Indirection-Bounded Call Graph Analysis

Madhurima Chakraborty
University of California, Riverside

Aakash Gnanakumar
University of California, Riverside

Manu Sridharan
University of California, Riverside

Anders Møller
Aarhus University

Abstract

Call graphs play a crucial role in analyzing the structure and behavior of programs. For JavaScript and other dynamically typed programming languages, static call graph analysis relies on approximating the possible flow of functions and objects, and producing usable call graphs for large, real-world programs remains challenging.

In this paper, 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.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases JavaScript, call graphs, points-to analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.27

Supplementary Material *Software*: `https://zenodo.org/doi/10.5281/zenodo.12720724`

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

1 Introduction

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 [7, 20, 18, 3, 29]. 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 [15]. 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 [2]. Objects are typically modeled using allocation-site abstraction [20, 6], or more coarsely using field-based analysis [7, 8]. Functions are tracked using variations of control-flow analysis [17]. In general, the analyses can be expressed using conditional subset constraint systems, which are solved using cubic-time algorithms [21, 25].

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 [21], that directly follows the levels of indirections, as we explain in detail in Section 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 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 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. [7]. 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 by the recent work of Mathiasen and Pavlogiannis [16] 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 operations 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.

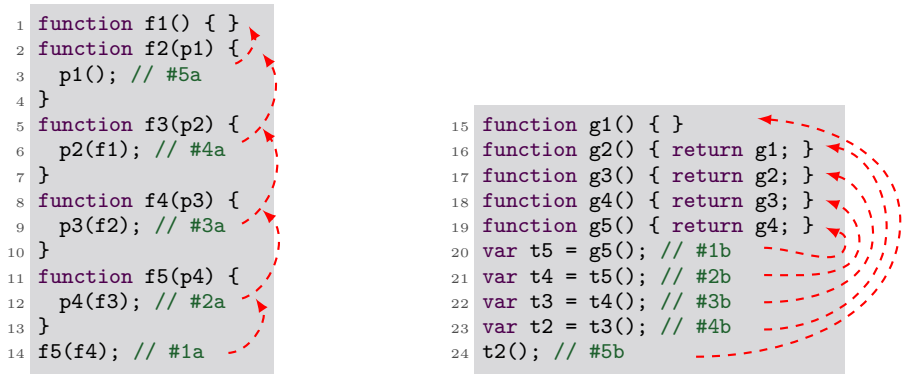
In summary, the contributions of this paper are:

- We propose the use of *indirection-bounded* analysis (Section 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) 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 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).

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 1, practical analyses are not fully sound.) Different use cases have motivated different metrics in the literature [5]. With the *call site targets* metric [7], 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



(a) Function arguments **(b)** Function return values

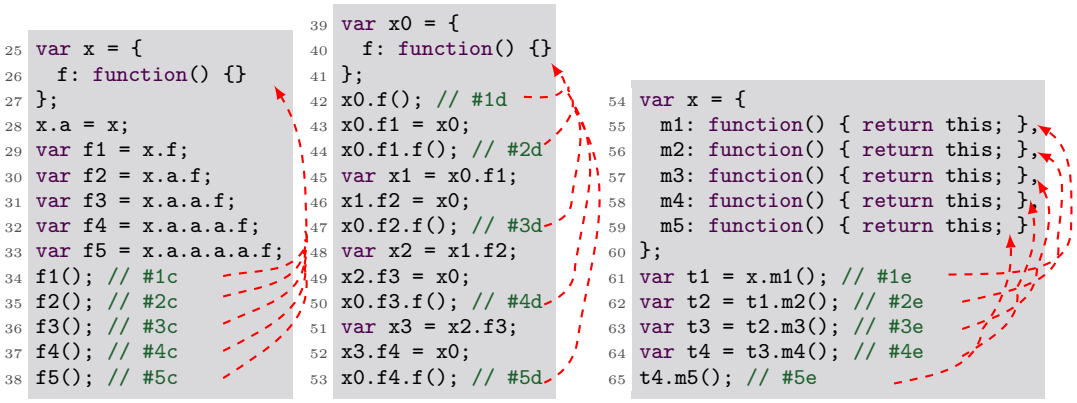
Figure 1 Example programs that illustrate indirection levels for function calls.

call sites that are semantically reachable. A variant is the *call edge sets* metric [30], 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 [26] and the *reachable edges* metric [10] instead compare the sets of functions or call edges, respectively, that are reachable from the program entry points (e.g., application modules).

