarrow \texttt{x}\}$
end procedure

The clean separation between propagation and edge addition is a key characteristic of wave propagation; it leverages this structure to efficiently eliminate cycles in the constraint graph and compute a topological ordering to minimize propagation [89].

Algorithm 5 Algorithm 4 modified to bound indirections except for field reads.

```
1:  bound: bound on indirections
2:  i: current iteration, initially 0
3:  procedure AddEdges()
4:      G' ← G
5:      for each x = y.f ∈ P, o_i ∈ pt(y) do
6:          G ← G ∪ {o_i.f → x}
7:      if G' = G ∧ i < bound then
8:          for each x.f = y ∈ P, o_i ∈ pt(x) do
9:              G ← G ∪ {y → o_i.f}
10:         for each x = y(z) ∈ P, f_i ∈ pt(y) do
11:             G ← G ∪ {z → p_i, ret_i → x}
12:         i ← i + 1
13: end procedure
```

3.4.3 Adding Bounds

Given the structure of the wave propagation algorithm, adding bounding of all indirections is straightforward. Algorithm 4 already separates its handling of conditional constraints, which correspond to indirections, into the AddEdges procedure. So, bounding indirections simply requires limiting the number of times that AddEdges runs to be less than the bound. We have found that in real implementations that use the wave propagation structure, adding bounding is similarly straightforward.

Recall that Section 3.3.3 showed that field reads often require higher bounds than other indirections, matching the intuition of Section 3.2. Algorithm 5 is a variant that only bounds indirections via field writes and calls, while leaving handling of field reads unbounded; the changes compared to Algorithm 2 are emphasized with blue (lines 4, 7 and 12). Variables *bound* and i are introduced, and only the AddEdges procedure of Algorithm 4 is modified. The modified code first adds edges to G based on field reads without checking the bound (lines 5–6). Then, field write and call statements are processed,

but only if the field read processing added no new edges to G (the $G' = G$ check on line 7). If handling of reads adds new edges to G, then AddEdges returns without incrementing i, and another phase of propagation is run (see line 8). Hence, the algorithm only handles stores and calls and increments i (lines 8–12) once propagation and edge addition from field reads have iterated to a fixed point.

The $G' = G$ check on line 7 is crucial for getting the full benefit of unbounded field reads. Consider the following example:

```
1 var y = {...};
2 var x = y.a.b.c;
3 x.m = p => {...};
4 x.m(...);
```

We have three nested field reads at line 2 and one field write on the resulting object at line 3. With unbounded field reads, one would expect the call at line 4 could be discovered with an indirection bound of 1. But, without checking for $G' = G$ at line 7, the counter i for writes and calls would still be incremented while handling the reads, exhausting the bound before the relevant data flow from reads was discovered. Algorithm 5 discovers the call with $bound = 1$, as desired. In our implementation, constraints from JavaScript method calls (see Figure 3.2c in Section 3.2) are also handled in an unbounded manner, similar to handling of field reads in Algorithm 5.

3.4.4 Bounded ACG

Algorithm 6 Bounded ACG algorithm [26], as a modified version of Algorithm 4.

```
1: bound: bound on indirections
2: i: current iteration, initially 0
3: procedure Init()
4:     for each x = p =>_i {...} ∈ P do
5:         pt(x) ← pt(x) ∪ {f_i}
6:     for each x = y, x = z.y, or z.x = y ∈ P do
7:         G ← G ∪ {y → x}
8:     for each return_i x ∈ P do
9:         G ← G ∪ {x → ret_i}
10: end procedure
11: procedure AddEdges()
12:     if i < bound then
13:         for each x = y(z) ∈ P, f_i ∈ pt(y) do
14:             G ← G ∪ {z → p_i, ret_i → x}
15:         i ← i + 1
16: end procedure
```

The ACG algorithm of Feldthaus et al. [26] is a well-known technique for building JavaScript call graphs. ACG uses a field-based modeling of field accesses, unlike the field-sensitive formulation of Section 3.3.1. In ACG, reads and writes of object fields are modeled as assignments to and from global variables, and hence they do not introduce indirections for the analysis. Algorithmically, both the original ACG analysis and an indirection-bounded variant can be phrased as a simplified version of wave propagation. Pseudocode for indirection-bounded ACG is given in Algorithm 6; code related to bounding is again shown in blue (lines 12 and 15). The Analyze and Propagate procedures (elided) are identical to those in Algorithm 4. Field reads and writes are now handled similarly to assignments in Init (line 6). AddEdges is modified to remove all handling of field accesses. The only remaining indirections to handle in AddEdges are calls, as in 0-CFA [101]. Bounding is also simplified compared to Algorithm 5, as field reads do not require any special treatment.

3.5 Evaluation

We have implemented the techniques of Section 3.4 in two analysis frameworks. JELLY [49, 73], implements a field-sensitive call graph analysis using an algorithm like wave propagation [89], and is targeted at Node.js programs. WALA [27], has an implementation of the ACG algorithm and is targeted at browser-based applications. We modified these implementations to optionally use bounding as described in Sections 3.4.3 and 3.4.4.

With these implementations, we performed an experimental evaluation to assess the effectiveness of indirection-bounded call graph construction. We designed our evaluation to answer the following main research questions:

1. How are *analysis running time*, *recall* and *precision* impacted by different values of the indirection bound?

2. How does bounding of field reads and method calls (disabling the SELECTIVE flag of Section 3.3.2) impact the overall effectiveness of the analysis?

3.5.1 Benchmarks

For Node.js benchmarks, we further filtered the benchmarks used in the dynamic study based on the following three criteria. Initially, we required the dynamic call graph to contain at least 50 call edges to provide a sufficient basis for comparing the precision and recall of the static call graph. Then, we required JELLY could compute a static call graph with a recall of at least 20% as compared to the dynamic call graph; this eliminated some benchmarks where JELLY analyzed only a small portion of the code (typically due to unsupported test frameworks). This criterion also eliminated cases where JELLY ran out of memory. Second, we required that JELLY took at least 15 seconds to analyze the benchmark; for benchmarks that can be analyzed quickly, there is no need for bounding.

Table 3.3: Node.js benchmarks used for indirection-bounded static analysis experiments with JELLY.

Benchmark	#Pkgs	#Mods	#Funs	Code Size (kB)	Analysis Time (secs)	Precision (%)	Recall (%)
node-glob	42	186	2,621	1,103	22.38	88.01	90.14
kraken-js	130	303	3,192	1,416	19.62	96.85	37.95
tern	22	233	3,318	1,469	17.67	92.69	80.70
doctoc	96	322	3,212	1,515	15.25	93.97	74.18
js-yaml	59	479	4,440	1,900	28.86	96.42	95.38
react-loadable	55	138	3,192	1,952	405.44	99.35	81.50
babel-plugin-module-resolver	67	537	4,886	2,063	55.55	99.70	91.82
scrape-it	198	390	4,337	2,297	32.80	95.71	80.08
express-react-views	64	489	4,987	2,390	79.19	96.30	98.81
node-oauth2-server	36	262	4,588	2,448	461.66	75.43	95.53
lost	71	994	4,597	2,670	34.28	96.13	98.66
json2csv	112	428	6,743	2,692	29.99	97.24	81.45
homebridge	91	378	5,680	2,787	27.15	92.86	76.51
sharedb	37	266	5,630	3,164	47.64	76.54	65.95
big.js	1	26	1,718	3,306	15.88	96.02	100.00
normalizr	142	747	9,845	3,694	136.44	85.77	81.45
react-refetch	86	408	7,691	3,829	643.23	97.78	90.60
baobab	113	618	9,081	3,957	125.39	92.51	92.79
react-fontawesome	84	410	7,487	4,015	98.99	99.25	99.10
You-Dont-Need-Momentjs	63	742	8,024	5,404	48.10	85.91	84.43
eslint-plugin-compat	80	1,349	7,625	5,513	32.81	97.05	96.20
rewire	103	837	12,316	6,813	82.47	94.62	89.38
bootlint	110	741	10,022	6,835	57.96	96.85	80.46
webpacker	92	922	13,048	6,930	317.69	98.67	83.71
webpack-dashboard	122	1,182	17,783	9,823	834.18	96.31	76.44

Table 3.4: Web and Mobile benchmarks used for experiments with WALA.

Benchmark	#Pkgs	#Mods	#Funs	Code Size (kB)	Analysis Time (secs)	Precision (%)	Recall (%)
Vanillajs	2	8	131	30	0.32	90.13	98.64
Mithril	3	8	136	57	0.35	82.77	90.83
Vue	4	6	623	247	0.68	88.70	95.64
Knockoutjs	4	4	586	308	0.69	89.65	97.31
Jquery	5	5	869	371	2.51	82.75	98.27
Backbone	6	11	950	387	3.93	78.93	97.77
Canjs	5	7	1,105	559	10.15	76.55	97.49
Knockback	8	11	1,798	764	29.25	80.49	96.35
Angularjs	5	9	1,488	1,093	17.08	83.53	95.87
React	5	10	1,876	1,168	45.73	67.67	97.65
Blank-app	–	–	5,203	2,030	309.77	62.33	67.45
Chat-app	–	–	7,119	3,185	1,537.29	55.12	64.85
Bidding-app	–	–	7,073	3,194	1,627.19	55.18	65.02
Todolist-app	–	–	11,678	5,704	8,340.14	57.98	68.28

This filtering led to a set of 25 benchmarks, whose details are given in Table 3.3. The benchmarks are large, with thousands of functions and ranging up to more than 9.8MB of code. For each benchmark, the table states the numbers of packages, modules and functions, the code size, and the analysis time, precision, and recall for Jelly when run without bounding.

For assessing WALA, we constructed a suite of 14 web and mobile benchmarks, shown in Table 3.4. We included the 10 programs from the TodoMVC suite that were used by Chakraborty et al. in their study [15], and we re-used their test harness to exercise the programs. We also included four sample React Native applications, encompassing the starter app, a to-do app, a chat app, and a bidding app, all gathered from GitHub.

We chose apps we could run successfully in a simulator and that worked with our analysis infrastructure, and we manually developed a test harness for each to exercise

the code. For these apps, we constructed call graphs for the final shipping version of the code, which is bundled as a single file; hence, package and module counts are omitted in the table. The mobile benchmarks are much larger and more complex than the web benchmarks, explaining the higher analysis times and lower recall for those programs. While some of the smaller benchmarks in this suite can be analyzed very quickly, we retained them in the suite due to the challenge of manually exercising dynamic behaviors in web and mobile apps.

3.5.2 Experimental Configuration

In our experiments we measure precision and recall using the call site targets metric described in Section 3.2, i.e., by comparing sets of possible call targets at call sites. We found that this metric most robustly captured the data flows discovered by the static analysis. We also experimented with the reachable functions and reachable edges metrics (see Section 3.2). These metrics showed similar trends as for the call site targets metric, but are also more fragile since discovery of one additional call graph edge sometimes dramatically impacts overall reachability, making the results harder to interpret.

We ran the Jelly experiments on a machine with an 8-core Intel Core i7-11700 processor and 32GB of RAM, running Ubuntu 20.04.6 LTS. For the WALA experiments we used a Google Cloud virtual machine with a 4-core Intel Broadwell Xeon CPU and 64GB RAM running Ubuntu 20.04.6 LTS.

