Correlation Tracking for Points-To Analysis of JavaScript

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip

IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
{msridhar,dolby,satishchandra,mschaefer,ftip}@us.ibm.com

Abstract. JavaScript poses significant challenges for points-to analysis, particularly due to its flexible object model in which object properties can be created and deleted at run-time and accessed via first-class names. These features cause an increase in the worst-case running time of field-sensitive Andersen-style analysis, which becomes $O(N^4)$, where N is the program size, in contrast to the $O(N^3)$ bound for languages like Java. In practice, we found that a standard implementation of the analysis was unable to analyze popular JavaScript frameworks.
We identify *correlated dynamic property accesses* as a common code pattern that is analyzed very imprecisely by the standard analysis, and show how a novel *correlation tracking* technique enables us to handle this pattern more precisely, thereby making the analysis more scalable. In an experimental evaluation, we found that correlation tracking often dramatically improved analysis scalability and precision on popular JavaScript frameworks, though in some cases scalability challenges remain.

Keywords: Points-to analysis, call graph construction, JavaScript

1 Introduction

JavaScript is rapidly gaining in popularity because it enables programmers to write rich web applications with full-featured user interfaces and portability across desktop and mobile platforms. Recently, pointer analysis for JavaScript has been used to enable applications such as security analysis [10, 12], bug finding [14], and automated refactoring [8].

Real-world web applications increasingly make use of framework libraries like *jquery*[1] that abstract away browser incompatibilities and provide advanced DOM manipulation and user interface libraries. A recent survey [27] found that more than half of all surveyed sites use one of the nine most popular frameworks. These frameworks present a formidable challenge to static analysis, since they are relatively large and make frequent use of highly dynamic language features.

In particular, JavaScript's flexible object model presents a major challenge for scalable and precise points-to analysis. In JavaScript, an *object* has zero or

[1] http://jquery.com

more *properties* (corresponding to *instance fields* in languages like Java), each identified by a unique name. Properties may contain any kind of value, including first-class functions, and programmers may define a "method" on an object by assigning a function to one of its properties, as in the following example:

```
o.foo = function f1() { return 23; };
o.bar = function f2() { return 42; };
o.foo();
```

To identify the precise call target for the call `o.foo()`, an analysis must be able to compute points-to targets for `o.foo` and `o.bar` separately and not conflate them. This is usually achieved by a *field-sensitive* analysis [16].

Field-sensitive analysis for JavaScript is complicated by the fact that properties can be accessed by computed names. Consider the following code:

```
f = p() ? "foo" : "baz";
o[f] = "Hello, world!";
```

Here, `f` is assigned either the value `"foo"` or the value `"baz"`, depending on the return value of `p`. A *dynamic property access* is then used to store a string value into a property of object `o` whose name is determined by the value of `f`. If `f` is `"foo"`, the existing property `o.foo` is overwritten, otherwise a new property `o.baz` is created and initialized to the given value.

In the presence of dynamic property accesses, performing a field-sensitive analysis poses both theoretical and practical challenges. We show that, surprisingly, extending a standard implementation of field-sensitive Andersen's points-to analysis [3] to handle dynamic property accesses causes the implementation to run in worst-case $O(N^4)$ time, where N denotes the size of the program, compared to the typical $O(N^3)$ bound for other programming languages.

This increased complexity is not merely of theoretical interest—we were unable to scale a traditional field-sensitive analysis to handle JavaScript frameworks like *jquery*. These frameworks generally make heavy use of dynamic property accesses to reflectively access object properties. In combination with other features of JavaScript such as first-class functions, these operations cause an explosion in analysis imprecision that makes call graph construction intractable in practice, as we illustrate on an example in Section 2. And while these operations are most idiomatic and common in JavaScript, exactly the same operations can be written in other scripting languages like Python.

We have devised a technique that helps address issues caused by dynamic property accesses by making the points-to analysis *more precise*. We observed that for property writes that cause imprecision in practice, there is often an obvious correlation between the updated location and the stored value that is ignored by the points-to analysis. For example, for the statement `x[p] = y[p]` (which copies the value for property `p` in `y` to property `p` in `x`), a standard points-to analysis does not track the fact the same property `p` is accessed on both sides of the assignment. If `p` ranges over many possible values, this leads to conflation of many unrelated property-value pairs and cascading imprecision. Our technique regains precision by tracking such correlated read/write pairs and

analyzing them separately for each value of `p`. The analysis thus ends up only copying values between properties of the same name, dramatically improving precision *and* performance in many cases. This *correlation tracking* is achieved by extracting the relevant code into new functions and analyzing them with targeted context sensitivity.

We implemented our technique as an extension to the Watson Libraries for Analysis (WALA) [26] and conducted experiments on five widely-used JavaScript frameworks: *dojo*, *jquery*, *mootools*, *prototype.js*, and *yui*. On these benchmarks, WALA's default implementation of a field-sensitive Andersen-style analysis usually is not able to complete analysis within a reasonable amount of time and produces very imprecise results. We show that correlation tracking significantly improves both analysis performance and precision: most benchmarks can now be analyzed in seconds, though some scalability challenges remain.

The presence of `eval` and other constructs for executing dynamically generated code means that a (useful) static analysis for JavaScript cannot be sound, and ours is no exception. Nevertheless, there are interesting applications for unsound call graphs in security analysis and bug finding [12, 24], and we expect existing tools to benefit significantly from our techniques.

The contributions of this paper can be summarized as follows:

- We show that a standard implementation of field-sensitive Andersen's points-to analysis extended to handle dynamic property accesses has $O(N^4)$ worst-case running time, in contrast to the $O(N^3)$ bound for other languages.
- We demonstrate that this increased complexity has practical repercussions by showing that a previously developed field-sensitive implementation of Andersen's points-to analysis for JavaScript is unable to analyze several widely-used JavaScript frameworks.
- We present a technique to address scalability issues caused by dynamic property accesses by enhancing the points-to analysis to track correlated dynamic property accesses that must at run-time refer to properties with the same name.
- We report on an implementation of our correlation tracking technique on top of WALA and its application to JavaScript frameworks, demonstrating significantly improved scalability and precision in many cases.

