这个论文的问题：
1、如何定位造成精度不好的地方
2、实验为什么要使用那三个指标，指标的值代表什么
3、有什么问题

Revamping JavaScript Static Analysis via Localization and Remediation of Root Causes of Imprecision

Shiyi Wei
University of Maryland
USA
wei@cs.umd.edu

Omer Tripp*
Google
USA
trippo@google.com

Barbara G. Ryder
Virginia Tech
USA
ryder@cs.vt.edu

Julian Dolby
IBM Research
USA
dolby@us.ibm.com

ABSTRACT

Static analysis is challenged by the dynamic language constructs of JavaScript which often lead to unacceptable performance and/or precision results. We describe an approach that focuses on improving the practicality and accuracy of points-to analysis and call graph construction for JavaScript programs. The approach first identifies program constructs which are sources of imprecision (i.e., root causes) through monitoring the static analysis process. We then examine and suggest specific context-sensitive analyses to apply. Our technique is able to to find that the root causes comprise less than 2% of the functions in JavaScript library applications. Moreover, the specialized analysis derived by our approach finishes within a few seconds, even on programs which can not complete within 10 minutes with the original analysis.

CCS Concepts

•Software and its engineering → **Software testing and debugging; •Theory of computation** → *Program analysis;*

Keywords

JavaScript; program analysis

1. INTRODUCTION

Dynamic programming languages such as JavaScript are now in widespread use for both client-side and server-side applications, often together with cloud services and/or mobile devices. The flexibility of these languages, for example for building prototypes, is key to their popularity, but their dynamic nature presents real challenges to static program analyses. These challenges affect analyses used to ensure the security of applications (e.g., [5, 6]), to optimize code for good performance (e.g., [8]) and to aid program understanding (e.g., [4]). For example, JavaScript allows object property accesses as associative arrays where the property name may be set as a result of execution. Making assumptions more accurate than worst-case here may be difficult. Moreover, JavaScript supports multiple programming paradigms including object-oriented, functional and procedural programming. Each programming paradigm requires specialized techniques for accurate analysis [23]. These are examples of how JavaScript features render accurate static analysis very difficult for many real-world programs.

The effectiveness of a static analysis often is evaluated as a combination of its precision (i.e., low false positive rate) and performance (i.e., efficiency in a limited time/space budget). With complex, medium-sized JavaScript programs, especially those using real-world libraries such as *jQuery*, static analysis cannot reach a useful solution within a reasonable time budget despite recent progress on improving the state-of-the-art (e.g., [20, 2, 23, 14]). Often it is even difficult to obtain a good approximation of the program call graph, which is a foundation of inter-procedural static analysis and useful for tasks such as dead-code elimination [12].

We present a new approach to tackle this problem for JavaScript programs. By applying an imprecise static call graph construction and points-to analysis algorithm to the program, extra information about points-to propagation is gathered in the points-to graph under construction.[1] A heuristic process observes the analysis propagation phase in order to capture anomalous behavior (i.e., when the analysis is becoming too approximate through propagation of inaccuracy). A diagnosis algorithm is applied to trace this "bad" behavior back to its root causes linked to specific functions. By applying a well-chosen context sensitivity policy to use on these functions during a fresh analysis pass, the anomalous behavior may be circumventable. This process utilizes dynamic analysis results in addition to the static analysis self-inspection to help choose the kinds of context sensitivity to propose. We call this entire process *root-cause localization and remediation*, and it is crucial for designing effective, new static analysis algorithms for JavaScript as well as for tuning existing analysis algorithms.

More specifically, our systematic support for root-cause localization and remediation focuses on points-to analysis, an enabling static analysis for various automated software tools. When confronted with an unscalable and/or too imprecise static points-to analysis on a target program, we keep

*The author was employed by IBM Research when the paper was written.

[1] Note that JavaScript points-to analysis and call graph construction are often interleaved. The accuracy of the points-to graph directly determines the accuracy of the call graph.

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950338

track of the history information in the propagation system, labeling the origins of the points-to relations. While examining the analysis propagation as it is performed, heuristics are applied to decide when to identify the sources of imprecision (i.e., when the analysis result begins to diverge). Our automatic root-cause identification uses the intermediate points-to results and the history labels to infer the variables and/or reference properties that have big impact on overall analysis precision and performance as the *root causes*.

In addition, we have designed an automatic approach that suggests specialized refinements to improve analysis precision for these root causes. Using a dynamic trace of program execution, we build a set of dynamic points-to graphs using various kinds of context sensitivity. The idea is that the dynamic points-to graphs can simulate the possible effects of applying a particular kind of context sensitivity to the root-cause function in the re-started analysis.

The automatic root-cause localization relieves a static analysis designer from the chores of manually inspecting the program and the analysis implementation to understand the sources of imprecision. Moreover, the automatic improvement suggestion provides possible context-sensitive analysis choices that may significantly improve the overall analysis performance and precision. The specialized analysis configurations derived from the results of our approach, with possible adjustment from the static analysis designer, can be executed on the same program to observe if the performance and/or precision issues have been resolved. If necessary, the same process can be iteratively performed to locate sufficiently many of the sources of imprecision in the analysis on the target program to achieve the desired accuracy and performance. We have conducted experiments to evaluate the accuracy of the automatic root-cause location and the subsequent improvement suggestions on real-world JavaScript libraries and applications.

The major contributions of this work are:

- We present the *first* research that focuses on supporting static analysis design with automatic root-cause localization, identifying the sources significantly affecting analysis precision and performance.

- Our approach is the *first* to use dynamic information to automatically suggest the kind of context sensitivity needed for significant precision improvement on identified root causes of analysis inaccuracy.

- We present an evaluation of the proposed approaches on several benchmarks. The experimental results on JavaScript library applications demonstrate that our root-cause localization algorithm accurately identifies the program constructs that cause an initial static analysis to not finish in a 10 minute time allotment. Applying specialized context sensitivity on these constructs significantly improves the analysis performance. The results on JavaScript benchmarks also show that an analysis that *selectively* applies the recommended context sensitivity from our automatic improvement suggestion achieves a much better balance between precision and performance compared both to an imprecise analysis and a more precise analysis that applies context sensitivity over the entire program.

```
1  function extendBasic(target, source) {
2      var name;
3      target = target || {};
4      for (name in source) {
5          target[name] = source[name];
6      }
7      return target;
8  }
```

Figure 1: Modifed `extend` function of jQuery 1.6.1.

2. BACKGROUND & MOTIVATION

In this section, we first introduce JavaScript points-to analysis with an example. We then discuss an empirical study of interesting static analysis behaviors that guided our design. Finally, we illustrate our approach using a code example.

2.1 Background

Points-to analysis approximates the program's heap by calculating the set of abstract values a variable or reference property may have during execution. Context sensitivity is a general technique to achieve more precise program analysis by distinguishing between calls to a function [16]. It has been demonstrated that applying specialized context sensitivity in JavaScript points-to analysis is an effective approach for improving its precision and performance. For example, Sridharan et al. [20] used the values of a parameter `p` of a function as the calling context, if `p` is the property name in a property access (e.g., `v[p]`), a special treatment for dynamic property accesses in JavaScript. Because this algorithm is designed to address the challenges caused by a specific language feature and performs well for the programs where this feature is present, other program constructs found in real-world JavaScript applications that require more accurate handling may render the analysis ineffective.

