State-Sensitive Points-to Analysis for the Dynamic Behavior of JavaScript Objects

Shiyi Wei and Barbara G. Ryder

Department of Computer Science,
Virginia Tech, Blacksburg, VA, USA
{wei,ryder}@cs.vt.edu

Abstract. JavaScript object behavior is dynamic and adheres to prototype-based inheritance. The behavior of a JavaScript object can be changed by adding and removing properties at runtime. Points-to analysis calculates the set of values a reference property or variable may have during execution. We present a novel, partially flow-sensitive, context-sensitive points-to algorithm that accurately models dynamic changes in object behavior. The algorithm represents objects by their creation sites and local property names; it tracks property updates via a new control-flow graph representation. The calling context comprises the receiver object, its local properties and prototype chain. We compare the new points-to algorithm with an existing JavaScript points-to algorithm in terms of their respective performance and accuracy on a client application. The experimental results on real JavaScript websites show that the new points-to analysis significantly improves precision, uniquely resolving on average 11% more property lookup statements.

Keywords: JavaScript, program analysis, points-to analysis.

1 Introduction

Dynamic programming languages, including JavaScript, Ruby and PHP, are widely used in developing sophisticated software systems, especially Web applications. These languages share several dynamic features, including dynamic code generation and dynamic typing, used in real-world programs [20]. For example, JavaScript code can be generated at runtime using *eval* and JavaScript functions can be variadic (i.e., functions can be called with different numbers of arguments). Despite the popularity of these dynamic languages, there is insufficient tool support for developing and testing programs because their dynamic features render many traditional analyses, and tools which depend on them, ineffective.

In addition, instead of class-based inheritance JavaScript supports prototype-based inheritance [16,27] that results in a JavaScript object inheriting properties from a chain of (at least one) prototype objects. The model also allows the properties of a JavaScript object to be added, updated, or deleted at runtime. This means that JavaScript objects can exhibit different behaviors at different

R. Jones (Ed.): ECOOP 2014, LNCS 8586, pp. 1–26, 2014.

times during execution. Moreover, object constructors may be polymorphic so that objects created by the same constructor may have distinct properties. These aspects of JavaScript further complicate tool building.

Some tools have been developed to support JavaScript software development (e.g., [19,22]). Points-to analysis is the enabling analysis for such tools. Researchers have proposed several points-to analyses that handle different features of JavaScript (e.g., [8,17,26]). Nevertheless, there are opportunities to significantly improve the precision of points-to analysis through better modeling of object property set changes.

In this paper, we present a novel points-to algorithm that can accurately model JavaScript objects. Changes to object properties are tracked more accurately to reflect object run-time behavior at different program points. A new graph decomposition for control-flow graphs is used to better track object property changes. Prototype-based inheritance is more accurately modeled to locate delegated properties. The analysis identifies objects by their creation site as well as their local property names upon construction, more accurately than the per-creation-site representation. To distinguish polymorphic constructors, this analysis can incorporate dynamic information collected at runtime (see Section 3.6). Technically, the analysis is *partially flow-sensitive* (on our new control-flow graph structure) and *context-sensitive*, using a new form of object sensitivity [18].[1] Rather than using the receiver object creation site as a calling context in the analysis, we use an approximation of the receiver object and its properties at the call site (i.e., *obj-ref state*).

In order to compare this algorithm with previous techniques, we instantiated our new points-to analysis as the static component of the JavaScript Blended Analysis Framework (*JSBAF*) [28]. Blended analysis collects run-time information through instrumentation to define the calling structure used by a subsequent static analysis, and to capture dynamically generated code. It has been demonstrated that blended analysis is practical and effective on JavaScript programs [28]. We measured the performance and accuracy of our new analysis on a statement-level points-to client (*REF* analysis) that calculates how many objects are returned by a property lookup (e.g., a read of $x.p$).

The major contributions of this work are:

- We have designed a novel *state-sensitive points-to analysis that accurately and safely handles dynamic changes in the behavior of JavaScript objects.* This algorithm presents a new program representation that enables partially flow-sensitive analysis, a more accurate object representation, and an expanded points-to graph that facilitates strong updates for the statements changing object properties.

[1] Informally, a flow-sensitive analysis follows the execution order of statements in a program; a flow-sensitive analysis can perform strong updates, but a flow-insensitive one cannot. A context-sensitive analysis distinguishes between different calling contexts of a method, producing different analysis results for each context [21,24]. A context-insensitive analysis calculates one solution per method.

– Experimental results from our new analysis compared to a recent points-to analysis [26], both implemented in *JSBAF*, showed that *state-sensitive analysis significantly improved precision*. On average over all the benchmarks (i.e., 12 popular websites), 48% of the property lookup statements were resolved to a single object by our new analysis, while the existing analysis [26] uniquely resolved only 37% of these statements. Although our analysis incurred a 127% time overhead on average to achieve the increased precision, it was able to analyze each of the programs in the benchmarks in under 5 minutes, attesting to its scalability in practice.

Overview. Section 2 defines the notion of *obj-ref state* and then uses an example to illustrate the sources of imprecision in JavaScript points-to analysis. Section 3 describes our new points-to analysis algorithm and implementation. Section 4 presents the *REF* analysis and the experimental results. Section 5 discusses related work, and Section 6 offers conclusions and future work.

2 Definitions and Motivating Example

In this section we define key concepts and use an example to illustrate the sources of imprecision in current points-to analyses for JavaScript.

2.1 JavaScript Object-Reference State

JavaScript is a dynamically typed programming language whose object behavior can change as object properties are added or deleted at runtime. In strongly typed programming languages, the notion of *type* is used to abstract the possible behavior of an object (e.g., the class of an object in Java) [23]; however, in dynamically typed languages, the type of an object can change during execution. In order to avoid confusion, we call the type of a JavaScript object its *obj-ref state*.[2]

> **Definition 1.** The *obj-ref state* at a program point denotes all of its accessible properties and their non-primitive values.

The accessible properties of an object conform to the property lookup mechanism implemented in JavaScript. Every JavaScript object includes an internal reference to its prototype object from which it inherits non-local properties. A JavaScript object may have a sequence of prototype objects (i.e., a prototype chain) whose properties it can inherit. When reading a property p of an object o, the JavaScript runtime checks the local properties of o to see if o has a property named p. If not, the JavaScript runtime checks to see if the prototype object of o has a property named p, continuing to check along the prototype chain from object to object until the property is found (or not) [7].

[2] This general notion can be used for other dynamic languages and is related to structured typing for strongly typed languages [23].

Definition 2. *State-update statements* are: (1) property write statement (i.e., $x.p = y$ or $x['p'] = y$), (2) property delete statement (i.e., *delete* $x.p$ or *delete* $x['p']$), and (3) an invocation that directly or indirectly results in execution of (1) and/or (2).

The *state-update statements* are the set of statements in JavaScript that may affect the *obj-ref states*. In Figure 1, we illustrate the *obj-ref state* with an example that shows the objects connected to O_1 at a program point. The local properties of object O_1 are named p_1 and p_2 and O_4 is its prototype object. O_7 is visible from O_1 by accessing $O_1.p_4$ while O_6 is not visible from O_1 by accessing $O_1.p_2$ because a local property named p_2 exists for O_1. To sum up, the shaded nodes (i.e., O_6 and O_9) are not accessible from O_1 and the unshaded nodes constitute O_1's reference state.

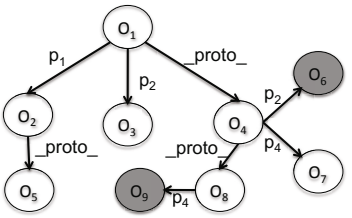


Fig. 1. *obj-ref state* for O_1. (Unshaded nodes only)

2.2 Imprecision of Points-to Analysis

A flow-insensitive analysis may produce imprecise results when *obj-ref state* changes, because it must safely approximate points-to relations and it cannot do strong updates. Existing context-sensitive analyses may produce imprecise results because they lack the power to distinguish between different *obj-ref states* for the same JavaScript object. In Figure 2, we present a JavaScript example to illustrate the sources of imprecision of a flow-insensitive, context-insensitive points-to analysis resulting from several dynamic features of JavaScript. We also demonstrate that an existing context-sensitive analysis using the same object representation as [18] is ineffective at distinguishing the function calls in the example.