3.5.3 Results

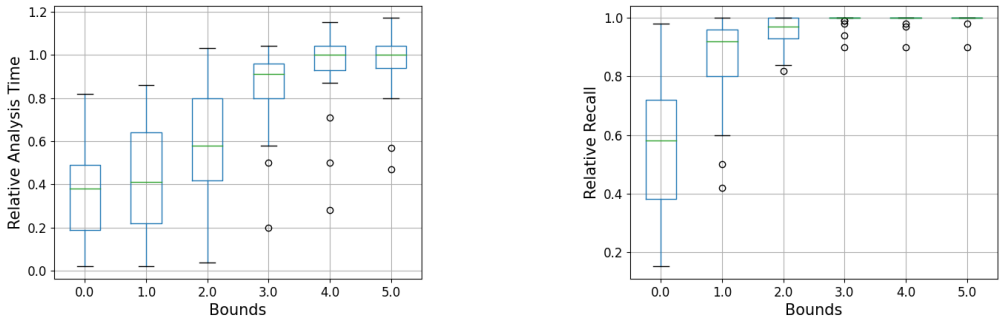
Figure 3.3 presents our main results for the Jelly experiments. For these experiments, field read and method call indirections were unbounded, the preferred configuration as discussed in Sections 3.2 and 3.3.3. The box plots give analysis time and recall relative to the unbounded Jelly analysis. The ideal for a bounded analysis would be to have recall as close to 1.0 as possible, so no recall is lost compared to unbounded, with analysis time as close to 0.0 as possible, maximizing performance. The analysis time data points sometimes extend above 1.0 primarily due to noise in the running time measurements. Additionally, the number of cycle elimination runs in wave propagation [89] can be affected by the use of different types of bounds in the analyzer. This variance can slightly increase or decrease

overall analysis running time, depending on the number of cycles in the constraint graph.

Studying Figures 3.3a and 3.3b, indirection bound 2 gives the best balance of analysis time improvement and recall. The average analysis time speed-up is roughly 2X (ranging from 0.9X–23.9X), while the average relative recall is 95% (82%–100%) of the unbounded analysis. The high relative recall matches the results of our dynamic study (Section 3.3.3), where we observed that 96% of dynamic calls were discoverable within a bound of 2. At bound 1, recall loss is significant at 17%, whereas bound 3 shows a minimal recall loss of only 1%, although the analysis time increases significantly. At bound 2, the recall loss is moderate at 5%, offering a balance between recall and analysis time.

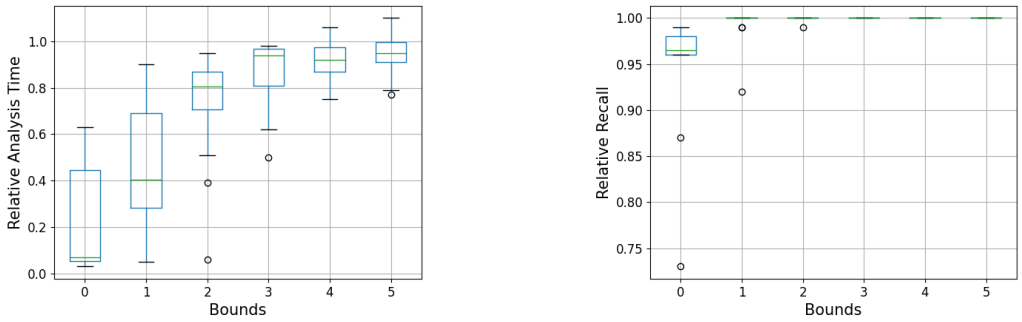
Figure 3.4 gives results for our WALA experiments. Here, we see that bound 1 yields a good overall trade-off, with a roughly 2X average speed-up (1.1X–21.4X) with an average relative recall of 99% (91%–100%). Bound 2 yields 99.99% relative recall on average with a smaller average speedup of 1.6X. We believe the higher recall numbers at lower bounds compared to Jelly are due to the fact that the ACG algorithm has fewer types of indirections to bound (see Section 3.4.4), and hence more data flow is discovered within a lower bound. For the web benchmarks, the largest observed performance improvements were for the largest benchmarks (e.g., for React we saw a 21X improvement). It would be useful future work to evaluate bounding on a suite of larger web and mobile benchmarks, but exercising such benchmarks to get good coverage of dynamic behaviors can be challenging (due to many user interactions, server-side state, etc.).

Finally, Figure 3.5 gives data for our second research question, showing results for Jelly with all indirections bounded, including field reads and method calls. Here, the bound with the closest analysis time / recall trade-off to our main configuration is bound 5, with a relative recall distribution fairly similar to bound 2 in Figure 3.3b. However, at this bound we see some higher outliers in analysis time, ranging up to 89% of the unbounded analysis time. Further, as shown in Section 3.3.3, with this configuration there is a long tail of calls that require a much higher bound to discover; in Figure 3.5b, even at bound 5, there is one benchmark with relative recall below 80% and five below 90%. Given these



(a) Analysis time (b) Recall

Figure 3.3: Analysis time and recall for JELLY with different indirection bounds and SELECTIVE enabled, relative to JELLY's unbounded analysis.

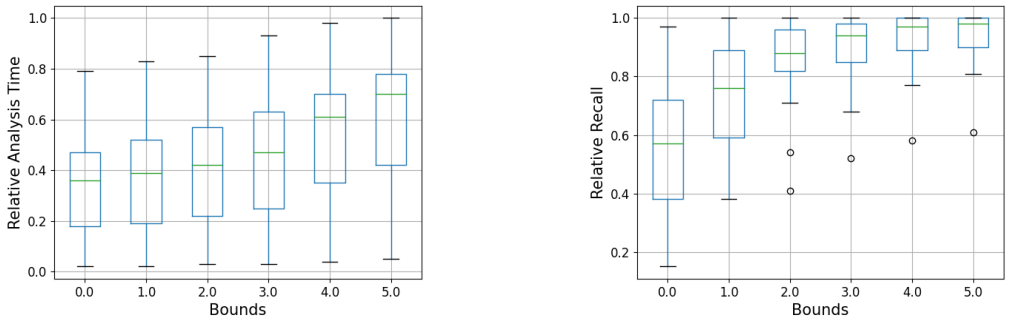


(a) Analysis time (b) Recall

Figure 3.4: Analysis time and recall for WALA with different indirection bounds, relative to WALA's unbounded analysis.

considerations, and the naturalness of code patterns with high numbers of field read and method call indirections (Sections 3.2 and 3.3.3), we believe analysis with selective bounding is the better choice for more robust results.

Without indirection bounding, there were 31 benchmarks that could not be analyzed because they caused the analyzer to run out of memory. With indirection bounding, 7 of those benchmarks can be successfully analyzed without running into memory issues.



(a) Analysis time

(b) Recall

Figure 3.5: Relative analysis time and recall for JELLY with reads and methods calls also bounded (SELECTIVE disabled).

3.5.4 Threats to Validity

A threat to the external validity of our evaluation is our choice of benchmarks. For Node.js we chose a large set of realistic benchmarks in a principled manner (see also Section 3.3.3). For web and mobile we have smaller sets of benchmarks, due to challenges in exercising such benchmarks to collect dynamic data. It is possible that on other types of benchmarks, bounding will be less effective. Our results may also be internally invalid due to bugs in our implementation. We have a variety of regression tests to check correctness of our results and we have done extensive manual inspection of complex examples, reducing this threat. Finally, our choice of call site targets as the precision/recall metric is another threat to external validity. We chose this metric since it best measures the overall effectiveness of the static analysis in capturing function data flows. But, there could be scenarios where a client relies on certain critical edges being present in the static call graph, and those particular edges require a bound greater than 2 to discover. In such cases, a higher indirection bound (or other heuristics) would be required to produce a useful static call graph, leading to higher analysis time.

3.6 Related Work

The most closely related work is the recent study by Mathiasen and Pavlogiannis [61] on the complexity of different variants of Andersen's classic pointer analysis. Most importantly, they presented an algorithm for solving Andersen-style pointer analysis instances with bounded numbers of store operations in witnesses of points-to relations, and proved that the algorithm runs in almost quadratic time. In comparison, the indirection-bounded analysis that is based on wave propagation or ACG remains cubic time for a fixed bound. As mentioned in Section 3.1, Mathiasen and Pavlogiannis conjecture that the level of indirection is typically small but without giving empirical evidence and without experimentally evaluating the effects on analysis time, precision and recall. Furthermore, their work focuses on a C-like language with field-insensitive analysis and without involving higher-order functions, whereas we consider field-sensitive hybrid call graph and pointer analysis for JavaScript. It remains an open problem whether their complexity results can be adapted to field-sensitive analysis. The algorithm and theoretical complexity results by Mathiasen and Pavlogiannis are based on Dyck reachability and matrix multiplications, whereas we base our approach on the wave propagation algorithm that is known to work well in practice. For example, wave propagation is used in the SVF analysis tool for LLVM [107] and also constitutes the core of the PUS constraint solver [56].

Mathiasen and Pavlogiannis [61] additionally showed that their bounded analysis technique is perfectly parallelizable, in contrast to ordinary Andersen pointer analysis. It will be interesting in future work to investigate whether that theoretical property can be exploited in practice to parallelize indirection-bounded call graph analysis.

Horwitz [40] studied a similar notion of levels of pointer indirection, but for reasoning about the analysis precision loss that may occur when normalizing pointer operations, which is not immediately related to bounded analysis techniques.

Utture and Palsberg [116] introduced a mechanism for analyzing library code only partially to speed up whole-program static analysis of application code. Their technique retains precision but, like indirection-bounded analysis, may lose some recall. We believe such approaches could be combined with indirection-bounded analysis to speed up analysis even further.

Utture et al. [114] (and follow-up work [51]) propose the notion of a call-graph pruner, which aims to improve analysis precision by eliminating call edges that are likely to be false positives. Like indirection-bounded analysis, that technique may negatively affect recall but in practice often achieves a good balance between precision and recall. The technique works as a post-processing phase and as such does not improve analysis time, and it relies on a learning algorithm that does not provide semantics-based, predictable outcomes.

Bounds have often been used in configuring the abstraction used by a static analysis, e.g., k-limiting for context sensitivity or access path length [102]. Indirection bounding is fundamentally different, in that it heuristically terminates the core fixpoint computation of the analysis before a fixed point is reached. Hence, unlike the aformentioned types of k-limiting, indirection bounding impacts analysis soundness, trading off a small amount of recall for improved scalability.

As pointed out in Section 3.1 it is well known that practically all whole-program static analyzers have imperfect recall [57], but the analysis results are still useful for many use cases. Reif et al. [93] and Sui et al. [106] investigated this phenomenon empirically for state-of-the-art analyzers for Java, and Antal et al. [8] have made a similar study for JavaScript call graph analysis tools. The more recent work by Chakraborty et al. [15] introduced a method for quantifying the root causes of missing edges in call graphs produced by a field-based static analysis for JavaScript [26]. The dynamic analysis used in [17] inspired the technique presented in Section 3.3.

