Efficient Construction of Approximate Call Graphs for JavaScript IDE Services

Asger Feldthaus*, Max Schäfer†**, Manu Sridharan‡, Julian Dolby‡, and Frank Tip§**

*Aarhus University, Denmark
asf@cs.au.dk

†Nanyang Technological University, Singapore
schaefer@ntu.edu.sg

‡IBM T.J. Watson Research Center, USA
{msridhar,dolby}@us.ibm.com

§University of Waterloo, Canada
ftip@uwaterloo.ca

***Abstract*—The rapid rise of JavaScript as one of the most popular programming languages of the present day has led to a demand for sophisticated IDE support similar to what is available for Java or C#. However, advanced tooling is hampered by the dynamic nature of the language, which makes any form of static analysis very difficult. We single out efficient call graph construction as a key problem to be solved in order to improve development tools for JavaScript. To address this problem, we present a scalable field-based flow analysis for constructing call graphs. Our evaluation on large real-world programs shows that the analysis, while in principle unsound, produces highly accurate call graphs in practice. Previous analyses do not scale to these programs, but our analysis handles them in a matter of seconds, thus proving its suitability for use in an interactive setting.**

I. Introduction

Over the past decade, JavaScript has turned from a niche language for animating HTML pages into an immensely popular language for application development in many different domains. Besides being the enabling technology for Web 2.0 applications such as Google Mail, it is becoming a popular choice for server-side programming with Node.js, for writing cross-platform mobile apps with frameworks like PhoneGap, and even for implementing desktop applications.

This increase in popularity has led to a demand for modern integrated development environments (IDEs) for JavaScript providing smart code editors, software maintenance tools such as automated refactorings, and code analysis tools. While a variety of mature IDEs exist for languages like Java and C#, such tools have only just begun to appear for JavaScript: existing IDEs such as Eclipse, Komodo IDE and NetBeans are starting to support JavaScript, while new IDEs specifically targeted at web programming such as WebStorm and Cloud9 are also gaining traction.

Compared with their Java counterparts, however, these IDEs are fairly bare-bones. Code navigation and completion use heuristics that sometimes fail unexpectedly, while refactoring and analysis is all but unsupported. There has been some work on more principled tools and analyses [7, 9, 11, 14], but these approaches do not yet scale to real-world applications.

A key impediment to advanced tooling for JavaScript is the difficulty of building *call graphs*, that is, determining the functions that a given call may invoke at runtime. Such reasoning is required in IDEs both for basic features like “Jump to Declaration”, and also for refactoring or analysis tools that need to reason interprocedurally.

**This paper was written while the authors were with IBM Research.

To be useful in an IDE, a call graph construction algorithm must be *lightweight* and *scalable*: modern programmers expect IDE services to be constantly available, so it should be possible to quickly compute call graph information on demand, even for large, framework-based JavaScript applications.

Demands on *precision* and *soundness* may vary between clients: for Jump to Declaration, listing many spurious call targets is perhaps even worse than occasionally missing one, while analysis tools may prefer a fairly conservative call graph. Achieving absolute soundness is, however, almost impossible in JavaScript due to widespread use of `eval` and dynamic code loading through the DOM. A useful compromise could be an unsound call graph construction algorithm that can, to some extent, quantify its degree of unsoundness, for instance by indicating call sites where some callees may be missing; the IDE can then pass this information on to the programmer.

In Java, call graphs can be efficiently constructed using class hierarchy analysis [6], which uses type information to build a call graph. However, since JavaScript is dynamically typed and uses prototype-based inheritance, neither class hierarchy analysis nor its more refined variants that keep track of class instantiations [5, 20] are directly applicable. Additionally, these analyses cannot easily handle first-class functions.

An alternative are flow analyses such as CFA [17] or Andersen’s points-to analysis [4] that statically approximate the flow of data (including first-class functions) to reason about function calls. While typically not fast enough for interactive use, such analyses could still be used for non-interactive clients like analysis tools. However, while state-of-the-art flow analyses for JavaScript can handle small framework-based applications [19], they do not yet scale to larger programs.

In this work, we present a lightweight flow analysis specifically designed to efficiently compute approximate call graphs for real-world JavaScript programs. Its main properties are:

1) The analysis is *field-based* [13, 15], meaning that it uses a single abstract location per property name. Thus, two

functions that are assigned to properties of the same name will become indistinguishable as call targets.
2) It *only tracks function objects* and does not reason about any non-functional values.
3) It *ignores dynamic property accesses*, i.e., property reads and writes using JavaScript's bracket syntax.

These design decisions significantly reduce the number of abstract locations and objects, thus dramatically improving scalability. While precision could, in principle, suffer, we show in our evaluation that this does not happen in practice.

Clearly, ignoring dynamic property reads and writes makes the analysis unsound, but this is a consequence of the first two design decisions: since we only track function objects, we cannot reason about the possible string values of `p` in a dynamic property access `e[p]`, and using a field-based approach means that imprecision cannot be contained by reasoning about aliasing. However, previous work has shown that many dynamic property accesses are *correlated* [19], i.e., they copy the value from property p in one object to property p of another. With a field-based approach, such a copy is a no-op, since the analysis uses a single representation for all p properties anyway. Our evaluation indicates that in practice, very few call targets are missed due unsoundness.

Like any flow analysis, our analysis faces a chicken-and-egg problem: to propagate (abstract) argument and return values between caller and callee we need a call graph, yet a call graph is what we are trying to compute. We explore two analysis variants that tackle this problem in different ways.

The first is a standard optimistic analysis in the terminology of Grove and Chambers [8] that starts out with an empty call graph, which is gradually extended as new flows are discovered until a fixpoint is reached.

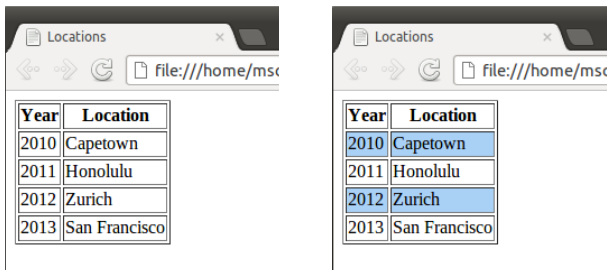
The second variant is a pessimistic analysis that does not reason about interprocedural flow at all and simply gives up on call sites whose call target may depend on such flow, except in cases where the callee can be determined purely locally.

We have implemented both of our techniques and performed an extensive empirical evaluation on ten large, real-world JavaScript web applications, many of them based on popular frameworks. To show the feasibility of using our analyses in an IDE, the implementation operates on abstract syntax trees (ASTs) as IDE-based tools normally do, rather than on an intermediate code representation as is typical for flow analyses.