Figures 1 and 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 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 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 in Section 4, analysis algorithms like wave propagation [21], ACG [7], or 0-CFA [24] 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 2a, lines 25–28 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 33. 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 2b shows an example where call site `#5d` has 5 levels 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



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

Figure 2 Example programs that illustrate indirection levels for object field accesses.

established by the preceding lines.[1]

Finally, Figure 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 [9]), which suggests that method calls should be treated in the same way as field reads in indirection-bounded analysis.

3 Dynamic Indirection Bound Estimation

To confirm the intuitions from Section 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.

3.1 Language

The first column of Table 1 defines the types of canonical statements in a core language. (The constraint rules in the second column will be explained in Section 4.1.) The statement types are standard for a flow- and context-insensitive Andersen-style points-to analysis [2] 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

[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.

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

Table 1 Statement types for our analysis and the corresponding static analysis constraint rules (discussed in Section 4).

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, 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.

3.2 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 1 gives pseudocode for our dynamic analysis. We assume the analysis is implemented via an interface similar to that provided by frameworks like Jalangi [23], where a callback provided by the analysis is invoked before and possibly after the execution of each program statement. In Algorithm 1, the callback is the HandleStmt procedure. For all statement types, we assume HandleStmt is invoked before the statement s executes,

Algorithm 1 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
```

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 1 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 $V[\mathtt{y}, f]$.

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), $V[\mathtt{x}, i]$ 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`, $V[\mathtt{x}, v]$ is set to be the minimum of its current value and $V[\mathtt{y}, v]$ (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 1a in Section 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 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 2a and 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 1 since our core language (Table 1) contains only function calls (with no receiver argument), not method calls.

Algorithm 1 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 1 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.

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

Table 2 Number of call edges with each minimum bound across all benchmarks, for configurations with SELECTIVE enabled and disabled, respectively.

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 Node.js dynamic analysis [28]. 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 1 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 $V[\texttt{y}, v]$. Then, when handling the subsequent write of v to `x`, it uses $S[v]$ instead of $V[\texttt{y}, v]$ for the bound update (line 10 in Algorithm 1).

Benchmarks and methodology We created a suite of 74 Node.js benchmarks for our study, as no standard benchmark suite was available. These were randomly selected among the top 1,000 highest ranked JavaScript projects on GitHub which both had unit tests available and that worked correctly 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.

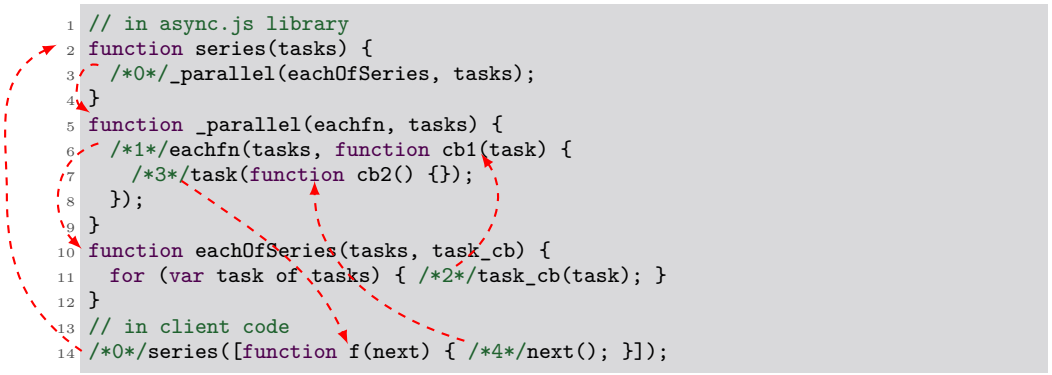
Results Table 2 gives the results from our study. We present results for two configurations. The first is our preferred configuration, with SELECTIVE enabled, so reads and method calls are treated as unbounded. The second row gives results when all indirections are bounded. Each column shows how many 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 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 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 2, and are challenging to extract due to their complexity.