Therefore, an important stage of static analysis design is to identify the causes of unexpected results. Intuitively, a *root cause* is a program construct that is a source of the precision and/or performance loss for a static analysis. Specifically, if the overall precision and performance of an analysis `A` improves significantly via specialized handling of the program construct in a specific program location, this construct is the root cause of imprecision of the analysis `A` for the program. For example, Figure 1 shows a small piece of code from *jQuery*, the most widely used JavaScript library [21]. A whole-program 1-CFA analysis [17] that separately analyzes each different call site of a function has scalability problems for any simple application that uses *jQuery* [20]. Applying the technique proposed by Sridharan et al. [20] only to the property accesses at line 5 resolves the performance issues. Therefore, this program construct is a root cause of imprecision of 1-CFA analysis for *jQuery* applications.

Unfortunately, identifying root causes is a costly process, requiring extensive experience in designing static analyses as well as a deep understanding of the target program. To the best of our knowledge, there is little tool support for this process, making it time-consuming and unprincipled. For example, identifying the property accesses at line 5 in Figure 1 as the root cause for 1-CFA analysis is difficult because (i) the *jQuery* library consists of about 9,000 lines of code, and (ii) similar program constructs are used throughout the

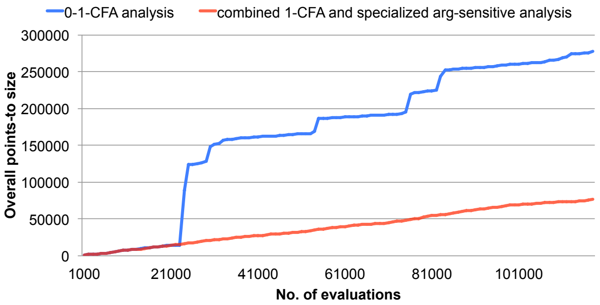

Figure 2: Points-to size growth during the analysis lifetime.

jQuery library with less impact on the overall analysis results, whereas this particular program location is critical. Therefore, we are motivated to develop new techniques to assist in *root-cause localization* by automating a significant part of the process of identifying the sources of imprecision.

2.2 Static Analysis Behavior

We have performed a brief empirical study to understand the behavior of JavaScript static analysis. Figures 2 and 3 show the different behaviors of two points-to analyses of a simple application that uses *jQuery*. The two points-to analyses in comparison are the 0-1-CFA analysis (i.e., only use 1-CFA analysis for the constructors to name objects by their allocation sites) and the combined 1-CFA and specialized argument-sensitive analysis [20]. Overall, the 0-1-CFA analysis experiences performance and precision issues (e.g., the analysis cannot finish analyzing the program within a time budget of 10 minutes), while the combined context-sensitive analysis performs significantly better.

Figure 2 shows the trend of overall points-to size (i.e., total number of points-to edges) growth for these two analyses during their lifetimes. The x axis presents the number of evaluations[2] and y axis presents the total points-to size of all variables in the program. The points-to size of the good combined context-sensitive analysis grows steadily "linear" throughout its lifetime. On the other hand, the overall points-to size growth of 0-1-CFA analysis exhibits "jumps", periods during which its overall points-to size dramatically increases. For example, between rounds 23,000 and 25,000 of evaluation, the overall points-to size of the 0-1-CFA analysis grows about ten times. The existence of such "jumps" indicates that the overly-approximated results are frequently propagated, resulting in significant overall precision loss. In addition, since the 0-1-CFA analysis experiences scalability issues, it remains incomplete at the end of the allocated analysis time and its overall points-to size continues to grow after 120,000 evaluations in Figure 2.

Figure 3 shows the distributions of the points-to sizes for each variable in the program. The x axis presents the points-to size of a variable and the y axis presents the percentage of the variables in the program with the corresponding points-to size. For the combined context-sensitive analysis, the points-to size of the majority of the variables (i.e., 81%) is less than 6; few variables are associated with large points-to sets. For the 0-1-CFA analysis, the points-to sizes of only 40% variables are less than 6 and there are condensed occurrences of variables with extremely large points-to sets (e.g., about 20% of the variables' points-to sizes are 150). This result also indicates that overly-approximated results of specific program constructs may pollute many places in the program due to copying during the points-to propagations.

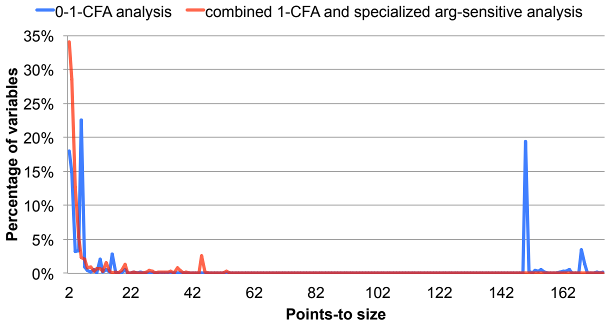

Figure 3: Points-to size distribution.

The above results demonstrate the significantly different behaviors between points-to analyses when their performance and precision vary. These results motivated us to design an automated approach to identify root causes of imprecision via the differences in static analysis behavior.

2.3 Root-cause Localization & Remediation

We now use the example in Figure 1 to illustrate the ideas on localizing root causes. First, the root-cause localization should be performed during the period in which overall precision of the points-to analysis starts to decrease, reflected as the "jumps" in terms of the overall points-to size in Figure 2. Second, we use the history information of points-to propagations and the incomplete points-to results to locate the program constructs that are root causes of imprecision. Intuitively, two conditions should be met: (i) the program construct has a wide reach within the propagation system (e.g., the values of the property access `source[name]` are assigned to `target[name]` at line 5 and are transitively propagated to about 500 other program variables or reference properties), and (ii) the impact of its wide reach is significant (e.g., looking up the property `name` of `source` at line 5 produces the points-to size of 150). Therefore, the imprecision of this property access results in the overall imprecision of the 0-1-CFA analysis when analyzing *jQuery* applications, becoming a root cause of imprecision.

In addition, to further assist in the process of remedying static analysis, we design an improvement suggestion algorithm that uses dynamic information to suggest appropriate context sensitivity to improve the analysis precision on the identified root causes. Dynamic information is used to simulate the benefits of different context-sensitive analyses, a generally applicable idea to quantify the potential precision of a specific context sensitivity.

3. TECHNICAL OVERVIEW

Figure 4 summarizes the root-cause localization and remediation process. We first run the static analysis in its initial configuration, which may lead to performance and/or precision issues. To monitor the behavior of the analysis, we run it in *diagnostic* mode by instrumenting the propagation system with labels (i.e., *labeled propagation system*) that keeps track of the history of points-to propagations by saving the

[2] An evaluation in the points-to analysis solves a constraint that may result in changes of the points-to results.

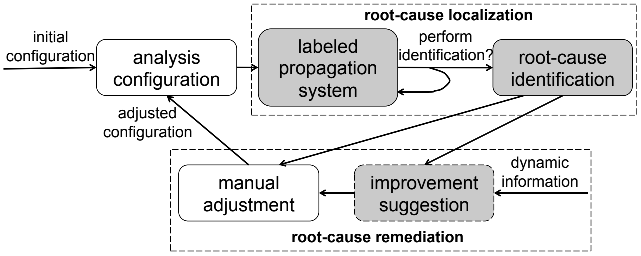

Figure 4: Overview of the root-cause localization and remediation process.

origins of points-to relations. We avoid the common situation that the analysis crashes (i.e., runs out of time/space budget), causing the loss of the metadata for root-cause localization, by obtaining periodic snapshots of the analysis state.