3.7 Conclusion

Indirection-bounded analysis is a simple but effective approach for speeding up call graph analysis while missing relatively few call edges. The approach complements the theoretical results of Mathiasen and Pavlogiannis by providing a practical algorithm and empirical evidence, and it generalizes their bounded mechanism to a language with higher-order functions and to field-sensitive analysis. The results of the dynamic analysis presented in Section 3.3 indicate that real-world JavaScript code tends to have low levels

of indirection of function calls and field writes, which gives a semantic justification of the approach. We have demonstrated that indirection-bounded analysis is straightforward to incorporate into Pereira and Berlin's wave propagation algorithm and also into the field-based ACG algorithm by Feldthaus et al., and that it can be advantageous to choose a fixed bound independent of the individual programs being analyzed.

For future work, it may be interesting to explore the potential of indirection-bounded analysis for other programming languages, and to investigate whether the parallelizability results of Mathiasen and Pavlogiannis also hold in presence of higher-order functions and field-sensitive analysis.

Data Availability

The supplementary material is available online[2]. It contains the benchmarks used in the experimental evaluation and instructions for using indirection-bounded analysis with the open source analysis tools Jelly and WALA.

[2] `https://zenodo.org/doi/10.5281/zenodo.12720724`

Chapter 4

Promise Unkept: Detecting Mismanaged Promises in JS

Promises in JavaScript have become the standard mechanism for handling asynchronous operations, replacing callback-based patterns to avoid deeply nested code structures. However, promise-based code introduces new classes of programming errors. Failure to attach rejection handlers can result in unhandled promise rejections that crash applications, while the absence of fulfillment handlers may cause important asynchronous results to go unnoticed. Existing approaches for detecting these issues include runtime checks, linters, and dynamic analysis, but each has limitations: runtime mechanisms (e.g., in Node.js) only flag issues when faulty paths are actually executed—often too late in production; linters rely on syntactic patterns and often miss deeper semantic issues; and dynamic analyses depend heavily on test coverage, making them blind to rarely executed paths.

These limitations motivate the need for a static analysis that can reason about promise behavior across all program paths without relying on execution. This chapter presents such an approach for detecting missing promise handlers in JavaScript applications. We implement our technique in PromisePatrol, which uses abstract interpretation to model promise creation, state transitions, and handler attachment, tracking these properties interprocedurally through function calls. A key contribution is the Entry Anchor technique, which identifies functions that directly manipulate promises as targeted analysis starting

points, improving both focus and scalability. We evaluated our approach on 100 popular npm packages and identified 15 true instances of missing promise handlers across 7 distinct projects. These findings demonstrate that statically detecting promise-related issues in JavaScript is both feasible and effective, providing actionable insights into asynchronous defects that often evade traditional testing.

4.1 Introduction

JavaScript plays a central role in the modern software ecosystem, powering applications that range from interactive front-end interfaces to mission-critical backend systems. To support responsiveness in such diverse environments, JavaScript employs an event-driven, non-blocking concurrency model. Long-latency operations such as file and network access are delegated to the event loop, allowing the main thread to continue executing other tasks.

Early JavaScript programs managed asynchrony using callbacks, in which the continuation of a computation was passed as an argument to asynchronous functions. Although flexible, this approach often led to deeply nested and difficult-to-maintain control structures—commonly referred to as *callback hell*. Such patterns made code complex, hard to read, and error-prone, complicating both error propagation and control flow reasoning. To address these shortcomings, ECMAScript 2015 (ES6) standardized *Promises*, introducing a more structured abstraction for asynchronous programming. The design of JavaScript's Promise mechanism was influenced by prior work in the *E* language [70], and the Promises/A+ specification [90].

In addition to improving code clarity, the ES6 `Promise` API provided a standardized set of methods for creating and composing asynchronous operations. Developers can use `Promise.resolve()` and `Promise.reject()` to quickly produce fulfilled or rejected promises. For coordinating multiple tasks, `Promise.all()` waits for all promises to fulfill, while `Promise.race()` settles as soon as one does. Later additions like `Promise.any()` and `Promise.allSettled()` introduced further flexibility for aggregating results. The `async/await` syntax introduced in ES2017 built on Promises to offer a more readable, sequential style of asynchronous programming.

While `Promise`s offer a structured model for asynchronous programming, they also give rise to subtle and often non-obvious correctness issues. One such issue arises from *unresolved promises*, which neither fulfill nor reject and can cause memory retention or indefinite suspension of computation. A more critical concern involves *unhandled rejections*, where a rejected promise lacks an associated rejection handler. Beginning with version 15.0.0 [81], Node.js classifies unhandled rejections as fatal errors [111], reflecting extensive community debate and real-world failures observed in production environments [20]. Such errors may result in resource exhaustion, inconsistent program state, or loss of control-flow integrity.

A second, less widely discussed category of errors arises from *missing fulfillment handlers*—cases where a Promise is fulfilled but its result is never consumed. These cases are often indicative of developer oversights, such as unintentionally dropped control or data flow. Although not fatal, such omissions can lead to lost progress, inconsistent application state, and performance degradation. These issues are rarely flagged by linters or runtime checks, yet they indicate a clear mismatch between program behavior and developer intent. Prior work has paid comparatively less attention to these cases, and existing dynamic approaches struggle to reliably detect them due to dependence on runtime coverage and test oracles.

Existing techniques for identifying these defects have significant limitations. Runtime monitoring can detect problems only when specific execution paths are triggered, which may be rare in production settings. Linters and pattern checkers operate on purely syntactic heuristics, lack path sensitivity, and are limited to intra-procedural reasoning. Tools like ESLint [122], while widely used, do not track interprocedural Promise propagation or model handler attachment across chains. Dynamic analyses on the other hand offer greater expressiveness but their effectiveness is ultimately constrained by test coverage and input diversity. These limitations motivate the need for static analyses that provide broader coverage and deeper semantic reasoning about Promise behavior enabling detection of these defects early and reliably.

Our Contribution. In this work, we present PromisePatrol, a static analysis framework for detecting missing fulfillment and rejection handlers in JavaScript programs that use Promises. Built atop Jelly, a state-of-the-art static analysis infrastructure, PromisePatrol performs interprocedural analysis and models the semantics of asynchronous constructs to reason deeply about promise creation, resolution, and handler propagation across function boundaries. This enables detection of subtle defects that arise from complex control and data flow patterns in asynchronous code. Our key contributions are:

- A novel interprocedural static analysis for detecting missing fulfillment and missing rejection handlers, addressing subtle misuse patterns that elude traditional linting or runtime-based detection.
- A call graph-guided entry point discovery mechanism, *Entry Anchors*, that improves analysis scalability by isolating promise-relevant roots in the call graph.
- An empirical evaluation of PromisePatrol on 100 widely-used *npm* packages, uncovering 15 true positives across 7 projects, and demonstrating the practical effectiveness of static Promise mismanagement detection.

These contributions advance the state of static reasoning about asynchronous behavior in JavaScript and provide a foundation for more robust and scalable tooling for Promise misuse detection.

4.2 Promise Mismanagement Defects

Promises provide a structured interface for handling asynchronous computations. A Promise encapsulates a computation that may eventually complete with a value (fulfillment) or terminate with an error (rejection). Handlers for these outcomes can be registered via methods such as `then(onFulfilled, onRejected)`, `catch(onRejected)`, or by using the `await` keyword. It is worth noting that the `await` construct implicitly acts as a fulfillment handler, resuming execution when the Promise resolves.

Figure 4.1 illustrates common ways to handle fulfilled and rejected JavaScript Promises. Fulfillment is typically handled using .then(onFulfilled) or the await keyword,

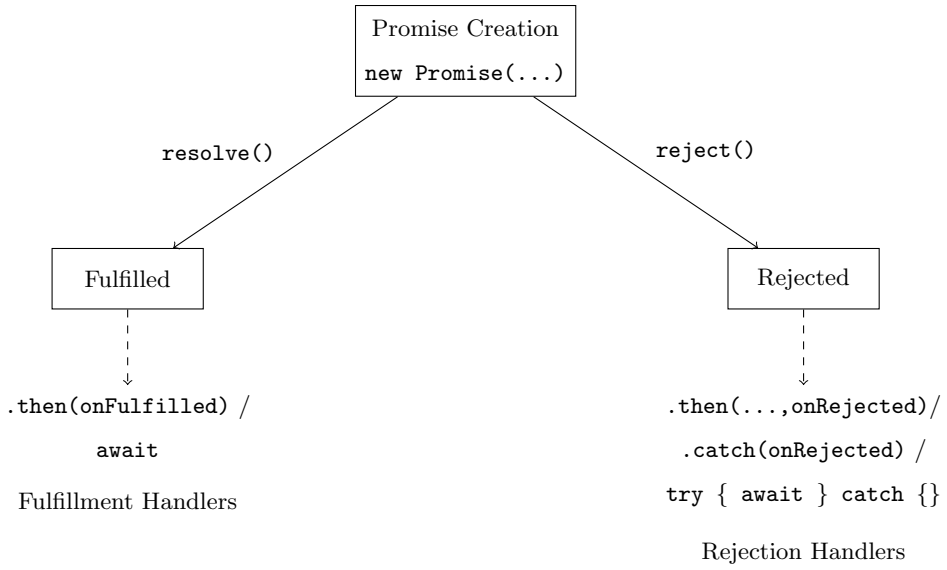


Figure 4.1: Lifecycle of a JavaScript Promise and points of handler attachment.

while rejection is handled via .then(..., onRejected), .catch(), or try/catch blocks surrounding await. The figure emphasizes that fulfillment and rejection are distinct outcomes, each handled through different language constructs. When such handlers are omitted, the program may proceed without responding to success or failure, resulting in missed effects or unhandled exceptions. These omissions are often difficult to detect, especially when they arise under rare timing conditions or in deeply nested asynchronous flows.

To guide our analysis, we formalize two core categories of Promise-related defects: *Missing Fulfillment Handlers* (MFH) and *Missing Rejection Handlers* (MRH). These defects arise when a Promise may fulfill or reject along some execution path, yet no corresponding handler is present to observe or respond to that outcome. MFH and MRH reflect violations of robust asynchronous design principles, where each deferred computation is expected to have a defined and observable lifecycle. Identifying such omissions early can prevent subtle data loss, control-flow inconsistencies, and runtime failures in production systems.

Missing Fulfillment Handler (MFH). A Missing Fulfillment Handler arises when a Promise may transition to the fulfilled state, yet no handler is registered to observe or make use of the resulting value.

Definition. Let P denote the set of all Promise instances in the program. For each $p \in P$, let Π_p represent the set of feasible execution paths originating from the creation of p. We say that p exhibits an MFH defect if there exists a path $\pi \in \Pi_p$ such that:

1. p may reach the *fulfilled* state along π; and
2. no effective fulfillment handler is reachable on π, where a fulfillment handler is defined as:
   - an invocation of `.then(onFulfilled)` attached to p, or
   - an occurrence of `await` applied to p within a reachable control-flow context.

Failure to register a fulfillment handler may result in lost computation, unobserved side effects, or inconsistent program state if subsequent operations assume the Promise was processed.

Example: Consider an asynchronous operation that updates a user's profile and returns the updated profile object or a status.

Listing 4.1: Missing Fulfillment Handler leading to unobserved operation status.

```
1 async function updateUserProfile(userId, data) {
2   return db.updateUser(userId, data);
3 }
4 updateUserProfile('user123', { lastLogin: new Date() });
5 // The operation's success is unobserved, may mask critical state transitions.
```

In Listing 4.1, the call to `updateUserProfile` produces a Promise that may fulfill if the database update succeeds. However, the returned Promise is neither awaited nor chained with a `.then()` handler. Along this feasible execution path, the fulfilled value is unobserved and unused. According to the MFH definition, this constitutes a missing fulfillment handler, which may lead to inconsistencies if subsequent logic implicitly assumes that the update has completed successfully.