Lines 2-6 show a constructor function $X()$. Objects created by $X()$ may or may not have the local property named p or q (lines 4 and 5) depending on the value of its argument. The statement in line 12 updates the value of local property p of an object pointed to by x if p exists; otherwise, the statement adds the local property named p to the object. Figures 3(a) and 3(b) show the points-to graphs that reflect the run-time behavior of this code. We use the line number to represent the object created (e.g., the object created at line 7 (*new* X) is O_7). We focus on two program points in the execution, lines 10 and 15. The nodes O_7, O_4, and O_3 and O_9 constitute the *obj-ref state* of O_7 at line 10 and the nodes O_7, O_{12}, O_3 and O_{14} constitute the *obj-ref state* of O_7 at line 15.

```
1  function P(){ this.p = new Y1(); }
2  function X(b){
3      this.__proto__ = new P();
4      if(b) { this.p = new Y2(); }
5      else this.q = new Y3();
6  }
7  var x = new X(true);
8  x.bar = function(v, z){ v.f = z; }
9  var z1 = new Z();
10 x.bar(x.p, z1);
11 ...
12 x.p = new A();
13 ...
14 var z2 = new Z();
15 x.bar(x.p,z2);
```

Fig. 2. JavaScript example

Note that O_1 is not visible from O_7 at lines 10 or 15 because of the existence of the local property named p. The *obj-ref state* of object O_7 is different at these two program points.

Constructor polymorphism (lines 2-6), object property change (line 12) and function invocations (lines 10 and 15) in the example make precise static points-to analysis hard to achieve with current techniques. Figure 3(c) shows a points-to graph for the example built by a flow- and context-insensitive points-to analysis. Dashed nodes and edges are imprecise points-to relations that cannot exist at runtime.

There are several sources of imprecision. Line 7 creates an object pointed to by variable x by invoking the polymorphic constructor $X()$. Not knowing the value of b, static analysis conservatively builds all the points-to relations possible from execution of $X()$.[3] When reading the property p of x (line 10), static analysis returns objects O_4 and O_1 because a conservative analysis cannot distinguish whether or not O_4 actually exists. Furthermore, because of the imprecise result of the read of $x.p$, invoking the $bar()$ function results in imprecise property reference from O_1 to O_9. Flow-insensitive points-to analysis simply adds O_{12} to $O_7.p$ (line 12) because it cannot perform strong updates. Because the analysis does not distinguish which objects v and z point to on different calls of $bar()$, line 15 results in additional imprecision with respect to $O_4.f$ and $O_{12}.f$.

Flow-sensitive analysis is not sufficient to resolve the imprecision in the example without an appropriate context for call sites. First, indirect assignment statements cannot be strongly updated in general. Second, assuming $x.p$ is strongly updated to point to O_{12} at line 12, a context-insensitive analysis does not remove the imprecise edges $(< O_4, f >, O_{14})$ and $(< O_{12}, f >, O_9)$ because calls to $bar()$ (lines 10 and 15) are not distinguished by calling contexts. Object sensitivity [18]

[3] In this short example, constant propagation of parameters would help static analysis precision but clearly this is not always possible.

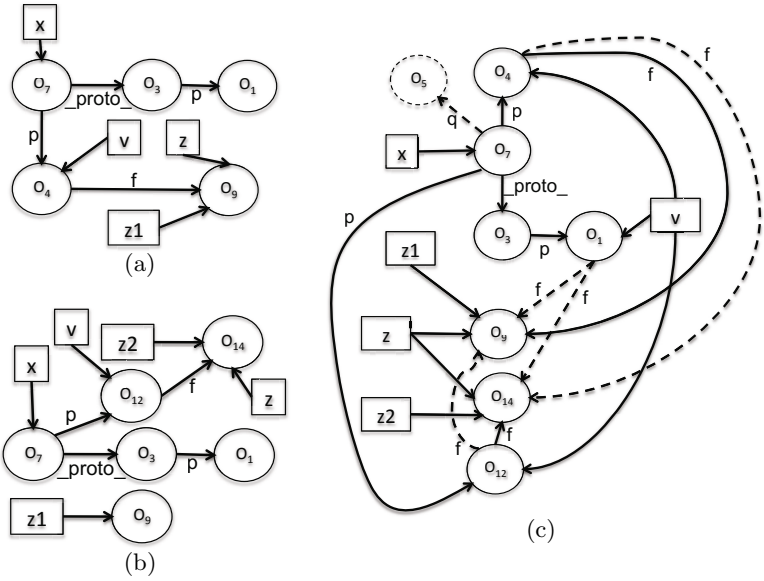


Fig. 3. Imprecision of static points-to analysis. (a) Run-time points-to graph at line 10. (b) Run-time points-to graph at line 15. (c) Flow- and context-insensitive points-to graph.

has been shown to perform better than call-string context sensitivity [24] for the idioms used in object-oriented languages [15]. However, object-sensitive analysis is not able to differentiate these two call sites because they have the same receiver object O_7, which has two different *obj-ref states* at these call sites. Our new points-to analysis is designed to handle these constructs more accurately and to address the challenges raised by *obj-ref state* updating and prototype-based inheritance.

3 State-Sensitive Points-to Analysis

In this section we will present our *state-sensitive points-to analysis* for JavaScript. We will explain key ideas used in the analysis, including the intra-procedural program representation (i.e., the block-sensitive decomposition of control-flow graphs), the solution space (i.e., the annotated points-to graph with access path edges and in-construction nodes), the transfer functions of the state-update statements as well as the state-preserving statements, state sensitivity (i.e., a form of context sensitivity based on object sensitivity that captures changes in object behavior during execution) and block sensitivity (i.e., a partial flow sensitivity performed on the transformed CFG). Finally, we will discuss the implementation details of our algorithm.

3.1 State-Preserving Block Graph

A flow-insensitive analysis ignores the control flow of a program while a flow-sensitive analysis typically uses an intra-procedural control-flow graph (CFG). Our analysis aims to provide a better model of a JavaScript object whose reference state exhibits flow-sensitive characteristics (e.g., allowing addition and deletion of object properties at any program point). Cognizant of the possible overhead introduced by a fully flow-sensitive analysis, we designed a partially flow-sensitive analysis that only performs strong updates when possible on state-update statements using a transformed CFG, called the *State-Preserving Block Graph (SPBG)*. Recall that the state-update statements, including the property write (i.e., add or update a property) and delete (i.e., remove a property), directly change the *obj-ref state* in JavaScript; all other statements (e.g., property read) are state-preserving statements.

Figure 4 shows an example *SPBG* (Figure 4(b)) compared to its original CFG (Figure 4(a)). An *SPBG* is a transformed control-flow graph whose basic blocks are aggregated into region nodes according to whether or not they contain a state-update statement. The *SPBG* also contains state-update statements as special singleton statement nodes (i.e., *state-update* nodes). An example of a region node (i.e., *state-preserving* node) is 2-4-5-7 in Figure 4(b) whereas node $x = new\ A()$ is an example of a *state-update* node. Note that in creating singleton nodes the algorithm breaks apart former basic blocks (e.g., $1 \rightarrow \{1', x = new\ A(), 1''\}$).