The remainder of this paper is organized as follows. Section 2 presents a motivating example that illustrates the complexity of points-to analysis for JavaScript. Section 3 formulates field-sensitive points-to analysis for JavaScript and shows the $O(N^4)$ worst-case running time of a standard implementation extended with handling of computed property names. Section 4 presents our approach for improved handling of dynamic property accesses. Experimental results are presented in Section 5. Section 6 discusses how similar scalability issues may arise in other languages, demonstrating that the techniques presented in this paper may be more widely applicable. Finally, related work is discussed in Section 7 and conclusions are presented in Section 8.

```
 6 function extend(destination, source) {
 7   for (var property in source)
 8     destination[property] = source[property];
 9   return destination;
10 }
11
12 extend(Object, {
13   extend:        extend,
14   inspect:       inspect,
15   ...
16 });
17
18 Object.extend(String.prototype, (function() {
19   function capitalize() {
20     return this.charAt(0).toUpperCase()
21       + this.substring(1).toLowerCase();
22   }
23   function empty() {
24     return this == '';
25   }
26   ...
27   return {
28     capitalize:     capitalize,
29     empty:          empty,
30     ...
31   };
32 })());
33 "javaScript".capitalize(); // == "Javascript"
```

Fig. 1. The `extend` function and some of its uses in *prototype.js*; the definition of `inspect` is omitted.

2 Motivation

In this section, we illustrate how some of JavaScript's dynamic features impact points-to analysis and call-graph construction. We illustrate these points using Figure 1, which shows a few fragments of the widely used *prototype.js* library.[2]

2.1 JavaScript Objects and Functions

JavaScript's model of objects and functions is extremely flexible:

– Unlike class-based languages like Java, JavaScript has no built-in concept of an object instance method. Instead, functions are first-class values, and methods are simply functions stored in object properties. For instance on

[2] `http://www.prototypejs.org/`

lines 12-16 in Figure 1, an object is created with two properties, `extend` and `inspect`, and `extend` is bound to the function defined on line 6.
- Object properties are dynamic in that they are not declared and can be accessed with first-class names. On line 8, the `destination` object has properties assigned based on the `property` variable; if a given property exists in `destination`, it is overwritten; if not, it is created. Thus, the set of properties an object may have is not evident from the code, unlike in a static language like Java.
- The notion of a method call is idiosyncratic; since functions are assigned dynamically to objects, the notion of a receiver is defined by the call itself. Consider the `extend` function defined on line 6. On line 18, this function will be invoked by the `Object.extend` call, and `this` will be bound to `Object`. However, on line 12, `extend` is called directly by name, and `this` defaults to the global object.

Since there is no a priori distinction between properties containing values and properties containing methods, a *field-sensitive analysis*, which represents each property of each abstract object separately, is necessary for obtaining a precise call graph for JavaScript programs. A field-insensitive analysis, which uses a merged representation for each object's properties, would conclude that the invocation of `Object.extend` could possibly invoke `Object.inspect` as well, a very imprecise result.

Several techniques used in other languages to remove obviously invalid results from points-to sets are not applicable in JavaScript. The language lacks classes and declared types for variables, so type filters [9] cannot be employed. Properties are created upon first write, so assignments to non-existent properties cannot be discarded as invalid by the analysis. Finally, although functions declare formal parameters, they can be invoked with any number of actual arguments. If too few arguments are passed, the remaining parameters are assigned the value `undefined`. All arguments (including those not corresponding to any declared parameter) can be accessed via the built-in `arguments` array. This flexibility makes it impossible to perform arity matching to narrow down the set of functions that may be invoked at a given call site.

As a consequence, local imprecision in the analysis can easily cascade and pollute much of the analysis result.

2.2 Dynamic Property Accesses

Figure 1 illustrates how JavaScript's *dynamic property accesses* pose a major challenge for points-to analysis. In particular, the example illustrates how *prototype.js* uses such accesses to dynamically extend objects, a feature often used within *prototype.js* itself. Several other frameworks, including *jquery*, offer similar functionality.

The program of Figure 1 declares a function `extend` on lines 6–10. It uses a `for-in` loop to iterate over the names of all properties of the object bound to its `source` parameter, assigning them to the loop variable `property`. The

value of each property is then read using a *dynamic property access* expression `source[property]`, and stored in a property of the same name in object `destination` by assigning it to `destination[property]`. The following aspects of JavaScript's semantics should be noted:

- The name of the property accessed by a dynamic property access expression is *computed at run-time*.
- As mentioned above, a write to a property *creates* that property if it does not exist yet.

Together, these observations imply that it is possible for a JavaScript program to create objects with an *unbounded* number of properties, which is impossible in statically typed languages such as Java or C#.

Figure 1 also shows two examples of how `extend` is used inside the *prototype.js* library itself.

- On lines 12–16, `extend` is called to bind several functions to properties in the built-in `Object` object. Note that the `extend` function itself is bound to a property `extend` of `Object`.
- On lines 18–32, the `extend` function is invoked as `Object.extend` to extend the `prototype` property of the built-in `String` object with properties `capitalize` and `empty`. As shown on line 33, these properties then become available on all `String` objects, since they have `String.prototype` as their prototype object and hence inherit all its properties.

Now, consider applying Andersen's points-to analysis [3] to the example in Figure 1. As discussed earlier, field-sensitive analysis is necessary to obtain sufficient precision. To handle dynamic property accesses, the analysis must furthermore track the possible values of property-name expressions (like `property` on line 8) and use that information to reason about what properties a dynamic access can read or write. Section 3 formulates such an analysis in more detail and discusses the effect of dynamic property accesses on worst-case complexity.

For the example of Figure 1, variable `property` on line 8 may be bound to the name of any property of any object bound to `source`. In particular, `property` may refer to any property name of the object passed as the second argument in the call on line 12 (`"extend"`, `"inspect"`, etc.) and the one passed on line 18 (`"capitalize"`, `"empty"`, etc.). This means that the points-to set for the dynamic property expression `source[property]` must include *all properties* of the `source` objects. The write to `destination[property]` therefore causes Andersen's analysis to add *all* of these functions to the points-to sets for properties `"extend"`, `"inspect"`, `"capitalize"` etc. in all the `destination` objects (recall that a write to a non-existent property creates the property). In particular, such an analysis would conclude, very imprecisely, that the call `Object.extend(...)` on line 18 might invoke any of the functions `Object` is extended with on line 12.

By the same reasoning, it can be seen that due to the invocation of `extend()` at line 18, this points-to analysis would compute for each property added to `String.prototype` a points-to set that includes `capitalize`, `empty`, and any

```
34 function extend(destination, source) {
35   for (var property in source)
36     (function(p){
37       destination[p] = source[p];
38     })(property);
39   return destination;
40 }
```

Fig. 2. Transformed example

other function stored in the object literal on line 32. Consequently, an invocation of a function read from one of these properties would be approximated as a call to any one of them by the analysis. The resulting loss of precision is detrimental because `String` objects are used pervasively.

This kind of precision loss arose for several widely-used JavaScript frameworks that we attempted to analyze (see Section 5), making straightforward field-sensitive points-to analysis intractable due to the long time it takes to compute the highly imprecise points-to relation, and the excessive space required to store it. This problem is exacerbated by the fact that JavaScript frameworks use mechanisms such as the `extend` function of Figure 1 internally during initialization, which means that merely including the code for these libraries in a web page will trigger the problem.