In Section 4, we describe and motivate our heuristics to determine whether the observed propagation behavior is anomalous. Intuitively, an anomaly is flagged if a given variable or reference property, collectively referred to as a *program pointer*, is associated with a large points-to set, and the label corresponding to the pointer propagates to many other pointers within a small number of evaluations of the propagation system. The labels record the transitive propagation of abstract objects across pointers. Our identification algorithm ranks the results by order of their impact on precision, thereby reflecting how likely they are to be among the root causes of imprecision.

For root-cause remediation, the optional *improvement suggestion* step that provides recommendations for improvement accepts as input (i) the candidate root causes, (ii) a set of context sensitivity policies, and (iii) one or more dynamic execution traces of the program. The dynamic traces provide precise points-to information, such that the effect of different sensitivity policies can be evaluated.

Intuitively, the improvement suggestion process maps between the root causes identified statically and the concrete points in the trace t, and simulates the effect of different context sensitivity policies on t. For example, given statement ℓ: $\mathsf{y} = \mathsf{x}[\mathsf{p}]$ in method m as the root cause, the first step is to simulate the least precise treatment of ℓ by merging together concrete points-to information from all its occurrences in t. What follows is an iterative process, wherein different kinds of context sensitivity and combinations thereof are applied, and their effects are simulated. A suggestion is output for ℓ to utilize a certain combination C of context-sensitive analyses if (i) C partitions the points-to set of $\mathsf{x}[\mathsf{p}]$ effectively, and (ii) other combinations are only marginally better.

The final step is for the user (i.e., either the analysis designer or an end user capable of configuring the analysis) to manually adjust the analysis configuration based on the localization results and/or the improvement suggestions. Upon doing so, the analysis can be rerun under the adjusted configuration to observe if the performance and/or precision issues have been resolved. The same process can be performed iteratively to locate all the root causes, thereby leading to a specialized analysis that meets the performance/-precision requirements of the user.

In the next two sections, we describe the details of the two automatic algorithms: root-cause localization and improvement suggestion. We discuss each in turn.

4. ROOT-CAUSE LOCALIZATION

First we explain more technically how the labeled propagation system is implemented. Then we describe our indicators based on the labels, which tell if anomalous propagation behavior is occurring. The workflow of root-cause localization is shown in Procedure 1.

Proc 1 Root-cause localization workflow.

```
Input: config: analysis configuration
Input: i: evaluation interval
Output: R: set of root causes
 1: sys ← initialize propagation system with config
 2: while (c ← sys.<next constraint>) != NULL do
 3:   for each assignment v1 = v2 from c do
 4:     $pts_{v1}$= $pts_{v1} \bigcup pts_{v2}$ //points-to propagation
 5:     $l_{v1}$ = $l_{v1} \bigcup l_{v2} \bigcup$ {v2} //label propagation
 6:   end for
 7:   if (k ← # of evaluations) mod i = 0 then
 8:     grow ← points-to size growth in past i evaluations
 9:     if grow > threshold then
10:       g ← intermediate static labelled points-to graph
11:       for each pointer node n in g do
12:         $impact_n$ ← compute the impact of n on g
13:       end for
14:       R ← high impact nodes in g
15:       return
16:     end if
17:   end if
18: end while
```

4.1 Labeled Propagation System

A propagation system for the points-to analysis solves the constraints to reach a fixed point, propagating the points-to relations of the variables and reference properties in the program. A majority of the constraints that exist in the propagation system are assignments. For example, to process an invoke instruction, the generated constraints include assignments from the actual arguments to the formal parameters, from the callee's return values to the left-hand side variable of the invoke instruction, etc.

We assume a subset-based (aka inclusion-based) propagation system [1], which means that the system solves a constraint that assigns a program pointer v2 to another pointer v1 by adding the points-to set of v2 to that of v1. In a standard propagation system, there is no provenance. The points-to set associated with a pointer may be the result of direct or transitive assignments, and there is no telling in general how it evolved to its current state. Such information is critical, however, to identify root causes, since frequent assignments involving an inaccurate points-to set may pollute the overall precision of the points-to solution as depicted in Figures 2 and 3.

To address this loss of information, in our labeled propagation system each constraint that assigns the values of v2 to v1 results in changes to v1's points-to relations. Moreover, a label v2 is associated with the points-to set of v1 that indicates (some of) the points-to relations of v1 were propagated through v2 (lines 3-6 in Procedure 1). As an example, in WALA [22] intermediate representation (IR), which is in SSA form [3], the statement at line 5 in Figure 1 reduces to (a) $\mathtt{v}_{\mathtt{tmp}}$ = source[name] and (b) target[name] = $\mathtt{v}_{\mathtt{tmp}}$, where $\mathtt{v}_{\mathtt{tmp}}$, source, target and name are all local variables of the function.

For the property read instruction (a), the analysis would first query the points-to set of `source`, P_{source}, and the points-to set of `name`, P_{name}. The pairs of each element in P_{source} and P_{name} (e.g., $p_{source\_i} \cdot p_{name\_j}$) are returned as the results of looking up the reference properties of `source[name]`. Note that the values of `name` iterate over all the property names of `source`; in the `extend` function, these values are the large set function names loaded in *jQuery*. This ultimately results in a large points-to set for v_{tmp} due to multiple assignments from various reference properties. We retain all these reference properties (e.g., $p_{source\_i} \cdot p_{name\_j}$) as labels attached to the points-to set of v_{tmp}.

For the property write instruction (b), the analysis similarly looks up the reference properties of `target[name]` (e.g., $p_{target\_i} \cdot p_{name\_j}$) and adds the points-to relations of v_{tmp} to each of these reference properties, resulting in an overly approximated points-to set for each function name in *jQuery*. We retain both v_{tmp} and the existing transitive labels from v_{tmp} (e.g., $p_{source\_i} \cdot p_{name\_j}$) in the points-to set of each reference property of `target[name]`.

In addition to tracking the set of labels that are immediately or transitively propagated to the points-to set of a given pointer, we generate a *propagation-history graph* for the points-to set of that pointer, which organizes the points-to propagation history into a hierarchical structure. Intuitively, a node in the propagation-history graph is a program pointer that has bearing on the specific points-to set. An edge from node n_1 to another node n_2 represents that the points-to set of n_2 was explicitly added to that of n_1. The entry points of the graph are the program pointers whose points-to sets are directly propagated into the corresponding points-to set.

For the same example in Figure 1, the propagation-history graph for the points-to set of v_{tmp} consists of all the reference properties of `source[name]` as entry points. v_{tmp} is the entry point of the propagation-history graph for the points-to set of each reference property of `target[name]` (e.g., $p_{target\_i} \cdot p_{name\_j}$), while there also are edges from v_{tmp} to the properties of `source[name]`.

As motivated above, during the propagation process, the labeled system is paused regularly upon completing cycles of `i` evaluations, for a fixed value of `i` (line 7 in Procedure 1). We base our analysis of root causes of precision on the resulting intermediate states.

Specifically, we count the total number of points-to relations (i.e., edges) in the intermediate points-to graph for the k_{th} pause (i.e., after n×k evaluations), `actual(k)`. We then compare the value of `actual(k)` with the value of `predict(k)` to decide whether to perform root-cause identification.