Missing Rejection Handler (MRH). A Missing Rejection Handler arises when a Promise may transition to the rejected state, yet no handler is registered to observe or respond to the error.

Definition. Let p be a Promise instance created in the program, and let Π_p denote the set of feasible execution paths originating from the creation of p. We say that p exhibits an MRH defect if there exists a path $\pi \in \Pi_p$ such that:

1. p may reach the *rejected* state along π; and
2. no effective rejection handler is reachable on π, where a rejection handler is defined as:
   - an invocation of `.catch()` or `.then(..., onRejected)` attached to p, or
   - an enclosing `try { await p } catch {...}` block on a path that observes the rejection.

Unhandled rejections are treated as fatal errors in many JavaScript environments (e.g., Node.js v15 and later), and are notoriously difficult to trace during testing.

Example: Consider an asynchronous operation that reads a configuration file and parses its content.

Listing 4.2: Missing Rejection Handler leading to unhandled rejection.

```
1 import fs from 'fs/promises';
2 async function getConfig() {
3   const configData = await fs.readFile("app-config.json");
4   return JSON.parse(configData);
5 }
6 getConfig();
```

In Listing 4.2, the call to `getConfig()` produces a Promise that may reject if the file `"app-config.json"` does not exist or cannot be read. The `await` within `getConfig` propagates this rejection by returning a rejected Promise. However, the call to `getConfig()` is neither wrapped in a `try/catch` block nor chained with a `.catch()` or `.then(..., onRejected)` handler. Along this feasible execution path, the rejection remains unobserved. According to the MRH definition, this constitutes a missing rejection handler, potentially leading to program termination or silent failure in a larger system.

MRH defects are a direct violation of robust error handling principles, akin to uncaught exceptions in synchronous programming. In modern JavaScript environments

like Node.js, an unhandled Promise rejection is a critical error that can terminate the application [80]. In browser environments, while they might not always crash the tab, they lead to errors logged in the console and can leave the application in an undefined or inconsistent state, potentially resulting in silent failures or degraded user experience [68]. The failure to handle potential rejections means that the program is not prepared for exceptional circumstances, violating the principle of "fail-safe" or "fail-fast" where errors are caught, logged, and potentially recovered from, rather than propagating silently or causing abrupt termination.

Detecting MFH and MRH defects statically, before runtime, is crucial for building reliable and maintainable asynchronous JavaScript applications, however it presents significant challenges. First, Promise-based asynchronous flows often span multiple modules, requiring precise interprocedural analysis and robust call-graph construction. Second, Promise rejection handlers exhibit unique propagation semantics, complicating the identification of effective handlers, like a single `catch` may handle rejections for an entire chain. Additionally, `try-catch` blocks enveloping `await` expressions necessitate context-sensitive analysis to accurately ascertain error-handling coverage across layered asynchronous calls. Lastly, JavaScript's dynamic nature, including higher-order functions and dynamic property access, exacerbates call-graph construction complexities, increasing the likelihood of missed defects (false negatives) or excessive warnings (false positives). Addressing these challenges demands sophisticated static analysis frameworks capable of precise, scalable interprocedural reasoning about asynchronous Promise usage.

4.3 Approach

Our goal is to detect the two classes of unhandled Promise usage in JavaScript programs, namely MFH and MRH that we identified in Section 4.2. We achieve this through a staged static analysis pipeline. The key stages of our analysis are:

1. call graph construction using a static call graph generator `Jelly`,
2. identification of *Promise-relevant functions*,
3. computation of analysis *Entry Anchors* to guide the abstract interpretation, and

4. a callstack-based abstract interpretation over the program's abstract syntax tree (AST) tailored for Promise semantics.

4.3.1 Call Graph Construction

We use `Jelly`, a recent static analysis framework for JavaScript/TypeScript programs targeting Node.js to construct a conservative, over-approximated call graph. Jelly performs field-sensitive, flow-insensitive points-to analysis and models most core ECMAScript constructs. Its design is inspired by techniques from TAPIR [71], JAM [79], and ACG [26], while supporting recent advances in approximate interpretation [49] and indirection bounding [14]. Jelly constructs a call graph $G = (V, E)$, where each node $v \in V$ represents a function and each edge $(v_1, v_2) \in E$ indicates a possible call from v_1 to v_2. We rely on this call graph as the foundation for subsequent entry anchor discovery and interprocedural reasoning.

4.3.2 Identification of Promise-Relevant Functions

We define a function as *Promise-relevant* if it creates, returns, or consumes a Promise object. Let $R \subseteq V$ denote the set of Promise-relevant functions. A function $f \in V$ belongs to R if it satisfies any of the following syntactic or semantic criteria:

1. It directly constructs a Promise, e.g., via `new Promise(...)`.
2. It returns a Promise, either explicitly or by virtue of being declared `async`.
3. It receives a value that may be a Promise through:
   - parameter passing,
   - variable capture in closures,
   - or the return value of another function.

Promise-relevance also propagates transitively: if a function calls another Promise-relevant function and uses its result in subsequent computation (e.g., via chaining, awaiting, or other control/data flow), it is also marked as Promise-relevant. This transitive closure ensures that wrapper functions and pass-through layers are accurately captured. If Promises

Algorithm 7 Entry Anchor Determination for Promise Analysis

```
1: function ComputeEntryAnchors
2:     E ← ∅
3:     Visited ← ∅
4:     function Explore(f)
5:         if f ∈ Visited then return
6:         Visited ← Visited ∪ f
7:         hasRelevantCaller ← false
8:         for all c ∈ C(f) do
9:             if c ∈ P ∧ c ∈ R then
10:                hasRelevantCaller ← true
11:                Explore(c)
12:        if ¬hasRelevantCaller then
13:            E ← E ∪ f
14:    end function
15:    for all f ∈ P ∩ R do
16:        Explore(f)
17:    return E
18: end function
```

are created or consumed in the global or module scope (i.e., outside any function), we treat the global scope as a synthetic function and include it in R as well.

The set R is computed via interprocedural static analysis over dataflow paths involving Promise allocations, usages, and return flows.

This classification plays a critical role in our analysis. First, it excludes large portions of the program that cannot influence Promise outcomes, improving scalability. Second, it enables identification of semantically meaningful analysis roots—*Entry Anchors*. Unlike traditional whole-program analyses that begin from global entry points, our approach initiates modular reasoning from Promise-relevant functions, making Promise-relevance a necessary precondition for precise and scalable static analysis.

4.3.3 Entry Anchor Determination

Traditional static analyses often begin from top-level entry scripts (e.g., the main file executed by Node.js) and traverse the call graph to infer behavior. However, in real-world projects, this approach can lead to scalability issues—especially when Promises are deeply nested within the call stack. To avoid redundant traversals and inefficient back-and-forth flow between callers and callees, we introduce the concept of *Entry Anchors* : Promise-relevant entry points that enable more precise and targeted analysis of asynchronous effects.

By initiating abstract interpretation from Entry Anchors rather than the global entry point, our analysis benefits from two key advantages. First, it localizes reasoning to semantically meaningful Promise flows, reducing unnecessary propagation through unrelated code. Second, it naturally supports modular analysis of asynchronous behavior, enabling scalable analysis across large codebases.

We compute Entry Anchors by performing a reverse traversal of the call graph. For each Promise-relevant function that is reachable from the application entry point, we explore its callers recursively. A function is marked as an Entry Anchor if none of its reachable callers are also Promise-relevant. This identifies functions from which new Promise behavior originates and ensures that analysis starts at relevant locations. The procedure is summarized in Algorithm 7.

4.3.4 Promise-Aware Abstract Interpretation

Our static analysis framework detects missing Promise handlers by applying abstract interpretation over the Babel AST of JavaScript programs [11]. The analysis models the lifecycle of Promises—including their creation, resolution, rejection, and handler registration—across interprocedural control flows. To ensure scalability, the analysis is seeded at the *Entry Anchors* computed in Section 4.3.3.

Algorithm 8 Promise Handler Analysis

Require: JavaScript promises P, Call graph G, Entry Anchors E

Ensure: Set of warnings $\mathcal{W}$ for promises with missing handlers

$\mathcal{W} \leftarrow \emptyset$

for all entry anchor $e \in E$ **do**

    ANALYZE$(e, \emptyset)$

return $\mathcal{W}$

function ANALYZE$(f, \mathcal{C})$ ▷ $\mathcal{C}$ is call stack

    if $f \in \mathcal{C}$ **then return** $\bot$

▷ Recursion detection

    $\sigma \leftarrow \bot$ ▷ Promise environment for f

    $\mathcal{C}' \leftarrow \mathcal{C} \cup \{f\}$

▷ Analyze function body with interprocedural calls

    for all statement $s \in f$ **do**

        $\sigma \leftarrow \mathcal{F}(s, \sigma)$ ▷ Intraprocedural transfer

        if s is call expression **then**

            for all target $g \in$ targets(s, G) **do**

                $\sigma_g \leftarrow$ ANALYZE$(g, \mathcal{C}')$

                $\sigma \leftarrow \sigma \sqcup$ TRANSFER(σ_g, s) ▷ Merge callee effects

    CHECKHANDLERS(f, σ)

    return σ

end function

function CHECKHANDLERS(f, σ)

    global $\mathcal{W}$

    $\mathcal{W}_f \leftarrow \emptyset$

    for all promise $p \in \sigma$ where $\neg$escapes(p, f) **do**

        if $\sigma[p]$.mayFulfill $\wedge \neg\sigma[p]$.HasFulfillmentHandler **then**

            $\mathcal{W}_f \leftarrow \mathcal{W}_f \cup \{$missing fulfillment handler$\}$

        if $\sigma[p]$.mayReject $\wedge \neg\sigma[p]$.HasRejectionHandler **then**

            $\mathcal{W}_f \leftarrow \mathcal{W}_f \cup \{$missing reject handler$\}$

    $\mathcal{W} \leftarrow \mathcal{W} \cup \mathcal{W}_f$

end function

4.3.4.1 Abstract Domain

To track Promise usage in a sound and tractable manner, we employ an allocation-site based abstraction where each promise is identified by its syntactic creation location. For each statically allocated Promise p, we define an abstract domain as:

$$\mathcal{D}_{\text{Promise}}(p) = \langle \texttt{MayFulfill}, \texttt{MayReject}, \texttt{HasFulfillmentHandler}, \texttt{HasRejectionHandler} \rangle.$$

Each component is a Boolean flag that approximates the Promise's lifecycle behavior across all control-flow paths.

The abstract state flags `MayFulfill` and `MayReject` capture whether a Promise may settle with a fulfillment or rejection outcome, respectively. These flags are joined using logical disjunction, so they evaluate to `true` if the transition is feasible on any path. This ensures the analysis conservatively accounts for all possible settlement behaviors observable along any control-flow path, without requiring assumptions about which paths are more likely to execute.

In contrast, `HasFulfillmentHandler` and `HasRejectionHandler` flags are joined using logical conjunction, requiring that the handler be attached along all paths for the flag to be `true`. This approach preserves soundness by marking a handler as present only if it is attached along all control-flow paths; if any path omits the handler, the analysis conservatively reports it as potentially missing. This prevents the analysis from overlooking unhandled Promises in the presence of complex data flow.

