Change-aware Dynamic Program Analysis for JavaScript

Dileep Ramachandrarao Krishna Murthy
Department of Computer Science
TU Darmstadt
Darmstadt, Germany
dileep.rk730@gmail.com

Michael Pradel
Department of Computer Science
TU Darmstadt
Darmstadt, Germany
michael@binaervarianz.de

***Abstract*—Dynamic analysis is a powerful technique to detect correctness, performance, and security problems, in particular for programs written in dynamic languages, such as JavaScript. To catch mistakes as early as possible, developers should run such analyses regularly, e.g., by analyzing the execution of a regression test suite before each commit. Unfortunately, the high overhead of these analyses make this approach prohibitively expensive, hindering developers from benefiting from the power of heavyweight dynamic analysis. This paper presents change-aware dynamic program analysis, an approach to make a common class of dynamic analyses change-aware. The key idea is to identify parts of the code affected by a change through a lightweight static change impact analysis, and to focus the dynamic analysis on these affected parts. We implement the idea based on the dynamic analysis framework Jalangi and evaluate it with 46 checkers from the DLint and JITProf tools. Our results show that change-aware dynamic analysis reduces the overall analysis time by 40%, on average, and by at least 80% for 31% of all commits.**

***Index Terms*—program analysis, JavaScript**

I. Introduction

Dynamic analyses help developers identify programming mistakes by analyzing the runtime behavior of a program. For example, the Valgrind analysis framework [1] is the basis for various tools, e.g., to identify references of undefined values via a memory analysis and unintended flows of data via a taint analysis. More recently, the Jalangi framework [2] has popularized dynamic analysis for dynamic languages, in particular, JavaScript, and is the basis for several tools that find, e.g., inconsistent types [3], optimization opportunities [4], memory leaks [5], and various other code quality issues [6].

Given the power of these analyses, it would be desirable to use them frequently during the development process. For example, one could apply such analyses on each commit, e.g., by running a regression test suite and analyzing its execution. By running each analysis after each commit, the analyses could warn developers about mistakes right when the mistakes are introduced, similar to static checks performed by a compiler or lint-like tools. An important prerequisite for this promising usage scenario is that an analysis produces results quickly.

Unfortunately, most dynamic analyses implemented on top of general-purpose analysis frameworks, such as Valgrind and Jalangi, are not only powerful but also heavyweight. Typically, such an analysis imposes a significant runtime overhead, ranging between factors of tens and hundreds. For projects with large regression test suites, this overhead practically prohibits running the analysis after every change. Such large regression test suites are particularly common for dynamic languages, where static checks are sparse and likely to miss problems. For example, applying two dynamic analyses for JavaScript, DLint [6] and JITProf [4], to the regression test suite of the widely used underscore.js library[1] takes over five minutes on a standard PC, whereas running the test suite without any analysis takes only about five seconds.

As a motivating example, consider Figure 1, which shows a change in a piece of JavaScript code. Suppose to use a dynamic analysis that warns about occurrences of `NaN` (not a number), which may result in JavaScript when arithmetic operations are applied to non-numbers, but it does not trigger any runtime exception. Furthermore, suppose that the project's regression test suite has three tests that call the functions `a`, `b`, and `e`, respectively, as indicated in the last three lines of the example. In the original code, the analysis reports a `NaN` warning at line 22. In the modified code, the analysis warns again about line 22 and also about the newly introduced `NaN` at line 12. Naively applying the dynamic analysis after the change will analyze the executions of all code in Figure 1. However, most of the code is not impacted by the change, and there is no need to re-analyze the non-impacted code.

Existing approaches that reduce the overhead of dynamic analysis use sampling. One approach is to sample the execution in an analysis-specific way, e.g., based on how often a code location has been executed [4]. While such sampling can significantly reduce the overhead, it still remains too high for practical regression analysis. Another approach is to distribute the instrumentation overhead across multiple users [7]. However, this approach is not suitable for regression

This work was supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the ConcSys and Perf4JS projects, and by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

[1] http://underscorejs.org

Old version:

```
1  var x=4, y, z;
2
3  function a() {
4    x = 42;
5  }
6  function b() {
7    c();
8    d();
9  }
10 function c() {
11
12   y = x / 2;
13 }
14 function d() {
15   /* expensive
16    * computation
17    * independent
18    * of x, y, z */
19 }
20 function e() {
21   // NaN warning
22   var n = "hi" - 23;
23   return n;
24 }
25
26 a();
27 b();
28 z = e();
```

New version:

```
1  var x=4, y, z;
2
3  function a() {
4    x = "aaa"; // changed
5  }
6  function b() {
7    c();
8    d();
9  }
10 function c() {
11   // new NaN warning
12   y = x / 2;
13 }
14 function d() {
15   /* expensive
16    * computation
17    * independent
18    * of x, y, z */
19 }
20 function e() {
21   // old NaN warning
22   var n = "hi" - 23;
23   return n;
24 }
25
26 a();
27 b();
28 z = e();
```

Fig. 1: Code change in a program to be analyzed dynamically.

analysis, where results are expected in short time and without relying on users.

As a technique complementary to reducing the overhead of dynamic analysis, regression test selection [8] can reduce the number of tests to execute, which will also reduce the overall analysis time. Regression test selection partly addresses the problem but has the drawback that even after reducing the number of tests, the analysis may still consider code that is unaffected by a change. For example, in Figure 1, selecting the test that calls function `b` will nevertheless cause the dynamic analysis of function `d`, which is called by `b` but not affected by the change. Furthermore, as a practical limitation, we are not aware of any generic regression test selection tool available for JavaScript, which is the target language of this paper.

This paper addresses the problem of reducing the time required to run a dynamic analysis that has already been applied to a previous version of the analyzed code. We present *change-aware dynamic analysis*, or short CADA, an approach that turns a common class of dynamic analyses into change-aware analyses. The key idea is to reduce the number of code locations to dynamically analyze by focusing on functions that are impacted by a code change. When entering a function at runtime, the dynamic analysis checks whether this function is impacted, and only if it is impacted, analyzes the function. We find that in practice, most code changes impact only a small percentage of functions, enabling CADA to avoid analyzing most functions.