Our Technique In Section 4, we propose *correlation tracking* as a solution to this problem that can dramatically improve both precision and performance. The key idea is to enhance Andersen's analysis to track correlations between dynamic property reads and writes that use the same property name. For our example, the value read from `source[property]` is written into `destination[property]`; since the value of `property` cannot have changed between the read and the write, the enhanced analysis can reason that a property value from `source` may only be copied to the property with the same name in `destination`.

This is implemented by first extracting the relevant code—in this case the body of the `for-in` loop—into a new function with the property name as its only parameter, as shown for function `extend` in Figure 2.[3] The new function is then analyzed context-sensitively with a separate context for each value of the property name parameter, thereby achieving the desired precision.

This context-sensitivity policy is reminiscent of Agesen's Cartesian Product Algorithm [1] and object-sensitive analyses [18, 20] in the sense that different contexts are introduced for a function based on the values passed as arguments (further discussion in Section 4.2). These enhancements enable our analysis to efficiently compute call graphs for framework-based JavaScript applications in many cases that could not be handled by the baseline field-sensitive analysis.

[3] Other variables of the surrounding scope remain accessible in the extracted code, since JavaScript supports lexical scoping.

Statement	Constraint	
`x = {}`^{i}	$\{o^i\} \subseteq pt(x)$	[Alloc]
`v = "name"`	$\{\texttt{name}\} \subseteq pt(v)$	[StrConst]
`x = y`	$pt(y) \subseteq pt(x)$	[Assign]
`x[v] = y`	$\dfrac{o \in pt(x) \qquad \texttt{s} \in pt(v)}{pt(y) \subseteq pt(o.\texttt{s})}$	[StoreField]
`y = x[v]`	$\dfrac{o \in pt(x) \qquad \texttt{s} \in pt(v)}{pt(o.\texttt{s}) \subseteq pt(y)}$	[LoadField]
`v = x.nextProp()`	$\dfrac{o \in pt(x) \qquad o.\texttt{s} \text{ exists}}{\{\texttt{s}\} \subseteq pt(v)}$	[PropIter]

Table 1. Our formulation of field-sensitive Andersen's points-to analysis in the presence of first-class fields

3 Field-Sensitive Points-To Analysis for JavaScript

In this section, we formulate a field-sensitive points-to analysis for a core language based on the object model of JavaScript. This formulation describes the existing points-to analysis implementation in WALA [26], which we use as our baseline. Then, we show that a standard implementation of Andersen's analysis runs in worst-case $O(N^4)$ time for this formulation, where N is the size of the program, due to computed property names. Finally, we give a minimal example illustrating the imprecision that our techniques address.

Formulation The relevant core language features of JavaScript are shown in the leftmost column of Table 1. Note that property stores and loads act much like array stores and loads in a language like Java, where the equivalent of array indices are string constants.[4] Property names are first class, so they can be copied between variables and stored and retrieved from data structures. As discussed in Section 2, properties are added to objects when values are first stored in them. The `v = x.nextProp()` statement type is used to model the JavaScript `for-in` construct (see Section 2); it updates `v` with the next property name of

[4] In full JavaScript, not all string values originate from constants in the program text; as discussed further in Section 5.1, we handle this by introducing a special "unknown" property name that is assumed to alias all other property names.

the object `x` points to.[5] So, assuming a corresponding `hasNextProp` construct, `for (v in x) { B }` could be modeled as:

```
while (x.hasNextProp()) { v = x.nextProp(); B }
```

The second column of Table 1 presents Andersen-style points-to analysis rules for the core language. The only way in which this differs from a standard Andersen-style analysis for Java [21] is that it supports tracking of property names as they flow through assignments. We represent the points-to set of a program variable x as $pt(x)$. The rules are presented as inclusion constraints over points-to sets of program variables and of properties of abstract objects (e.g., $o.\texttt{name}$). We assume that object allocations are named with one abstract heap object per static statement, e.g., abstract object o^i for statement i. Note that *pt*-sets track not just abstract objects, but also string constants possibly representing property names.[6]

Complexity Computing an Andersen-style points-to analysis can be viewed as solving a *dynamic transitive closure* (DTC) problem for a graph of constraints similar to those in Table 1: $o \in pt(x)$ iff x is reachable from o in the graph. Reachability information is stored by maintaining points-to sets for variables and for fields of abstract-locations, and "propagating" abstract locations to points-to sets based on the constraint edges [21]. The problem requires *dynamic* transitive closure since the StoreField and LoadField rules introduce new constraints based on other points-to facts, which translates to adding new graph edges based on other reachability facts. Most efficient implementations of Andersen's analysis essentially work by computing a dynamic transitive closure; see previous work for details [21].

For Java-like languages, the worst-case complexity of the DTC computation for points-to analysis is $O(N^3)$. The key constraint rules to consider are for field accesses, e.g., the StoreField rule for a statement `x.f = y` (reasoning about LoadField is similar):

$$\frac{o \in pt(x)}{pt(y) \subseteq pt(o.\texttt{f})}$$

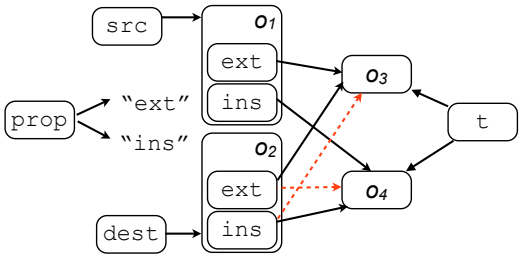
Note that since the field name is manifest in the Java statement, the field-name precondition seen in Table 1 is not required in this rule. Via this rule, the algorithm may add $O(N)$ constraints of the form $pt(y) \subseteq pt(o.\texttt{f})$ to the graph in the worst case (since $|pt(x)|$ is $O(N)$). Considering $O(N)$ abstract locations that may be propagated across each such generated constraint, and $O(N)$ field-write statements in the program, we obtain an $O(N^3)$ worst-case bound on running time.

[5] Property names from objects in the prototype chain are also considered [7, §12.6.4], but we elide this detail here for clarity.

[6] If a non-`String` object o is used as a property name in a dynamic property access, a name is obtained by coercing o to a `String` [7, §11.2.1]; we elide modeling of this behavior here for clarity.

```
1 src = {}
2 dest = {}
3 src["ext"] = {}
4 src["ins"] = {}
5 prop = (*) ? "ext" : "ins";
6 t = src[prop];
7 dest[prop] = t;
```

(a)



(b)

Fig. 3. Imprecisely analyzed property accesses, and the corresponding points-to graph computed by the analysis in Table 1; the red, dashed edges are spurious.