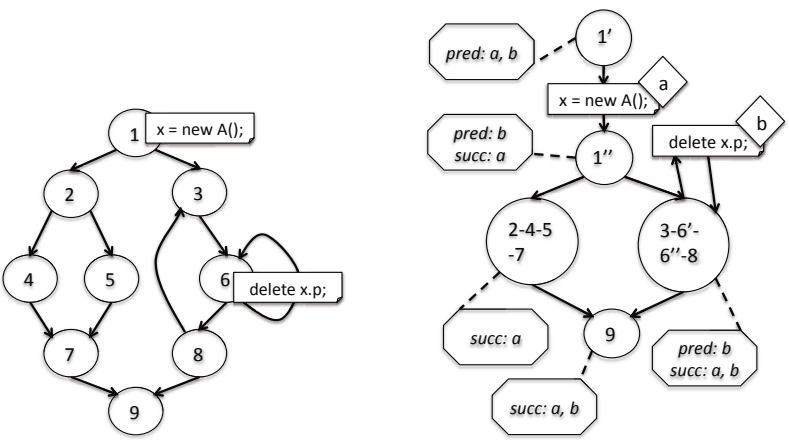


Fig. 4. *SPBG* generation. (a) CFG. (b) *SPBG*.

We first split any basic blocks in the CFG that contain at least one state-update statement (see Definition 2 in Section 2), obtaining a *split-CFG*. State-update statements (1) and (2) can be detected syntactically and invocations that

may result in an *obj-ref state* change (i.e., category (3)) are found by a linear call graph traversal.[4] We then use a variant of the standard CFG construction algorithm [1] to build the split-CFG. The header nodes used include the standard headers [1] plus (i) any state-update statement is a region header of a state-update node containing only that statement, and (ii) any state-preserving statement that immediately follows a state-update statement is a region header of a state-preserving node.

In an *SPBG*, state-preserving region nodes are formed based on grouping nodes in the split-CFG that share the same control-flow relations with respect to state-update nodes. The possible control-flow relations of node n_1 and n_2 in a split-CFG include: (1) n_1 is a successor of n_2, (2) n_1 is a predecessor of n_2, (3) n_1 is both a successor and a predecessor of n_2 (i.e., n_1 and n_2 exist in a loop) and (4) n_1 and n_2 have no control-flow relation (e.g., n_1 and n_2 are present in different branches). We label each node in a split-CFG with its relations to each state-update node via depth-first searches. The set of labels form a signature for that node. If nodes share the same signature it means that they have the same control-flow relationship(s) to a (set of) state-update statement(s) so that they can be collapsed to a state-preserving node in the *SPBG*. Figure 4(b) shows the signatures of the state-preserving regions in the generated *SPBG*; *a* and *b* represent the state-update statements *x=new A()* and *delete x.p*, respectively. Basic blocks 2, 4, 5 and 7 are aggregated because they only appear as successors of *x=new A()* and have no control-flow relation to *delete x.p*. The region node 2-4-5-7 is not further aggregated with basic block 9 because 9 is a successor of *delete x.p* but 2-4-5-7 is not.

3.2 Points-to Graph Representation

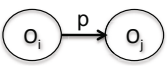
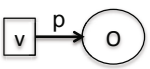
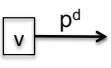
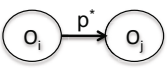
Our points-to graph representation includes constructs that facilitate the handling of strong updates by our analysis. Our algorithm design allows strong updates when possible for state-update statements. In contrast, most flow-sensitive Java analysis algorithms cannot perform strong updates for indirect assignment statements (e.g., $x.p = y$) and few analyses consider property delete statements, which are uncommon in object-oriented languages. Two existing techniques help to enable strong updates for such statements in JavaScript: recency abstraction and access path maps.

Recency abstraction [2,11] associates two memory-regions with each allocation site. The most-recently-allocated block, a concrete memory-region, allows strong updates and the not-most-recently-allocated block is a summary memory-region. We adapt the idea of recency abstraction to enable strong updates during analysis of constructor functions.

De *et al.* [6] performed strong updates at indirect assignments by computing the map from access paths (i.e., a variable followed a sequence of property accesses) to sets of abstract objects. This work demonstrated the validity of using access path maps to perform strong updates for indirect write statements in

[4] Our analysis requires a pre-computed call graph as input. See Section 3.6 for details.

Table 1. Expanded points-to graph with annotations

points-to graph G	node N	variable v	
		abstract object o	
		in-construction object $@o$	
	edge E	variable reference $(v, \phi o)$	
		property reference $(< \phi o_i, p >, \phi o_j)$	
		access path $(< v, p >, \phi o)$	
	annotation A	d annotation p^d	
		$*$ annotation p^*	

Java. We adapt this approach to points-to analysis for JavaScript by expanding the points-to graph representation instead of using separate maps.

Table 1 lists the nodes, edges and annotations in our points-to graph. In addition to variable nodes v and abstract object nodes o, our points-to graph contains *in-construction object nodes* $@o$.[5] Details of the in-construction objects will be discussed in Section 3.3. For sake of simplicity, we use ϕo to represent either kind of object node (i.e., o or $@o$).

There are three kinds of edges. Variable reference and property reference edges exist in a traditional points-to graph. An *access path edge*, $(< v, p >, \phi o)$, denotes that the property p of variable v refers to object ϕo. $< v, p >$ represents an access path with length of 2 (i.e., a variable followed by one field access $v.p$).[6]

Our analysis calculates *may* pointer information, meaning that a points-to edge in the graph may or may not exist at runtime. To better approximate the *obj-ref states* of JavaScript objects, we introduce annotations on property reference edges as well as access path edges. The annotations help to calculate *must exist* information for object property names. In our analysis, the d annotation on a property name p (i.e., p^d) denotes that the local property named p must not exist. This annotation only applies to access path edges in our points-to graph. The other annotation, $*$, applies to both property reference edges and

[5] Similar to the recency abstraction, an in-construction object always describes exactly one concrete object. In our analysis, it exists only during analysis of a constructor.

[6] The length of an access path is one more than the number of field accesses [6].

access path edges. p^* denotes that the local property named p may not exist. Property reference edges without annotation or access path edges without annotation represent *must exist* information for the property names. We use p^ϕ to represent any kind of p^d, p^* or p edge. These annotated edges help us perform a more accurate property lookup (see Section 3.3).

$Pt(x)$ denotes the points-to set of x and $Pt(< \phi o, p >)$ denotes the points-to set of the property p of ϕo. $Pt(< v, p >)$ denotes the points-to set of access path $v.p$. We also define the operation $Alias(v)$ which returns the set of variables W such that v and $w \in W$ point to the same object. *apset(v)* denotes the set of all access path edges of v (i.e., *apset(v)* $= \forall q : \{(< v, q^\phi >, \phi o)\}$).

In addition to the points-to graph, we use a mapping data structure to store intermediate information in the analysis. The map M is used to record the list of property names when an object is constructed. An abstract object (e.g., o) is the key in M whose value is the set of local property names that exist when the constructor function of the abstract object returns (e.g., {p1, p2, p3}).

3.3 Points-to Analysis Transfer Functions

In this section we describe the data-flow transfer functions for the statements shown in Table 2.

Object creation ($x =$ *new X(*a_1*,* a_2*, ...,* a_n*)*)**.** In our analysis, an object creation statement (i.e., *new* statement) is modeled in three steps. $x =$ *new X* creates an in-construction object $@o_i$. Then the invocation of the constructor *new* $X((a_1, a_2, ..., a_n))$ is modeled as a function call on $@o_i$. Upon the return of the constructor (i.e., ret_X), the analysis removes the in-construction object from the points-to graph and redirects all points-to relations from $@o_i$ to an abstract object (i.e., *remove(G,* $@o_i$*)*). If the local property set of the in-construction object matches that of an existing abstract object with the same allocation site, the in-construction object is merged into the abstract object; otherwise, a new abstract object is created to replace the in-construction object. There is at most one in-construction object for each creation site.[7]

The transfer function of the object creation statement ensures that abstract objects are based on their allocation site as well as their constructed local properties (i.e., an approximation of actual *obj-ref state*); in other words, the objects created at the same allocation site that contain the same set of local property names share the same abstract object in our analysis. This object representation is more precise than using one abstract object per creation site.

Property write ($x.p = y$)**.** In general, strong updates cannot be performed on the property write statement because an abstract object may summarize multiple

[7] Recursive constructor calls involve the creation of an in-construction object when the in-construction object for the same allocation site already exists (before it resolves into an abstract object). In our analysis, the existing in-construction object is resolved into a *special abstract object* whose set of properties upon construction is *unknown*. A fixed point calculation is done using the special abstract object.