```
// in async.js library
function series(tasks) {
  /*0*/_parallel(eachOfSeries, tasks);
}
function _parallel(eachfn, tasks) {
  /*1*/eachfn(tasks, function cb1(task) {
    /*3*/task(function cb2() {});
  });
}
function eachOfSeries(tasks, task_cb) {
  for (var task of tasks) { /*2*/task_cb(task); }
}
// in client code
/*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 2 show, calls like these requiring bound 4 or greater are quite rare across our benchmarks (only 0.6% of call edges).

In contrast, with SELECTIVE disabled, natural code patterns can lead to high bounds, as discussed in Section 2. For example, consider the following code from the `express` benchmark:

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

This code uses a fluent interface [9]: 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.

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 [21] (Section 4.1), and then present a simplified version for solving the constraints (Section 4.2). In Section 4.3 we extend the algorithm with indirection bounding. Finally, in Section 4.4 we show how further simplifications yield a bounded version of the ACG algorithm of Feldthaus et al. [7]. The static analyses presented in this section work independently of the dynamic analysis presented in Section 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.

4.1 Analysis Formulation

The second column of Table 1 gives constraint rules for computing an Andersen-style points-to analysis for our statement types. The rules are standard; see [25] 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 3.2), 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 [1]—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 $pt(\mathtt{y})$, 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 2 and 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.

4.2 Simplified Wave Propagation

Algorithm 2 presents pseudocode for a simplified version of the wave propagation algorithm [21], 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 2 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 $n \to n'$ in G represents a subset constraint $pt(n) \subseteq pt(n')$ from Table 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 Table 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 $n \to n'$ 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 Table 1. 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 work [21].

4.3 Adding Bounds

Given the structure of the wave propagation algorithm, adding bounding of all indirections is straightforward. Algorithm 2 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

Algorithm 2 Simplified wave propagation algorithm.

```
1:  P: program to analyze
2:  pt: points-to relation, initially empty
3:  G: flow graph, initially empty
4:  procedure Analyze()
5:      Init()
6:      repeat
7:          pt' ← pt, G' ← G
8:          Propagate()
9:          AddEdges()
10:     until pt' = pt ∧ G' = G
11: end procedure
12: procedure Init()
13:     for each x = {}_i ∈ P do
14:         pt(x) ← pt(x) ∪ {o_i}
15:     for each x = p =>_i {...} ∈ P do
16:         pt(x) ← pt(x) ∪ {f_i}
17:     for each x = y ∈ P do
18:         G ← G ∪ {y → x}
19:     for each return_i x ∈ P do
20:         G ← G ∪ {x → ret_i}
21: end procedure
22: procedure Propagate()
23:     repeat
24:         pt' ← pt
25:         for each edge n → n' in G do
26:             pt(n') ← pt(n') ∪ pt(n)
27:     until pt' = pt
28: end procedure
29: procedure AddEdges()
30:     for each x = y.f ∈ P, o_i ∈ pt(y) do
31:         G ← G ∪ {o_i.f → x}
32:     for each x.f = y ∈ P, o_i ∈ pt(x) do
33:         G ← G ∪ {y → o_i.f}
34:     for each x = y(z) ∈ P, f_i ∈ pt(y) do
35:         G ← G ∪ {z → p_i, ret_i → x}
36: end procedure
```

have found that in real implementations that use the wave propagation structure, adding bounding is similarly straightforward.

Recall that Section 3.3 showed that field reads often require higher bounds than other indirections, matching the intuition of Section 2. Algorithm 3 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 2 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

Algorithm 3 Algorithm 2 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
```

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 3 discovers the call with $bound = 1$, as desired. In our implementation, constraints from JavaScript method calls (see Figure 2c in Section 2) are also handled in an unbounded manner, similar to handling of field reads in Algorithm 3.

4.4 Bounded ACG

The ACG algorithm of Feldthaus et al. [7] 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 Table 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 4; 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 2. 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 [24]. Bounding is also simplified compared to Algorithm 3, as field reads do not require any special treatment.

Algorithm 4 Bounded ACG algorithm [7], as a modified version of Algorithm 2.