To find the functions affected by a change, a lightweight, static change impact analysis reasons about dependences between functions based on their read and write operations, and produces a function dependence graph. Starting from functions that are modified in a particular change, the approach then propagates the change to all possibly impacted functions. Key to the success of our approach is that the change impact analysis is efficient. To this end, we use a scalable yet not provably sound analysis. We find that, despite the theoretical unsoundness, the analysis is sufficient in practice, in the spirit of soundiness [9].

For the example in Figure 1, the change impact analysis determines that function `a` has been changed and that, as a result of this change, also function `c` is impacted by the change. By only applying the dynamic analysis to these functions, but not to any others, the approach reduces the number of analyzed functions from five to two. As a result, the change-aware analysis reports the newly introduced warning at line 12. To obtain the set of all warnings in the program, one can combine the warnings found by the change-aware analysis for the current version with warnings found for previous versions of the program.

To evaluate the CADA approach, we apply it to the version histories of seven popular JavaScript projects. Without our approach, analyzing the changed versions of the projects takes 133 seconds, on average over all commits. CADA reduces the analysis time by 40%, on average, and by at least 80% for 31% of all commits. Comparing CADA to naively running non-change-aware analyses on every version, we find that CADA does not miss any warnings reported by the class of dynamic analyses targeted in this paper.

In summary, this paper contributes the following:

- An approach for turning a common class of dynamic analyses into change-aware dynamic analyses without modifying the analyses themselves.
- A lightweight, static change impact analysis for JavaScript. We believe this analysis to be applicable beyond the current work, e.g., for helping developers to understand the impact of a potential change or for test case prioritization.
- Empirical evidence that the approach significantly reduces the time required to apply heavyweight dynamic analysis. As a result, such analyses become more amenable to regular use, e.g., at every commit.

II. Problem Statement

The following section describes the problem that this paper tackles. We consider dynamic analyses that reason about the behavior of a program to identify potential programming mistakes and that report warnings to the developer. Given a program P and inputs I, such a dynamic analysis A produces a set of warnings $A(P, I) = \mathcal{W}$. We assume that each warning is a pair of a code location $l \in \mathcal{L}$ and a description $d \in \mathcal{D}$ of the problem, i.e., $\mathcal{W} = \mathcal{L} \times \mathcal{D}$. For example, for the left-hand side of the motivating example in Figure 1, an analysis that identifies occurrences of `NaN` produces a warning (line 22, "NaN created").

This paper focuses on dynamic analyses that, to identify a particular warning, reason about runtime events triggered by a locally confined set of code locations, such as all code in a single function. We call this class of analyses *locally confined dynamic analyses*. Many existing analyses fall into this category, including checkers of code quality rules, such as those in DLint [6], and performance profilers, such as those in JITProf [4]. In contrast, locally confined dynamic analyses do not include analyses that require tracking values through the entire program, e.g., information flow analysis or whole program dependence analysis.

During the development process, a program evolves by applying changes. Let $P, P', P'', ...$ be different versions of a program. The goal of our work is to apply a dynamic analysis to every new version that appears during the development process, so developers can immediately react to any warnings caused by the most recent change. For a change from P to P', suppose to have a mapping $\mu : \mathcal{W} \rightarrow \mathcal{W}'$ of warnings from version P to warnings for the subsequent version P'. There are three sets of warnings:

- Old warnings that remain present in the new version: $\mathcal{W}_{old} = \{w' \in \mathcal{W}' \mid \exists w \in \mathcal{W} \text{ so that } \mu(w) = w'\}$.
- Newly introduced warnings: $\mathcal{W}_{new} = \{w' \in \mathcal{W}' \mid \nexists w \in \mathcal{W} \text{ so that } \mu(w) = w'\}$.
- Warnings that got fixed by the change: $\mathcal{W}_{fixed} = \{w \in \mathcal{W} \mid \nexists w' \in \mathcal{W}' \text{ so that } \mu(w) = w'\}$.

A naive but computationally expensive approach to obtain these three sets is to apply the analysis to every version, which yields a set of warnings for each version:

$$A(P, I) = \mathcal{W}, \quad A(P', I) = \mathcal{W}', \quad ...$$

The problem addressed in this paper is how to reduce the time required to apply a locally confined dynamic analysis to a new version of a program, after a previous version has already been analyzed. Specifically, we are looking for a change-aware variant of the analysis called A_{CA}, with the following two properties:

$$\mathcal{W}'_{CA} = \mathcal{W}_{old} \cup \mathcal{W}_{new} \text{where } A_{CA}(P, P', I) = \mathcal{W}'_{CA} \quad (1)$$

$$time(A_{CA}(P, P', I)) < time(A(P', I)) \quad (2)$$

The first condition guarantees that the change-aware analysis does not miss any newly introduced warnings compared to a full analysis and reports all warnings remaining from the previous version. The second condition ensures that the execution time taken by the change-aware analysis is lower than the time taken by the full analysis. For the example in Figure 1, the change-aware analysis should find the old warning at line 22 and the newly introduced warning at line 12, while taking less time than an analysis of the entire code.

Given the abundance of existing dynamic analyses, we aim for an approach that does not focus on a specific analysis but that applies across several locally confined dynamic analyses and that can be used without changing the analyses themselves.

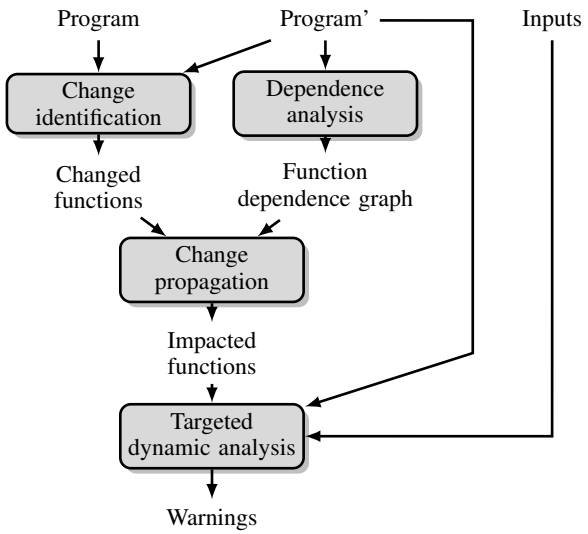

Fig. 2: Overview of the approach.