This interpretation ensures that the analysis flags missing handlers whenever their absence is possible along any control-flow path, and marks a Promise as capable of fulfilling or rejecting only if the corresponding transitions are syntactically reachable. This design reflects a sound, flow-sensitive analysis: a Promise is considered capable of fulfilling or rejecting if such a transition is possible along any path, while handler presence is confirmed only when it holds on all paths. This avoids unsound assumptions about handler coverage and ensures that potential settlement behaviors are not under-approximated.

4.3.4.2 Abstract Semantics

Our static analysis is implemented as an abstract interpreter that traverses the JavaScript AST and propagates abstract promise states interprocedurally, as outlined in

Algorithm 8. The analysis begins from a set of statically computed *Entry Anchors*, and maintains precision by tracking abstract environments per function call context.

The interpreter processes each function body in a forward, statement-by-statement manner, applying transfer functions $\mathcal{F}(s, \sigma)$ to update the abstract environment σ. The environment maps allocation-site identifiers to abstract promise states as defined in 4.3.4.1. Control-flow branching constructs such as conditionals are handled by exploring all feasible branches and joining the resulting environments using domain-specific merge operators. Loops are abstractly approximated by evaluating their bodies once and conservatively merging the result with the incoming state.

4.3.4.3 Transfer Functions

This section defines the transfer functions $\mathcal{F}(s, \sigma)$ for a selected set of Promise-related constructs that affect the abstract environment during analysis. These rules conservatively model critical aspects of the Promise lifecycle, including creation, fulfillment/rejection, and handler attachment. Further refinements—such as the propagation of handler state through chained promises and `await` expressions—are discussed separately in Section 4.3.4.5.

[TF-NewPromise] *Promise Allocation*

$$\frac{s \equiv \texttt{p = new Promise(...)} \quad \langle f, r \rangle = \texttt{HeuristicGuess}(p)}{\sigma' = \sigma[p \mapsto \langle f, r, \bot, \bot \rangle]}$$

Allocates a fresh abstract promise with handler flags unset. The initial values of `MayFulfill` and `MayReject` are determined using lightweight heuristics (see Section 4.3.4.7); if the outcome cannot be conclusively inferred, both flags are conservatively set to `true`.

[TF-AsyncReturn] *Async Function Invocation*

$$\frac{s \equiv \texttt{v = f(...)}}{\sigma' = \sigma[p \mapsto \langle f, r, h_f, h_r \rangle]} \quad \text{where} \begin{cases} f & = \bigvee_i \sigma_i(f).\texttt{MayFulfill} \\ r & = \bigvee_i \sigma_i(f).\texttt{MayReject} \\ h_f & = \bigwedge_i \sigma_i(f).\texttt{HasFulfillmentHandler} \\ h_r & = \bigwedge_i \sigma_i(f).\texttt{HasRejectionHandler} \\ \text{if } f \text{ returns one or more promises; otherwise:} & \\ \langle f, r \rangle & = \texttt{HeuristicGuess}(f), \quad h_f = \bot, \quad h_r = \bot \end{cases}$$

Every `async` function invocation introduces a synthetic abstract promise at the callsite. If the callee `f` explicitly returns one or more Promises, the resulting abstract state is computed by merging the states of all returned Promises. The flags `MayFulfill` and `MayReject` are joined using disjunction, ensuring that any possible resolution is conservatively preserved. On the other hand, the handler flags `HasFulfillmentHandler` and `HasRejectionHandler` are joined using conjunction, reflecting that the synthetic promise is considered handled only if all returned Promises are appropriately handled.

If no explicit Promise is returned by async function `f`, the analysis invokes `HeuristicGuess(f)` to estimate its settlement behavior. In this case, the handler flags are conservatively set to `false` ($\bot$), indicating that rejection or fulfillment may go unhandled unless explicitly observed.

[TF-Then] *Handler Attachment via* ***`then`***

$$\frac{s \equiv \texttt{q = p.then(onFulfilled, onRejected)} \quad \langle f, r \rangle = \texttt{HeuristicGuess}(q)}{\begin{array}{l} \sigma[p].\texttt{HasFulfillmentHandler} \leftarrow (\texttt{onFulfilled} \neq \texttt{undefined}) \\ \sigma[p].\texttt{HasRejectionHandler} \leftarrow (\texttt{onRejected} \neq \texttt{undefined}) \\ \sigma \leftarrow \sigma[q \mapsto \langle f, r, \bot, \bot \rangle] \end{array}}$$

Attaching a fulfillment or rejection handler via `then` updates the corresponding flags on `p`. A fresh derived promise `q` is allocated, initialized with state flags from `HeuristicGuess`, and no attached handlers. This ensures consistent treatment of derived promises.

[TF-Catch] *Rejection Handler via* `catch` **[TF-Catch]** *Rejection Handler via* `catch`

$$\frac{s \equiv \texttt{q = p.catch(onRejected)} \quad \langle f, r \rangle = \texttt{HeuristicGuess}(q)}{\begin{array}{l} \sigma[p].\texttt{HasRejectionHandler} \leftarrow (\texttt{onRejected} \neq \texttt{undefined}) \\ \sigma \leftarrow \sigma[q \mapsto \langle f, r, \bot, \bot \rangle] \end{array}}$$

The `catch` statement guarantees that rejection is handled on `p`. As with `then`, a new derived promise `q` is introduced into the environment and initialized without any attached handlers. Crucially, rejection handlers attached via `catch` (or the second argument of `then`) affect not only the immediate promise `p`, but also propagate backward through the chain: all ancestor promises leading to `p` are marked as having effective rejection handlers. This transitive marking reflects JavaScript's semantics, where a rejection handler installed at any point downstream can prevent an unhandled rejection warning for the entire chain.

In our implementation, complex chained expressions (e.g., `p.then(...).catch(...)`) are first normalized by introducing temporary variables to isolate intermediate results. Once normalized, the appropriate transfer functions are applied to each intermediate step individually.

These representative rules define how the environment is updated at key Promise-related constructs. Transfer functions for 'await' and try-catch constructs are discussed in more detail in Section 4.3.4.5, which elaborates on control- and data-flow interactions in chained and implicit handler contexts.

4.3.4.4 Interprocedural Analysis

Function calls are handled using a three-phase interprocedural analysis protocol designed to propagate promise states across function boundaries. The analysis uses context-sensitive environments to distinguish between different call sites and to model how Promises are passed, manipulated, and returned across functions.

Phase 1: Parameter Setup. When analyzing a call expression s, the analysis first resolves the set of possible target functions using the static call graph G. For each target function g, a new abstract promise environment is constructed by mapping actual arguments at the call site to the corresponding formal parameters of g. This mapping uses scope-aware

variable identifiers, which include module and lexical scope context to prevent conflation of bindings across unrelated functions. Promises that are passed as arguments are marked as escaping to the caller to enable proper tracking of their lifecycle across functions.

Phase 2: Callee Analysis. The callee function g is analyzed recursively using the newly initialized environment from Phase 1. To maintain context sensitivity, the analysis uses a call stack $\mathcal{C}$ to represent the current analysis context, preventing reanalysis of recursive functions within the same call chain. Each function body is interpreted using the same intraprocedural strategy described earlier—statements are processed sequentially, and local control flow (e.g., conditionals and loops) is modeled using conservative approximations.

Phase 3: Effect Integration. Once callee analysis completes, the effects of the callee are integrated back into the caller environment. Promises that escape the callee scope—either by being returned or by being modified in-place through parameter aliasing—are identified and merged into the caller's environment. This transfer preserves important metadata such as handler flags and chain relationship. This ensures that the interprocedural effects of handler attachment or Promise chaining are correctly propagated to all relevant calling contexts.

Special handling is applied for `async` functions. When a call site invokes an `async` function, the analysis allocates a fresh synthetic abstract promise in the caller environment to represent the eventual result, as discussed in Section 4.3.4.3.

This interprocedural strategy is critical for handling real-world JavaScript idioms, where Promises are frequently passed as arguments, returned from functions, or manipulated through callbacks. By explicitly modeling these flows, the analysis can track handler attachment and settlement behavior with high fidelity across function boundaries.

4.3.4.5 Promise Chain and Await Semantics

The analysis uniformly models both explicit and implicit mechanisms for attaching handlers in JavaScript. Explicit chaining operations such as `then` and `catch` are directly modeled by allocating fresh abstract promises for the chained result, establishing parent-child links between the source and derived promises, and updating handler flags based on the provided arguments. Rejection handlers attached via `catch` or the second argument to

`then` are propagated backward through the chain so that upstream promises are marked as handled if any downstream promise installs a rejection handler (see Section 4.3.4.3).

To implement this propagation, the analysis maintains an auxiliary structure that maps each abstract promise to the set of reference variables currently pointing to it, as well as to all downstream references created from it via chaining. When a handler is added to a promise reference (e.g., `q`), the analysis consults this auxiliary structure to identify all abstract promises whose reference sets include `q` and conservatively marks them as having the corresponding handler. This allows upstream promises in a chain to be updated as handled even if the handler is attached at a later point in the chain.

For example:

```
const p = Promise.resolve("value");
const q = p.then(val => val.toUpperCase());
const r = q.catch(e => e);
```

After `q.catch(...)` statement is analyzed, the analysis marks `p` and `q` as having rejection handlers, even though only `q` is explicitly annotated. This reflects the JavaScript semantics where a downstream `catch` suppresses unhandled rejection warnings for all upstream promises in the chain.

Await expressions are treated as syntactic sugar for chaining with implicit fulfillment and rejection handlers. When the analysis encounters `await` e, it marks the associated abstract promise returned by the function call as having a fulfillment handler. If the await is enclosed within a syntactic `try-catch` block, the rejection is considered handled as well. Promises returned by `async` functions are made to escape their caller function boundary if the caller does not attach a handler. To remain faithful to JavaScript semantics—where any caller along an awaited call chain can install a rejection handler—the analysis treats such escaping promises as implicitly passed upward through the call chain. When an `await` may result in rejection and is not protected by a `try-catch` or `.catch()`, the rejection is conservatively marked as escaping to the caller. This transitive escape continues through any sequence of awaited, unguarded calls. Propagation halts in these cases: (i) the rejection is caught, in which case it is considered handled; (ii) the callee is invoked without awaiting its result, making the rejection unobservable to the caller; or (iii) the rejection reaches the

outermost function, leaving no remaining opportunity for propagation or handling. In cases (ii) and (iii), the rejection is left unhandled.

This unified treatment of chaining and `await` semantics enables the analysis to track rejection and fulfillment handler coverage across both explicit and implicit asynchronous flows, ensuring sound detection of missing handlers.

4.3.4.6 Missing Handler Warnings

The final phase identifies potential Missing Fulfillment Handler (MFH) and Missing Rejection Handler (MRH) warnings by inspecting the final promise state at each function exit. For each promise p that remains local to the function and does not escape its scope, we check the following:

- **MFH Detection**: If $\text{mayFulfill}(p) = \top$ and $\text{HasFulfillmentHandler}(p) = \bot$, then report a potential Missing Fulfillment Handler.
- **MRH Detection**: If $\text{mayReject}(p) = \top$ and $\text{hasRejectionHandler}(p) = \bot$, then report a potential Missing Rejection Handler.