Now, consider the StoreField rule from Table 1, which includes an additional pre-condition $\mathtt{s} \in pt(v)$ to handle computed property names. Unlike Java, this rule may introduce $O(N^2)$ new constraints, one for each (abstract location, property name) pair. Factoring in $O(N)$ worst-case propagation work for each constraint and $O(N)$ store statements in the program now yields an $O(N^4)$ running-time bound for the analysis, worse than that for Java. This bound assumes that the analysis may find each abstract location to have $O(N)$ fields in the worst-case. In a language with classes, an assumption of a constant number of fields per object becomes reasonable [21]. But, with JavaScript's lack of classes and semantics of creating fields when written, such an assumption cannot be made. In fact, our techniques are designed to address a common pattern that causes a blowup in the number of fields per object, as discussed below.

$$\dfrac{\dfrac{\dfrac{o_1 \in pt(src)}{o_4 \in pt(o_1.\mathtt{ins})}\,\text{L4} \qquad \dfrac{o_1 \in pt(src) \qquad \mathtt{ins} \in pt(prop)}{pt(o_1.\mathtt{ins}) \subseteq pt(t)}\,\text{L6}}{o_4 \in pt(t)} \qquad \dfrac{\mathtt{ext} \in pt(prop) \qquad o_2 \in pt(dest)}{pt(t) \subseteq pt(o_2.\mathtt{ext})}\,\text{L7}}{o_4 \in pt(o_2.\mathtt{ext})}$$

Fig. 4. Imprecise derivation of $o_4 \in pt(o_2.\mathtt{ext})$ for the example of Figure 3(a), using the rules of Table 1. Rule applications are labeled with the corresponding line number from Figure 3(a) as appropriate.

Imprecision Example Consider the sequence of statements in Figure 3(a). This program is intended to model normalized statements corresponding to the `for-in` loop in Figure 1 as they would appear to Andersen's analysis; identifier names have been abbreviated.[7] The program creates properties `ext` and `ins` in the object o_1 (named by the line number of its allocation), and then copies one of these properties to object o_2. Figure 3(b), which gives the points-to relation computed

[7] We avoid full normalization to the statement types of Table 1 to ease readability.

for the program, shows that the analysis has imprecisely conflated the `ext` and `ins` properties in o_2, concluding that they both may point to either o_3 or o_4.

Figure 4 shows in detail how the analysis imprecisely concludes that $o_4 \in pt(o_2.\texttt{ext})$, based on the rules of Table 1. Here, the imprecision stems from failing to take into account that lines 6 and 7 of Figure 3(a) must read the same property name from variable *prop*—the rule applied for line 6 uses $\texttt{ins} \in pt(prop)$, while the line 7 rule uses $\texttt{ext} \in pt(prop)$.

Note that conversion of JavaScript statements to their normal form for Andersen's analysis (see Table 1) may in fact *introduce* the imprecision illustrated above. Say that lines 6 and 7 of Figure 3(a) were written as a single statement `dest[prop] = src[prop]`. A points-to analysis that processed this statement directly could avoid the above imprecision—even without flow sensitivity, it is clear that `prop` cannot be re-defined between its two uses in this statement.[8] However, a flow-insensitive analysis must allow for `prop` to be re-defined between the normalized statements `t = src[prop]; dest[prop] = t`, adding imprecision. In contrast to a previous study showing that this normalization does not cause imprecision for points-to analysis of C in practice [5], we have observed real examples (e.g., that of Figure 1) where normalization causes imprecision for JavaScript. Our techniques can recover precision in these cases, and also in cases where multiple source-level statements are relevant, as discussed in the next section.

4 Correlation Tracking

We now discuss our *correlation tracking* technique for improving the scalability of JavaScript points-to analysis in practice. We first illustrate the technique on a small example (Section 4.1), and then detail how we achieve the correlation tracking by extracting code into new functions and analyzing them with targeted context sensitivity (Section 4.2).

4.1 Example

Recall that in the example from Figure 3 of Section 3 the field-sensitive points-to analysis of Table 1 imprecisely concluded that $o_4 \in pt(o_2.\texttt{ext})$ since it did not track the fact that the property read on line 6 must refer to the same property name as the write on line 7.

We can force the analysis to recognize this by splitting variable `prop`, which contains the property name, into two variables `prop1` and `prop2`, respectively corresponding to `prop`'s possible values of `"ext"` and `"ins"`, as shown in Figure 5(a).

Considering again the derivation from Figure 4, we see that the analysis can derive a modified constraint $pt(t_1) \subseteq pt(o_2.\texttt{ext})$ for Figure 5(a) (via line 8), but it cannot derive the corresponding $o_4 \in pt(t_1)$ fact required to imprecisely

[8] Unfortunately, no polynomial-time algorithm is known for such an analysis [5].

```
1 src = {}
2 dest = {}
3 src["ext"] = {}
4 src["ins"] = {}
5 if (*) {
6   prop1 = "ext";
7   t1 = src[prop1];
8   dest[prop1] = t1;
9 } else {
10  prop2 = "ins";
11  t2 = src[prop2];
12  dest[prop2] = t2;
13 }
```

(a)

```
1 src = {}
2 dest = {}
3 src["ext"] = {}
4 src["ins"] = {}
5 prop = (*) ? "ext" : "ins";
6 (function(ff) {
7   t = src[ff];
8   dest[ff] = t;
9 })(prop);
```

(b)

Fig. 5. Transformed versions of the example of Figure 3, to illustrate our technique

conclude that $o_4 \in pt(o_2.\texttt{ext})$. Instead, the analysis can only derive $o_4 \in pt(t_2)$ (via line 11), leading to $o_4 \in pt(o_2.\texttt{ins})$ (line 12), which is in fact feasible.

This example in Figure 5(a) is handled more precisely since the cloning enables the points-to analysis to *track the correlation* of the property name between the copied dynamic property reads and writes—it only copies `src["ext"]` to `dest["ext"]` and `src["ins"]` to `dest["ins"]`.

In general, of course, it is not straightforward to determine the set of possible property names a correlated read/write pair may refer to. Thus, instead of cloning the relevant section of code once for every property name, we extract it into a fresh anonymous function taking the property name as a parameter as shown in Figure 5(b).[9] Combined with a special context sensitivity policy that analyses the fresh function separately for every (abstract) parameter value, we obtain the same precision as with cloning.

4.2 Implementing Correlation Tracking

Identifying Correlations A dynamic property read r and a property write w are said to be *correlated* if w writes the value read at r, and both w and r must refer to the same property name.

We identify such correlated read/write pairs by a data flow analysis on an intra-procedural def-use graph. Starting from a dynamic property read r of the form `o[p]`, we follow def-use edges to track where the read value flows. If it may flow into a dynamic property write w of the form `o'[p'] = e` and we can prove that `p` must have the same value as `p'`, then r and w are correlated. In practice, we have found it sufficient to only consider cases where `p` and `p'` refer to the same local variable p, and p is not redefined between r and w.