III. Approach

To address the problem defined in Section II, we present CADA, an approach that turns a locally confined dynamic analysis into a change-aware analysis. Figure 2 summarizes our approach. The input to CADA are two versions of the same program and a set of inputs to exercise the program, such as a regression test suite. The core of the approach is a static change impact analysis that, given the two versions of the program, determines the set of code locations that are potentially affected by the change. The change impact analysis consists of three steps. First, CADA analyzes the differences between the two versions to determine the set of modified code locations ("Change identification"). This step is purely syntactic and does not reason about the effects of changes. Second, a dependence analysis of the modified program extracts a graph that summarizes the dependences between code locations ("Dependence analysis"). This step reasons about the semantics of the code based on calls and on read and write operations. Third, CADA propagates the information that a code location has been modified through the dependence graph to compute the set of all code locations impacted by the change ("Change propagation"). Finally, the set of impacted code locations is provided to the dynamic analysis, which analyzes the modified program while focusing on the impacted code ("Targeted dynamic analysis").

An important design decision is at what level of granularity to compute the impact of a change and which code locations to analyze dynamically. Three possible options are at the file level, at the function level, and at the statement level. A file level approach would determine for each source code file whether any code in this file is impacted by the change, and then perform the dynamic analysis on all code in the impacted files. A disadvantage of a file level approach is that, because a single file may contain large amounts of code, the approach may unnecessarily analyze large amounts of code. For example, many JavaScript libraries are implemented in a

single file to ease their distribution. In this case, any change in the library would cause the entire code to be analyzed, which defeats the purpose of change-aware analysis.

A statement level approach would compute the dependences between all statements in the code and then analyze only those statements that are impacted by a change. Even though this approach is the most precise among the three options, we discard it because of its computational cost. The ultimate goal of CADA is to save time, and precisely computing all statement-level dependences is likely to take a significant amount of time for complex programs.

To balance the tradeoff between precision and computational cost, CADA computes dependences at the function level. That is, the static change impact analysis determines which functions are affected by a change, and the dynamic analysis selectively analyzes only the impacted functions. To ease the presentation, we assume that all code is inside some function. This assumption is true for JavaScript, except for top-level scripts, which our implementation handles as special, top-level functions.

The remainder of this section presents the steps outlined in Figure 2 in detail.

A. Change Identification

The first step of CADA compares the original and the modified program to determine the *changed functions*, that is, the set of functions that have been syntactically modified. At first, the approach uses a standard line differencing tool, such as the Unix "diff" utility, to compute which lines in the modified program have been modified. These lines also include all newly added lines. Next, the approach parses the modified program into an AST and traverses the AST in a depth-first manner. During the traversal, CADA visits all nodes that represent functions, i.e., for JavaScript, function declaration and function expression nodes[2]. The approach keeps track of the code locations that mark the beginning and the end of each function. Finally, after traversing the ASTs of all files of the program, the approach matches each modified line to the innermost function that contains the line, and reports functions with at least one changed line as changed functions.

For the example in Figure 1, the line differencing tool identifies line 4 as the only modified line. After traversing the AST, the approach identifies function `a` as the only changed function.

B. Dependence Analysis

Based on the changed functions, CADA computes the set of *impacted functions*. Ideally, this set contains exactly all those functions whose execution is affected by the changed functions. To determine the impacted functions, we compute a graph that represents the dependences between functions:

Definition 1 (Function dependence graph)**.** *A function dependence graph is a directed graph $G_{dep} = (\mathcal{N}, \mathcal{E})$ where*

- *each node $f \in \mathcal{N}$ represents a function, and*
- *each edge $(f_1, f_2) \in \mathcal{E}$ represents that function f_2 depends on function f_1, i.e., that f_1 influences f_2.*

[2]See ES6 tree specification: https://github.com/estree/estree.

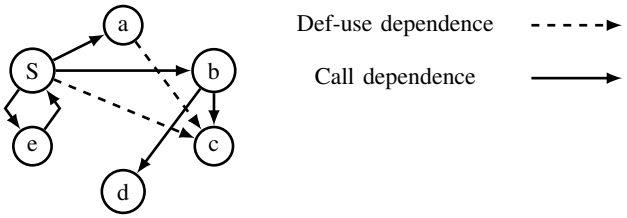

Fig. 3: Function dependence graph for the new version of the program in Figure 1.

CADA considers two kinds of dependences for the function dependence graph: definition-use dependences, which express that one function reads a value that may be defined in the other function, and call dependences, which express that one function may be called by another function. We represent these two kinds of dependences as two separate function dependence graphs G_{DU} and G_C, respectively. The overall function dependence graph is the union of both graphs, i.e., there is an edge between two particular functions if at least one of G_{DU} and G_C contains such an edge.

1) Definition-use Dependences: To compute definition-use dependences, we build upon an inter-procedural, flow-insensitive analysis of read-write relations between code locations. Specifically, given a set $\mathcal{L}$ of code locations, the analysis extracts *definition-use pairs* (d, u), where $d, u \in \mathcal{L}$ and where d writes a value that may be read by u. For example, consider the code on the right-hand side of Figure 1. The assignment at line 4 writes into variable `x`, which may be read by the expression `x / 2` at line 12. Hence, the pair of locations $(line\ 4, line\ 12)$ is a definition-use pair.

CADA accesses the results of the definition-use analysis via two maps. The map $usesOf : \mathcal{L} \mapsto 2^{\mathcal{L}}$ assigns to a code location that writes a value the set of code locations that may read this value. Conversely, the map $defsOf : \mathcal{L} \mapsto 2^{\mathcal{L}}$ assigns to a code location that uses a value the set of code locations that may define this value.

To compute the function-level definition-use dependences of a program, CADA iterates over all locations of the program that write a value into memory. For each such location $d \in \mathcal{L}$, the analysis computes the set of locations $usesOf(d)$ that may read the value. Next, the analysis lifts these locations to the function level using an auxiliary map $fctOf : \mathcal{L} \to \mathcal{F}$ that assigns to each location its syntactically enclosing function $f \in \mathcal{F}$. Finally, the analysis adds function level edges that represent the definition-use relations to G_{DU}, specifically, an edge $(fctOf(d), fctOf(u)))$ for each $u \in usesOf(d)$.

For example, consider Figure 3, where the dashed edges show the definition-use dependences between the functions in the running example in Figure 1. The S node represents the entire script; all other nodes represent functions. There are edges (a, c) and (S, c) because the use of `x` at line 12 may read either the value written at line 4, which is contained in

function `a`, or the value written at line 1, which is contained in the script S.