Both analyses scale very well and are able to build call graphs for even fairly large programs in a few seconds. As expected, the pessimistic analysis is faster than the optimistic one, since it does not need to iterate to a fixpoint.

To evaluate precision and scalability, we compared our analysis results to dynamic call graphs that we obtained by manually exercising instrumented versions of our subject programs. The optimistic analysis achieves high precision ($\geq 66\%$) and very high recall ($\geq 85\%$) with respect to these dynamic call graphs, but what is perhaps surprising is that the pessimistic analysis does just as well.

This suggests that in many cases the pessimistic analysis may be preferable: not only is it faster, but it also clearly



Year	Location
2010	Capetown
2011	Honolulu
2012	Zurich
2013	San Francisco

(a) (b)

Fig. 1. Test page before (a), and after (b) applying the plugin

indicates which call sites it cannot reason about precisely, whereas the optimistic analysis gives a result for every call site that may sometimes be very imprecise.

Finally, we evaluate several possible client applications of our algorithms: we show that the call graphs they generate are much more complete than what current IDEs offer and could be used to significantly improve the "Jump to Declaration" feature. Moreover, our call graphs also facilitate the implementation of bug finding and smell detection tools that check for common programming mistakes.

Contributions: To the best of our knowledge, this paper presents the first static analysis capable of computing useful call graphs for large JavaScript applications. Specifically, we make the following contributions:

- We propose two variants of a field-based flow analysis for JavaScript that only tracks function objects and ignores dynamic property reads and writes.
- We show that both scale to large, real-world programs.
- Comparing against dynamic call graphs, we find that the analyses, while in principle unsound, produce accurate call graphs in practice.
- We demonstrate several client applications to show the usefulness of our approach.

The remainder of the paper is organized as follows. Section II motivates our techniques on a simple but realistic JavaScript program. Section III explains our analyses in detail, and Section IV evaluates them on real-world applications. Section V surveys related work, and Section VI concludes.

II. Motivating Example

As an example to introduce our techniques, we present a simple plugin to the popular jQuery framework that can be used to highlight alternating rows of HTML tables for easier on-screen reading as shown in Fig. 1. We discuss some challenges for call graph construction illustrated by this example, and explain how our analysis handles them.

A. Example Walkthrough

The jQuery framework provides a wide variety of functionality to simplify cross-browser application development. Most of its features are exposed through the global `jQuery` function, which can be used to register event handlers, parse snippets of HTML, or perform CSS queries over the DOM.

```
 1 (function() {
 2  function jQuery(n) {
 3   var res = Object.create(jQuery.fn);
 4   var elts = document.getElementsByTagName(n);
 5   for(var i=0;i<elts.length;++i)
 6    res[i] = elts[i];
 7   res.length = elts.length;
 8   return res;
 9  }
10 
11  jQuery.fn = {
12   extend: function ext(obj) {
13    for(var p in obj)
14     jQuery.fn[p] = obj[p];
15   }
16  };
17 
18  jQuery.fn.extend({
19   each: function(cb) {
20    for(var i=0;i<this.length;++i)
21     cb(this[i], i);
22   }
23  });
24 
25  window.jQuery = jQuery;
26 })();
```

Fig. 2. A small subset of jQuery

The result of such a query is a special jQuery result object, which provides array-like access to the result elements through numerical indices and offers many utility methods, some of them defined by jQuery itself, and others defined by plugins.

Our simplified version of jQuery, shown in Fig. 2, implements a `jQuery` function as well. Following a common pattern, it is first defined as a local function within a surrounding closure (lines 2–9), and later stored in a global variable to make it accessible to client code (line 25). Our `jQuery` function only provides a very simple form of querying: when passed a string argument, it finds all DOM elements with this tag name (line 4), stores them into the result object, sets its `length` property to indicate how many elements were found, and returns it. For instance, `jQuery('tbody')` returns all table body elements in the document.

The result object itself is created on line 3 using the built-in function `Object.create`, which takes as its argument an object p and returns a new object o that has p as its prototype. In this case, the prototype object will be `jQuery.fn`, which is defined on line 11. Thus, any property defined on `jQuery.fn` is available on all jQuery result objects via JavaScript's prototype-based inheritance mechanism.

Initially, the `jQuery.fn` object contains a single property: a method `extend` that adds all property-value pairs of its argument object `obj` to `jQuery.fn`. This is done through a `for-in` loop (lines 13–14) that iterates over all properties p of `obj`, and uses dynamic property reads and writes to copy the value of property p on `obj` into a property of the same name on `jQuery.fn`. If no such property exists yet, it will be created; otherwise, its previous value will be overwritten.

```
27 (function($) {
28  $.fn.highlightAlt = function(c) {
29   this.each(function(elt) {
30    for(var i=1;i<elt.children.length;i+=2)
31     elt.children[i].style.backgroundColor = c;
32   });
33  };
34 
35  window.highlightAltRows = function() {
36   $('tbody').highlightAlt('#A9D0F5');
37  };
38 })(jQuery);
```

Fig. 3. A jQuery plugin to highlight alternating children of DOM elements

On line 18, the `extend` method is used to add a method `each` to `jQuery.fn`, which iterates over all elements contained in a result object and invokes the given callback function `cb` on it, passing both the element and its index as arguments.

The plugin, shown in Fig. 3, uses the `each` method, passing it a callback that in turn iterates over all the children of every result element, and sets the background color of every second element to a given color `c` (line 31). This functionality is exposed as a method `highlightAlt` added to the `jQuery.fn` object, and hence available on every jQuery result object. The plugin also defines a global function `highlightAltRows` that clients can invoke to apply highlighting to all tables in the document: it uses the `jQuery` function to find all table bodies, and then invokes `highlightAlt` on each of them. Notice that a closure is used to make the global `jQuery` variable available as a local variable `$`.

Our example illustrates several important features of JavaScript: variables have no static types and may, in general, hold values of different types over the course of program execution. Objects in JavaScript do not have a fixed set of properties; instead, properties can be created simply by assigning to them (e.g., the plugin adds a method `highlightAlt` to `jQuery.fn`), and can even be deleted (not shown in the example). Functions are first-class objects that can be passed as arguments (as with the `each` function), stored in object properties to serve as methods, and even have properties themselves. Finally, dynamic property reads and writes allow accessing properties by computed names.

B. *Challenges for Call Graph Construction*

As discussed in Section I, call graphs are widely useful in IDEs, for example to implement "Jump to Declaration" or to perform lightweight analysis tasks. Unfortunately, neither standard coarse approaches nor more precise flow analyses work well for building JavaScript call graphs, as we shall explain using our running example.