[9] Note that local variables `src` and `dest` are accessible inside the anonymous function due to JavaScript's lexical scoping discipline.

In some cases, r and w may be in different functions. To capture this situation, we conservatively assume that any called function may perform a dynamic property write. Thus, if both the value read by r and the value of `p` flow into a function call c, we consider r and c to be correlated. The reverse situation, with the property read occurring in a callee, does not appear to occur in practice and we do not handle it.

While this analysis is not complete and hence will not identify all correlated pairs, it is simple to implement and suffices in practice.

Function Extraction Once we have identified a correlation between r and w on property name p, we extract the snippet of code containing the two accesses into a new function with p as its parameter. In practice, it turns out to be sufficient to only consider the case where the read r and the write w (or the call c for inter-procedural correlations) occur in two statements s_r and s_w such that s_r precedes s_w inside the same block of statements.

If the extraction regions corresponding to different correlated pairs overlap, they are merged and extracted into a function taking all the relevant property name parameters as arguments.

Some language constructs need to be treated specially. In particular, the `this` value of the enclosing method needs to be passed as a separate parameter to the extracted function if there are any `this` references in the extraction region, and these references must be rewritten to access the parameter instead. We currently do not extract code that references the `arguments` array. Finally, unstructured control flow (such as `continue` or `break`) across the boundaries of the extraction region needs to be rewritten to instead return a special flag value; this value is checked by the enclosing function, and the appropriate jump is performed. All these checks and transformations are of the same kind as those performed by implementations of the Extract Method refactoring [19].

Context Sensitivity To ensure that correlated pairs are analyzed once per property name, we analyze the extracted function using the following context sensitivity policy:

> If function f uses a parameter p as the property name in a dynamic property access, f is analyzed context-sensitively, with a separate context for each value of p.

For the example of Figure 5(b), the policy creates a separate context for each value of the `ff` parameter for the function at line 6. This policy effectively clones the extracted function for each possible value of `ff` (`"ext"` and `"ins"`), matching the cloning in Figure 5(a) and hence adding the desired correlation tracking. Our context-sensitivity policy can be viewed as a variant of object sensitivity [18, 20], using the property name parameter instead of the `this` parameter to distinguish contexts.[10]

[10] In the case where a function uses multiple parameters as property names in dynamic accesses, we choose one such parameter arbitrarily to distinguish contexts. We have not observed this case in practice.

The policy is not restricted to functions generated by the extraction of correlated pairs. Thus it is able to handle cases where correlated reads and writes happen in different functions. For instance, consider the following example (loosely based on code found in the *mootools* framework):

```
function doWrite(d,p,v) { d[p] = v; }
function extend(destination, source) {
  for (property in source)
    doWrite(destination,property,source[property]);
}
```

Here, the read in `extend` is correlated with the write in `doWrite`. Our intraprocedural analysis will identify this correlation due to its conservative treatment of function calls, hence the call to `doWrite` will be extracted into a fresh function with parameter `property`. Both the fresh function and `doWrite` use their parameter as a property name, hence they are both analyzed context sensitively, yielding the desired precision.

This additional context sensitivity does not improve the worst-case running time of the analysis; in fact, the analysis could in principle become slower since more constraints are generated for functions analyzed under the new contexts. As the next section shows, however, the technique dramatically improves scalability in practice because we end up creating much sparser points-to graphs.

5 Evaluation

Here we present an experimental evaluation of the effectiveness of our techniques to make field-sensitive points-to analysis for JavaScript scale in practice.

5.1 Implementation

Our analysis implementation is built on top of WALA [26]. WALA provides a points-to analysis implementation for JavaScript, which we extended with correlation tracking. WALA's JavaScript points-to analysis is built on a highly-tuned constraint solver also used for Java points-to analysis [21], and it has already been used in production-quality security analyses for JavaScript [12, 24]. Our work was motivated by the fact that WALA's analysis could not scale to analyze many JavaScript frameworks. By building on WALA, we were able to re-use its handling of various intricate JavaScript language constructs such as the prototype chain and `arguments` array (also discussed in previous work [10, 14]). WALA also provides handwritten models of various pre-defined JavaScript objects and standard library functions.

Default Context Sensitivity WALA's JavaScript points-to analysis uses context sensitivity by default to handle two key JavaScript language features, and we preserved these techniques in our modified version of the analysis. The first construct is `new`, used to allocate objects. The `new` construct has a complex semantics in JavaScript based on dispatch to a first-class function value [7, §11.2.2].[11]

[11] In some cases, a `new` expression may not even create an object [7, §15.2.2.1].

WALA handles `new` by generating synthetic functions to model the behaviors of possible callees. As any one of these synthetic functions may be invoked for multiple `new` expressions, they must be analyzed with one level of call-string context in order to achieve the standard allocation-site-based heap abstraction of Andersen's analysis.

Accesses to variables in enclosing lexical scopes are also handled via context sensitivity by WALA. Handling lexical scoping for JavaScript can be complicated, as nested functions may read and/or write variables declared in enclosing functions [7, §10.2]. WALA employs contexts to ensure that a lexical access only reads or writes the appropriate clones of the accessed variable in the call graph. Without this approach, lexical accesses may lead to merging across call graph clones, muting the benefits of other context-sensitivity policies. Also, lexical information is stored in abstract-location contexts for function objects as needed, to handle closure accesses performed by the function after it is returned from its enclosing lexical scopes. Note that our function extraction technique is eased by WALA's precise treatment of lexical accesses, as fewer parameters and return values need to be introduced (see Section 4.2).

Unknown Properties While our analysis formulation in Section 3 allowed for only constant strings as property names, in a JavaScript property access `a[e]`, `e` may be an arbitrary expression, computed using user inputs, arithmetic, complex string operations, etc. Hence, in some cases WALA cannot compute a complete set of constant properties that a statement may access, i.e., the statement may access an *unknown* property. WALA handles such cases conservatively via *abstract object properties*, each of which represents the values stored in all properties of some (abstract) object. When created, an abstract property is initialized with all possible property values discovered for the object thus far. A read of an unknown object property is modeled as reading the object's abstract property, while a write to an unknown property is treated as possibly updating the object's abstract property and any other property whose name is known. This strategy avoids pollution in the case where all reads and writes are to known constant property names.

`call` *and* `apply` JavaScript function objects provide two built-in methods `call` and `apply` to reflectively invoke the represented function [7, §15.3.4]. Whenever the analysis determines that a call may dispatch to one of these methods, it creates a synthetic stub function that models the reflective call taking place at this call site.

Soundness WALA's points-to analysis attempts to treat most commonly used JavaScript constructs conservatively. However, unsoundness will occur in some cases:

- We currently do not handle `with` blocks, which put the properties of an object in the local scope. Of the frameworks we evaluate, only one (*dojo*) uses `with` blocks in two places. We manually desugared these uses in a similar way as suggested in [13].