For meaningful comparison, we utilize the results from the previous `k-1` pauses in performing linear regression analysis to find the fitted line `y = intercept + slope` × `x`, where `x` is the number of evaluations and `y` is the total number of points-to relations. The number of points-to relations of the k_{th} pause can then be predicted: `predict(k) = intercept + slope` × `kn`. If `actual(k) > threshold` × `predict(k)`, then we perform the root-cause identification at the k_{th} pause; otherwise, we continue the points-to analysis under the labeled propagation system. In practice we use `threshold = 110%`, effective for our evaluation in Section 6.

Figure 5 shows this prediction model when running the 0-1-CFA analysis on a *jQuery* application. The analysis is paused every 1000 evaluations and the fitted line in Figure 5 is calculated from the results of the first 23 pauses (i.e., 1000 to 23,000 evaluations). The predicted total points-to edges at the 24,000 evaluations is 16,234, while `actual(24)` is 88,857, significantly passing the threshold to perform root-cause identification. This prediction model can capture the "jump" phase of the points-to analysis shown in Figure 2, which is important to localize the root causes accurately.

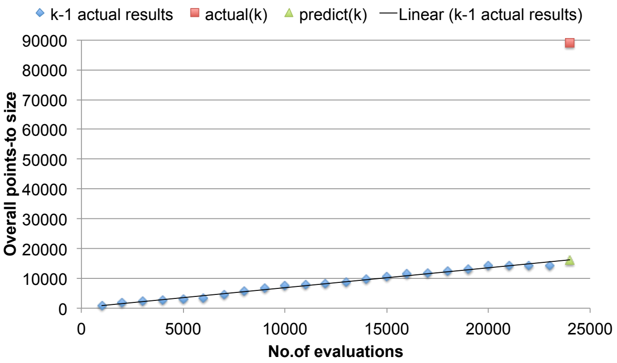

Figure 5: Prediction via linear regression.

If localization is performed too early, the over-approximate results may not have surfaced yet. If localization is performed too late during the analysis, then the over-approximate results may have polluted a large portion of the overall points-to results, making it difficult to identify the root causes of imprecision. Moreover, the evaluation interval to pause the analysis, `i`, as well as the decision threshold, may be tuned based on the budget and goals of root-cause localization.

4.2 Identifying Root Causes for Divergence

When performing root-cause identification, we use the intermediate points-to graph and the associated labels to identify possible root causes (line 10 in Procedure 1). For each variable or reference property `v`, we count (i) its points-to size, $|P_v|$, as well as (ii) its number of occurrences as labels in the points-to sets of other variables and/or reference properties, $|L_v|$. $|L_v|$ measures how widely `v` reaches within the propagation system, and $|P_v|$ measures if the impact of its wide reach is significant. We therefore use the scoring heuristic S_v=$|P_v|\times|L_v|$ as a measurement of the possibility of `v` being a root cause (lines 11-13 in Procedure 1).

The root-cause identification stage reports a set of variables as root causes in descending order of their scores. For example, v_{tmp}, the result of the property read instruction at line 5 in Figure 1, achieves an extremely high score (73,950) with $|P_{v_{tmp}}|$=150 and $|L_{v_{tmp}}|$=493 when localization is performed after 24,000 evaluations. Its score is x75 higher than the variable with the second highest score (990), rendering it the only root-cause candidate for imprecision of the 0-1-CFA analysis for this *jQuery* application. We report the variables and/or reference properties whose scores are at least half the highest score as *candidate (or suspicious) root causes*.

5. IMPROVEMENT SUGGESTION

The next step is to automatically compute suggestions for how to improve the analysis per the program, configuration and context sensitivity at hand. This is the focus of this section, whose workflow is shown in Procedure 2.

Proc 2 Improvement suggestion workflow.

```
Input: R = {$r_1$, ..., $r_n$}: root causes
Input: CS: context sensitivity policies
Output: <V, S>={($r_1$, $s_1$),...,($r_n$, $s_n$)}: suggestions
 1: t ← collect a dynamic trace
 2: for each cs ∈ CS do
 3:   G = G $\bigcup$ {gen(t, cs)}
 4: end for
 5: for each r ∈ R do
 6:   $A_{<r, G>} \leftarrow \emptyset$
 7:   for each g ∈ G do
 8:     $a_{<r, g>}$ ← query r's points-to size in g and measure
        its accuracy corresponding to g's context sensitivity
 9:     $A_{<r, G>} = A_{<r, G>} \bigcup \{a_{<r, g>}\}$
10:   end for
11:   $a_{<r, g_{min}>} \leftarrow \min(A_{<r, G>})$
12:   <V, S> = <V, S> $\bigcup$ {<r, $g_{min}$.cs>}
13: end for
```

First, the target program is executed to collect a dynamic trace (line 1 in Procedure 2), recording the following runtime artifacts in order of occurrence: (i) function entries and exits; (ii) invocations; and (iii) property reads and writes. At a property read/write instruction, we record (i) the instruction location in the program; (ii) the allocation site of the base object (as its program location); (iii) the property name, and (iv) the allocation site of the value, if it is a reference object, or the type of the value, if it is a primitive value. At an invoke instruction, we record (i) the location of the call site; (ii) the location of the target function; (iii) the allocation site of the receiver object; and (iv) the allocation sites and/or the types of the actual arguments.

Second, dynamic points-to graphs based on the dynamic trace are generated for the available kinds of context sensitivity (lines 2-4 in Procedure 2). Procedure 3 captures the algorithm that produces a dynamic points-to graph with respect to a specific context-sensitive analysis (e.g., 1-CFA, 1st-argument-sensitive [23], or context-insensitive analysis). The inputs are the dynamic trace, `trace`, and the kind of context sensitivity, `cs`. The algorithm iterates through each instruction `i` recorded in the dynamic trace. If `i` is an invoke instruction, at line 5 the call site and the argument are pushed into the call stack, `stack`. If `i` exits a function, then top element of `stack` is removed at line 7.

If `i` is a property read or write instruction, then lines 9 to 15 show how the appropriate calling context is determined; the calling context is *everywhere* for context-insensitive analysis. In the dynamic points-to graph, a variable is represented by (i) the location of the instruction, (ii) the part of the instruction (i.e., base, property or value), and (iii) the calling context. At lines 16 to 18, the object allocation site, represented by program location, of each part of the instruction collected at runtime is assigned to the corresponding variable node in the dynamic points-to graph.