2) Call Dependences: In addition to definition-use dependences, CADA also considers caller-callee dependences between functions and summarizes them into a function dependence graph G_C. For a call of f_{callee} by a function f_{caller}, the analysis creates two kinds of dependence edges. First, it adds an edge (f_{caller}, f_{callee}) because a change in f_{caller} may impact whether and how f_{callee} is called. Second, the analysis adds an edge (f_{callee}, f_{caller}) if and only if f_{callee} returns a value back to f_{caller}. To resolve function calls, CADA again builds upon the definition-use analysis by querying *defsOf* for the location where a function is defined.

We refine this basic approach by removing a particular kind of call dependence that arises due to a common coding idiom in JavaScript. The coding idiom addresses the lack of encapsulation in JavaScript files by wrapping all code of a file into an immediately invoked function expression (IIFE) of the form: `(function() { /* code */ })()`. Since all code inside the function is called from the IIFE, there would be a direct or indirect call dependence from the IIFE to all functions invoked in the file. As a result, any change of code in the IIFE would cause all functions to be marked as impacted. To avoid adding these overly conservative dependences, CADA identifies IIFEs and does not consider any call dependence edge starting from an IIFE.

For our running example, the solid edges in Figure 3 show the call dependences. Most dependences are uni-directional, except for (S, e) and (e, S) because function `e` returns a value to the script S.

3) Computing Dependences Efficiently: Because the overall goal of CADA is to speed up the dynamic analysis, it is crucial that the static dependence analysis is efficient, in particular for complex programs. Ideally, the function dependence graph contains an edge (f, g) if and only if f definitely influences g. However, soundly and precisely computing this graph is practically impossible for real-world JavaScript code, e.g., because of dynamically computed property names, dynamically loaded code, and effects of native functions that are unknown to the static analysis. Instead of aiming for a sound static dependence analysis, CADA efficiently approximates a sound and precise dependence graph. Specifically, the analysis does not handle the `with` construct, code that is dynamically loaded via `eval`, dynamically computed property names, and side effects of getter and setter functions.

The analysis transforms the program into an AST and computes dependences in three passes through the AST. The first two passes compute the definition-use pairs by computing the scope and type of both variables and object properties through an inter-procedural, flow-insensitive analysis. The result of the first two passes are the *defsOf* and *usesOf* maps.

Based on these maps, the third pass computes the function-level definition-use dependences and call-graph dependences. Table I summarizes the actions performed during the third pass, which traverses the AST in a depth-first manner. The actions add edges to the two graphs G_{DU} and G_C, and they update two auxiliary data structures: a stack F of enclosing functions and the set $\mathcal{F}_{returns}$ of functions that return a value. For each relevant node type, the table shows how the analysis updates these data structures when entering the node before traversing the node's children ("Pre") and after having traversed its children ("Post"). Actions 1 and 2 maintain the stack of enclosing functions. Actions 3 and 4 implement the extraction of definition-use dependences described in Section III-B1. Actions 5 and 6 extract call dependences as described in Section III-B2. After the AST traversal, to finalize the call dependence graph G_C, the analysis adds an inverse edge for each existing edge (f_{caller}, f_{callee}) in G_C if and only if f_{callee} is in the set $\mathcal{F}_{returns}$ of functions that return a value.

C. Change Propagation

Based on the function dependence graph, the next step of CADA is to propagate the impact of a source code change through the graph. At first, the approach marks all changed functions, as described in Section III-A, in the function dependence graph. Then, CADA computes the transitive closure of the marked nodes by following the dependence edges in the graph. As a result, all nodes that are directly or indirectly impacted by the source code change are marked. CADA reports the set of functions that correspond to the marked nodes as the *impacted functions*.

For the running example, function `a` is the only changed function. Propagating this change along the edges in Figure 3 yields a set of two impacted functions: `a` itself and `c`, which is influenced by `a` via a definition-use dependence.

D. Targeted Dynamic Analysis

The final step of CADA is to reduce the overhead of a dynamic analysis by focusing the analysis on the impacted parts of the code. The main idea is to selectively analyze only those functions of the program that are impacted by a change. That is, whenever the execution of the analyzed program enters a function, the analysis checks whether to analyze this function. If the function is not an impacted function, then the function body is executed without any analysis, i.e., our change-aware approach avoids the analysis overhead for this function.

Selectively analyzing code at the function level has the advantage that the check whether to analyze a piece of code is done only once per function execution. More fine-grained selective analysis, e.g., at the level of individual statements, would involve many more checks. As each check imposes additional runtime overhead, a fine-grained selective analysis might defeat the purpose of selectively analyzing particular code locations.

For example, suppose to execute the right-hand side of Figure 1. CADA focuses the analysis on the two impacted functions `a` and `c`. That is, the other three functions and the main script of the program are not analyzed, effectively reducing the overhead imposed by the analysis. Furthermore, suppose that the dynamic analysis checks for operations that produce a `NaN` value. The targeted dynamic analysis detects

TABLE I: Actions performed to extract dependences during a depth-first AST traversal.

	Node type	When	Action
1	Script, function declaration, or function expression	Pre	Push the script or function to stack F
2	Script, function declaration, or function expression	Post	Pop the script or function from stack F
3	Variable declaration or assignment expression d	Pre	For each $u \in usesOf(d)$, add edge (f_d, f_u) to G_{DU} where $f_d = topOf(F)$ and $f_u = fctOf(u)$
4	Object expression $\{p_1 : v_1, \ldots, p_n : v_n\}$	Pre	For each p_i and for each $u \in usesOf(p_i)$, add edge (f_d, f_u) to G_{DU} where $f_d = topOf(F)$ and $f_u = fctOf(u)$
5	Call expression with callee f_{callee}	Pre	For each function $f^i_{callee} \in defsOf(f_{callee})$, add edge $(f_{caller}, f^i_{callee})$ to G_C where $f_{caller} = topOf(F)$
6	Return statement	Pre	Add $topOf(F)$ to $\mathcal{F}_{returns}$

the newly introduced `NaN`-warning at line 12. In contrast, the dynamic analysis does not again find the `NaN`-warning at line 22, which is already known from analyzing the previous version of the program.