Table 2. Transfer functions of program statements

Statement	Transfer function
$s_i : x = new\ X(a_1, a_2, ..., a_n)$	(1) $x = new\ X : (G - apset(x)) \bigcup (x, @o_i)$ (2) $new\ X((a_1, a_2, ..., a_n)) : G \bigcup \{invoke(G, X, @o_i, a_1, a_2, ..., a_n)\}$ (3) $ret_X : remove(G, @o_i)$
$x.p = y$	(1) $if\ \|Pt(x)\| = 1\ and\ \{\phi o_i \in @O \| \phi o_i \in Pt(x)\}$: $G - \{(< @o_i, p^\phi >, \phi o_j) \| @o_i \in Pt(x) \wedge \phi o_j \in Pt(< @o_i, p^\phi >)\}) \bigcup \{(< @o_i, p >, \phi o_j) \| @o_i \in Pt(x) \wedge \phi o_j \in Pt(y)\}$ (2) $otherwise$: (2.1) $(G - \{(< x, p^\phi >, \phi o_i) \| \phi o_i \in Pt(< x, p^\phi >)\}) \bigcup \{(< x, p >, \phi o_j) \| \phi o_j \in Pt(y)\}$ (2.2) $G \bigcup \{(< \phi o_i, p^* >, \phi o_j) \| \phi o_i \in Pt(x) \wedge \phi o_j \in Pt(y)\}$ (2.3) $G \bigcup \{(< z, p^* >, \phi o_i) \| z \in Alias(x) \wedge Pt(z, p^\phi) \neq \emptyset \wedge \phi o_i \in Pt(y)\}$
$delete\ x.p$	(1) $if\ \|Pt(x)\| = 1\ and\ \{\phi o_i \in @O \| \phi o_i \in Pt(x)\}$: $G - \{(< @o_i, p^\phi >, \phi o_j) \| @o_i \in Pt(x) \wedge \phi o_j \in Pt(< @o_i, p^\phi >)\}$ (2) $otherwise$: (2.1) $(G - \{(< x, p^\phi >, \phi o_i) \| \phi o_i \in Pt(< x, p^\phi >)\}) \bigcup \{(< x, p^d >, null)\}$ (2.2) $G \bigcup \{(< \phi o_i, p^* >, \phi o_j) \| \phi o_i \in Pt(x) \wedge \phi o_j \in Pt(< \phi o_i, p >)\} - \{(< \phi o_i, p >, \phi o_j) \| \phi o_i \in Pt(x) \wedge \phi o_j \in Pt(< \phi o_i, p >)\}$ (2.3) $G \bigcup \{(< z, p^* >, \phi o_i) \| z \in Alias(x) \wedge Pt(z, p) \neq \emptyset \wedge \phi o_i \in Pt(< z, p >)\} - \{(< z, p >, \phi o_i) \| z \in Alias(x) \wedge Pt(z, p) \neq \emptyset \wedge \phi o_i \in Pt(< z, p >)\}$
$x = y$	$(G - apset(x)) \bigcup \{(x, \phi o_i) \| \phi o_i \in Pt(y)\}$
$x = y.p$	$(G - apset(x)) \bigcup \{(< x, \phi o_i >) \| o_i \in lookup(y, p)\}$
$x = y.m(a_1, a_2, ..., a_n)$	$(G - apset(x)) \bigcup \{invoke(G, M, \phi o_i, a_1, a_2, ..., a_n) \| \phi o_i \in Pt(y) \wedge M \in lookup(y, m)\}$

run-time objects; however, use of in-construction objects and access path edges enable strong updates in our analysis. In the points-to graph G, if x only refers to one object and the object is an in-construction object, we know that x refers to a specific concrete object. The analysis then performs strong updates on the property reference edges by removing the points-to edges in G denoting $@o_i.p$ (if they exist) and adding the new edges implied by $Pt(y)$. In other cases (i.e., the cardinality of *Pt(x)* is more than 1 or x refers to an abstract object), we use access path edges to enable strong updates on property write statements. First, the access path of $x.p$ can be strongly updated by removing the access path edges in G denoting $x.p$ (if they exist) and adding the new edges (e.g., $(< x, p >, o_j)$ where o_j is referred to by y). Second, the object(s) x points to are weakly updated (e.g., the edge $(< o_i, p^* >, o_j)$ is inserted if x points to o_i and y points to o_j). The property reference edges are inserted with the $*$ annotation because the property write statement may not affect all variables pointing to the updated object. Last, the access path edges of the variables that have a may alias relation to x need to be weakly updated. For example, $(< z, p^* >, o_i)$ is inserted to G if z may be an alias of x, and there exists at least an edge denoting $z.p$ (with or without annotation).

In Figure 5, we show an example of the effects of a property write statement on the points-to graph. Figure 5(a) illustrates the input points-to graph for the property write statement $x.p = y$. In Figure 5(b), our analysis performs a strong update on the access path $x.p$ (i.e., delete $(< x, p >, O_4)$ and add $(< x, p >, O_2)$, $(< x, p >, O_3)$) and inserts the edges $(< O_1, p^* >, O_2)$, $(< O_1, p^* >, O_3)$ (i.e., weak updates). The updated points-graph shows that the property p *must* exist on x, while either $(< x, p >, O_2)$ or $(< x, p >, O_3)$ *may* exist.

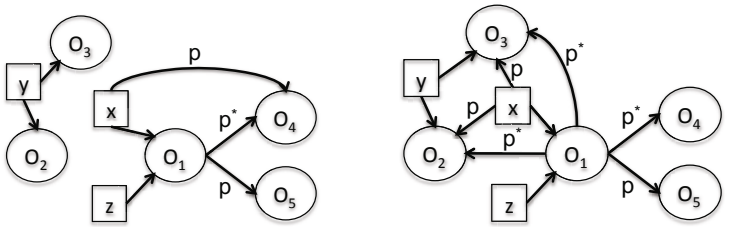


Fig. 5. Property write example. (a) Input points-to graph. (b) Updated points-to graph.

Property delete (*delete x.p*)**.** The transfer function of the delete statement is similar to the property write statement. Our analysis strongly updates the access path edges by removing the existing edges and adding a new edge (i.e., $(< x, p^d >, null)$) that denotes x must not have a local access path $x.p$. When performing weak updates on the property reference edges of an object o_i that is referred to by x, all existing edges denoting $o_i.p$ should be annotated by $*$ because

the property named p may not exist locally for o_i. The same rule applies when updating the access path edges of the aliases of x.

Direct write ($x = y$)**.** The effects of direct variable assignment on the points-to graph are relatively straightforward. $x = y$ creates points-to edges from x to all objects pointed to by y. Note that we perform weak updates on direct assignments. Although the analysis removes all the access path edges of x from the points-to graph (i.e., G - *apset(x)*), soundness is ensured because lookups through the abstract objects reflect *less precise, yet safe approximations* (see Procedure 1). Also, the access path edges of y cannot be copied to x because access path edges can only be added via strong updates.

Property read ($x = y.p$)**.** JavaScript enforces an asymmetry between reading and writing property values. When writing the value of a property or deleting a property, JavaScript always uses the local property, ignoring the prototype object. When reading a property of a variable (e.g., $x = y.p$), recall that JavaScript supports prototype-based inheritance. In some existing points-to analyses for JavaScript, when reading property p of an object, the property lookup mechanism is modeled by reporting *all* properties named p in the prototype chain of the object to ensure analysis safety.