```
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
```

5 Evaluation

We have implemented the techniques of Section 4 in two different analysis frameworks. The first, Jelly [19, 12], implements a field-sensitive call graph analysis using an algorithm like wave propagation [21], and is targeted at Node.js programs. The second, WALA [8], 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 4.3 and 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 3.2) impact the overall effectiveness of the analysis?

5.1 Benchmarks

For Node.js benchmarks, we further filtered the benchmarks used in the dynamic study (Section 3.3) 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.

This filtering led to a set of 25 benchmarks, whose details are given in Table 3. The benchmarks are large, with thousands of functions and ranging up to more than 9.8MB of code. For each benchmark, the table gives the numbers of packages, modules and functions, the code size, and the analysis time, precision, and recall for Jelly when run without bounding.

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

For assessing WALA, we constructed a suite of 14 web and mobile benchmarks, shown in Table 4. We included the 10 programs from the TodoMVC suite that were used by Chakraborty et al. in their study [4], 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.

5.2 Experimental Configuration

In our experiments we measure precision and recall using the call site targets metric described in Section 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

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

Table 4 Web and Mobile benchmarks used for experiments with WALA.

also experimented with the reachable functions and reachable edges metrics (see Section 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.

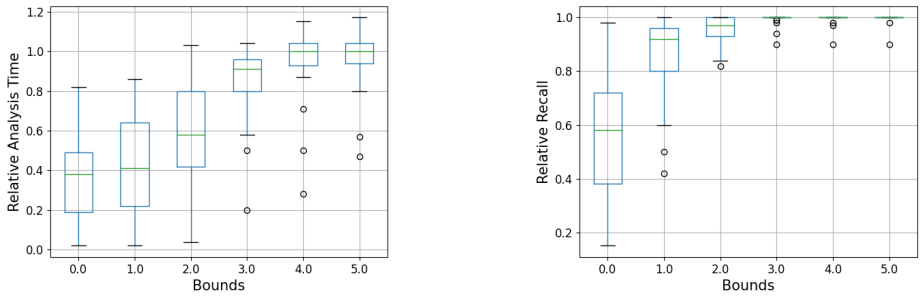
We ran the Jelly experiments on a machine with an 8-core Intel Core i7-11700 processor and 32GB of RAM, running Ubuntu 20.04.6 LTS. For the WALA experiments we used a Google Cloud virtual machine with a 4-core Intel Broadwell Xeon CPU and 64GB RAM running Ubuntu 20.04.6 LTS.

5.3 Results

Figure 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 2 and 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 [21] 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.

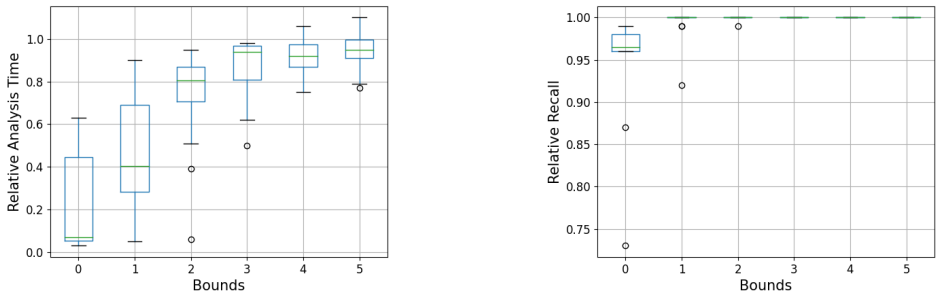
Studying Figures 3a and 3b, indirection bound 2 gives the best balance of analysis time improvement and recall. The average analysis time speed-up is roughly 2X (ranging from 0.9X–23.9X), while the average relative recall is 95% (82%–100%) of the unbounded analysis. The high relative recall matches the results of our dynamic study (Section 3.3), 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 4 gives results for our WALA experiments. Here, we see that bound 1 yields a



(a) Analysis time **(b)** Recall

Figure 3 Analysis time and recall for JELLY with different indirection bounds and SELECTIVE enabled, relative to JELLY's unbounded analysis.

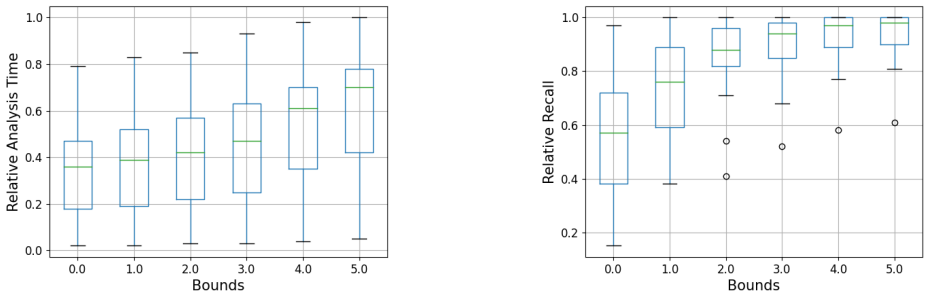


(a) Analysis time **(b)** Recall

Figure 4 Analysis time and recall for WALA with different indirection bounds, relative to WALA's unbounded analysis.

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 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 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 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, with this configuration there is a long tail of calls that require a much higher bound to discover; in Figure 5b, even at bound 5, there is one benchmark with relative recall below 80% and five below 90%. Given these considerations, and the naturalness of code patterns with high numbers of field read and method call indirections (Sections 2 and 3.3), we believe analysis with selective bounding is the better choice for more



(a) Analysis time **(b)** Recall

Figure 5 Relative analysis time and recall for JELLY with reads and methods calls also bounded (SELECTIVE disabled).

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.

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). 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.

6 Related Work