This algorithm is general in that various dynamic points-to graphs can be generated under different context sensitivity policies. Now that we have obtained different context-sensitive dynamic points-to graphs, we can determine which of the policies (including combinations thereof) are beneficial. For a program pointer `r` that is identified as a root cause, we locate its corresponding nodes via its program location in each dynamic points-to graph, and collect (i) the number of calling contexts associated with `r`, cc_r, and

Proc 3 Dynamic points-to graph generation, `gen(t, cs)`.

```
Input: t: dynamic trace
Input: cs: context sensitivity policy
Output: g: dynamic points-to graph
 1: stack ← $\emptyset$
 2: while (i ← next(t)) != NULL do
 3:   switch (kindOf i)
 4:     case INVOKE:
 5:       stack.push(call site and first arg of i)
 6:     case FEXIT:
 7:       stack.pop
 8:     case PREAD || PWRITE:
 9:       if cs = 1-CFA then
10:         context ← immediate call site on stack
11:       else if cs = 1st-argument-sens then
12:         context ← immediate argument on stack
13:       else if cs = context-insens then
14:         context ← everywhere
15:       end if
16:       $g_{(i_{loc}, base, context)} \rightarrow i_{base}$
17:       $g_{(i_{loc}, property, context)} \rightarrow i_{property}$
18:       $g_{(i_{loc}, value, context)} \rightarrow i_{value}$
19:   end switch
20: end while
```

(ii) the sum of points-to sizes under all calling contexts, $\sum |p_r|$. We then count $a_r = \frac{\sum |p_r|}{cc_r}$, the average dynamic points-to size per calling context, to measure the accuracy of the points-to relations of `v` under the given context sensitivity (line 8 in Procedure 2). The context sensitivity with the smallest a_r is chosen for the function that contains `r` as the improvement suggestion (lines 11 and 12 in Procedure 2).

6. EVALUATION

We have implemented our root-cause localization and remediation algorithms, and conducted experiments to assess their efficacy. In this section, we present the experimental results of our analysis on various JavaScript benchmarks.

6.1 Experimental Setup

6.1.1 Metrics

In our evaluation, we compared the performance and precision of points-to analysis that usually constructs a call graph as well as a points-to graph. For precision on a call graph, we measured (i) `HIGH_POLY`: the number of highly polymorphic call sites (i.e., call sites with more than 5 targets), (ii) `AVG_TARG`: the number of targets averaged over all call sites, and (iii) `REAC_FUNC`: the number of reachable functions, as in [20]. For precision on a points-to graph, we measured `PTS_SIZE`: the overall points-to size. This is the total number of points-to set sizes summed over all local variables in the program. The points-to set of a variable is the set of abstract objects (i.e., allocation sites) it refers to, representing the set of values it may have at runtime. In addition, we measured the performance of the points-to analysis with its running time (in seconds).

6.1.2 Benchmarks

We use two sets of JavaScript benchmarks.

Benchmarks I. Benchmarks I consists of applications that use JavaScript libraries, generated by Sridharan et al.

Table 1: Benchmarks I precision and performance results.

library	1-CFA			1-CFA + 1st-arg-sens *(whole-combined)*			1-CFA + selective 1st-arg-sens *(selective)*			
	REAC_FUNC	AVG_TARG	HIGH_POLY	REAC_FUNC	AVG_TARG	HIGH_POLY	REAC_FUNC	AVG_TARG	HIGH_POLY	time (sec)
jQuery	312	2.2	441	206	25.0	1235	206	1.2	16	16.3
prototype.js	451	12.9	259	170	3.3	124	170	1.5	39	2.9
script.aculo.us	617	14.4	188	179	2.7	145	179	1.4	47	4.6

[20]. Included are 11, 5 and 1 simple web applications that invoke *jQuery*, *prototype.js* and *script.aculo.us* libraries, respectively. These libraries are among the most popular JavaScript libraries for developing real-world web applications, especially *jQuery* [21]. Sridharan et al. [20] performed manual code rewriting for improving precision and performance of their specialized analysis on these libraries. In our experiments, we reuse these manual transformations.

Benchmarks II. Benchmarks II are JavaScript applications collected by Kashyap et al. [9]. Twelve out of the 28 programs from the original benchmarks were selected for our evaluation. These programs are collected from open-source JavaScript repositories, standard JavaScript benchmarks (e.g., SunSpider[3]), and the Emscripten LLVM test suite[4], the results of which benefit from various context-sensitive analyses [9, 23].

6.1.3 Experimental Design

Root-cause localization. We performed experiments on Benchmarks I to illustrate the accuracy of the root-cause localization algorithm. In this experiment, we used WALA's whole-program 1-CFA analysis as the *baseline* analysis, which experiences scalability and precision problems for JavaScript library applications. For each of the Benchmark I programs, we performed the localization algorithm on the 1-CFA analysis, finding a set of functions that contain the root causes. Then for the root-cause functions, we applied additional argument sensitivity on the first arguments (i.e., 1st-argument sensitivity [23]). For the rest of the functions, 1-CFA analysis was performed, resulting in a 1-CFA and *selective* 1st-argument-sensitive analysis. We compare the precision results among the 1-CFA, the whole-program 1-CFA and 1st-argument-sensitive analysis, and the *selective* analysis using the call graph metrics for Benchmarks I. We also compare the differences in terms of analysis performance.

Improvement suggestion. For the programs in Benchmarks II, we applied the root-cause localization as well as improvement suggestion algorithms on the *baseline* 0-1-CFA analysis. For the root-cause functions, we applied the suggested context sensitivity and for the rest of the functions, the 0-1-CFA analysis was performed, comprising an *auto-selective* analysis. Possible suggestions for root-cause functions include: (i) a context-insensitive analysis, (ii) a single context-sensitive analysis (i.e., 1-CFA or argument sensitivity on any argument of the function[5]), and (iii) a combined context-sensitive analysis (e.g., 1-CFA + 1st-argument sensitivity). We compare the performance and precision of the *auto-selective* analysis with the 0-1-CFA analysis and the *full-sensitive* analysis (i.e., a whole-program combined context-sensitive analysis that applies 1-CFA and argument sensitivity on all arguments) for Benchmarks II to illustrate the effectiveness of the improvement suggestion.

Our comparison experiments explored the following two hypotheses:

Hypothesis I: The root-cause localization algorithm can accurately identify a small set of program constructs as the root causes of imprecision.

Hypothesis II: Guided by the root causes identified by the localization algorithm, the improvement suggestion algorithm can recommend appropriate kinds of context sensitivity to significantly improve the analysis precision.

The experimental results were obtained on a 2.5 GHz Intel Core i5 MacBook Pro with 16 GB memory running the Mac OS X 10.11 operating system.

6.2 Benchmarks I Results

Tables 1 and 2 show the experimental results of Benchmarks I. For each JavaScript library, the results are arithmetically averaged over all the applications that use the specific library. For example, the 312 reachable functions of *jQuery* library from the 1-CFA analysis (i.e., column 2 of the jQuery row in Table 1) is calculated by averaging the number of reachable functions the 1-CFA analysis obtained for all 11 *jQuery* applications.[6]

Precision and performance. Table 1 shows the precision and performance results of the analyses on Benchmarks I. Columns 2-4, 5-7, and 9-11 show the results of call graph precision metrics for the 1-CFA analysis, the whole-program combined 1-CFA and 1st-argument-sensitive analysis (i.e., *whole-combined* analysis), and the 1-CFA and *selective* 1st-argument-sensitive analysis, respectively. For each analysis, we present its number of reachable functions (i.e., columns 2, 5, and 8), average number of targets per call site (i.e., columns 3, 6 and 9), and the number of highly polymorphic call sites (i.e., columns 4, 7, and 10). In addition, column 11 shows the points-to analysis time in seconds of the *selective* analysis. Given the time budget of 10 minutes, both the 1-CFA analysis and the *whole-combined* analysis failed to complete analyzing each of the programs in Benchmarks I; therefore, their precision results were calculated from the incomplete call graphs obtained after the timeout.

The REAC_FUNC results of the *whole-combined* and the *selective* analyses in Table 1 are the same for all three libraries;

[3] https://webkit.org/perf/sunspider/sunspider.html

[4] http://kripken.github.io/emscripten-site/

[5] Object sensitivity [13] applies calling contexts on the receiver argument.