Procedure 1. Optimized object property lookup: $lookup(v, p)$

Output: accessible objects $v.p$: P

if $Pt(< v, p >) \neq \emptyset$ **then**
  $P \cup Pt(< v, p >) \cup Pt(< v, p^* >)$
  return
else if $Pt(< v, p^* >) \neq \emptyset$ **or** $Pt(< v, p^d >) \neq \emptyset$ **then**
  $P \cup Pt(< v, p^* >)$
  for each object ϕo in $lookup(v, \_proto\_)$ **do**
    $S.push(\phi o)$
  end for
else
  for each object ϕo in $Pt(v)$ **do**
    $S.push(\phi o)$
  end for
end if
while S is not empty **do**
  $\phi o_i \leftarrow S.pop()$
  $P \cup Pt(< \phi o_i, p >) \cup Pt(< \phi o_i, p^* >)$
  if $|Pt(< \phi o_i, p >)| = 0$ **and** $(Pt(< \phi o_i, \_proto\_ >) \neq null$ **or** $Pt(< \phi o_i, \_\_proto\_\_^* >) \neq null)$ **then**
    for each object ϕo_j in $Pt(< \phi o_i, \_proto\_ >) \cup Pt(< \phi o_i, \_\_proto\_\_^* >)$ **do**
      $S.push(\phi o_j)$
    end for
  end if
end while

Procedure 1 illustrates our potentially more precise property lookup procedure enabled by our edge types and their annotations. This worklist algorithm iterates through all the accessible objects in the points-to graph when property p of variable v is read. Intuitively, it favors the use of access path edges in property lookup because they reflect the results of strong updates, before examining property reference edges. Lines 1 to 12 initialize the algorithm upon three conditions. (1) If there exist access path edges for $v.p$ without annotation (i.e., property p must exist locally), the objects in the $Pt(< v, p >)$ and $Pt(< v, p^* >)$ are considered to be accessible properties (line 2) and the algorithm returns (line 3). (2) If there exist access path edges for $v.p$ with either annotation, the algorithm needs to lookup objects in the prototype chain. In this case, the objects in the $Pt(< v, p^* >)$ (if $v.p^*$ exists) are considered to be accessible properties (line 5) and the algorithm pushes all the immediate prototype objects of v onto the worklist (lines 6 to 8). (3) Otherwise (i.e., no access path edge for $v.p$ exists), only the abstract objects are used for looking up so that all the objects in the $Pt(v)$ are pushed onto the worklist (lines 10 to 12). Lines 14 to 22 iterate the worklist. All the objects in $Pt(< \phi o, p >)$ and $Pt(< \phi o, p^* >)$ are considered to be accessible properties by our analysis (Line 16). Since an edge annotated with $*$ means that the property may not exist locally, the algorithm will continue looking up the prototype chain, until it reaches at least one points-to edge named p without annotation or the end of the prototype chain (Line 17 to 21). Thus, instead of finding *all* the properties named p in the prototype chain (i.e., *lookup_all(v, p)*), our algorithm can stop when it finds an *existing* property p (i.e., a property named p without annotation).

This new property lookup algorithm *lookup(v, p)* mimics the run-time property lookup mechanism of JavaScript while still assuring the safety of our analysis. For the example in Figure 5(b), *lookup(z, p)* results in O_2 and O_3 through the access path while *lookup(x, p)* results in O_2, O_3, O_4 and O_5 through the abstract object O_1. In Table 2, the transfer function of the property read statements refers to this optimized object property lookup algorithm. Because we perform weak updates on the property read statements, similar to direct writes, the analysis removes all the access path edges of x from the points-to graph to ensure safety.

Method invocation (x = *y.m(*a_1*,* a_2*, ...,* a_n*)*)**.** The method invocation (e.g., x = *y.m(*a_1*,* a_2*, ...,* a_n*)*) resolves for every receiver object pointed to by y. The invoked methods are determined by reading the property $y.m$ through our optimized lookup algorithm. Upon the return of method invocation, x is weakly updated by removing all its access path edges from G.

3.4 State Sensitivity

State sensitivity for JavaScript is a new form of context sensitivity derived from the notion of object sensitivity for languages such as Java [18]. In object sensitivity, each method is analyzed separately for each object on which it may be invoked. For strongly typed languages like Java, often object sensitivity

identifies objects in the analysis by their creation sites. Calls of a method using two different receiver objects (i.e., created at different sites) will result in two separate analyses of the method, even if the calls originated from the same call site. However, this is insufficient for JavaScript analysis, because object behavior may change dynamically at any program point during execution.

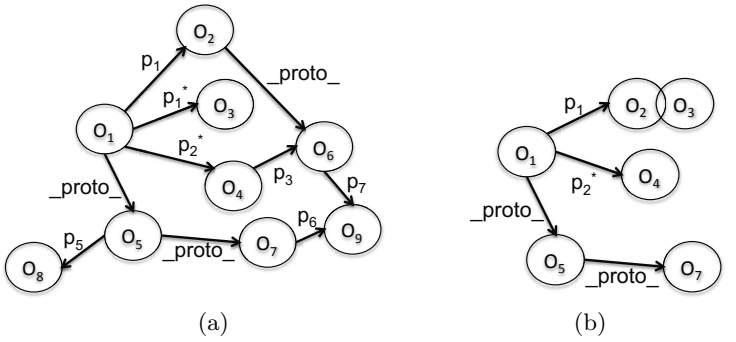


Fig. 6. Approximate *obj-ref state* as a context. (a) *obj-ref state* of O_1. (b) Approximate *obj-ref state* of O_1.

Ideally, state sensitivity would analyze each method separately for each *obj-ref state* on which it may be invoked. However, the graph representation of *obj-ref state* may contain many edges and nodes both locally and along prototype chains (e.g., *obj-ref state* of O_1 in Figure 6(a)), which would be prohibitively expensive to use as a context. Therefore, we use an *approximation* of the *obj-ref state* of the receiver object to differentiate calls that will be analyzed separately. Our approximation consists of the object, its local properties and their object values plus its chain of prototype objects. In Figure 6(b) we show the approximation corresponding to the *obj-ref state* of object O_1 in Figure 6(a). Note that the edges with the same local property name (annotated and not annotated) in the points-to graph are merged in the approximate *obj-ref state* (e.g., $(< O_1, p_1 >, O_2)$ and $(< O_1, p_1^* >, O_3)$ in Figure 6(a)). An object-sensitive analysis groups the calls using a receiver object created at the same allocation site and our state-sensitive analysis more accurately groups the calls where receiver objects have the same approximate *obj-ref state*. We intend to study the effects of using different *obj-ref state* approximations as calling contexts in future work.

3.5 Block-Sensitive Analysis

Our new points-to analysis algorithm is a fixed point calculation on the call graph, initialized with an empty points-to graph on entry to the JavaScript program, in which every constitutent *SPBG* is traversed in a flow-sensitive manner. Essentially, we have designed the points-to algorithm to emphasize precision for the *obj-ref state* information in the points-to graph and the *SPBG* to hide control flow not relevant to reference state updates.

Table 3. Union rules. (a) Access path edges union rules. (b) Property reference edges union rules.

(a)

$\bigcup$	$\emptyset$	$v.p^d$	$v.p^*$	$v.p$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$v.p^d$	$\emptyset$	$v.p^d$	$v.p^*$	$v.p^*$
$v.p^*$	$\emptyset$	$v.p^*$	$v.p^*$	$v.p^*$
$v.p$	$\emptyset$	$v.p^*$	$v.p^*$	$v.p$

(b)

$\bigcup$	$\emptyset$	$o.p^*$	$o.p$
$\emptyset$	$\emptyset$	$o.p^*$	$o.p^*$
$o.p^*$	$o.p^*$	$o.p^*$	$o.p^*$
$o.p$	$o.p^*$	$o.p^*$	$o.p$

More specifically, our analysis solves for the points-to graph on exit of each *SPBG* node. The transfer function for a node in the *SPBG* is one of two kinds: (1) for a state-update node perform strong update of the changed property, if possible (as in Table 2), or (2) for a state-preserving node perform a flow-insensitive analysis of the statements in that node, using an initial points-to graph (IN) and storing the fixed point reached in points-to graph OUT.

Normally in a points-to analysis, we would form IN as a union of the OUT points-to graphs of predecessors of a node. In our algorithm, we need to maintain the invariant of our annotated property edges, namely that a property name without an annotation means that property exists and a property name with the d annotation means that property must not exist.