Java IDEs take a type-based approach to call-graph construction [6]: the possible targets of a method call are simply those admitted by the program's class hierarchy. Since variables and properties are not statically typed in JavaScript, type-based call graph construction algorithms are not immediately applicable. While prototype objects are superficially similar to Java classes, properties can be dynamically added or overwritten. For instance, the `jQuery.fn` object in our example starts out with only one property (`extend`) to which two others (`each` and `highlightAlt`) are later added, defeating any simple static type inference. Type inference algorithms for JavaScript that can handle such complications have been proposed [12, 14], but do not yet scale to real-world programs.

An very naïve way to construct call graphs would be to use name matching, and resolve a call `e.f(...)` to all functions named `f`. This approach, which is used by Eclipse JSDT, fails when functions are passed as parameters or stored in properties with a different name, like the `extend` function on line 12. Consequently, JSDT is unable to resolve any of the call sites in our example. Other IDEs employ more sophisticated techniques, but we do not know of any current IDE that can handle callbacks and discover targets for the call on line 21.

A more conservative approach suggesting any function with the right number of parameters as a call target would likely be too imprecise in practice, yet still fails to be sound, since JavaScript allows arity mismatching: the call on line 21 passes two parameters, while the callback only declares one.

A flow analysis, like an Andersen-style pointer analysis [19] or an inter-procedural data flow analysis [14], can avoid these issues. Such analyses work by tracing the flow of abstract values through abstract memory locations based on the relevant program statements (primarily assignments and function calls). A call graph is then derived by determining which function values flow to each invoked expression.

However, building a precise flow analysis that scales to large JavaScript programs is an unsolved challenge. In the example of Fig. 2, the flow of functions to invocations is non-trivial, due to the use of the `extend` function to modify the `jQuery.fn` object. Precise modeling of dynamic property accesses like those in `extend` and other complex constructs is required to obtain a useful flow analysis result, but this precise modeling can compromise scalability; see [19] for a detailed discussion. In particular, we know of no JavaScript flow analysis that can analyze real-world jQuery-based application.[1]

C. Our Approach

In this paper, we show that a simple flow analysis suffices to construct approximate call graphs that are, in practice, sufficiently accurate for applications such as IDE services. Our analysis only tracks the flow of function values, unlike most previous flow analyses, which track the flow of all objects. Ignoring general object flow implies that for a property access `e.f`, the analysis cannot reason about which particular (abstract) object's `f` property is accessed. Instead, a *field-based* approach is employed, in which `e.f` is modeled as accessing a single global location `f`, ignoring the base expression `e`.

Our analysis uses a standard flow graph capturing assignments of functions into variables and properties, and of one variable into another. For instance, the function declaration on line 2 adds a flow graph edge from the declared function to the local variable `jQuery`, while the assignment on line 25 adds an edge from that variable to the abstract location **Prop**(`jQuery`) representing all properties named `jQuery`. The function call on line 38, in turn, establishes a flow from **Prop**(`jQuery`) into the parameter `$`, leading the analysis to conclude that the call on line 36 may indeed invoke the `jQuery` function. Details of how to construct the flow graph and how to extract a call graph from it are presented in the next section.

[1]The analysis in [19] could only analyze a manually rewritten version of jQuery with handling of certain JavaScript features disabled.

At first glance, dynamic property accesses present a formidable obstacle to this approach: for a dynamic property access `e[p]`, the analysis cannot reason about which names `p` can evaluate to, since string values are not tracked. A conservative approximation would treat such accesses as possibly reading or writing any possible property, leading to hopelessly imprecise analysis results. However, we observe that dynamic property accesses in practice often occur as correlated accesses [19], where the read and the write refer to the same property, as on line 14 in our example. A field-based analysis can safely ignore correlated accesses, since like-named properties are merged anyway. Our analysis goes further and simply ignores *all* dynamic property accesses.

This choice compromises soundness, as seen in this example (inspired by code in jQuery):

```
arr = ["Width","Height"];
for (var i=0;i<arr.length;++i)
 $.fn["outer"+arr[i]] = function() { ... };
$.fn.outerWidth();
```

The dynamic property write inside the loop corresponds to two static property writes to `$.fn.outerWidth` and `$.fn.outerHeight`, which the analysis ignores; hence it is unable to resolve the call to `outerWidth`.

But, as we shall show in our evaluation (Section IV), such cases have little effect on soundness in practice. Furthermore, unlike more precise flow analyses, our approach scales easily to large programs, which makes it well suited for use in an IDE, where a small degree of unsoundness can be tolerated.

III. Analysis Formulation

We now present the details of our call graph construction algorithm. We first explain the intraprocedural parts of the analysis, and then present two contrasting approaches to handling interprocedural flows, one pessimistic and one optimistic.

A. Intraprocedural Flow

Our algorithm operates over a *flow graph*, a representation of the possible data flow induced by program statements. The vertices of the flow graph represent functions, variables and properties, while the edges represent assignments. To emphasize the suitability of our techniques for an IDE, we show how to construct the flow graph directly from an AST, as is done in our implementation.

Abstracting from a concrete AST representation, we write Π for the set of all AST positions, and use the notation t^π to mean a program element t (such as an expression, a function declaration, or a variable declaration) at position $\pi \in \Pi$.

	node at π	edges added when visiting π
(R1)	`l = r`	$V(r) \to V(l)$, $V(r) \to \mathbf{Exp}(\pi)$
(R2)	`l || r`	$V(l) \to \mathbf{Exp}(\pi)$, $V(r) \to \mathbf{Exp}(\pi)$
(R3)	`t ? l : r`	$V(l) \to \mathbf{Exp}(\pi)$, $V(r) \to \mathbf{Exp}(\pi)$
(R4)	`l && r`	$V(r) \to \mathbf{Exp}(\pi)$
(R5)	$\{\overline{\texttt{f: e}}\}$	$V(e_i) \to \mathbf{Prop}(f_i)$
(R6)	function expression	$\mathbf{Fun}(\pi) \to \mathbf{Exp}(\pi)$, if it has a name: $\mathbf{Fun}(\pi) \to \mathbf{Var}(\pi)$
(R7)	function declaration	$\mathbf{Fun}(\pi) \to \mathbf{Var}(\pi)$

Fig. 4. Intraprocedural flow graph edges generated for AST nodes

We assume a lookup function λ for local variables such that $\lambda(\pi, x)$ for a position π and a name x returns the position of the local variable or parameter declaration (if any) that x binds to at position π. For any position π, $\phi(\pi)$ denotes the position of the innermost enclosing function (excluding π itself).