– We do not model the semantics of `eval` and the `Function` constructor as well as several other constructs for executing dynamically generated code. This is analogous to how analyses of Java commonly ignore complex reflection and dynamic code loading.
– Some implicit conversions prescribed by the language standard are not yet modeled. In particular, some of these conversions can result in calls to `toString` or `valueOf` methods that we currently ignore.
– Our model of the JavaScript library and the DOM is incomplete, which can lead to unsoundness. Again, this is similar to how analyses of Java work, few of which model the intricate native implementation of portions of the libraries.

In spite of possible unsoundness, the points-to analysis is still useful for a variety of clients, e.g., bug-finding tools [12, 24]. Furthermore, we expect that correlation tracking would provide significant benefits for a sound approach to JavaScript points-to analysis as well.

Framework	Home Page	Version	LOC	Correlated Pairs
dojo	`http://www.dojotoolkit.org`	1.6.1	4748	20
jquery	`http://jquery.com`	1.6.1	5896	34
mootools	`http://mootools.net`	1.4.0	3815	41
prototype.js	`http://prototypejs.org`	1.7	4956	9
yui	`http://developer.yahoo.com/yui`	2.9	24088	31

Table 2. Overview of the JavaScript frameworks used in our experiments. The "LOC" column gives the number of lines of non-blank, non-comment source code as determined by CLOC (`http://cloc.sf.net`), and the "Correlated Pairs" column gives the number of correlated access pairs extracted by our technique.

5.2 Experimental Setup

We evaluated our approach on five popular JavaScript frameworks listed in Table 2, which are among the most widely used frameworks according to a recent survey [27]. For each framework, we collected six small benchmark applications that use the framework, ranging from trivial web pages that do nothing but load the framework scripts to toy web applications of up to 155 lines of code.[12] Note that even just loading each framework already causes significant initialization code to run.

For each benchmark, we attempted to construct call graphs (and hence points-to graphs) using both WALA's standard points-to analysis and our improved technique. We found that most of the frameworks contain sophisticated

[12] A complete list of the benchmark applications used and all of our experimental data is available online at `http://tinyurl.com/JSPointers`.

uses of the above-mentioned reflective methods `call` and `apply`. To more clearly separate out the impact of these features (which is orthogonal to the issue addressed by correlation tracking) we additionally ran our analysis once with modeling of `call` and `apply`, and once without, thus yielding a total of four different configurations.

From now on, we will refer to the configuration using WALA's standard analysis without `call`/`apply` support as "Baseline$^-$" and to the one with support as "Baseline$^+$"; "Correlations$^-$" and "Correlations$^+$" are the corresponding configurations using correlation tracking.

Table 2 also shows, in its last column, the number of correlated access pairs that our technique extracts into fresh functions, which is relatively modest. The benchmark applications themselves did not contain any correlated access pairs.

We performed a separate manual transformation of the `extend` function in *jquery* to eliminate its complex use of the `arguments` array, which again is orthogonal to our focus in this paper. Here is an excerpt of the relevant code:

```
jQuery.extend = function () {
  var target = arguments[0] || {}, i = 1,
      length = arguments.length, deep = false;
  // Handle a deep copy situation
  if ( typeof target === "boolean" ) {
    deep = target;
    target = arguments[1] || {};
    // skip the boolean and the target
    i = 2;
  }
  ...
  // extend jQuery itself if only one argument is passed
  if ( length === i ) {
    target = this;
    --i;
  } ...
}
```

The function explicitly tests both the number of arguments and their types, with significantly different behaviors based on the results. If the first argument is a boolean, its value determines whether a deep copy is performed, and if there is only one argument, then its properties are copied to `this`. Any sort of traditional flow-insensitive analysis of this function gets hopelessly confused about what is being copied where, since `target`, the destination of the copy, can be an argument, a fresh object, or `this` depending upon what is passed.

We manually specialized the above function for the different possible numbers and types of arguments, and this specialized version is analyzed in all four configurations of the points-to analysis. Without the specialization, neither the baseline analysis nor our modified version is able to build a call graph for `jquery`. We leave it to future work to build an analysis to automatically perform these specializations.

All our experiments were run on a Lenovo ThinkPad W520 with a 2.20 GHz Intel Core i7-2720QM processor and 8GB RAM running Linux 2.6.32. We used the OpenJDK 64-Bit Server VM, version 1.6.0_20, with a 5GB maximum heap.

5.3 Results

Framework	Baseline$^-$	Baseline$^+$	Correlations$^-$	Correlations$^+$
dojo	* (*)	* (*)	3.1 (30.4)	6.7 (*)
jquery	*	*	78.5	*
mootools	0.7	*	3.1	*
prototype.js	*	*	4.4	4.5
yui	*	*	2.2	2.1

Table 3. Time (in seconds) to build call graphs for the benchmarks, averaged per framework; '*' indicates timeout. For *dojo*, one benchmark takes significantly longer than the others, and is hence listed separately in parentheses.

Performance We first measured the time it takes to generate call graphs for our benchmarks using the different configurations, with a timeout of ten minutes. The results are shown in Table 3. Since our benchmarks are relatively small, call graph construction time is dominated by the underlying framework, and different benchmarks for the same framework generally take about the same time to analyze. For this reason, we present average numbers per framework, except in the case of *dojo* where one benchmark took significantly longer than the others; its analysis time is not included in the average and given separately in parentheses.

Configuration Baseline$^-$ does not complete within the timeout on any benchmark except for *mootools*, which it analyzes in less than a second on average. However, once we move to Baseline$^+$ and take `call` and `apply` into consideration, *mootools* also becomes unanalyzable.

Our improved analysis fares much better. Correlations$^-$ analyzes most benchmarks in less than five seconds, except for one *dojo* benchmark taking half a minute, and the six *jquery* benchmarks, which take up to 80 seconds. Adding support for `call` and `apply` again impacts analysis times: the analysis now times out on the `jquery` and `mootools` tests, along with the `dojo` outlier (most likely due to a sophisticated nested use of `call` and `apply` on the latter), and runs more than twice as slow on the other *dojo* tests; on *prototype.js* and *yui*, on the other hand, there is no noticeable difference. However, our precision measurements indicate that some progress has been made even for the cases with timeouts in Correlations$^+$ (see below).

Our timings for the "+" configurations do not include the overhead for finding and extracting correlated pairs, which is very low: on average, the former takes about 0.1 seconds, and the latter even less than that.

In summary, correlation tracking speeds up the analysis dramatically in most cases, though reflective language features like `call` and `apply` still present a challenge for some of the benchmarks.