Table 3 shows the union rules for the access path edges and property reference edges when two points-to graphs are unioned. For the access path edges: (1) if access path $v.p^\phi$ does not exist in at least one predecessor, then $v.p^\phi$ does not exist after union; (2) if $v.p^d$ or $v.p$ exists in both predecessors, then $v.p^d$ or $v.p$ respectively exists after union; (3) otherwise, $v.p^*$ exists after union. For the property reference edges: (1) if $o.p$ exists in both predecessors, then $o.p$ exists after union; (2) otherwise, if $o.p$ or $o.p^*$ exists in at least one predecessor, then $o.p^*$ exists after union. These rules ensure analysis safety when property lookup is performed.

3.6 Implementation of State-Sensitive Analysis in *JSBAF*

Our new points-to analysis was implemented with a client as the static component of the *JavaScript Blended Analysis Framework* (*JSBAF*), a general-purpose analysis framework for JavaScript [28]. This framework was designed to strongly couple dynamic and static analyses to account for the effects of the dynamic features of JavaScript. We chose this implementation platform because blended analysis has been demonstrated to be more efficient and effective in analyzing real JavaScript programs.[8]

JSBAF can be applied to analyze a JavaScript program (i.e., JavaScript code on a webpage) automatically in the presence of a good test suite. The dynamic phase gathers run-time information by executing tests. A trace of each test

[8] A static analysis was not able to finish analyzing most webpages in [28].

is collected, including call statements, object creations, variadic function calls with parameters, and dynamically created code. The implementation separates each trace into its constituent page traces. Each subtrace on a page is analyzed separately in the static phase. Data-flow solutions from different page subtraces are combined into a entire solution for that page.

Blended analysis uses only the observed calling structure as a basis to model the JavaScript program. Knowledge of unexecuted calls or object creations can be used to prune other unexecuted code sharing the same control dependence. For example, knowing $Y3()$ is not called at line 5 in Figure 2, blended analysis prunes this unexecuted statement so that the imprecise node O_5 and its connected edges will not be created. Thus, blended analysis is unsafe because not all executions are explored, but sound on the observed executions.

Our points-to algorithm was implemented on the *IBM T.J. Watson Libraries for Analysis (WALA)* open-source static analysis framework[9] which contains several existing static points-to analysis algorithms. *WALA* has been extended to enable blended analysis by providing dynamic information (i.e., a run-time collected call graph, dynamically generated code, object creation sites) [28].

Our algorithm takes as inputs the run-time collected calling structure (i.e., call graph[10]) and source code including dynamically generated code. Code pruning was performed on function bodies so that the code in polymorphic constructors and variadic functions was specialized. Hence, constructor polymorphism was handled by our improved object representation combined with dynamic information (i.e., objects created at the same allocation site with different sets of property names are represented as separate abstract objects).

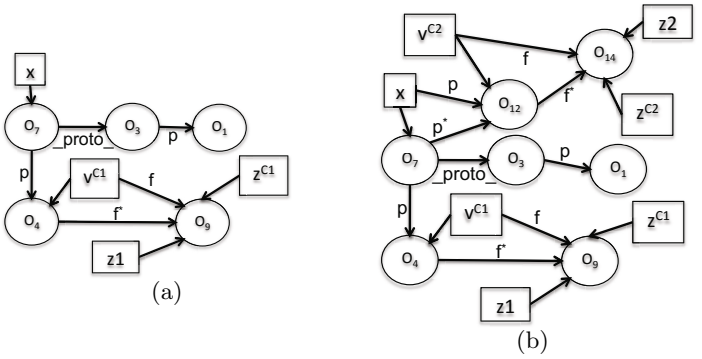


Fig. 7. Blended state-sensitive points-to analysis. (a) points-to graph at line 10. (b) points-to graph at line 15.

Example. In comparison to the inaccurate points-to solution of a flow- and context-insensitive analysis for the JavaScript code in Figure 2, we now

[9] `http://wala.sourceforge.net/`

[10] Each node in the call graph is associated with the object creations observed during its execution.

demonstrate the results of our state-sensitive points-to analysis in the context of blended analysis. Figures 7(a) and 7(b) show the points-to graphs obtained at lines 10 and 15, respectively. Because blended analysis executes the program and does not observe an object created by the constructor *Y3*, the code at line 5 is pruned so that our analysis does not generate the inaccurate node O_5 nor the edge $(< O_7, q >, O_5)$. For the call statement at line 10, our points-to analysis calculates the *obj-ref state* approximation of O_7, namely $C1$: {O_7, p:O_4, *_proto_*: O_3}. Also, when looking up $x.p$ at line 10, our algorithm returns O_4 because there is no annotation on the property reference edge so that further lookup through the prototype chain is not necessary. Note that the points-to graph in Figure 7(a) is as precise as the run-time points-to graph (Figure 3(a)).

At line 12, $x.p$ is strongly updated via the access path edge $(< x, p >, O_{12})$. For the call statement at line 15, our points-to analysis calculates the *obj-ref state* approximation of O_7, $C2$: {O_7, p:[O_4, O_{12}], *_proto_*: O_3}. Our points-to algorithm distinguishes this call site from line 10 because O_7 has a different *obj-ref state* here. The lookup of $x.p$ at line 15 follows the access path edge so that the node O_{12} is returned. Thus, in this example our analysis results in none of the inaccurate edges in the flow- and context-insensitive analysis (Figure 3(c)) and reflects the actual run-time behavior of JavaScript objects (Figure 3(b)).

4 Evaluation

In this section, we present experiments using *JSBAF* with our state-sensitive points-to analysis compared to an existing points-to analysis [26], evaluating both with a *REF* client.

4.1 Experimental Design

REF **Analysis.** To evaluate the precision and performance of our points-to analysis, we implemented a JavaScript reference analysis (*REF*). The *REF* client calculates the set of objects returned by property lookup at a property read statement (i.e., $x = y.p$) or call statement (i.e., $x = y.p(...)$).[11] For each of these statements s in a function being analyzed in calling context c, we compute $REF(s, c)$, the set of objects returned by a property lookup for each $o.p$ where o is pointed to by y. The cardinality of the *REF* set depends on the precision of the points-to graph and the property lookup operation; the smaller the set returned, the more useful for program understanding, for example.

In Figure 2, assume we add the function property

$$x.foo = function(){var\ a = this.p;\ return\ a;}$$

Effectively, $foo()$ returns the property lookup result for $this.p$. If $x.foo()$ is called at line 11 before the property update statement *x.p=new A()*, it will return O_4.

[11] All source code instances of property lookups (e.g., *return y.p*) occur as one of these two statements in the *WALA* intermediate code.

If $x.foo()$ is called at line 13 after *x.p=new A()*, it will return O_{12}. For an analysis that is flow-insensitive or that cannot distinguish these call sites by calling context, the return value of each of these function calls will contain at least two objects (i.e., O_4 and O_{12}).

Comparison with Points-to Analysis in [26]. We use the term *Corr* to refer to a blended version of correlation-tracking points-to analysis [26] (see Section 5 for more details) and its *REF* client. To demonstrate the additional precision of our analysis over *Corr*, we applied the correlation extraction transformation to our JavaScript benchmarks before performing our points-to analysis. We use the term *CorrBSSS* to refer to a blended version of this augmented new points-to analysis and its *REF* client. For each algorithm, an object property lookup returns a *REF* set whose cardinality $|REF(s, c)|$ is calculated. For *Corr*, the *lookup_all*() approximate algorithm described in Section 3.3 is used. For *CorrBSSS*, we use our optimized lookup algorithm *lookup*() in Procedure 1.

Benchmarks. We conducted the experiments with the benchmarks collected from 12 websites among the top 25 most popular sites on *alexa*, reusing website traces originally used in [28]. The results in [28] showed that the collected traces covered a large portion of the executable JavaScript code in those websites, including dynamically generated code. Although the benchmarks we used cover the most popular websites, it will require further investigation to determine how representative these benchmarks are of other websites. The experimental results were obtained on a 2.53GHz Intel Core 2 Duo MacBook Pro with 4GB memory running the Mac OS X 10.5 operating system.