There are four basic types of vertices:

$$\begin{array}{lcll} V & ::= & \mathbf{Exp}(\pi) & \text{value of expression at } \pi \\ & \mid & \mathbf{Var}(\pi) & \text{variable declared at } \pi \\ & \mid & \mathbf{Prop}(f) & \text{property of name } f \\ & \mid & \mathbf{Fun}(\pi) & \text{function declaration/expression at } \pi \end{array}$$

We define a function V that maps expressions to corresponding flow graph vertices:

$$V(t^\pi) = \begin{cases} \mathbf{Var}(\pi') & \text{if } t \equiv x \text{ and } \lambda(\pi, x) = \pi' \\ \mathbf{Prop}(x) & \text{if } t \equiv x \text{ and } \lambda(\pi, x) \text{ undefined} \\ \mathbf{Prop}(f) & \text{if } t \equiv e.f \\ \mathbf{Exp}(\pi) & \text{otherwise} \end{cases}$$

To build the flow graph, we traverse the AST and add edges as specified by the rules in Fig. 4.[2] For our example, by rule (R7) the declaration of `jQuery` on line 2 yields an edge **Fun**(2) → **Var**(jQuery), where we use line numbers as positions and refer to local variables by name for readability. Likewise, the function expression on line 12 yields two edges **Fun**(12) → **Var**(ext) and **Fun**(12) → **Exp**(12) by (R6). Some of the other edges generated for our example are shown as solid arrows in the partial flow graph in Fig. 5.

B. Interprocedural Flow

To handle interprocedural flow, the set of vertices needs to be extended as follows:

$$\begin{array}{lcll} V & ::= & \ldots & \\ & \mid & \mathbf{Callee}(\pi) & \text{callee of call at } \pi \\ & \mid & \mathbf{Arg}(\pi, i) & i\text{th argument of call at } \pi \\ & \mid & \mathbf{Parm}(\pi, i) & i\text{th parameter of function at } \pi \\ & \mid & \mathbf{Ret}(\pi) & \text{return value of function at } \pi \\ & \mid & \mathbf{Res}(\pi) & \text{result of call at } \pi \end{array}$$

The function V mapping expressions to vertices is likewise extended: if $\lambda(\pi, x)$ is the ith parameter of the function declared at π', then $V(x^\pi) = \mathbf{Parm}(\pi', i)$, and $V(\texttt{this}^\pi) = \mathbf{Parm}(\phi(\pi), 0)$, i.e., `this` is considered to be the 0th parameter. Rules for connecting **Arg** and **Ret** vertices with **Exp** vertices are given in Fig. 6.

[2](R4) is somewhat subtle: in JavaScript, the result of `l && r` can only be `l` if `l` evaluates to a false value, but in this case it is not a function, and thus does not have to be tracked.

Returning to our example, the function call on lines 27–38, yields, by rule (R8), an edge **Prop**(jQuery) → **Arg**(38,1). This edge, and some of the other edges that are generated by the rules of Fig. 6 are shown as dotted arrows in Fig. 5.

We now introduce two approaches for connecting **Parm** to **Arg** and **Ret** to **Res** vertices to track interprocedural flow.

Algorithm 1 PESSIMISTIC CALL GRAPH CONSTRUCTION

Output: call graph C, escaping functions E, unresolved call sites U

```
1: $C_i := \{(\pi, \pi') \mid t^\pi$ is a one-shot call to a function $f^{\pi'}\}$
2: $E_i := \{\pi' \mid \neg\exists\pi.(\pi, \pi') \in C_i\}$
3: $U_i := \{\pi \mid \neg\exists\pi'.(\pi, \pi') \in C_i\}$
4: $G := \emptyset$
5: ADD INTERPROCEDURAL EDGES$(G, C_i, E_i, U_i)$
6: add edges from Fig. 4 and 6
7: $C := \{(\pi, \pi') \mid \mathbf{Fun}(\pi') \overset{\text{opt}}{\rightsquigarrow}_G \mathbf{Callee}(\pi)\}$
8: $E := \{\pi \mid \mathbf{Fun}(\pi) \rightsquigarrow_G \mathbf{Unknown}\}$
9: $U := \{\pi \mid \mathbf{Unknown} \rightsquigarrow_G \mathbf{Callee}(\pi)\}$
```

C. Pessimistic Approach

The pessimistic call graph construction algorithm (Alg. 1) only tracks interprocedural flow in the important special case of *one-shot calls*, i.e., calls of the form

$$\texttt{(function(}\overline{x}\texttt{) \{ ... \})(}\overline{e}\texttt{)}$$

where an anonymous function (the *one-shot* closure) is directly applied to some arguments. In all other cases, interprocedural flow is modeled using a special **Unknown** vertex.

We start call graph construction from an initial call graph C_i that only contains edges from one-shot calls to one-shot

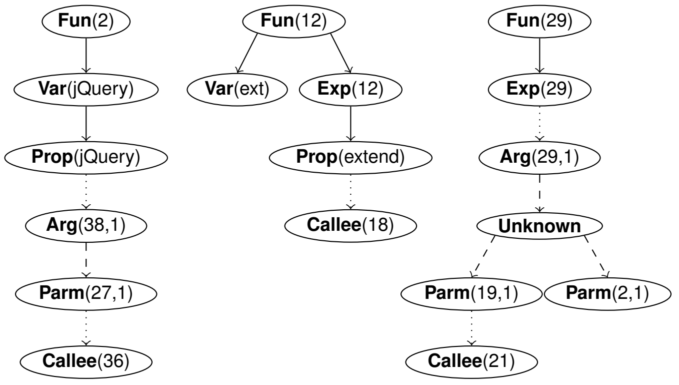


Fig. 5. Partial flow graph for Fig. 2 and 3. Solid edges are added by the rules of Fig. 4, dotted edges by the rules of Fig. 6, and dashed edges by Alg. 2.

	node at π	edges added when visiting π
(R8)	$f(\overline{e})$ or `new` $f(\overline{e})$	$V(f) \to \mathbf{Callee}(\pi)$, $V(e_i) \to \mathbf{Arg}(\pi, i)$, $\mathbf{Res}(\pi) \to \mathbf{Exp}(\pi)$
(R9)	$r.p(\overline{e})$	as (R8), plus $V(r) \to \mathbf{Arg}(\pi, 0)$
(R10)	`return` e	$V(e) \to \mathbf{Ret}(\phi(\pi))$

Fig. 6. Flow graph edges generated for calls and returns

closures. All other functions are considered *escaping functions* (set E_i), and all other call *unresolved call sites* (set U_i). The flow graph G is initially empty.