E. Reporting Warnings

After analyzing the changed version of the program with a targeted dynamic analysis, CADA reports all warnings detected by the dynamic analysis as well as warnings found in the previous version. To map warnings across versions, we use the Unix "diff" utility to map lines from the previous version of the code to lines from the changed version of the code. Reporting all mapped warnings from the previous version would be incorrect because a change may fix a warning. To avoid incorrectly reporting fixed warnings, we only map previously detected warnings that occur in functions not impacted by the change. Because these warnings certainly have not been fixed by the change (otherwise, their surrounding function would be an impacted function), propagating them from the previous version is correct. In contrast, all warnings within impacted functions are detected by the targeted dynamic analysis, and we do not have to map them from previously found warnings.

IV. Implementation

We implement CADA for dynamic analyses of JavaScript code. The static change impact analysis builds upon Esprima[3] for parsing and upon Tern [10] for computing definition-use information. The output of the static analysis is a list of functions that need to be analyzed dynamically. The selective dynamic analysis is implemented on top of Jalangi [2]. It supports selective analysis at the function level by generating two variants of each function body: one with the original code and one with instrumented code. Our implementation is open source and available for download.[4]

V. Evaluation

Our evaluation aims at answering the following research questions:

- How effective is CADA in reducing the overall analysis time of dynamic analyses?
- Does CADA miss any warnings that the non-change-aware version of a dynamic analysis finds, and if yes, which ones?
- How much does the configuration of each part of our approach impact the execution time and the correctness of CADA?
- How is the total time that CADA takes distributed across the individual steps of the approach?

[3]http://esprima.org

[4]https://github.com/drill89/Change-aware-Dynamic-Analysis

TABLE II: Projects used for the evaluation.

Project	Release	LoC	Test cases	Test exec. time (millisecs)	Changed fcts. (avg.)
Backbone	1.3.3	1,946	1,101	508	3.6
Backbone LayoutMgr	1.0.0	1,159	756	6367	1.5
BigNumber	2.4.0	2,737	80,355	1,432	2.3
Bootstrap Datepicker	1.7.0	2,115	900	2,107	2.7
Cal-Heatmap	3.5.4	3,476	1,244	4,179	1.8
Countdown	2.6.0	1,366	3,456	702	4.7
Underscore	1.8.3	1,626	1,541	4,826	2.3

A. Experimental Setup

To answer these questions, we evaluate our approach by applying 46 dynamic checkers to the version histories of seven popular JavaScript projects. Table II lists the projects we consider. To obtain different versions of these projects, we download ten consecutive commits just before the commit that corresponds to the release in Table II, excluding commits that do not change any code. The table shows the number of test cases in the developer-provided test suites and how long it takes to execute these test suites for the release given in the table. The last column of the table reports the percentage of functions that are syntactically changed in a single commit, on average.

To dynamically analyze these projects, we use the analyses provided by the DLint [6] and JITProf [4] tools. Both tools consist of a set of dynamic checkers, 39 in DLint and seven in JITProf, giving a total of 46 checkers. We include all checkers of these tools, without modifying them in any way. Almost all checkers are locally confined dynamic analyses. One exception is a checker in JITProf that searches for "inconsistent object layouts", i.e., an object usage pattern that often spans across multiple functions. The checker reports a warning only if it observes both the creation of an object and its usage. We

do not remove such non-locally confined checkers during our experiments to study their impact on our results (Section V-C).

All experiments are done on a machine with an Intel Core i5-3230 CPU with 2.6GHz and 6GB of memory, running 64-bit Ubuntu 14.04, node.js 4.4.5, and Chromium 50.0.2661.102.

B. Reduction of Analysis Time

The primary goal of our work is to reduce the time required to run a heavyweight dynamic analysis. To evaluate the effectiveness of CADA in achieving this goal, we analyze each version of each project in two ways. First, we run the full dynamic analysis on the given code, without taking into account what parts of the code have changed compared to the previous version. This run takes an amount of time t_{full}. Second, we run CADA, which analyzes what parts of the code have changed compared to the previous version. Running CADA, including its static change impact analysis, takes an amount of time t_{CADA}. To measure the reduction of analysis time by CADA, we compute the *performance improvement* as $1 - \frac{t_{CADA}}{t_{full}}$. This measure of reduction ignores the one-time effort of running the full analysis because this effort amortizes over time.

Figure 4 summarizes the performance improvements of CADA over full dynamic analysis. Each graph shows the improvements for all ten commits of a project, giving the results for DLint and JITProf separately. For example, applying the change-aware variant of DLint to BigNumber reduces the analysis time by over 99% for seven of the ten commits. CADA significantly reduces the analysis time for most commits, projects, and analyses. Overall, the approach reduces the analysis time by 40%, on average, and by at least 80% for 31% of all commits.

Even though CADA proves effective in most cases, the performance improvement is negative for some commits. For example, the approach adds between 8% and 12% of analysis time when analyzing four commits of Countdown with JITProf. The reason is that the additional time spent on static change impact analysis sometimes does not outweigh its benefits, because the set of functions found to be impacted contains most functions of the program. Fortunately, these cases are rare and only impose a relatively small negative performance impact. Section V-D discusses variants of our approach that further reduce the analysis time, at the expense of precision.

C. Warnings Missed by Change-Aware Analysis

Our approach may miss warnings introduced in a change for two reasons. First, due to the unsoundness of the static change impact analysis, CADA may mark a function as not impacted even though the change impacts the function. As a result, dynamically analyzing only the impacted functions would miss any warnings found in the miss-classified function. Second, applying CADA to a non-locally confined dynamic analysis may lead to missed warnings because the occurrence of a warning at an impacted code location l_w may depend on whether another, non-impacted code location l_o gets executed. For example, consider an analysis that tracks objects and warns about a particular object usage pattern, but that tracks an object only if the analysis observes the object creation. If the program creates an object at a non-impacted location l_o, a change-aware variant of the analysis will miss any misbehavior that occurs at an impacted code location l_w. Such an analysis falls outside of the class of locally confined dynamic analyses that CADA is designed for (Section II), and we therefore accept missing such warnings.

To evaluate how prevalent the two kinds of missed warnings are, we compute the set of all warnings introduced in a particular commit and compare this set to the warnings reported by CADA. For this purpose, we execute the full dynamic analysis on each version of the program, which yields sets $\mathcal{W}_1$, $\mathcal{W}_2$, etc. of warnings. Next, we compute for each pair of consecutive versions i and $i+1$, the set $\mathcal{W}_\Delta$ of warnings that are present in $\mathcal{W}_{i+1}$ but not in $\mathcal{W}_i$. After computing $\mathcal{W}_\Delta$ for a particular pair of consecutive versions, we check whether the set of warnings reported by CADA for the newer version lacks any of the warnings in $\mathcal{W}_\Delta$. If we find such warnings, we call them *missed warnings*.