4.2 Experimental Results

Improved *REF* Precision. Table 4 shows the *REF* client results for the 12 websites. Columns 2-4 present the results for *Corr* and columns 5-7 present the results for *CorrBSSS*. For each website, columns 2 & 5, 3 & 6, and 4 & 7 in Table 4 correspond to the percentage of property lookup statements that return 1 object, 2-4 objects, and more than 4 objects, respectively. The result shown for each website is averaged over the corresponding percentage numbers for all the webpages in that domain; for example, the 38% entry for *facebook.com* in column 2 is the average for *Corr* over the 27 webpages analyzed of the percentage of property lookup statements returning only 1 object.

Comparing columns 2-4 with 5-7 in Table 4 for each website, we see the relative precision improvement of *CorrBSSS* over *Corr*. For *REF* analysis, the best result is that the lookup returns only one object and the property lookup is more precise if the number of objects returned is smaller. On average over all the websites, *Corr* reported **37%** of the property lookup statements were resolved to a single object, while *CorrBSSS* improved this metric to **48%**, *a significant improvement*. In addition, *REF* analysis results may become too approximate to be useful if too many objects are returned. Although **15%** of the statements on average returned more than 4 objects for *Corr*, *CorrBSSS* reduced that number to **7%**.

Table 4. *REF* analysis precision

Website	*Corr*			*CorrBSSS*		
	1	2-4	≥ 5	1	2-4	$\geq$5
facebook.com	38%	52%	10%	50%	47%	3%
google.com	32%	51%	17%	53%	42%	5%
youtube.com	41%	47%	12%	54%	41%	5%
yahoo.com	48%	46%	6%	52%	45%	3%
wikipedia.org	29%	45%	26%	43%	39%	18%
amazon.com	45%	52%	3%	46%	51%	3%
twitter.com	32%	53%	15%	39%	49%	12%
blogspot.com	35%	34%	31%	53%	36%	11%
linkedin.com	34%	49%	17%	44%	50%	6%
msn.com	40%	36%	24%	48%	37%	15%
ebay.com	30%	40%	30%	46%	40%	14%
bing.com	41%	34%	25%	54%	37%	9%
Geom. Mean	**37%**	**44%**	**15%**	**48%**	**43%**	**7%**

Table 5. *REF* analysis cost (in seconds) on average per webpage

Website	*Corr*	*CorrBSSS*	overhead
facebook	17.4	45.9	163%
google	13.0	30.4	134%
youtube	31.2	75.3	141%
yahoo	28.5	54.1	90%
wiki	16.0	40.1	151%
amazon	15.1	24.2	61%
twitter	38.1	94.5	148%
blog	15.9	42.4	137%
linkedin	27.8	62.0	167%
msn	34.4	57.9	68%
ebay	8.3	27.2	227%
bing	22.1	50.4	128%
Geom. Mean	**20.4**	**46.7**	**127%**

These improved precision results indicate the potential for greater practical use of state-sensitive points-to information by client analyses.

We also investigated the average number of objects returned by a property lookup statement. For each website, we calculated the number of objects per statement on average over all its webpages. Over all the benchmarks, *Corr* produced on average **2.8** objects and *CorrBSSS* only reported on average **2.3** objects. Intuitively, this means that on average fewer objects at each property lookup statement must be examined to gain better understanding of the code.

REF **Performance.** An analysis approach is practical if it scales to real-world programs, such as JavaScript code from actual websites. Because *CorrBSSS* is partially flow-sensitive and context-sensitive, it is important to demonstrate that this analysis is scalable. Table 5 shows the time performance of *Corr* versus *CorrBSSS*.[12] Columns 2 and 3 present the average webpage analysis time for each website, averaging over all of its webpages. Both *Corr* and *CorrBSSS* completely analyzed all the benchmark programs. On average over all the websites, *Corr* completely analyzed a webpage in **20.4** seconds, while *CorrBSSS* did so in **46.7** seconds, incurring an 127% average time overhead per webpage, acceptable for a research prototype implementation which has not been optimized.

Discussion. We collected data characterizing benchmark program structure and complexity to relate these characteristics to observed analysis precision and performance. The entries in Table 6 all represent averages per webpage that are averaged over an entire website. Column 2 shows the average number of functions in a JavaScript program. Column 3 shows the percentage of functions containing at least one state-update statement. Column 4 shows the percentage of statements that are state-update statements. Column 5 shows the number of contexts produced by *CorrBSSS* as a multiplier for column 2. On average over all the websites, **9%** of the functions contained local state-update statement(s); these averages ranged from **4%** for *yahoo.com* to **18%** for *msn.com*. This suggests that *the state-update statements are localized in a relatively small portion of the JavaScript program (e.g., in constructor functions)*. Manual inspection of several websites (i.e., *facebook*, *google* and *youtube*) revealed there were significant object behavior changes in JavaScript code outside of constructors. On average over all the websites, **8%** of the statements were identified as state-update statements. The relatively small number of state-update statements means that our *SPBG* contained many fewer nodes than the corresponding CFGs; therefore the flow-sensitive analysis was more practical in cost on the *SPBG*s.

Now we compare the analysis precision observed in Table 4 with the number of contexts generated on average per function per page (column 5 in Table 6) to observe the effect of state sensitivity. *google*, *blog*, and *ebay* were the websites for which *CorrBSSS* improved precision the most, whereas *amazon*, *yahooo*, *twitter*, and *msn* were the websites for which *CorrBSSS* produced similar results to *Corr*. For the former websites, *CorrBSSS* generated the greatest number of contexts per function per webpage. For the latter websites, *CorrBSSS* generated the fewest. We observe strong correlation between the precision gain and the number of contexts generated by *CorrBSSS*, demonstrating that *state sensitivity significantly increased analysis precision on these benchmarks*, and suggesting that state sensitivity will be an effective form of context sensitivity for JavaScript analysis.

[12] The time cost in Table 5 reflects the performance of the static phase of blended analysis. In the experiments, the dynamic phase of *Corr* and *CorrBSSS* is the same for both analyses. The work in [28] has demonstrated that the static phase dominates the blended analysis cost.

Table 6. Benchmark and context statistics. (Total contexts per website is approximately column 2 times column 5.)

Website	No. of functions	% of functions w/ update(s)	% of state-update stmt	No. of contexts
facebook	2123	9%	8%	4.0
google	1002	17%	6%	6.7
youtube	1329	7%	6%	3.9
yahoo	3810	4%	4%	2.4
wiki	270	10%	19%	4.8
amazon	729	6%	6%	1.9
twitter	618	15%	5%	3.4
blog	583	14%	14%	6.1
linkedin	920	8%	11%	3.6
msn	1537	8%	8%	2.8
ebay	581	18%	13%	7.5
bing	1131	7%	11%	4.9
Geom. Mean	**972**	**9%**	**8%**	**4.0**

As shown in Table 5, the *CorrBSSS* time overhead differed significantly for different websites, from 61% (*amazon.com*) to 227%(*ebay.com*). We investigate several program characteristics to reason about such differences. First, the *SPBG*s created by *CorrBSSS* determine the efficiency of the flow-sensitive analysis. On average over all the websites, an *SPBG* was comprised of about **6** nodes, explaining why *CorrBSSS* scaled on real websites. Functions with large numbers of nodes in their *SPBG* usually contained multiple state-update statements and complex control flow. The largest number of nodes for an *SPBG* was **23** in *linkedin*. Second, the websites with the least performance overhead from *CorrBSSS* were *amazon*, *msn* and *yahoo*. These websites contained a relatively small percentage of update statements (i.e., all below average) and *CorrBSSS* generated the lowest number of contexts for them. The website that incurred the most overhead (i.e., *ebay*) contained **13%** update statements, (i.e., the third highest percentage in our benchmarks), and the greatest number of contexts per function (i.e., **7.5**) generated by *CorrBSSS*. These results support the reasoning that more complex block structure and more context comparisons contribute to the higher overhead for *CorrBSSS*.

5 Related Work

Due to space limitations, we present only the work most closely related to our state-sensitive points-to algorithm.