Algorithm 2 ADD INTERPROCEDURAL EDGES

Input: flow graph G, initial call graph C, escaping functions E, unresolved call sites U

```
1: for all $(\pi, \pi') \in C$ do
2:     add edges $\mathbf{Arg}(\pi, i) \to \mathbf{Parm}(\pi', i)$ to $G$
3:     add edge $\mathbf{Ret}(\pi') \to \mathbf{Res}(\pi)$ to $G$
4: for all $\pi \in U$ do
5:     add edges $\mathbf{Arg}(\pi, i) \to \mathbf{Unknown}$ to $G$
6:     add edge $\mathbf{Unknown} \to \mathbf{Res}(\pi)$ to $G$
7: for all $\pi' \in E$ do
8:     add edges $\mathbf{Unknown} \to \mathbf{Parm}(\pi', i)$ to $G$
9:     add edge $\mathbf{Ret}(\pi') \to \mathbf{Unknown}$ to $G$
```

Now we add interprocedural edges to G as described in Alg. 2: **Arg** vertices are connected to **Parm** vertices along the edges in C_i, and similar for **Ret** and **Res**, thus modeling parameters and return values. Argument values at unresolved call sites flow into **Unknown**, and from there into every parameter of every escaping function. Conversely, the return value of every escaping function flows into **Unknown**, and from there into the result vertex of every unresolved call site.

In Fig. 5, this step adds the dashed edges. Note that $\mathbf{Arg}(38, 1)$ is connected to $\mathbf{Parm}(27, 1)$, precisely modeling the one-shot call at line 38, whereas $\mathbf{Arg}(29, 1)$ is conservatively connected to **Unknown**, since this call site is unresolved.

Intraprocedural edges are now added as per Fig. 4 and 6.

To extract the final call graph, we need to compute the transitive closure of G to determine all function vertices $\mathbf{Fun}(\pi)$ from which a call site π' is reachable. However, if we consider flows through **Unknown**, the resulting call graph will be very imprecise. Instead, we want to produce a call graph that gives reasonably precise call targets for many call sites, and marks sites for which no precise information is available.

Writing $\rightsquigarrow_G$ for the transitive closure of G, and $\overset{\text{opt}}{\rightsquigarrow}_G$ for the *optimistic* transitive closure which does not consider paths through **Unknown**, we define the call graph C, the set E of escaping functions, and the set U of unresolved call sites: a call may invoke any function that directly flows into its callee vertex without going through **Unknown**; if **Unknown** flows into a site, then that site is unresolved and the information in C may not be complete; and if a function flows into **Unknown**, it may be invoked at call sites not mentioned in C.

In the partial flow graph in Fig. 5, we can see that $\mathbf{Fun}(12) \overset{\text{opt}}{\rightsquigarrow}_G \mathbf{Callee}(18)$, so the call at line 18 may invoke the function at line 12, and likewise $\mathbf{Fun}(2) \overset{\text{opt}}{\rightsquigarrow}_G \mathbf{Callee}(36)$. However, $\mathbf{Fun}(29) \overset{\text{opt}}{\not\rightsquigarrow}_G \mathbf{Callee}(21)$, and since there are no other flows into $\mathbf{Callee}(21)$, the pessimistic call graph does not provide a call target for this call.

D. Optimistic Approach

The pessimistic approach produces a call graph triple (C, E, U) from an initial triple (C_i, E_i, U_i), which could be done repeatedly. This is what the optimistic approach does, but instead of starting from a conservative triple that considers all calls unresolved and all functions escaped unless they are one-shot, the optimistic approach starts with the empty triple $(\emptyset, \emptyset, \emptyset)$. The flow graph is built using the same rules as for the pessimistic approach and a new triple is extracted in the same way, but then the whole procedure is repeated until a fixpoint is reached.

For our example, this leads to a more complete call graph; in particular, the optimistic approach can show that the call on line 21 may invoke the function passed on line 29.

E. Discussion

One would expect the pessimistic approach to be more efficient but less precise than the optimistic approach, and past work on call graph construction for other languages supports this conclusion [8]. As we will show in our evaluation, however, the loss in precision is actually fairly minor in practice, hence the pessimistic approach may be preferable for some applications.

Many call graph algorithms only produce call graphs for code that is deemed reachable from a given set of entry points, which can improve precision, particularly for optimistic call graph construction. We choose not to do so for two main reasons. Firstly, we want our algorithms to be usable in an IDE while developing a program; at this point, some code may not yet have been integrated with the rest of the program and hence appear to be dead, but a programmer would still expect IDE services to be available for this code.

Secondly, reasoning about reachability requires a fairly elaborate model of the JavaScript standard library and the DOM: for instance, event handlers should always be considered reachable, and reflective invocations using `call` and `apply` must also be accounted for. By analyzing *all* code instead, we can make do with a very coarse model that simply lists all known methods defined in the standard library and the DOM. For a standard library function such as `Array.prototype.sort`, we then simply introduce a new vertex $\mathbf{Builtin}(\texttt{Array\_sort})$ with an edge to $\mathbf{Prop}(\texttt{sort})$.

IV. EVALUATION

We have implemented both the pessimistic and the optimistic call graph algorithm in CoffeeScript,[3] a dialect of JavaScript. In this section, we evaluate our implementation with respect to the following three evaluation criteria:

(EC1) How scalable are our techniques?
(EC2) How accurate are the computed call graphs?
(EC3) Are our techniques suitable for building IDE services?

To evaluate these criteria, we run both our algorithms on ten real-world subject programs and measure their performance.

[3]http://coffeescript.org/

TABLE I
SUBJECT PROGRAMS

Program	Underlying Framework	LOC	Num. of Functions	Num. of Calls	Dyn. CG Coverage
3dmodel	none	4880	29	109	55.17%
beslimed	MooTools	4750	703	2017	86.05%
coolclock	jQuery	6899	548	1747	81.25%
flotr	Prototype	4946	743	2671	68.98%
fullcalendar	jQuery	12265	1089	4083	70.83%
htmledit	jQuery	3606	389	1253	62.00%
markitup	jQuery	6471	557	1849	71.43%
pacman	none	3513	152	485	79.61%
pdfjs	none	31694	965	3570	67.77%
pong	jQuery	3646	375	1324	75.00%

To measure accuracy, we compare the resulting static call graphs against dynamic call graphs obtained by manually exercising the programs. Finally, we informally compare our analyses with existing IDEs, and report on experiments with two client analyses implemented on top of our call graphs.[4]

A. Subject Programs

Table I lists our subjects, which are ten medium to large browser-based JavaScript applications covering a number of different domains, including games (*beslimed*, *pacman*, *pong*), visualizations (*3dmodel*, *coolclock*), editors (*htmledit*, *markitup*), a presentation library (*flotr*), a calendar app (*fullcalendar*), and a PDF viewer (*pdfjs*). As shown in the table, all but three of them rely on frameworks; these frameworks are the three most widely-used ones according to a recent survey [22], which found that 56% of all surveyed websites used jQuery, 5% used MooTools, and 4% used Prototype.