The last column of the "CADA" block of Table III shows the percentage of all newly introduced warnings that the approach misses, on average per commit. For most projects and analyses, no warnings are missed. An exception to this overall result are runs of JITProf on Backbone LayoutManager, Bootstrap Datepicker, and Underscore. The reason for these missed warnings is the "inconsistent object layouts" checker in JITProf [4]. As discussed in Section V-A, this checker falls outside of the class of locally confined dynamic analyses that CADA is designed for.

Beyond the warnings missed by this single checker, we do not observe any other kind of missed warnings. In particular, we do not observe any missed warnings due to the imprecision of the static change impact analysis. This result confirms our design decision to use a practical static analysis that favors efficiency over soundness.

D. Comparison with Alternative Approaches

To evaluate whether CADA's benefits can be achieved with simpler means, we compare our approach to four simpler variants of the approach:

- *No removal of IIFE call dependences*. In this variant, we do not remove call dependences related to immediately invoked function expressions, as described in Section III-B2.
- *Only definition-use dependences*. This variant builds the function dependence graph based on only definition-use dependences, i.e., without considering call dependences.
- *Only call dependences*. Inversely, this variant considers only call dependences, without considering definition-use dependences.
- *Only changed functions*. This variant removes most of CADA's change impact analysis and considers exactly those functions as impacted that are syntactically mod-

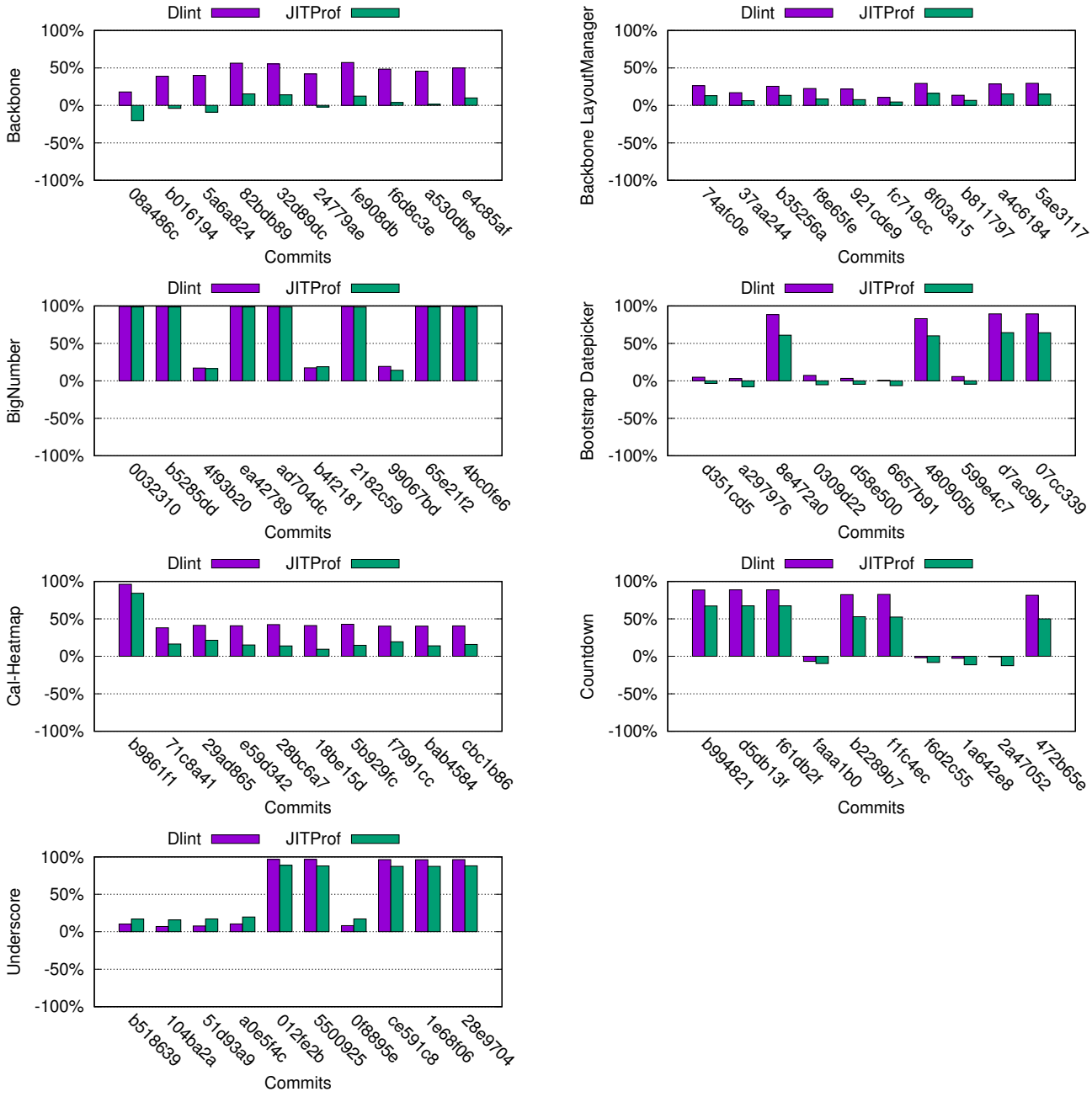

Fig. 4: Performance improvements of CADA over full dynamic analysis for different commits and projects.

ified by a change, i.e., we do not compute a function dependence graph.

We evaluate these variants of CADA with respect to their effectiveness in reducing the analysis time and their ability to find warnings introduced by a change. The execution time of each variant includes only those components of CADA necessary for the variant. For example, the "only changed functions" variant saves the time for computing a function dependence graph.

Table III summarizes the results of the comparison. Each block in the table presents the results of a variant, with the "CADA" block being the full approach. The values are averages over all commits. The results show that simplifying the approach reduces its effectiveness in one of two ways. On the one hand, the variant that keeps call dependences due to IIFEs increases the number of impacted functions, which reduces the performance improvements, while slightly reducing the number of missed warnings. Since the missed warnings are all due a single JITProf checker that is outside of the class of analyses targeted by this work, we use the full CADA approach as our default. On the other hand, the other three variants reduce the set of functions classified as impacted, which in turn increases the performance improvement. However, this improvement in performance comes at the cost of missing

TABLE III: Comparison of different variants of our approach. All values are averages over commits.