Memory Consumption Introducing a more sophisticated context sensitivity policy may in general lead to larger points-to graphs since the same function may now be analyzed under many more contexts than before, which in turn leads to increased memory consumption. For our benchmarks, we measured the number of points-to edges as a proxy for memory consumption. We found that correlation tracking usually *decreases* this number by two to three orders of magnitude, indicating that the less precise analysis configurations build much denser graphs.

There are two exceptions to this pattern. On *jquery*, the decrease only amounts to 60% to 90%. On *mootools*, there is a case where correlation tracking leads to larger points-to graphs: Correlations$^-$ on average yields about four times as many points-to edges as Baseline$^-$. This is not surprising, since *mootools* is the only benchmark that the imprecise analysis configurations can actually handle.

Framework	Baseline$^-$	Baseline$^+$	Correlations$^-$	Correlations$^+$
dojo	$\geq$ 60.8% ($\geq$60.4%)	$\geq$ 60.5% ($\geq$60.1%)	16.7% (24.5%)	18.8% ($\geq$28.3%)
jquery	$\geq$ 35.9%	$\geq$ 36.2%	26.7%	$\geq$ 31.5%
mootools	9.5%	$\geq$ 35.5%	9.5%	$\geq$ 10.9%
prototype.js	$\geq$ 40.5%	$\geq$ 40.7%	17.8%	18.7%
yui	$\geq$ 16.6%	$\geq$ 16.6%	12.0%	12.2%

Table 4. Percentage of functions considered reachable by our analysis, averaged by framework; '$\geq$' indicates that the number is a lower bound due to analysis timeout. As before, numbers for the outlier on *dojo* are given separately.

Reachable Functions To assess the quality of the generated call graphs, we measured the percentage of functions the analysis considers reachable. In cases of timeout, we base our measurements on the partial call graphs available after ten minutes; the numbers then represent lower bounds.

For every framework under every configuration, Table 4 shows the average percentage of reachable functions over all the benchmarks; variance between benchmarks was very low except for the same *dojo* benchmark that produced atypical timings, which is again listed separately. The baseline configurations consistently deem more functions to be reachable than the correlation tracking configurations, in many cases dramatically so, indicating poor call graph quality.

Polymorphism As a final measure of call graph quality, Table 5 shows the number of highly polymorphic call sites with more than five targets. Once more, these tend to be very similar for benchmarks based on the same framework, so we average over frameworks, except for the by now well-known outlier on *dojo*.

Framework	Baseline$^-$	Baseline$^+$	Correlations$^-$	Correlations$^+$
dojo	≥239.4 (≥240)	≥226.4 (≥225)	0.0 (1)	1.0 (≥11)
jquery	≥244.0	≥249.0	3.0	≥9.0
mootools	0.0	≥29.2	0.0	≥0.0
prototype.js	≥164.5	≥166.0	0.0	0.2
yui	≥29.0	≥34.5	0.0	0.0

Table 5. Number of highly polymorphic call sites (i.e., call sites with more than five call targets) for the benchmarks, averaged per framework; '≥' indicates that the result is a lower bound due to timeout. The outlier on *dojo* is separated out.

The correlation-tracking configurations report very few highly polymorphic call sites: the maximum number is 11 such sites on the problematic *dojo* benchmark under configuration Correlations$^+$, and the maximum number of call targets is 22 on some of the *jquery* benchmarks. We inspected several of these sites and found that they involved higher-order functions and callbacks, justifying the higher call graph fanout. The baseline configurations, on the other hand, produce very dense call graphs with many highly imprecisely resolved call sites, some with more than 300 call targets.

Note that even for cases where Correlations$^+$ times out, the number of highly-polymorphic call sites is dramatically reduced compared to Baseline$^+$. This result is an indication that correlation tracking is still helpful in these cases, even though further work on scalability is needed. For clients that do not require a full call graph, the partial call graph computed by Correlations$^+$ would likely be more useful than that of Baseline$^+$ due to its lower density.

In summary, these results clearly show that correlation tracking significantly improves scalability and precision of field-sensitive points-to analysis for a range of JavaScript frameworks.

6 Other Languages

We have shown that correlation tracking improves analysis of several common JavaScript frameworks. But while our work focuses on JavaScript, there are analogs in other languages. Some languages allow writing code equivalent to the `extend` function from *prototype.js*, and most languages provide string-indexed maps that can cause a similar precision loss. We briefly discuss both cases.

Dynamic property accesses in Python. Like JavaScript, Python is a highly dynamic scripting language with features for reflective property access: `dir` lists all properties of an object, and `getattr` and `setattr` provide first-class property access. An equivalent of the `extend` function of Figure 1 can easily be written:

```
def extend(a, b):
    for f in dir(b): setattr(a, f, getattr(b, f))
```

This style is less idiomatic and pervasive in Python, however, so the kind of imprecision we see when analyzing JavaScript is less likely to occur in practice.

Imprecision with maps. Maps are a core data structure in many applications and can cause precision loss in the same manner as dynamic property accesses in JavaScript. Most maps have accessors to get and set entries and to list all keys. Consider an example in Java: for server-side web applications, the `HttpSession` class stores state that persists across multiple user interactions with a server that share a session; the following code, to enhance security, sanitizes all this persistent state:

```
for(String n : session.getAttributeNames())
  session.setAttribute(n, sanitize(session.getAttribute(n)));
```

This is essentially the same pattern as `extend`, and will cause imprecision in modeling session state, unless techniques like correlation tracking are employed.

7 Related Work

We distinguish several threads of related work.

Complexity Chaudhuri [6] presents an optimization to CFL-reachability / recursive state machine algorithms (which can handle standard field-sensitive points-to analysis [22]) that yields $O(N^3/log(N))$ worst-case running time. We conjecture that similar techniques could shave a logarithmic factor from our $O(N^4)$ bound for points-to analysis in the presence of dynamic property accesses, but devising and analyzing such an algorithm remains as future work.

JavaScript Semantics Guha et al. [13] present a formalization of a core calculus λ_{JS} for JavaScript, which includes computed property names, prototype chains and other troublesome features, but excludes `eval`. Our implementation is not based on translating JavaScript to λ_{JS}, but even with such an approach the key analysis challenges that we face would remain. A complete formalization of JavaScript (again without `eval`) is described by Maffeis et al. [17], but their semantics is too complex to be useful for reasoning about static analyses.

Argument sensitivity The Cartesian product algorithm [1] (CPA) and object sensitivity [18] both inspired our context-sensitivity policy for extracted functions (see Section 4.2). These techniques create contexts based on the concrete types of arguments at call sites, thus allowing analysis of a function to be specialized based on what types of values are being passed to it. CPA does this for all parameters, and object sensitivity applies just to the receiver argument.