In many cases, the framework libraries included in the subject programs were in minified form. To aid debugging, we replaced these by their unminified development versions, which also more closely matches the development setting in which we envision our techniques to be used. Since minifiers typically do not rename properties, however, our analyses should not be significantly less precise for minified code.

For each program, we list three size measures: the number of non-blank, non-comment lines of code as determined by the `cloc` utility, as well as the number of functions and of call sites. The coverage number in the last column will be explained below. For the framework-based subjects, framework code contributes between 66% and 94% of code size.

B. Scalability (EC1)

To evaluate scalability, we measured the time it takes to build call graphs for our subject programs using both of our algorithms. As JavaScript's built-in time measurement functions turned out to be unreliable, we used the standard UNIX `time` command, measuring user time. This includes both time for parsing and for the analysis, so we separately measured the time it takes just to parse every program.

The results of these measurements are given in Fig. 7. All experiments were performed on a Lenovo ThinkPad W520 with an Intel Core i7-2720QM CPU and 8GB RAM, using version 3.1.8.22 of the V8 JavaScript engine running on

[4]Our experimental data is available online at http://tinyurl.com/jscallgraphs.

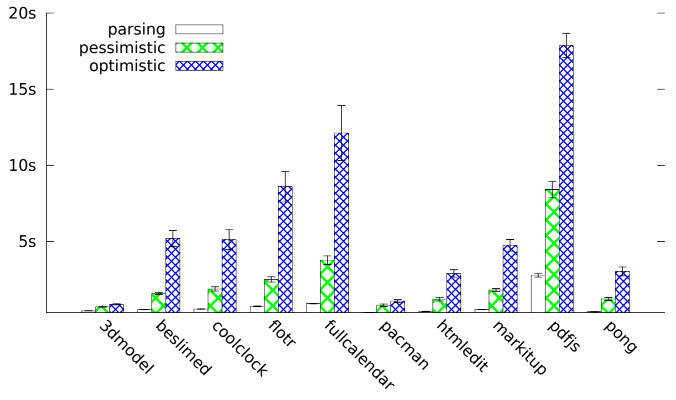


Fig. 7. Time measurements for parsing and analysis; averaged over ten runs, error bars indicate standard deviation

Linux 3.0.0-24 (64-bit version). Timings are averaged over ten runs, with error bars indicating standard deviation.

Both analyses scale very well, with even the largest program analyzed in less than 18 seconds using the optimistic, and less than nine seconds using the pessimistic approach (including about three seconds of parsing time in both cases). The pessimistic analysis in particular already seems fast enough for use in an IDE, where an AST would already be available.

C. Call Graph Accuracy (EC2)

Measuring the accuracy of our call graphs is not easy, since there is no existing analysis that can handle all our subject programs against which to compare our results. Instead we compare against dynamic call graphs and measure precision and recall with respect to dynamically observed call targets.

To obtain dynamic call graphs, we instrumented our subject programs to record the observed call targets for every call that is encountered at runtime, and manually exercised these instrumented versions. Additionally, we measured the function coverage achieved this way, i.e., the percentage of non-framework functions that were executed while recording the call graphs, which is shown in the last column of Table I. In all cases but one, coverage is above 60%, indicating that the dynamic call graphs are based on a reasonable portion of the code and hence likely to be fairly complete. We manually investigated the low coverage on *3dmodel* and found that most of the uncovered code does in fact seem to be dead.

Next, we used our two analyses to generate call graphs for all our subject programs, and computed precision and recall for every call site that is covered by the dynamic call graph. Writing D for the set of targets of a given call site in the dynamic call graph, and S for the set of targets determined by the analysis, the precision is computed as $\frac{|D \cap S|}{|S|}$ (i.e., the percentage of "true" call targets among all targets), while recall is $\frac{|D \cap S|}{|D|}$ (i.e., the percentage of correctly identified true targets). Averaging over all call sites for a given program, we obtain the results in Fig. 8.

Both analyses achieve almost the same precision on most programs, with the pessimistic analysis performing slightly

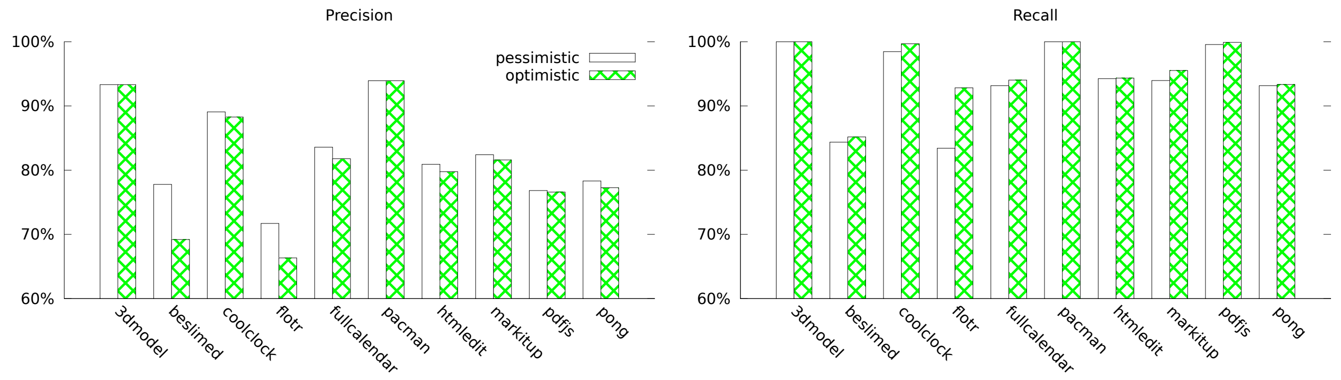


Fig. 8. Precision and recall measurements for optimistic and pessimistic call graphs compared to dynamic call graphs

better. Only on *beslimed* and *flotr*, the two non-jQuery programs, is the difference more marked, and we only achieve a relatively modest precision of between 65% and 75%, while on the others the precision is at least 80%.

For both approaches, the main sources of imprecision are functions that are stored in properties of the same name, which a field-based analysis cannot distinguish as call targets. Additionally, the optimistic approach may resolve callback invocations imprecisely. The pessimistic approach would give up on such call sites, returning zero call targets, which accounts for its better precision measure.