The effective rejection handler status accounts for handler propagation within promise chains. A promise has an effective rejection handler if any downstream promise in its chain has a rejection handler, since rejection handlers catch rejections from all upstream promises.

To avoid redundant reports, promises are grouped by their chains, and warnings are reported once per chain rather than per individual promise. This prevents duplicate warnings for promises that are semantically part of the same promise chain.

4.3.4.7 Heuristics for Determining Handler Necessity

A key challenge in Promise handler analysis is distinguishing genuinely problematic omissions from deliberate, semantically safe usage patterns. Without this distinction, our tool risks overwhelming developers with false positives in large codebases.

Consider the following cases:

- **Side-effecting operations**: Functions like `console.log()`, `fs.writeFile()`, or telemetry/reporting calls are often invoked without awaiting their completion. Requiring handlers in such cases would raise unnecessary warnings, as these are often used in non-critical contexts.
- **Deliberate fire-and-forget patterns**: Background tasks such as cache invalidation or analytics dispatch are intentionally launched without awaiting or error handling to avoid blocking control flow.
- **Top-level async execution**: Immediately Invoked Async Function Expressions (async IIFEs) are used to enable `await` at the module level. These typically do not require fulfillment handlers, though rejection handlers are sometimes attached for robustness.

To account for such patterns, our analysis incorporates lightweight syntactic heuristics to estimate whether a promise may fulfill or reject. These are implemented via a function called `HeuristicGuess`, which is invoked within the transfer functions defined in Section 4.3.4.3. For example, when analyzing `new Promise(...)` or chaining constructs like `then`/`catch`, the analysis calls `HeuristicGuess` to initialize the `MayFulfill` and `MayReject` flags of the resulting abstract promise.

A few examples would be:

- If the executor body of a `new Promise` lacks an obvious `reject()` call, `MayReject` may be initialized to `false`.
- If an async function does not contain any `throw` or `await` of rejecting calls, it is considered unlikely to reject.
- Known APIs such as `Promise.resolve()`, `Promise.allSettled()` that settle deterministically are treated accordingly (e.g., `MayReject` is set to `false`).

In ambiguous cases—e.g., when Promise behavior is inferred through opaque APIs or higher-order callbacks—the analysis conservatively sets both state flags to `true`, requiring handlers to be safe.

These heuristics are conservative yet effective, reducing warning fatigue while preserving the analysis' focus on actionable issues. As our evaluation in Section 4.4 shows, this filtering plays a critical role in the practical viability of the tool in real-world JavaScript codebases.

4.4 Evaluation

We evaluate PromisePatrol across a range of real-world JavaScript and TypeScript projects to answer the following research questions:

RQ1. How sound and precise is PromisePatrol in detecting missing `fulfillment` and `reject` handlers in real-world JavaScript programs?

RQ2. Can PromisePatrol be used to verify the absence of unhandled promise errors in practical applications?

RQ3. Does the analysis scale to large JavaScript/TypeScript codebases?

RQ4. What is the impact of handler necessity heuristics on reducing false positives in Promise misuse detection?

To address these questions, we applied our analysis to the *npm-100* benchmark suite, a curated collection of 100 actively maintained JavaScript and TypeScript packages from the npm ecosystem. We measured both the effectiveness and scalability of PromisePatrol, examined its ability to verify correct usage of Promises, and assessed the practical impact of optional heuristics designed to reduce false positives.

In addition to answering the research questions, we provide an illustrative case studies of real bugs detected by PromisePatrol and a discussion of the underlying challenges and limitations of Promise analysis.

The evaluation was conducted on a machine with an Intel Core i7-11700 CPU (8 cores, 16 threads, 2.5 GHz), 32 GiB RAM, running Ubuntu 20.04.6 LTS and Node.js v22.9.0, with a timeout of 10 minutes per project.

4.4.1 RQ1: Precision of Handler Detection

To evaluate the precision of PromisePatrol, we manually inspected all warnings across the subset of projects where promise usage was detected and static analysis succeeded. We exclude test and specification files from our analysis and only report warnings whose source locations belong to the main application code and not third-party dependencies.

A total of 34 warnings were reported across 7 projects. Each was labeled as a true positive (TP) or false positive (FP) based on whether a missing fulfillment or rejection handler could plausibly lead to an unhandled rejection or silently ignored resolution. Table 4.1 summarizes the classification. Of the 34 warnings, 15 were true positives and 19 were false positives, resulting in an overall precision of approximately 44%. Most false positives were caused by imprecise modeling of property writes, conditional resolution, and APIs that internally use Node.js-style callback conversion (e.g., `util.callbackify`). We provide illustrative case studies in Section 4.4.5 to show how true positives arise in real-world packages and why they are non-trivial to detect.

Table 4.1: Per-project classification of warnings produced by PromisePatrol

Project	**True Positives**	**False Positives**	**Total Warnings**
axios	0	6	6
execa	0	4	4
glob	6	5	11
mkdirp	2	1	3
ts-node	0	1	1
rimraf	7	0	7
yaml	0	2	2
Total	**15**	**19**	**34**

The results show that PromisePatrol is effective at identifying real bugs in projects such as `glob`, `mkdirp`, and `rimraf`. Notably, in `rimraf`, all reported warnings

Table 4.2: Summary of projects with Promise usage

Project	Size (KB)	Total Time (s)	Promise Time (s)	#Promises
axios	4587.5	697.26	2.47	17
execa	1310.7	294.63	0.05	3
glob	7075.8	667.22	5.35	17
mkdirp	256.0	22.37	0.03	10
react-dom	25743.4	4170.64	2.88	4
rimraf	6773.0	617.29	5.13	32
ts-node	1966.1	797.26	0.39	1
yaml	2089.0	564.28	0.03	1

are true positives arising from asynchronous operations where failures were not properly caught.

4.4.2 RQ2: Verifiability of Absence of Handler Bugs

Beyond bug detection, PromisePatrol aims to act as a lightweight verifier for promise-handling correctness. To evaluate this capability, we examined all 88 projects (out of 100) for which the analysis completed successfully and promise usage was detected.

Of the 63 total promises identified across these projects (see Table 4.2), 36 were statically verified to be correctly handled—i.e., they either fulfilled or rejected without side effects or were protected by appropriate handlers. The remaining 27 promises triggered one or more warnings as discussed in RQ1. These results confirm that PromisePatrol can be used not only to flag missing handlers but also to provide positive assurance.

Such verification is valuable in continuous integration settings where assurance of safe promise usage can be checked alongside traditional type and syntax checks.

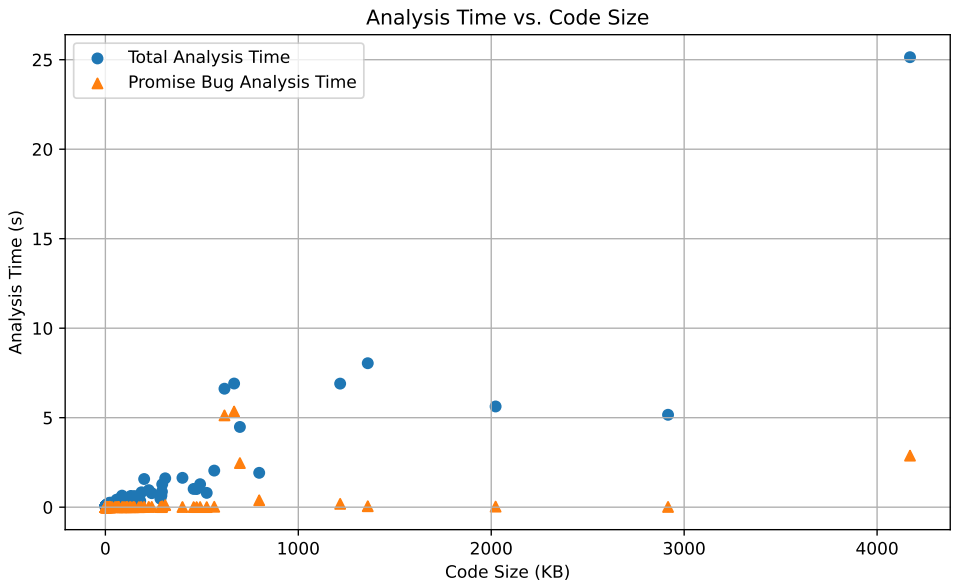


Figure 4.2: Relationship between code size and analysis time for `npm-100` benchmark.

4.4.3 RQ3: Scalability and Performance

To assess the practicality of adopting PromisePatrol in real-world development pipelines, we evaluated its scalability and runtime performance on the complete `npm-100` benchmark suite. Out of 100 projects, PromisePatrol successfully analyzed 88 projects with 10min timeout.

- The average total analysis time per project was 1.15 seconds, with a maximum of 25.14 seconds.
- The average Promise-specific analysis time was just 0.19 seconds, peaking at 5.4 seconds.
- The mean project size was 262,KB, with the largest project, `react-dom`, exceeding 42.0,MB in size.

Table 4.2 provides a breakdown of project sizes, Promise counts, and analysis times. Notably, large projects such as `rimraf` (32 Promises, 5.13s analysis time) and `react-dom` (4 Promises, 2.88s) demonstrate that Promise-specific analysis incurs minimal overhead even in complex codebases.

To evaluate scaling behavior, we plotted code size against both total and Promise-specific analysis times (Figure 4.2). While total analysis time grows approximately linearly with code size, Promise bug analysis time remains consistently low, confirming that PromisePatrol 's targeted checks are lightweight and scalable.

4.4.4 RQ4: Impact of handler necessity heuristics

To improve the practical precision of our analysis without sacrificing soundness, PromisePatrol incorporates a set of lightweight handler necessity heuristics described in Section 4.3.4.7. These heuristics aim to suppress spurious warnings in common programming patterns where missing handlers are semantically harmless or deliberately omitted.

To evaluate their impact, we re-ran the analysis with the heuristics disabled. In this configuration, PromisePatrol produced 73 warnings across the same 8 projects. This is a two-fold increase compared to the 34 warnings reported with heuristics enabled. Manual inspection revealed that the majority of the additional warnings correspond to patterns such as telemetry logging or background cleanup tasks—cases where omission of handlers is either benign or expected by design.

Crucially, the heuristics are purely conservative filters: they only suppress warnings in cases where certain syntactic or semantic indicators strongly suggest that the promise does not need a fulfillment or rejection handler. The results underscore the practical utility of the heuristics: while they do not change the core analysis logic, they effectively reduce false positives without introducing false negatives. As such, they make PromisePatrol suitable for integration into real-world development workflows by minimizing alert fatigue while preserving its ability to detect true issues.