				CADA			Include IIFE deps.			Only call deps.			Only def-use deps.			Only changed fcts.		
Project	Time of full analysis (milliseconds)	Number of functions	Number of warnings	Impacted functions (%)	Perf. improvement over full analysis (%)	Missed warnings (%)	Impacted functions (%)	Perf. improvement over full analysis (%)	Missed warnings (%)	Impacted functions (%)	Perf. improvement over full analysis (%)	Missed warnings (%)	Impacted functions (%)	Perf. improvement over full analysis (%)	Missed warnings (%)	Impacted functions (%)	Perf. improvement over full analysis (%)	Missed warnings (%)
Backbone																		
- DLint	5,651	149.8	3	11	45	0	46	9	0	10	46	0	4	52	67	2	51	67
- JITProf	2,774	149.8	26	11	2	0	46	-28	0	10	16	0	4	9	4	2	3	13
Backbone LayoutManager																		
- DLint	7,675	72.6	1	15	23	0	30	15	0	6	27	0	11	25	0	2	28	0
- JITProf	6,476	72.6	49	15	11	6	30	5	0	6	14	71	11	13	10	2	14	100
BigNumber																		
- DLint	874,448	74	5	30	75	0	87	27	0	26	76	0	24	88	0	3	78	100
- JITProf	390,638	74	0	30	74	0	87	25	0	26	75	0	24	89	0	3	52	8
Bootstrap Datepicker																		
- DLint	39,442	135	1	37	39	0	63	9	0	27	40	100	7	71	0	2	86	100
- JITProf	11,855	135	163	37	23	86	63	-3	17	27	29	91	7	50	85	2	60	94
Cal-Heatmap																		
- DLint	224,201	226	161	80	46	0	80	46	0	17	86	99	8	91	100	1	96	0
- JITProf	52,325	226	2346	80	22	0	80	22	0	17	73	93	8	78	96	1	83	87
Countdown																		
- DLint	8,508	21.9	1	50	57	0	80	22	0	43	59	0	30	78	0	21	99	0
- JITProf	3,040	21.9	12	50	37	0	80	9	0	43	43	0	30	55	0	21	98	0
Underscore																		
- DLint	178,664	170	1	27	53	0	39	52	0	26	54	0	2	97	0	1	96	100
- JITProf	53,302	170	4	27	53	50	39	53	50	26	54	0	2	88	50	1	88	100
Average	132,786	121.3	198	36	40	10	61	19	5	22	49	32	12	63	29	5	67	55

warnings. For example, the variant that analyzes only the changed functions, saves 67% of the execution time of a full dynamic analysis, but misses 55% of the warnings, on average.

We conclude from these results that all steps of our approach are indeed required to effectively reduce the time required to run dynamic analyses without unnecessarily missing warnings.

E. Time Breakdown

To better understand where CADA spends its efforts, Figure 5 shows a breakdown of the execution time spent in different steps of the approach. Each bar shows the percentage of the overall time spent on (1) statically analyzing the impact of changes, (2) instrumenting the source code, and (3) executing the test suite while dynamically analyzing its execution. Step 1 reduces the time taken for step 3. Step 2 is independent of the change-awareness of an analysis and must also be done for a full dynamic analysis. Overall, the results show that the static analysis is cheap relative to the dynamic analysis, which confirms our decision to build upon a lightweight static analysis. For the Backbone project, CADA spends on above-average percentage of time on the change impact analysis. The reason is that this project has a relatively high number of functions to analyze statically compared to the time taken by the full dynamic analysis (see Table III).

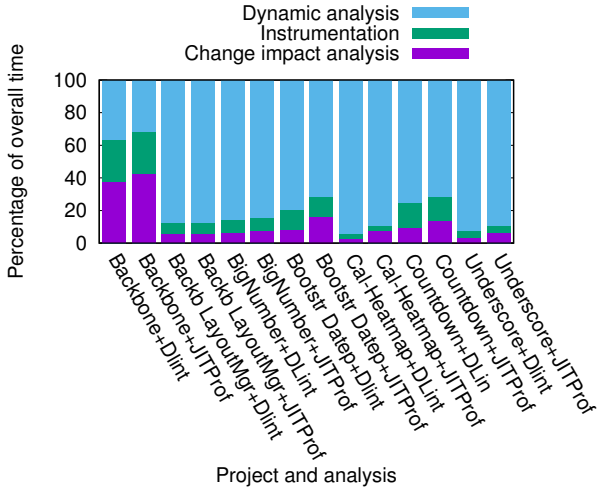

Fig. 5: Breakdown of the execution time spent in different steps of CADA.

VI. Related Work

a) Reducing the Overhead of Dynamic Analyses: A feedback-directed instrumentation technique reduces the overhead of crash diagnosis by distributing the analysis across multiple users [7]. Our work differs by considering arbitrary locally confined dynamic analyses instead of a single analysis, by reducing the overhead on a single machine instead of relying on a crowd of users, and by exploiting which code has been changed. JITProf [4] reduces its runtime overhead through function-level and instruction-level sampling. The sampling approach is oblivious of changes in the code and hence complementary to CADA.

Our approach targets instrumentation-based dynamic analyses. Alternative ways to implement dynamic analyses could also benefit from making the analysis change-aware. First, a dynamic analysis can be implemented by modifying the underlying runtime engine [11], [12]. Even though such an implementation typically yields lower overheads than a source-to-source translation, the overhead still remains non-negligible. Second, a dynamic analysis can be implemented by partly emulating the VM within the JavaScript application [11], which yields overheads comparable to those of Jalangi. Our lightweight, static change impact analysis could enable these approaches to selectively analyze particular functions, which would reduce the runtime overhead.