[6] Because the applications in Benchmarks I are relatively simple programs that use the JavaScript libraries, the analysis performance and precision results of these programs are dominated by the underlying libraries. Therefore, we report the average results based on the corresponding libraries.

they are significantly more precise than those of the 1-CFA analysis. Only 29% (for *script.aculo.us*) to 66% (for *jQuery*) functions considered reachable by the 1-CFA analysis are produced by the *whole-combined* and the *selective* analyses. Interestingly, the number of highly polymorphic call sites is below 50 for the *selective* analysis for all three libraries, while the 1-CFA analysis produces 188 (for *script.aculo.us*) to 441 (for *jQuery*) and the *whole-combined* analysis results in 145 (for *script.aculo.us*) to 1235 (for *jQuery*) highly polymorphic call sites. This result suggests that (i) 1-CFA analysis is imprecise to resolve the call targets in many cases, and (ii) because the *whole-combined* analysis applies 1st-argument sensitivity over all the program, it may create many calling contexts for the functions that are not identified as root causes which may not significantly increase the analysis precision (e.g., in terms of the `REAC_FUNC` metric). The average number of targets per call site from the *selective* analysis ranges from 1.2 (for *jQuery*) to 1.5 (for *prototype.js*), indicating the *selective* analysis precisely resolves the targets for most call sites. Although the *whole-combined* analysis reduces the `AVG_TARG` of the 1-CFA analysis from 12.9 to 3.3 and from 14.4 to 2.7 for *prototype.js* and *script.aculo.us*, respectively, it still results in a higher average number of targets per call site comparing to the *selective* analysis. For *jQuery* library, applying argument sensitivity over the entire program results in significant increase of `AVG_TARG` (i.e., on average 25 targets per call site) and `HIGH_POLY` (i.e., 1235 highly polymorphic call sites) due to the additional calling contexts.

The last column in Table 1 shows that the *selective* analysis finishes analyzing the libraries on average between 3 seconds (for *prototype.js*) and 16 seconds (for *jQuery*). Due to the fact that the 1-CFA analysis could not finish analyzing any of these libraries in under 10 minutes, we claim that the *selective* analysis has significantly better performance because of the precision gained by applying 1st-argument sensitivity to the root-cause functions. The *whole-combined* analysis also could not complete within the 10-minute time budget. Applying 1st-argument sensitivity over all the program generates too many calling contexts for the propagation system to quickly converge.

Table 2: Localization results of Benchmarks I.

library	no. of localized functions	no. of evaluations	slope
jQuery	2	25000	3.7
prototype.js	4	72000	5.1
script.aculo.us	4	96000	4.9

Root-cause localization characteristics. Table 2 shows additional information that characterizes the results of the root-cause localization algorithm. For the experiments on Benchmarks I, we paused the 1-CFA analysis every 1000 evaluations to decide if the root-cause localization algorithm should be performed. Columns 2, 3, and 4 present the number of functions identified as root causes, the number of evaluations until performing root-cause localization, and the slope of the last 1000 evaluations (i.e., the increase in the total number of points-to relations divided by 1000), respectively.

Our algorithm identifies 2, 4 and 4 functions as root causes for *jQuery*, *prototype.js*, and *script.aculo.us*, respectively. Comparing to the number of reachable functions computed by any analysis in Table 1 (i.e., more than 150 functions), very small fractions of these functions were identified as root causes. This result combined with the good precision and performance of the *selective* analysis support our intuition that a small number of complex constructs in the programs may contribute to significant loss of analysis performance and precision if not handled accurately. Therefore, it is extremely useful for our automated localization algorithm to pinpoint these root causes, as shown.

Column 4 shows that the slopes of the last 1000 evaluations range from 3.7 (for *jQuery*) to 5.1 (for *prototype.js*). For example for *prototype.js*, the large slope indicates that the precision of the 1-CFA analysis significantly decreases during this period (i.e., about 5 new points-to relations per constraint), while the slope of the simple linear regression for all previous evaluations is 0.9. This result suggests that our heuristics accurately decided when to perform the root-cause localization for JavaScript libraries.

Summary. The results of Benchmarks I suggest that (i) our root-cause localization algorithm is capable of locating a small number of functions where the 1-CFA analysis experiences significant precision and performance loss, and (ii) because the root causes are highly clustered in these JavaScript libraries, applying the selective 1st-argument-sensitive analysis only on the localized functions achieved a much better balance between precision and performance comparing to the *whole-combined* analysis. The above observations support **Hypothesis I**.

6.3 Benchmarks II Results

Figures 6 and 7 show the precision and performance results of Benchmarks II, respectively. Because the 0-1-CFA analysis finishes analyzing all 12 programs in Benchmarks II within the time budget of 10 minutes, the root-cause localization was performed after the 0-1-CFA points-to analysis completed on each program.

Figure 6 presents the `PTS_SIZE` precision improvement of the *full-sensitive* analysis (i.e., grey bars) and the *auto-selective* analysis (i.e., patterned bars) over the 0-1-CFA analysis. Because the *full-sensitive* analysis applies 1-CFA and argument sensitivity of all arguments over all the functions, its results are at least as precise as the *selective* analysis. In Figure 6, the y axis shows the precision improvement of the corresponding analysis `Y` over the 0-1-CFA analysis, calculated as follows

$$\mathtt{IMP}_{\mathtt{Y}} = \frac{\mathtt{PTS\_SIZE}_{\mathtt{0-1-CFA}} - \mathtt{PTS\_SIZE}_{\mathtt{Y}}}{\mathtt{PTS\_SIZE}_{\mathtt{0-1-CFA}}} \times 100\%$$

Therefore, $\mathtt{IMP}_{\mathtt{Y}}$ measures the precision of analysis `Y` in terms of removing the false positives from the 0-1-CFA analysis results. In Figure 6, the *full-sensitive* analysis improves the `PTS_SIZE` precision over the 0-1-CFA analysis by between 9% (for *sgefa*) to 65% (for *cryptobench*). For all but five programs (i.e., *fourinarow*, *cryptobench*, *linq_functional*, *linq_aggregate* and *linq_enumerable*), the differences in precision improvement percentages are within 3.5% between the *full-sensitive* and the *auto-selective* analyses, indicating that the *auto-selective* analysis produces similar results to the *full-sensitive* analysis for most programs. The results of the *linq_aggregate* program exhibit the largest differ-