Both analyses achieve very high recall: in every case, more than 80% of dynamically observed call targets are also found by the analysis, with recall above 90% for the jQuery-based programs and close to 100% for the framework-less programs. Missing call targets are due to the unsoundness of our approach with respect to dynamic property writes. These are often used in frameworks to define a group of closely related functions or to do metaprogramming, which is rare in non-framework code. On *flotr*, the optimistic analysis does significantly better than the pessimistic one; this seems to be due to a liberal use of callback functions, which are not handled by the pessimistic analysis.

D. Suitability for IDE Services (EC3)

We now evaluate the suitability of our analyses for three typical client applications.

Jump to Declaration: Java IDEs typically offer a "Jump to Declaration" feature for navigating from a field or method reference to its declaration. In JavaScript, there are no method declarations as such, but several JavaScript IDEs offer a similar feature to navigate from a function call to the function (or, in general, functions) that may be invoked at this place.

Our call graph algorithms could be used to implement such a feature. The pessimistic algorithm seems to be particularly well-suited, since it gives a small set of targets for most call sites. While no call target may be available for unresolved call sites, this is arguably better than listing many spurious targets.

To test this hypothesis, we measure the percentage of call sites with a single target, excluding native functions. The results, along with the percentage of call sites with zero, two,

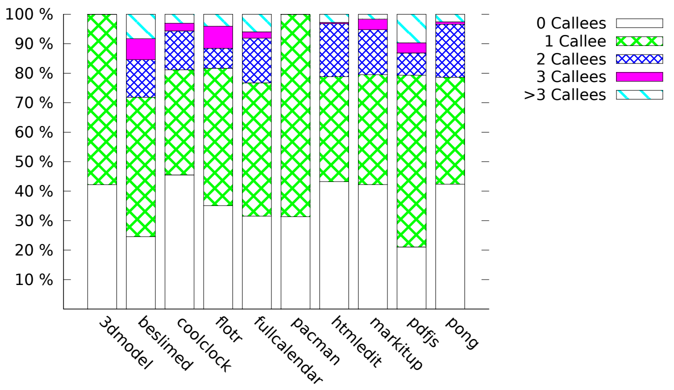


Fig. 9. Number of non-native call targets per site with pessimistic analysis

three, and more than three targets, are given in Fig. 9: on all benchmarks, more than 70% of call sites have at most one target, 80% have at most two and 90% at most three targets. This suggests that the pessimistic algorithm could be useful for implementing Jump to Declaration.

The relatively large percentage of call sites without targets is due to excluding native call targets. If they are included, the pessimistic analysis is on average able to find at least one callee for more than 95% of calls. The maximum number of non-native call targets is 20 callees for a small number of sites on *beslimed*; if native targets are considered, several call sites can have up to 124 callees: these are calls to `toString`, with 120 of the suggested callees being DOM methods.

We now compare our approach against three current JavaScript IDEs: Eclipse JSDT, Komodo IDE, and WebStorm.

The Eclipse JSDT plugin (we tested version 1.4.0 on Eclipse 4.2.0) provides a Jump to Declaration feature, which does not seem to handle method calls, severely limiting its practical usefulness: across all our subject programs, it can only find targets for about 130 call sites (less than 1%).

Komodo IDE (version 7.0.2) uses fairly intricate heuristics to resolve function and method calls that works well on our smaller subject programs such as *3dmodel*. However, it seems unable to handle larger, framework-based programs, where its Jump to Declaration feature usually fails.

WebStorm (version 4.0.2) is closed-source, precluding examination of its implementation. It seems to maintain a representation of variable and property assignments similar to our flow graph. No transitive closure is computed, hence Jump to Declaration only jumps to the most recent definition and it may take several jumps to find the actual callee. WebStorm has built-in support for the most popular frameworks, so it can understand commonly used metaprogramming patterns that foil our analyses. However, it performs no interprocedural reasoning at all (not even for one-shot closures), so it is impossible to jump to the declaration of a callback function.

Smell detection: As an example of a more complicated client, we implemented a simple tool for detecting global variables that are used like local variables, suggesting a missing `var` declaration. While this may not necessarily be a bug, it is considered bad practice and makes code less robust.

We check whether all functions using a global variable x definitely assign to x before reading it. Additionally, call graph information is used to check whether one function using x can call another. If so, the functions might see each others' updates to x, indicating that it may not be possible to turn x into a local variable without changing program behavior.

With pessimistic call graphs, the tool suggests 37 missing `var` declarations on our subject programs. One of these is a false positive due to unsoundness, but in all 36 other cases the global variable could indeed be turned into a local. With optimistic call graphs, there are only 24 true positives and the same false positive. Without interprocedural analysis, the number of false positives rises to nine: in all eight new cases, the global variable is a flag that is indeed modified by a callee and hence cannot be made local, highlighting the importance of interprocedural reasoning for this analysis.

Bug finding: We implemented a tool that looks for functions that are sometimes called using `new`, but as normal functions at other times. While there are functions that can be used either way, this is often indicative of a bug.

Using pessimistic call graphs, the tool reports 14 such functions. One of these is a true positive indicating a bug in *flotr*, four are true but harmless positives in jQuery, and nine are false positives due to imprecise call graph information. Using optimistic call graphs, the number of false positives increases to 16, with no additional true positives.

E. Summary and Threats to Validity

Our evaluation shows that both call graph construction algorithms scale very well. Even though our current implementation is written in CoffeeScript and does not use highly optimized data structures, it is able to build call graphs for substantial applications in a few seconds. The faster pessimistic algorithm may be more suitable for IDE use, but further optimizations to the optimistic algorithm are certainly possible.

Comparing against dynamic call graphs, we found that the vast majority of dynamically observed call targets are predicted correctly by our analyses, and on average the number of spurious call targets is low. Our analyses resolve most call sites to at most one callee (up to 90% on some programs), and compute no more than three possible targets for almost all sites. The only extreme outliers are calls to `toString`, which have more than 100 callees due to our field-based approach.

An informal comparison of our analyses with existing IDEs suggests that the pessimistic analysis outperforms most of them, while the optimistic analysis can handle cases that exceed the capabilities of all surveyed tools. We also discussed two examples of analysis tools that need call graphs. While these tools could be useful for developers, they did not find many bugs on our subject programs, which seem quite mature.

Finally, we discuss threats to the validity of our evaluation.

First, our subject programs may not be representative of other JavaScript code. We only consider browser-based applications, so it is possible that our results do not carry over to other kinds of JavaScript programs. Most of our subject programs use jQuery, with only two programs using other frameworks. We have shown that our approaches perform particularly well on jQuery-based and framework-less applications, and slightly less so on other frameworks. On the other hand, recent data [22] suggests that less than 20% of websites use a framework other than jQuery, so our approach should be applicable to most real-world, browser-based JavaScript code.