The most closely related work is the recent study by Mathiasen and Pavlogiannis [16] 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 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 [27] and also constitutes the core of the PUS constraint solver [14].

Mathiasen and Pavlogiannis [16] 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 [11] 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 [31] 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. [30] (and follow-up work [13]) 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 [25]. 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 1 it is well known that practically all whole-program static analyzers have imperfect recall [15], but the analysis results are still useful for many use cases. Reif et al. [22] and Sui et al. [26] investigated this phenomenon empirically for state-of-the-art analyzers for Java, and Antal et al. [3] have made a similar study for JavaScript call graph analysis tools. The more recent work by Chakraborty et al. [4] introduced a method for quantifying the root causes of missing edges in call graphs produced by a field-based static analysis for JavaScript [7]. The dynamic analysis used by Chakraborty et al. inspired the technique presented in Section 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 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 at `https://zenodo.org/doi/10.5281/zenodo.12720724` contains the benchmarks used in the experimental evaluation and instructions for using indirection-bounded analysis with the open source analysis tools Jelly and WALA.

References

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

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

3 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. `doi:10.1109/ACCESS.2023.3255984`.

4 Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic root cause quantification for missing edges in JavaScript call graphs. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 3:1–3:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.ECOOP.2022.3`.

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

6 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. `doi:10.1145/93542.93585`.

7 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. `doi:10.1109/ICSE.2013.6606621`.

8 Stephen Fink et al. WALA. `https://github.com/wala/WALA`, 2024.

9 Martin Fowler. FluentInterface. `https://www.martinfowler.com/bliki/FluentInterface.html`, 2005. Accessed: 2023-09-24.

10 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. `doi:10.1145/2001420.2001442`.

11 Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997. `doi:10.1145/239912.239913`.

12 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.

13 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. doi:10.1145/3540250.3549175.

14 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. doi:10.1145/3510003.3510075.

15 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. doi:10.1145/2644805.

16 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. doi:10.1145/3434315.

17 Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, 2012. doi:10.1145/2187671.2187672.

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

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

20 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. doi:10.1145/3460319.3464836.

21 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. doi:10.1109/CGO.2009.9.

22 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. doi:10.1145/3293882.3330555.

23 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. doi:10.1145/2491411.2491447.

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

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

26 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. doi:10.1145/3377811.3380441.

27 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. doi:10.1145/2892208.2892235.

28 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. doi:10.1145/3178372.3179527.

29 Kwangwon Sun and Sukyoung Ryu. Analysis of JavaScript programs: Challenges and research trends. *ACM Comput. Surv.*, 50(4):59:1–59:34, 2017. doi:10.1145/3106741.

30 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. doi:10.1145/3510003.3510166.

31 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. doi:10.1145/3510003.3510046.