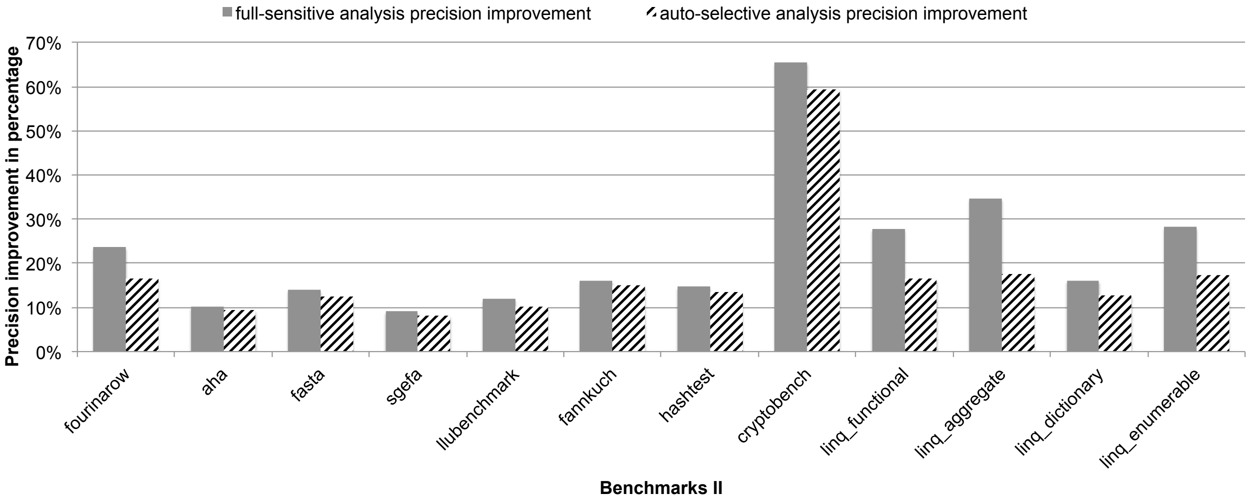

Figure 6: Benchmarks II precision results.

ence in terms of precision improvement (i.e., 17%) between the *full-sensitive* 35%) and the *auto-selective* (18%) analyses. The points-to relations in the 0-1-CFA solution that do not appear in the *full-sensitive* solution are false positives. More than 50% of these relations also did not appear in the *auto-selective* solution for *linq_aggregate*, demonstrating its greater accuracy over the 0-1-CFA analysis.

Figure 7 presents the performance results of the *full-sensitive* (i.e., grey bars) and the *auto-selective* (i.e., patterned bars) analyses comparing to the 0-1-CFA analysis performance. The y axis (in logarithmic scale) shows the overhead of the corresponding analysis `Y`'s time cost comparing to the 0-1-CFA analysis time (i.e., $\frac{\texttt{TIME}_{\texttt{Y}}}{\texttt{TIME}_{\texttt{0}-\texttt{1}-\texttt{CFA}}}$). The performance of an analysis on each benchmark program was obtained by averaging over 30 repeated executions.

In Figure 7, the *full-sensitive* analysis performs significantly worse than the 0-1-CFA analysis for all the benchmark programs. It could not finish analyzing *linq_aggregate* and *linq_enumerable* under the time budget of 10 minutes; therefore, the incomplete points-to results of these two programs were obtained after the timeout for comparison. The *full-sensitive* analysis is at least two orders of magnitude slower for another three programs (i.e., *linq_functional*, *aha*, and *linq_dictionary*) and is between 23 (for *fannkuch*) and 60 (for *llubenchmark*) times slower for six programs than the 0-1-CFA analysis. For example, it takes less than one second for the 0-1-CFA analysis to finish analyzing *linq_functional*, while the *full-sensitive* analysis needs almost 7 minutes to complete analyzing the same program. Despite of the fact that the *full-sensitive* analysis often results in relatively significant precision improvement (e.g., 28% for *linq_functional*), the performance issues outweigh the benefits of this whole-program combined context-sensitive analysis in many cases.

On the other hand, the *auto-selective* analysis is capable of analyzing the benchmarks under the same order of magnitude as the 0-1-CFA analysis for all but three programs (i.e., *fourinarow*, *aha* and *linq_dictionary*). For example, the *auto-selective* analysis improves the precision over the 0-1-CFA analysis by 59% for *cryptobench* and it also finishes analyzing this program almost as fast as the 0-1-CFA analysis. In most cases, the *auto-selective* analysis results in a much better balance between performance and precision than the *full-sensitive* analysis (e.g., the *full-sensitive* analysis is 65% more precise but performs 5 times slower than the 0-1-CFA analysis for *cryptobench*). For three programs the *auto-selective* analysis performs more than 10 times slower than the 0-1-CFA analysis (i.e., 10, 15 and 143 times slower for *aha*, *fourinarow* and *linq_dictionary*, respectively), the *auto-selective* analysis is still an order of magnitude faster than the *full-sensitive* analysis for *aha* and about 3 times faster than the *full-sensitive* analysis for *fourinarow*. An outlier is the *auto-selective* analysis for *linq_dictionary*, whose performance is similar to the *full-sensitive* analysis. The *auto-selective* analysis applied combined context sensitivity on 15 out of 84 (i.e., 18%) reachable functions of *linq_dictionary*, resulting in both precision improvement and performance overhead. Nevertheless, the *auto-selective* analysis has achieved significantly better performance than the *full-sensitive* analysis for most of the programs in Benchmarks II.

Our localization algorithm identifies between 6% (for *fourinarow*) to 27% (for *linq_aggregate*) of the functions as the sources of precision loss, with an average of 13% of the functions over all the programs in Benchmarks II, a relatively small fraction. Our improvement suggestion algorithm recommends combined context sensitivity on 72%, single context sensitivity on 25%, and context insensitivity on 3% of all the root-cause functions in Benchmarks II, respectively.

Summary. The *auto-selective* analysis obtains similar precision results to the *full-sensitive* analysis that improves the precision of the 0-1-CFA analysis. Moreover, the *auto-selective* analysis' performance is significantly better than the whole-program combined fully context-sensitive analysis. This result supports **Hypothesis II** that our improvement suggestion algorithm can choose the appropriate context sensitivity that benefits the results both in performance and precision for Benchmarks II.

6.4 Threats to Validity

Although we used benchmarks collected by Sridharan et al. [20] and Kashyap et al. [9] in our experiments, the representativeness of these benchmarks might threaten the validity of our conclusions as applicable to all JavaScript programs. (i) The simple web applications invoking the li-

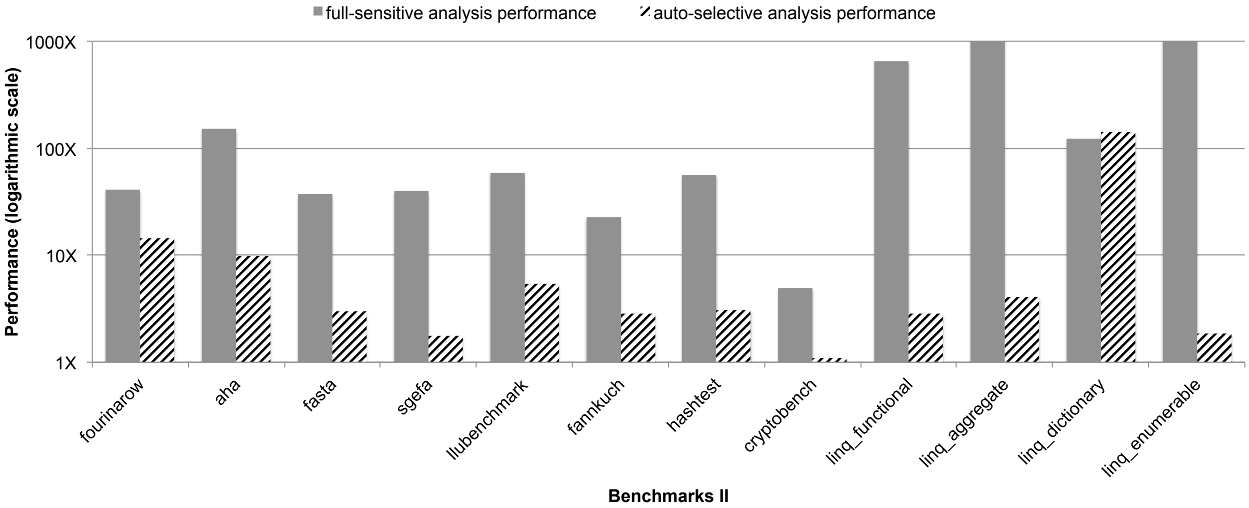

Figure 7: Benchmarks II performance results.

braries (i.e., Benchmarks I) may not be representative of later JavaScript library versions or the behavior of real-world JavaScript websites. (ii) Benchmarks II, consisting of small and dated JavaScript programs in standard benchmarks, may not be representative of non-website JavaScript applications.