Second, our accuracy measurements are relative to an incomplete dynamic call graph, not a sound static call graph. Hence the recall should be understood as an upper bound (i.e., recall on a more complete call graph could be lower), whereas precision is a lower bound (i.e., precision could be higher). Given the difficulty of scaling sound call graph algorithms to realistic programs, dynamic call graphs are the best data we can compare ourselves against at the moment. Moreover, the relatively high function coverage of the dynamic call graphs suggests that they are representative of the entire programs.

V. Related Work

Existing flow analyses for JavaScript [7, 9, 10, 14, 19, 21] generally do not scale to framework-based programs. Some of them, such as Gatekeeper [9], do not reason statically about dynamic property accesses, just like our analysis. Gatekeeper recovers soundness, however, by performing additional runtime instrumentation. All these systems track general object flow and use a more precise heap model than we do.

Recently, Madsen et al. presented an analysis that sidesteps the problem of analyzing complex framework code and modeling native APIs by inferring their behavior from uses in client code [16]. Our approach is scalable enough to directly analyze frameworks, and since we only track functions and do not reason about reachability, no elaborate models for native code seem to be necessary. We could, however, adopt their approach in cases where such modeling becomes important.

Wei and Ryder [23] propose a combined static-dynamic taint analysis of JavaScript programs. In their work, a number of traces is collected that contain information about method calls and object creation. This information is used to assist a static taint analysis with the construction of a call graph that includes code that is executed as the result of calls to `eval`, and excludes code in uncovered branches. In addition, the number of arguments supplied to methods calls is captured and used to counter some of the loss of precision due to function variadicity, by creating multiple distinct nodes in the call graph for certain methods. Like ours, their analysis is unsound, but it is likely to be less scalable than ours because of its reliance on a traditional static pointer analysis. An in-depth comparison of cost and accuracy of the two approaches is future work.

Agesen et al. presented a number of type inference techniques [1–3] for Self, a language with many similarities to JavaScript. They essentially compute highly context-sensitive flow graphs (from which call graphs could be extracted) to statically prove the absence of "message not understood" errors, where a method is not found on the receiver object or its prototypes. Our technique cannot do such reasoning, since it does not track the flow of most objects. Tracking general object flow for JavaScript leads to scalability and precision issues due to heavy usage of reflective idioms that seem not to be as frequently used in Self.

Grove and Chambers [8] present a general framework for call-graph construction algorithms. Our analysis does not fit directly in their framework since they do not discuss prototype-based languages, but roughly speaking, our analysis can be viewed as a variant of 0-CFA [17] where (1) only function values are tracked, (2) field accesses are treated as accesses to correspondingly-named global variables, and (3) all code is assumed to be reachable. Our key contribution is in showing that such an analysis works well for JavaScript in practice. Previous work has studied the effectiveness of field-based flow analysis for C [13] and Java [15, 18]. They exploit static type information to distinguish identically named fields of different struct/class types, which is impossible in JavaScript.

VI. Conclusions

We have presented a fast, practical flow analysis-based approach to call graph construction for JavaScript. Our analysis (i) is field-based, i.e., identically named properties of different objects are not distinguished; (ii) only tracks function values, ignoring the flow of other objects; and (iii) ignores dynamic property reads and writes. We have proposed two variants of this analysis: a pessimistic variant that makes conservative assumptions about interprocedural flow, and an optimistic variant that iteratively builds an interprocedural flow graph.

Both analyses scale extremely well and can handle far larger programs than any other static analysis for JavaScript that we are aware of. While unsound in theory, they produce fairly complete call graphs in practice. These properties make our approach well-suited for use in an IDE.

In such a setting, it would be wasteful to build a call graph from scratch every time it is needed, since large parts of the program typically remain unchanged. Instead, flow graphs could be precomputed and cached on a per-file basis, and then combined into a graph for the whole program when needed.

As future work, we plan to apply our approach in other settings besides IDEs, such as taint analysis [11]. Here, soundness is much more important, so we need to handle dynamic property accesses. Conservatively treating them as potentially accessing all properties will in general result in too much imprecision, so some form of string analysis for reasoning about property names is likely needed. Introducing this and other features (such as tracking of non-function objects) into our analysis while still keeping it scalable is an interesting challenge, which could provide valuable insights into the cost and benefit of different analysis features for JavaScript.

References

[1] O. Agesen. Constraint-Based Type Inference and Parametric Polymorphism. In *SAS*, pages 78–100, 1994.

[2] O. Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *ECOOP*, 1995.

[3] O. Agesen and D. Ungar. Sifting out the Gold: Delivering Compact Applications from an Exploratory Object-Oriented Programming Environment. In *OOPSLA*, 1994.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[5] D. Bacon and P. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA*, 1996.

[6] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP*, August 1995.

[7] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported Refactoring for JavaScript. In *OOPSLA*, 2011.

[8] D. Grove and C. Chambers. A Framework for Call Graph Construction Algorithms. *TOPLAS*, 23(6), 2001.

[9] S. Guarnieri and V. B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, 2009.

[10] S. Guarnieri and V. B. Livshits. Gulfstream: Incremental Static Analysis for Streaming JavaScript Applications. In *WebApps*, 2010.

[11] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the World Wide Web from Vulnerable JavaScript. In *ISSTA*, 2011.

[12] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *ESOP*, 2011.

[13] N. Heintze and O. Tardieu. Ultra-fast Aliasing Analysis Using CLA: A Million Lines of C Code in a Second. In *PLDI*, 2001.

[14] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *SAS*, 2009.

[15] O. Lhoták and L. Hendren. Scaling Java Points-to Analysis Using Spark. In *CC*, April 2003.

[16] M. Madsen, B. Livshits, and M. Fanning. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. MSR TR 2012-66, Microsoft Research, 2012.

[17] O. Shivers. Control Flow Analysis in Scheme. In *PLDI*, 1988.

[18] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-Driven Points-To Analysis for Java. In *OOPSLA*, 2005.

[19] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP*, 2012.

[20] F. Tip and J. Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. In *OOPSLA*, pages 281–293, 2000.

[21] D. Vardoulakis and O. Shivers. CFA2: A Context-Free Approach to Control-Flow Analysis. In *ESOP*, 2010.

[22] W^3 Techs. Usage of JavaScript Libraries for Websites. http://w3techs.com/technologies/overview/javascript_library/all, February 2013.

[23] S. Wei and B. G. Ryder. Practical Blended Taint Analysis for JavaScript. Technical report, Virginia Tech, 2013.