4.4.5 Illustrative Case Studies

To better understand the nature and validity of the warnings reported by PromisePatrol, we manually inspected the Node.js libraries where true positives were found: `mkdirp`, `glob`, and `rimraf`.

mkdirp. In `mkdirp`, a commonly used library for recursive directory creation, our tool reported a warning about a potentially unhandled rejection in a `.then(...)` chain. The rejection handler attached only handles certain error codes (e.g., `ENOENT`, `EEXIST`, `EROFS`), and rethrows others as shown below:

```
opts.mkdirAsync(path, opts).then(() => made || path, async (er) => {
if (er.code === 'ENOENT') {
return mkdirpManual(...);
}
if (er.code !== 'EEXIST' && er.code !== 'EROFS') {
throw er;
}
return opts.statAsync(path).then(...);
});
```

Manual testing confirmed that if the target path is an existing file, the operation fails with `ENOTDIR`, which is rethrown and must be caught by the caller. Thus, the warning reflects a genuine case where rejection is propagated to the top level, with no apparent enclosing handler.

glob. The `glob` library performs filesystem traversal using APIs such as `lstat()` and `realpath()`. Our analysis reported missing rejection handlers in certain parts of code. As filesystem operations can fail under normal conditions (e.g., missing files, permission issues), the analysis conservatively flags these as potentially unhandled. While we did not find documentation clarifying whether such errors are expected to propagate to the caller, their lack of explicit handling makes this a plausible concern.

rimraf. In `rimraf`, an asynchronous file and directory remover, our analysis surfaced both missing fulfillment and rejection handlers. We observed a call to `Promise.all(...)` whose result is used in a boolean reduction, but not explicitly guarded:

```
const removedAll = (await Promise.all(entries.map(...))).reduce((a,b) => a && b, true);
```

If any of the nested deletions reject, the rejection propagates unless caught upstream. Additionally, we observed that `rimraf` suppresses some errors internally (e.g.,

`ENOENT`) but allows others (e.g., `ENOTDIR`) to escape. However, this behavior was not consistently documented.

Across these case studies, we observe that Promise-related error handling in libraries is often partial and inconsistently documented. In some cases, specific error codes are explicitly suppressed (e.g., `ENOENT`), while others are allowed to propagate without clear indication. This lack of documentation places the burden on application developers to infer which rejections must be handled explicitly, leading to unintended behaviors, such as silent failures, incomplete operations, or user-visible errors that surface only under specific runtime conditions. These examples underscore both the practical relevance of our analysis and the difficulty developers face reasoning about error propagation in asynchronous code.

4.4.6 Discussion

Our analysis approach is fundamentally based on sound abstract interpretation principles. However, its overall accuracy and effectiveness are conditioned on the completeness and precision of the underlying call graph computed by Jelly. JavaScript's dynamic features—such as dynamic property access, higher-order functions, reflective constructs, and use of `eval`—introduce inherent uncertainty in static call graph construction. To remain scalable and broadly applicable, Jelly intentionally omits resolution of certain constructs, which can lead to missing call edges. These omissions may result in false negatives where some Promise interactions remain unanalyzed.

On the other hand, our abstraction of Promise semantics—including the handling of `async/await`, chaining via `then/catch`, and propagation through the event loop and microtask queue—is designed to capture common JavaScript Promise behaviors in a pragmatic and broadly applicable manner. While effective for idiomatic code, the abstraction may overapproximate in the presence of framework-specific APIs, native bindings, or implicit Promise construction via utilities like `util.promisify`. In such cases, the analysis can conservatively flag handlers as missing even when they are implicitly managed by the surrounding runtime or framework, leading to false positives.

A key challenge in Promise analysis is that correct usage often hinges on when a handler is attached relative to when a Promise settles. Ideally, one would verify that a fulfillment or rejection handler handler is attached before the corresponding resolution or rejection occurs—a liveness condition that is difficult to enforce statically. Promises may resolve many ticks after their creation, and handlers may be attached in a later tick as well. To make the property analyzable, we conservatively approximate this by requiring that handlers be attached within the same tick as the Promise's creation. This transforms the verification problem into a safety check: before exiting the event loop tick where a Promise is created, a fulfillment or rejection handler must be attached. This shift enables pragmatic static reasoning while still catching a wide range of bugs observed in practice.

Importantly, our definition of Missing Fulfillment and Rejection Handlers is strict and conservative: it flags the absence of handlers on all interprocedural paths from Promise creation until the Promise goes out of scope. This strict definition ensures comprehensive coverage at the expense of potential false positives, particularly in scenarios such as intentionally employed "fire-and-forget" patterns or when analyzing libraries intended to delegate Promise handling responsibility to consuming applications. In this contexts, PromisePatrol 's warnings may be considered false positives from a developer's perspective.

We position PromisePatrol as a conservative static checker that aids developers in reasoning about Promise correctness in complex, asynchronous JavaScript programs. Its goal is not to guarantee correctness in every scenario, but to highlight cases where rejection or fulfillment handling may be absent. By surfacing these potentially risky flows, PromisePatrol supports two key use cases: (i) enabling early detection of latent bugs

stemming from missing handlers, and (ii) prompting developers to document asynchronous contracts, especially in shared libraries. In this way, PromisePatrol complements both safety-focused workflows and robust API design practices.

4.5 Related Work

JS static analysis JavaScript program analysis has been an active field of research due to the language's unique challenges and widespread adoption on both client-side and server-side applications. JavaScript's dynamic and reflective nature makes sound static analysis difficult, particularly as the language evolved with the introduction of Node.js. Static and dynamic analysis frameworks have been developed to improve type inference, call graph construction, and general program understanding. Several of these frameworks have been designed to handle JavaScript's unique challenges using *abstract interpretation* [22]. Prominent among these are TAJS [45], JSAI [47], which use abstract interpretation to infer types in highly dynamic JavaScript code. SAFE and its successor, SAFEWAPI [12, 54], convert JavaScript code into an Intermediate Representation (IR), making it suitable for abstract interpretation. These frameworks have been instrumental in handling JavaScript's complex type system and program behaviors. However, they often encounter issues with scalability when applied to large, real-world applications.

JS call graph construction JavaScript call graph construction has been extensively studied, using static, dynamic, or hybrid analysis methods. Call graph construction for JavaScript applications poses unique challenges due to the language's dynamic features, requiring specialized techniques developed over the past decade [15]. Sridharan et al. [103] and Feldthaus et al. [26] demonstrated that traditional call graph construction algorithms designed for statically-typed languages like Java often produce unsound and incomplete results for JavaScript. Field-based flow analysis was introduced for constructing call graphs, enabling scalable support for JavaScript IDEs [26]. Nielsen et al. [78] scan Node.js applications to construct modular call graphs.

The primary challenge in constructing call graphs for dynamic languages like JavaScript is achieving a balance between precision, recall, and performance. Extensive research

has focused on each of these dimensions, as the ideal balance may vary depending on the downstream use of the call graphs. For example, Utture et al. [115] (and follow-up work [50]) proposed a call-graph pruner to enhance precision by filtering out likely false-positive call edges, balancing precision and recall, though it does not reduce analysis time and lacks predictable, semantics-based outcomes. Chakraborty et al. [15] quantified the impact of different factors on recall reduction, and later introduced a performance-focused call graph analysis that nearly doubles speed by bounding indirection levels, achieving high accuracy with minimal recall loss [14].

Analysis of JavaScript Promise-Based Bugs. The prevalence of event-driven and asynchronous programming in JavaScript makes reasoning about program behavior particularly difficult, especially in the presence of complex control flows and temporal dependencies. Fard et al. [25] demonstrated that asynchronous callbacks constitute a substantial portion of untested JavaScript code, revealing the limitations of conventional testing strategies.

To tackle the unique challenges of Promise-based concurrency, Madsen et al. [58] introduced a foundational formal semantics for a core subset of ECMAScript 6 Promises. This formalism enabled precise reasoning about the behavior of Promises and served as the basis for a novel program artifact called the promise graph. These graphs model the relationships and dependencies between Promises within an application. For example, "dead Promises"—Promises that are never resolved or rejected—can be identified by locating nodes in the graph that lack outgoing edges.

Building on this foundation, Alimadadi et al. [3] developed *PromiseKeeper*, a dynamic analysis tool that constructs Promise graphs from observed execution traces. The tool identifies anti-patterns such as unhandled rejections and misused chaining patterns by analyzing runtime behavior. However, because PromiseKeeper operates purely at runtime, it can only guarantee coverage for code paths exercised during execution.

More recently, Turcotte et al. [113] introduced *DrAsync*, which employs lightweight static analysis using CodeQL to detect Promise-related anti-patterns directly from source

code. Although this approach offers the advantages of static analysis, its scope is largely limited to syntactic patterns and lacks the deeper semantic guarantees provided by formal methods.

Additionally, Arteca et al. [10] proposed *Resynchronizer*, a tool that refactors asynchronous JavaScript code to enhance performance by enabling better parallelism and reducing blocking behavior. While effective for performance optimization, Resynchronizer does not target the sound detection of correctness bugs in Promise usage.

In contrast to prior work, PromisePatrol is the first static analysis framework to explicitly model both missing fulfillment and rejection handlers for JavaScript Promises. Unlike dynamic analyses such as PromiseKeeper, which are limited to observed executions, PromisePatrol applies call-stack-aware abstract interpretation to reason interprocedurally about Promise state transitions. While it does not offer complete soundness guarantees, its design captures several semantic patterns in large codebases making it practically effective for uncovering latent asynchronous defects.

4.6 Conclusion

This work demonstrates the feasibility of static analysis for detecting promise-related programming errors in JavaScript applications. PromisePatrol employs a call-stack-based abstract interpretation with Entry Anchor computation to identify missing promise handlers across function boundaries. Our evaluation on 100 npm packages shows that the approach scales to real-world codebases, detecting 15 genuine bugs with moderate precision (44%).

The analysis successfully handles standard promise usage patterns, including promise chains, async/await constructs, and aggregate operations such as `Promise.all()`. However, several limitations affect completeness: the analysis does not track promises stored in object properties or complex data structures, and it lacks models for implicit promise creation through APIs like `util.promisify` or framework-specific abstractions.

Despite these limitations, the results demonstrate that targeted static analysis can effectively identify promise-related bugs that are difficult to detect through testing

alone. The Entry Anchor technique proves valuable for focusing analysis effort on promise-relevant code regions, enabling scalable analysis of large JavaScript applications. Future work should extend coverage to more complex Promise flows, including Promises stored in object properties, passed through data structures, or created via framework abstractions and higher-order APIs.

Chapter 5

Conclusions

Static analysis of JavaScript presents fundamental challenges due to the language's dynamic and asynchronous nature. Precise call graph construction is a prerequisite for a wide range of interprocedural analyses, yet JavaScript's reflective features, dynamic dispatch mechanisms, and loosely structured object model make this task particularly difficult. As a result, many existing analyses suffer from unsoundness, poor scalability, or excessive imprecision—limiting their effectiveness in real-world software development.

This dissertation addressed these challenges by proposing principled techniques for improving the recall, performance, and applicability of static analyses for JavaScript. First, we introduced a root cause quantification framework to identify and categorize sources of unsoundness in call graphs. This approach provided empirical insight into why call graph algorithms fail and offered guidance for targeted improvements. Second, we presented an indirection-bounded call graph construction technique that significantly improves scalability without compromising recall or precision. This optimization makes it feasible to analyze large-scale JavaScript applications using whole-program static analysis. Third, we developed PromisePatrol, a static analysis framework based on abstract interpretation, designed to detect common promise misuses such as missing fulfillment or rejection handlers. Our evaluation demonstrated that these issues are prevalent in real-world code and difficult to uncover through testing alone.

Together, these contributions advance the state of JavaScript static analysis by improving the soundness and tractability of foundational analysis infrastructure, and by

demonstrating the utility of these improvements in analyzing asynchronous behavior. The techniques introduced in this thesis are practical, modular, and applicable to a wide range of real-world codebases, offering immediate value to both research and developer communities.