Related Analyses of JavaScript. Several approaches were proposed to analyze JavaScript programs. Sridharan *et al.* presented a points-to analysis for JavaScript that focused on handling correlated dynamic property accesses [26].

Correlated property accesses were identified and then extracted into a function. Using the property name as the calling context, points-to analysis tracking correlation was shown to be more precise and efficient than a field-sensitive Andersen's points-to analysis. In our experiments, *CorrBSSS* was augmented by correlation analysis (i.e., *Corr*) demonstrating a significant improvement in the analysis precision.

Jensen *et al.* presented a static analysis that can precisely model prototype chains [12]. In their analysis, the *absent* set indicated potentially missing properties. The property edges annotated with * play a similar role in our analysis. Jensen's analysis is context-sensitive similar to 1-object-sensitivity used in Java [18]. The static flow-sensitive analysis presented in [12] was not scalable on large JavaScript programs, whereas our experiments showed the *CorrBSSS* was practical for blended analysis of real-world websites.

Several points-to analyses were proposed to handle other important challenges introduced by JavaScript. Guarnieri and Livshits designed a points-to analysis to detect security and reliability issues in JavaScript widgets [8]. They used a subset of JavaScript language, $JavaScript_{SAFE}$, that can be statically approximated. Guarnieri *et al.* presented a static taint analysis based on a points-to analysis finding security vulnerabilities in real-world websites [10]. The points-to algorithm focused on addressing features of JavaScript including object creations and accesses through constructed property names. In these analyses, prototyping was modeled as *lookup_all* rather than our more accurate property lookup algorithm *lookup*. The points-to analysis in [10] and our algorithm are both implemented in *WALA*.

A hybrid analysis (i.e., a combination of static and dynamic analyses) is attractive when analyzing JavaScript programs. Chugh *et al.* presented a staged information flow analysis for JavaScript [4]. The approach analyzed the static code and incrementally analyzed the dynamically generated code. A similar approach was proposed by Guarnieri and Livshits [9]; their experiments studied the performance of the incremental analysis. In addition to collecting the dynamically generated/loaded code, blended analysis uses run-time information to make the analysis more precise (e.g., polymorphic constructors are distinguished via object initialization and some unexecuted code is pruned).

Several type-based points-to techniques have been proposed for JavaScript that support dynamic features such as prototype-based inheritance (e.g., [3,5,14]). It is difficult to compare our analysis with them as to practicality, because no empirical evidence on large JavaScript programs was presented.

Software tools supporting large JavaScript software including libraries are desirable. Schafer *et al.* provided an IDE support for JavaScript programming [22]. Points-to analysis was used to calculate code completion suggestions. Points-to analysis precision is crucial to determine the effectiveness of the tool. Madsen *et al.* presented a static analysis of JavaScript focusing on frameworks and libraries [17]. A novel use analysis was proposed to analyze libraries precisely and a points-to analysis was used to find aliases in the program. Our work is

complementary to these techniques, in that more precise points-to results would make them more practical.

Context-Sensitive Analysis. Our state-sensitive analysis is inspired by object sensitivity. Milanova *et al.* first introduced object sensitivity and implemented an object-sensitive points-to analysis for Java using a receiver object represented by its creation site as the calling context [18]. The experiments in [15] showed object sensitivity is the better choice as a calling context when analyzing an object-oriented language. Changes to object properties in JavaScript render object creation sites insufficient to represent object behavior, whereas state sensitivity captures object behavior changes better.

Smaragdakis *et al.* formalized object sensitivity, summarizing its variations [25]. They introduced type sensitivity where object type was used as the calling context. For dynamically-typed languages like JavaScript, type is a runtime notion, encapsulated in the idea of *obj-ref state* used as a calling context. Kastrinis *et al.* presented a hybrid context-sensitive analysis that combined object-sensitivity and call-site-sensitivity [13]. Hybrid context-sensitive analysis for JavaScript is planned for our future work.

6 Conclusion

JavaScript object behavior is difficult to analyze well because of prototype-based inheritance and allowed changes to object properties during execution. In this paper, we introduced a state-sensitive points-to analysis that models object behavior changes accurately by using a hierarchical program representation emphasizing state-update statements, by defining state sensitivity, a better context sensitivity mechanism for a dynamic language, and by enhancing the points-to graph representation for improving object property lookups. We implemented our new points-to algorithm as the static phase of a blended analysis in *JSBAF*. Experimental results on a *REF* client showed our analysis, *CorrBSSS*, significantly improved the precision of a previous good JavaScript points-to analysis *Corr* [26]. For example, 48% of property lookups were resolved to a single object by our analysis versus 37% by *Corr*. Although our research prototype implementation incurred on average 127% overhead versus *Corr* on the popular website benchmarks used, further optimization will improve the performance, which is in the practical range.

In future work, we intend to investigate variations of state sensitivity, to study the effects of different *obj-ref state* approximations on analyzing JavaScript programs. We are interested in exploring the capability of state-sensitive analysis to support program understanding. We also plan to generalize the proposed techniques to other dynamic programming languages.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison Wesley (1986)

2. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
3. Chugh, R., Herman, D., Jhala, R.: Dependent types for JavaScript. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, pp. 587–606 (2012)
4. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for JavaScript. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 50–62 (2009)
5. Chugh, R., Rondon, P.M., Jhala, R.: Nested refinements: a logic for duck typing. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 231–244 (2012)
6. De, A., D'Souza, D.: Scalable flow-sensitive pointer analysis for Java with strong updates. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 665–687. Springer, Heidelberg (2012)
7. Flanagan, D.: JavaScript: The Definitive Guide. O'Reilly Media, Inc. (2006)
8. Guarnieri, S., Livshits, B.: Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code. In: Proceedings of the 18th Conference on USENIX Security Symposium, pp. 151–168 (2009)
9. Guarnieri, S., Livshits, B.: Gulfstream: staged static analysis for streaming JavaScript applications. In: Proceedings of the 2010 USENIX Conference on Web Application Development, p. 6 (2010)
10. Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., Berg, R.: Saving the world wide web from vulnerable JavaScript. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 177–187 (2011)
11. Heidegger, P., Thiemann, P.: Recency types for analyzing scripting languages. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 200–224. Springer, Heidelberg (2010)
12. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for javaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009)
13. Kastrinis, G., Smaragdakis, Y.: Hybrid context-sensitivity for points-to analysis. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 423–434 (2013)
14. Lerner, B.S., Joe Gibbs, P., Guha, A., Shriram, K.: TeJaS: Retrofitting type systems for JavaScript. In: Proceedings of the 9th Symposium on Dynamic Languages (2013)
15. Lhoták, O., Hendren, L.: Context-sensitive points-to analysis: Is it worth it? In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 47–64. Springer, Heidelberg (2006)
16. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: Conference proceedings on Object-Oriented Programming Systems, Languages and Applications, pp. 214–223 (1986)
17. Madsen, M., Livshits, B., Fanning, M.: Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 499–509 (2013)
18. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM TOSEM 14(1), 1–41 (2005)
19. Orion, `http://www.eclipse.org/orion/`
20. Richards, G., Lebresne, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of JavaScript programs. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1–12 (2010)

21. Ryder, B.G.: Dimensions of precision in reference analysis of object-oriented programming languages. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 126–137. Springer, Heidelberg (2003)
22. Schafer, M., Sridharan, M., Dolby, J., Tip, F.: Effective smart completion for JavaScript. Technical Report RC25359, IBM (2013)
23. Sethi, R.: Programming Languages, Concepts & Constructs, 2nd edn. Addison Wesley (1996)
24. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, pp. 189–234 (1981)
25. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 17–30 (2011)
26. Sridharan, M., Dolby, J., Chandra, S., Schäfer, M., Tip, F.: Correlation tracking for points-to analysis of JavaScript. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 435–458. Springer, Heidelberg (2012)
27. Wegner, P.: Dimensions of object-based language design. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, pp. 168–182 (1987)
28. Wei, S., Ryder, B.G.: Practical blended taint analysis for JavaScript. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, pp. 336–346 (2013)