Smaragdakis et al. [20] conduct a thorough analysis of object sensitivity, classifying the prior work in terms of how it chooses contexts based on receiver objects. They also introduce type sensitivity in which contexts are distinguished not based on abstract objects themselves, but rather on their types. They show

that this is a promising approach for improving the cost/precision balance in analysis, but clearly it depends on having a useful notion of static types, which JavaScript lacks.

Other JavaScript Analyses JavaScript combines the program analysis challenges of a higher-order functional language with those of a very dynamic scripting language, and considerable work has focused on addressing some of these issues.

- Jensen et al. [14, 15] deal with issues arising from JavaScript's prototype-based inheritance and the use of automatic coercions. They construct a detailed lattice of types and adapt the recency abstraction of Balakrishnan et al. [4] to precisely handle writes to inherited properties in constructors. The JSRefactor refactoring tool for JavaScript [8] also includes a points-to analysis for JavaScript, which is influenced by this line of work.
- Vardoulakis and Shivers [25] introduce CFA2 to tackle the limitations of CFA with respect to the deep nesting of first-class function calls common in higher-order languages. They use a continuation passing style transformation of the code and a summarization scheme based on local state to match deeply-nested calls and returns. The DoctorJS type inference tool[13] is based on CFA2; it uses a special form of context sensitivity to analyze for-in loop bodies once for every abstract value of the loop variable, thus achieving (for this special case) a similar effect to our more general technique.

These techniques mostly address other challenges that arise when analyzing dynamic languages such as JavaScript, and are complementary to our work. There is also much work that focuses on problems that are specific to JavaScript:

- Zheng et al. [28] present a JavaScript analysis to find data races in code used in asynchronous ways in a Web browser. They analyze JavaScript code that uses several of the popular frameworks that we handle (*jquery*, *prototype.js*, and *yui*); however, they do not actually analyze the framework code, but instead design inference rules with the framework semantics encoded.
- Guarnieri and Livshits's Gatekeeper [10] and Gulfstream [11] tools perform points-to analyses for JavaScript for use in security analysis, with a focus on incremental analysis in the face of dynamically loaded code on Web pages. They treat dynamic property accesses precisely when a single possible property name can be determined statically (by a separate constant propagation pass), and otherwise assume that any property might be referenced. They do not focus on the problems caused by constructs such as `for-in` loops.

To the best of our knowledge, none of these systems is able to analyze JavaScript frameworks.

Dynamic Type Inference for Scripting Languages An et al. [2] present a dynamic analysis for inferring static types in Ruby,[14] sidestepping many of the challenges

[13] `www.doctorjs.org`.
[14] `http://www.ruby-lang.org/`

a static analysis for Ruby would have to face, which are similar to the issues arising in JavaScript. Despite being dynamic, however, their analysis is sound. Their focus is on type inference, so they do not track some information needed for our analysis, like the values of different string constants. Also, their technique requires test inputs, which are not readily available for JavaScript frameworks.

Field Sensitivity Tripp et al. [23] present a taint analysis for Java that implements a form of field sensitivity when handling common J2EE idioms.[15] J2EE uses a context structure that is essentially a hash table, and it is usually referenced in practice with constant strings as keys. They employ an abstraction of the semantics of the context object rather than the actual Java code, applying field sensitivity to distinguish different constant keys used in each context.

8 Conclusions

JavaScript is a uniquely challenging language for pointer analysis: ubiquitous use of dynamic property accesses and of idioms for iterating across all of an object's properties together complicate points-to analysis in a fundamental way.

We introduce *correlation tracking*, a technique for targeted context sensitivity to handle situations where values from dynamic property reads flow to dynamic writes of the same property. We show that this technique aids analyzing common JavaScript frameworks, dramatically improving scalability and precision in many cases. We also provide a sense of why analysis is so much more expensive by showing that extending a standard implementation of Andersen's analysis with support for dynamic property accesses increases its worst-case running time from $O(N^3)$ to $O(N^4)$, where N is the size of the program.

Work remains to further improve points-to analysis for JavaScript; while correlation tracking makes many popular frameworks tractable, there are some that still cannot be fully analyzed. Hence, our future work will focus on finding and solving the further causes of complexity in these frameworks, including better handling of `call` and `apply` and automation of the precise handling of the `arguments` array that was crucial for *jquery* (see Section 5). We also plan to integrate our current analysis with existing tools [12, 24], as we expect correlation tracking to significantly improve their effectiveness on framework-based web sites.

[15] `http://download.oracle.com/javaee`

Bibliography

1. Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *ECOOP*, 1995.
2. Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic Inference of Static Types for Ruby. In *POPL*, 2011.
3. Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.
4. Gogul Balakrishnan and Thomas W. Reps. Recency-Abstraction for Heap-Allocated Storage. In *SAS*, 2006.
5. Sam Blackshear, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Manu Sridharan. The Flow-Insensitive Precision of Andersen's Analysis in Practice. In *SAS*, 2011.
6. Swarat Chaudhuri. Subcubic Algorithms for Recursive State Machines. In *POPL*, 2008.
7. ECMA. ECMAScript Language Specification, 5th edition, 2009. ECMA-262.
8. Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported Refactoring for JavaScript. In *OOPSLA*, 2011.
9. David Grove and Craig Chambers. A Framework for Call Graph Construction Algorithms. *TOPLAS*, 23(6), 2001.
10. Salvatore Guarnieri and V. Benjamin Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, 2009.
11. Salvatore Guarnieri and V. Benjamin Livshits. Gulfstream: Incremental Static Analysis for Streaming JavaScript Applications. In *WebApps*, 2010.
12. Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the World Wide Web from Vulnerable JavaScript. In *ISSTA*, 2011.
13. Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *ECOOP*, 2010.
14. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *SAS*, 2009.
15. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural Analysis with Lazy Propagation. In *SAS*, 2010.
16. Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis Using Spark. In *CC*, April 2003.
17. Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *APLAS*, 2008.
18. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *TOSEM*, 14(1), 2005.
19. Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping Stones over the Refactoring Rubicon. In *ECOOP*, 2009.
20. Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick Your Contexts Well: Understanding Object-sensitivity. In *POPL*, 2011.

21. Manu Sridharan and Stephen J. Fink. The Complexity of Andersen's Analysis in Practice. In *SAS*, 2009.
22. Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-Driven Points-To Analysis for Java. In *OOPSLA*, 2005.
23. Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. In *PLDI*, 2009.
24. Omer Tripp and Omri Weisman. Hybrid Analysis for JavaScript Security Assessment. In *ESEC/FSE'11 (Industrial Track)*, 2011.
25. Dimitrios Vardoulakis and Olin Shivers. CFA2: A Context-Free Approach to Control-Flow Analysis. In *ESOP*, 2010.
26. T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`.
27. Web Technology Surveys. Usage of JavaScript libraries for websites. `http://w3techs.com/technologies/overview/javascript_library/all`. Accessed 30 March 2012.
28. Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *WWW*, 2011.