This work opens several promising avenues for future research. The root cause quantification technique can be extended to other dynamic languages, such as Python and Ruby, to study call graph unsoundness in the presence of higher-order functions and dynamic dispatch. Insights from this analysis may guide the development of improved call graph builders and more targeted static analyses. The indirection-bounded strategy, shown to be effective in JavaScript, could also be explored in other languages and analysis frameworks, particularly those involving field sensitivity and higher-order constructs. Finally, the Promise analysis framework can be enhanced to handle more complex flows, including Promises embedded in object properties, passed through data structures, or created via framework-specific abstractions. Extending the analysis to asynchronous paradigms beyond Promises—such as event emitters and reactive streams—offers further opportunities to improve static reasoning about concurrency in JavaScript. Together, these directions aim to expand the scope, precision, and applicability of static analysis techniques for dynamic and asynchronous programming environments.

Bibliography

[1] Addy Osmani and Contributors. Todomvc: Helping you select an mv* framework. `https://todomvc.com`, 2011. Accessed: 2025-05-27.

[2] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2-3):79–111, 1999.

[3] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous javascript programs. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–26, 2018.

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

[5] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

[6] 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.

[7] 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.

[8] Gábor Antal, Péter Hegedüs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. Is javascript call graph extraction solved yet? A comparative study of static and dynamic tools. *IEEE Access*, 11:25266–25284, 2023.

[9] Gábor Antal, Zoltán Tóth, Péter Hegedűs, and Rudolf Ferenc. Enhanced bug prediction in javascript programs with hybrid call-graph based invocation metrics. *Technologies*, 9(1):3, 2020.

[10] Ellen Arteca, Frank Tip, and Max Schäfer. Enabling additional parallelism in asynchronous javascript applications. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2021.

[11] Babel Contributors. Babel parser ast format. `https://github.com/babel/babel/blob/main/packages/babel-parser/ast/spec.md`, 2024. Accessed: 2025-05-27.

[12] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. Safewapi: Web api misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 507–517, 2014.

[13] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points-to analysis. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 115–125. ACM, 2003.

[14] Madhurima Chakraborty, Aakash Gnanakumar, Manu Sridharan, and Anders Møller. Indirection-Bounded Call Graph Analysis. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:22, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[15] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[16] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs (Artifact). *Dagstuhl Artifacts Series*, 8(2):7:1–7:5, 2022.

[17] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic root cause quantification for missing edges in javascript call graphs (extended version), 2022.

[18] 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.

[19] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 296–310. ACM, 1990.

[20] Various Contributors. Unhandled promise rejections should be fatal. `https://github.com/nodejs/node/issues/830`, 2015–2020. GitHub issue #830.

[21] 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.

[22] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[23] John-David Dalton and Contributors. Lodash: A modern javascript utility library. `https://lodash.com`, 2012. Accessed: 2025-05-27.

[24] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson R. Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 307–317. ACM, 2017.

[25] Amin Milani Fard and Ali Mesbah. Javascript: The (un) covered parts. In *2017 IEEE international conference on software testing, verification and validation (ICST)*, pages 230–240. IEEE, 2017.

[26] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 752–761. IEEE Computer Society, 2013.

[27] Stephen Fink et al. WALA. `https://github.com/wala/WALA`, 2024.

[28] Martin Fowler. FluentInterface, 2005. Accessed: 2023-09-24.

[29] Mohammad Ganji, Saba Alimadadi, and Frank Tip. Code coverage criteria for asynchronous programs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1307–1319, 2023.

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

[31] GitHub. Codeql for research, 2022. Accessed: 2022-05-13.

[32] GitHub. Codeql library for javascript: Call graph, 2022. Accessed: 2022-05-13.

[33] Google. Angular – a typescript-based web application framework. `https://angular.io`, 2016. Accessed: 2025-05-27.

[34] 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 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 177–187. ACM, 2011.

[35] S. Gude, M. Hafiz, and A. Wirfs-Brock. JavaScript: The used parts. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 466–475, 2014.

[36] 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.

[37] Marijn Haverbeke. Tern: Intelligent javascript tooling. `https://ternjs.net`, 2013. Accessed: 2025-05-27.

[38] 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.

[39] Zoltán Herczeg and Gábor Lóki. Evaluation and comparison of dynamic call graph generators for JavaScript. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2019*, pages 472–479, 2019.

[40] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997.

[41] Isaac Isaacs. mkdirp. `https://www.npmjs.com/package/mkdirp`, 2024. Accessed: 2025-05-26.

[42] 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.

[43] 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.

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

[45] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings 16*, pages 238–255. Springer, 2009.

[46] 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.

[47] 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 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 121–132, 2014.

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

[49] Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. Reducing static analysis unsoundness with approximate interpretation. *Proceedings of the ACM on Programming Languages (PACMPL)*, 4(PLDI):194:1–194:24, 2024.

[50] Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach D Le, and Quyet Thang Huynh. Autopruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 520–532, 2022.

[51] Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach Dinh Le, and Huynh Quyet Thang. AutoPruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 520–532. ACM, 2022.

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

[53] 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.

[54] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*, page 96. Citeseer, 2012.

[55] 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.

[56] Peiming Liu, Yanze Li, Bradley Swain, and Jeff Huang. PUS: A fast and highly efficient solver for inclusion-based pointer analysis. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1781–1792. ACM, 2022.

[57] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[58] Magnus Madsen, Ondřej Lhoták, and Frank Tip. A model for reasoning about javascript promises. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–24, 2017.

[59] 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.

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

[61] Anders Alnor Mathiasen and Andreas Pavlogiannis. The fine-grained and parallel complexity of Andersen's pointer analysis. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[62] MDN Contributors. Promise - javascript — mdn. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`, 2024. Accessed May 2024.

[63] MDN Web Docs. Object.getownpropertydescriptor(), 2021. Accessed: 2021-01-11.

[64] MDN Web Docs. with statement, 2021. Accessed: 2021-01-11.

[65] MDN Web Docs. Defining getters and setters, 2022. Accessed: 2022-05-13.

[66] MDN Web Docs. Function, 2022. Accessed: 2022-05-13.

[67] MDN Web Docs. Object.defineproperty(), 2022. Accessed: 2022-05-13.

[68] MDN Web Docs Contributors. Window: `unhandledrejection` event. `https://developer.mozilla.org/en-US/docs/Web/API/Window/unhandledrejection_event`, 2024. Accessed: 2025-05-27.

[69] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, 2012.

[70] Mark S. Miller et al. The e language: Secure distributed computing. `http://www.erights.org/`, 2006. Accessed 2025-06-09.

[71] 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.

[72] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. Detecting locations in javascript programs affected by breaking library changes. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–25, 2020.

[73] Anders Møller and Oskar Haarklou Veileborg. Jelly. `https://github.com/cs-au-dk/jelly`, 2024.

[74] Mozilla. Rhino: JavaScript in java, 2021. Accessed: 2021-01-11.

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

[76] 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.

[77] 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.

[78] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of node. js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 29–41, 2021.

[79] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of Node.js applications. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, pages 29–41, 2021.

[80] Node.js Contributors. `process: 'unhandledRejection'` Event. `https://nodejs.org/api/process.html#event-unhandledrejection`, 2024. Accessed: 2025-05-27.

[81] Node.js Technical Steering Committee. Node.js v15.0.0 release. `https://nodejs.org/en/blog/release/v15.0.0#throw-on-unhandled-rejections---33021`, 2020. Accessed May 2024.

[82] 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.

[83] Martin Odersky, Philippe Altherr, Vincent Cremet, and Burak Emir. An overview of the scala programming language. In *Technical Report IC/2004/64*. 2004.

[84] Stack Overflow. Stack overflow developer survey 2024, 2024. Accessed: 2024-11-05.

[85] OWASP. Juice shop, 2021. Accessed: 2021-12-01.

[86] 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.

[87] 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.

[88] Changhee Park, Sooncheol Won, Joonho Jin, and Sukyoung Ryu. Static analysis of JavaScript web applications in the wild via practical DOM modeling. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, pages 552–562, 2015.

[89] Fernando Magno Quintão Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*, pages 126–135. IEEE Computer Society, 2009.

[90] Promises/A+ Community. Promises/a+ specification. `https://promisesaplus.com/`, 2013. Accessed 2025-06-09.

[91] Q-Success. Usage statistics of javascript as client-side programming language on websites. `https://w3techs.com/technologies/details/cp-javascript`, 2025. Accessed: 2025-05-27.

[92] React Contributors. React – a javascript library for building user interfaces. `https://react.dev/`, 2024. Accessed: 2025-05-27.

[93] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 251–261. ACM, 2019.

[94] John Resig and The jQuery Team. jquery: The write less, do more, javascript library. `https://jquery.com`, 2006. Accessed: 2025-05-27.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

[99] 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.

[100] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 488–498. ACM, 2013.

[101] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[102] 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.

[103] 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.

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

[105] Steve Sanderson. Knockout.js: A javascript mvvm library. `https://knockoutjs.com`, 2010. Accessed: 2025-05-27.

[106] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1049–1060. ACM, 2020.

[107] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 265–266. ACM, 2016.

[108] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for Node.js. In Christophe Dubach and Jingling Xue, editors, *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, pages 196–206. ACM, 2018.

[109] Haiyang Sun, Daniele Bonetta, Filippo Schiavio, and Walter Binder. Reasoning about the node. js event loop using async graphs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 61–72. IEEE, 2019.

[110] Kwangwon Sun and Sukyoung Ryu. Analysis of JavaScript programs: Challenges and research trends. *ACM Comput. Surv.*, 50(4):59:1–59:34, 2017.

[111] Node.js Team. Node.js v15.0.0 release notes. `https://nodejs.medium.com/node-js-v15-0-0-is-here-deb00750f278`, 2020. Accessed May 2024.

[112] The Chromium Authors. Puppeteer – a node.js library for controlling headless browsers. `https://pptr.dev/`, 2025. Accessed: 2025-05-27.

[113] Alexi Turcotte, Michael D Shah, Mark W Aldrich, and Frank Tip. Drasync: identifying and visualizing anti-patterns in asynchronous javascript. In *Proceedings of the 44th International Conference on Software Engineering*, pages 774–785, 2022.

[114] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. Striking a balance: Pruning false-positives from static call graphs. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2043–2055. ACM, 2022.

[115] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. Striking a balance: pruning false-positives from static call graphs. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2043–2055, 2022.

[116] Akshay Utture and Jens Palsberg. Fast and precise application code analysis using a partial library. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 934–945. ACM, 2022.

[117] Shiyi Wei and Barbara G. Ryder. A practical blended analysis for dynamic features in JavaScript. Technical Report TR-12-18, Virginia Tech, 2012.

[118] Shiyi Wei and Barbara G Ryder. A practical blended analysis for dynamic features in javascript. Technical report, Department of Computer Science, Virginia Polytechnic Institute & State . . . , 2012.

[119] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for JavaScript. In *International Symposium on Software Testing and Analysis*, ISSTA, pages 336–346, 2013.

[120] 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.

[121] 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 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 487–498, 2016.

[122] Nicholas C. Zakas and the ESLint Team. Eslint: Pluggable javascript linter. `https://eslint.org`, 2013. Accessed: 2025-05-27.

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