7. RELATED WORK

To the best of our knowledge, we present the first work that focuses on automating the process of localizing and remedying the root causes when a JavaScript analysis is unscalable or too imprecise on a program. Nevertheless, our work is related to (i) recent context-sensitive static analyses that unveiled difficult JavaScript program constructs, and (ii) static analyses that identify the causes of precision loss for refinement.

7.1 JavaScript Context-sensitive Analysis

Sridharan et al. identified correlated dynamic property accesses (e.g., `x[p] = y[p]`) as a hard-to-analyze JavaScript code pattern and presented a specialized argument-sensitive analysis along with program transformation that dramatically improves analysis scalability and precision on JavaScript libraries [20]. This work motivated us to design an automatic localization algorithm that identifies such program constructs causing the analysis imprecision. We also have reused the library benchmarks collected by Sridharan et al. [20] as Benchmarks I in the evaluation.

Esben et al. inferred determinacy information (i.e., a variable and expression always has the same value at a given program point [15]) and improved static analysis precision on JavaScript libraries via various techniques [2]. For example, argument sensitivity was selectively applied for the arguments whose abstract value was a concrete string or a single object address. This specialized context sensitivity was proposed as a result of manually inspecting imprecise portions of an incomplete call graph obtained after a fixed number of evaluations. In our work, root causes of analysis imprecision are automatically localized and corresponding focused program constructs are provided for inspection.

Wei and Ryder presented a two-stage context-sensitive analysis for JavaScript that selectively applied specific kinds of context sensitivity on the function level [23]. Heuristics were used to choose the context sensitivity based on function characteristics extracted from the results of a pre-analysis. Our improvement suggestion may also result in applying different kinds of context sensitivity to the root-cause functions. However, the adaptive analysis in [23] required a pre-analysis that could finish analysis of the target program, while our approach does not. In addition, our goal is to apply the specialized context sensitivity only on the root-cause functions that may significantly improve the overall analysis scalability and precision, while adaptive analysis selects a specific context sensitivity for each of the functions..

Park and Ryu presented another JavaScript static analysis that improved precision via loop sensitivity [14]. The authors identified one root cause of scalability problems with JavaScript analysis as the combination of imprecise results in loops and in dynamic property accesses. Their analysis improved precision in loops by distinguishing each iteration of a loop with different contexts based on the analysis results of loop conditional expressions. Our approach systematically assists in the process of locating such code patterns as root causes of analysis scalability problems.

Madsen et al. presented a static analysis for event-driven *Node.js* applications, extending the traditional call graph with nodes and edges that reflect the flow of control due to event handling [11]. Three context sensitivity policies, varying in precision and cost, were introduced for constructing event-based call graphs. It would be interesting to explore if our root-cause localization technique may generally be applied to this work.

7.2 Refinement-based Analysis

Smaragdakis et al. presented introspective analysis that aimed to improve the performance of a context-sensitive analysis for Java [18]. Introspective analysis used heuristics to decide whether to refine an allocation site or a call site with context sensitivity, based on the metrics computed from context-insensitive points-to results. The heuristics focused on reasoning about the cost of applying additional context sensitivity. Instead, we focus on identifying the constructs that originate significant loss of performance and/or precision of an analysis. Therefore, applying more accurate but expensive analysis techniques only on these constructs may result in the improvement of the overall analysis performance and precision.

Sridharan and Bodík presented a refinement-based points-to analysis for Java that refined sensitivity for heap accesses and method calls [19]. The analysis was demand-driven and client-driven in that it focused on refining the analysis of relevant code. We focus on improving overall points-to analysis precision in terms of all possible queries for points-to sets of program variables, instead of precisely answering a specific query.

Guyer and Lin presented a client-driven analysis for C that automatically adjusted its precision in response to the needs of client analyses [7]. This client-driven analysis monitored polluting assignments (i.e., the program points that result inaccuracy in the analysis) and tuned context as well as flow sensitivity to improve precision. Our heuristics to locate the root causes consider not only the inaccurate results at a program point but also its impact on the overall points-to analysis precision by tracking the labels in the propagation system.

Liang et al. found minimal abstractions needed to prove a set of queries using machine learning algorithms and showed that very few components of an abstraction were needed to prove a query [10]. Our improvement suggestion, in contrast, is not driven by queries but rather by localization of the sources of imprecision.

8. CONCLUSIONS & FUTURE WORK

Static analysis of JavaScript is a challenging problem given the dynamic nature of this language. Hence, as we demonstrated experimentally, neither coarse nor precise analyses are able to cope with JavaScript applications with acceptable precision and performance. We have developed a technique to systematically identify root causes of analysis imprecision. Over root causes, we built an algorithm that automatically suggests a specific context sensitivity for a small fraction of the functions (i.e., root-cause functions). The results on library applications show that applying context sensitivity to a small set of root-cause functions resolves the scalability problems that occur on both coarse and precise analyses. The analysis that automatically applies the suggested kind of context sensitivity to the root-cause functions achieves a better balance between precision and performance for most programs in Benchmarks II, demonstrating the effectiveness of the improvement suggestion algorithm.

In the future, we intend to improve the usability of our approach, and make it more interactive, by enabling a visual interface to review root causes, etc. We also intend to extend our recommendation algorithm to support more forms of static analysis techniques (e.g., the ability to perform semantics-preserving transformations on the subject program for analysis purposes).

9. REFERENCES

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

[2] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 17–31, New York, NY, USA, 2014. ACM.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[4] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 752–761, Piscataway, NJ, USA, 2013. IEEE Press.

[5] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

[6] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 177–187, New York, NY, USA, 2011. ACM.

[7] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 214–236, 2003.

[8] B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM.

[9] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. Jsai: A static analysis platform for JavaScript. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132, New York, NY, USA, 2014. ACM.

[10] P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 31–42, New York, NY, USA, 2011. ACM.

[11] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven node.js javascript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 505–519, New York, NY, USA, 2015. ACM.

[12] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engg.*, 11(1):7–26, Jan. 2004.

[13] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, Jan. 2005.

[14] C. Park and S. Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *29th European Conference on Object-Oriented*

Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic, pages 735–756, 2015.

[15] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Dynamic determinacy analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 165–174, New York, NY, USA, 2013. ACM.

[16] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

[17] O. G. Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.

[18] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, 2014.

[19] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, 2006.

[20] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 435–458, 2012.

[21] W^{3}Techs. W^{3}Techs web technologies surveys: usage of JavaScript libraries for websites. http://w3techs.com/technologies/overview/javascript_library/all, 2016.

[22] WALA. T. J. Watson Libraries for Analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[23] S. Wei and B. G. Ryder. Adaptive context-sensitive analysis for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 712–734, 2015.