Staged analysis combines a dynamic analysis with a static analyses that eliminates unnecessary runtime checks. A staged analysis typically targets one particular dynamic analysis, such as a data detector [13], [14], a runtime monitor to check finite-state properties [15], and information flow analysis [16]. In contrast, CADA reduces the overhead of arbitrary locally confined dynamic analyses.

b) Change-Aware Program Testing and Analysis: KATCH [17] steers symbolic execution to code that has been changed. Our work differs by analyzing not only the changed but also other impacted code locations. There are several approaches for change-aware detection of concurrency bugs, e.g., by prioritizes schedules [18], by prioritizing tests [19] , by selecting interleavings to exercise [20], or by generating regression tests [21]. All these approaches focus on a particular kind of problem in concurrent programs, whereas CADA addresses a general class of dynamic analyses for arbitrary programs. Retrospec[5] is a prototype of a regression test selection tool for JavaScript. Regression test selection and change-aware dynamic analysis are complementary, and we leave combining them as future work.

c) Change Impact Analysis: Various existing analyses estimate the impact of a code change. A major difference to all existing change impact analyses is that we use the analysis to reduce the overhead of a dynamic analysis. Ryder and Tip [22] propose a call graph-based, static analysis that determines which tests exercise code impacted by a change and which changes may affect a particular test. Chianti [23] is an extended version of the idea. Both differ from CADA by reasoning about the impact of changes on tests, and vice versa, instead of the impact of changes on individual functions of the program. Another difference is that CADA considers both call dependences and definition-use dependences. Law and Rothermel [24] propose a dynamic change impact analysis that reports all code entities as impacted that are executed after a changed code entity. Other work reduces the runtime overhead of the technique based on an execute-after relation [25]. In contrast to both, CADA uses a static change impact analysis because the goal is to reduce the effort of dynamically analyzing the program. Our work also differs by computing impact based on definition-use dependences, finding code to be not impacted by a change even though this code executes after the changed code.

Tochal [26] is a change impact analysis for JavaScript that combines static and dynamic analysis to report to a developer which code entities may be affected by a change. In contrast to our work, Tochal relies on a dynamic analysis of the entire program. Our dependence analysis is related to static program slicing [27], [28]. In contrast to most slicing approaches, we do not aim for an executable subset of the program, but instead identify a subset of functions worth analyzing dynamically.

d) Dynamic Analysis of JavaScript: The dynamic nature of JavaScript has motivated various dynamic analyses beyond those considered here, including type inconsistency analysis [3], analysis of library conflicts [29], determinacy analysis [30], profilers to detect performance problems [5], [31], and dynamic data race detectors [32], [33], [34], [12]. A recent survey [35] gives a comprehensive summary of more dynamic analyses for JavaScript.

e) Static Analysis of JavaScript: Static analyses for JavaScript roughly fall into two categories. On the one hand, sound static analyses overapproximate the possible behavior of a program [36], [37], often at the price of considering only a subset of the language[38], [39], [40]. On the other hand, various analyses sacrifice soundness for practicality [41], [42], [43], e.g., by assuming that some dynamic language features are not used [44] or by adding runtime checks to compensate for the unsoundness of the static analysis [45]. The change impact analysis of CADA belongs to the second category, enabling the analysis to efficiently run on real-world code.

VII. Conclusions

This paper presents change-aware dynamic analysis, a technique that significantly reduces the runtime overhead of dynamic analyses by focusing the analysis on changed code locations. The approach is enabled by a lightweight static change impact analysis that computes a set of functions to analyze dynamically. We evaluate CADA by applying 46 dynamic checkers to the version histories of seven widely used JavaScript projects. Our results show that making the analyses change-aware reduces the overall analysis time by 40%, on average, and by at least 80% for 31% of all commits. Our work is a step toward bringing dynamic analysis at the fingertips of developers, who will benefit from finding problems early by frequently running these analyses.

[5]https://github.com/ericsteele/retrospec.js

REFERENCES

[1] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2007, pp. 89–100.

[2] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for JavaScript," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 488–498.

[3] M. Pradel, P. Schuh, and K. Sen, "TypeDevil: Dynamic type inconsistency analysis for JavaScript," in *International Conference on Software Engineering (ICSE)*, 2015.

[4] L. Gong, M. Pradel, and K. Sen, "JITProf: Pinpointing JIT-unfriendly JavaScript code," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 357–368.

[5] S. H. Jensen, M. Sridharan, K. Sen, and S. Chandra, "Meminsight: platform-independent memory debugging for javascript," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 345–356.

[6] L. Gong, M. Pradel, M. Sridharan, and K. Sen, "DLint: Dynamically checking bad coding practices in JavaScript," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 94–105.

[7] M. Madsen, F. Tip, E. Andreasen, K. Sen, and A. Møller, "Feedback-directed instrumentation for deployed javascript applications," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 899–910.

[8] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test., Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, 2012.

[9] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[10] M. Haverbeke, "Tern code analysis engine," http://ternjs.net/, Accessed in January 2017.

[11] E. Lavoie, B. Dufour, and M. Feeley, "Portable and efficient run-time monitoring of javascript applications using virtual machine layering." in *ECOOP*, 2014, pp. 541–566.

[12] E. Mutlu, S. Tasiran, and B. Livshits, "Detecting javascript races that matter," in *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2015.

[13] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," in *Conference on Programming Language Design and Implementation (PLDI)*, 2002, pp. 258–269.

[14] C. Flanagan and S. N. Freund, "Redcard: Redundant check elimination for dynamic race detectors," in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 255–280.

[15] E. Bodden, L. J. Hendren, and O. Lhoták, "A staged static program analysis to improve the performance of runtime monitoring," in *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2007, pp. 525–549.

[16] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for JavaScript," in *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009, pp. 50–62.

[17] P. D. Marinescu and C. Cadar, "Katch: high-coverage testing of software patches." in *ESEC/SIGSOFT FSE*, 2013, pp. 235–245.

[18] V. Jagannath, Q. Luo, and D. Marinov, "Change-aware preemption prioritization," in *ISSTA*, 2011, pp. 133–143.

[19] T. Yu, W. Srisa-an, and G. Rothermel, "Simrt: an automated framework to support regression testing for data races," in *ICSE*, 2014, pp. 48–59.

[20] V. Terragni, S. Cheung, and C. Zhang, "RECONTEST: effective regression testing of concurrent programs," in *ICSE*, 2015, pp. 246–256.

[21] M. Pradel, M. Huggler, and T. R. Gross, "Performance regression testing of concurrent classes," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 13–25.

[22] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, 2001, pp. 46–53.

[23] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. C. Chesley, "Chianti: a tool for change impact analysis of Java programs," in *OOPSLA*, 2004, pp. 432–448.

[24] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, 2003, pp. 308–318.

[25] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, 2005, pp. 432–441.

[26] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Hybrid DOM-sensitive change impact analysis for JavaScript," in *ECOOP*, 2015, pp. 321–345.

[27] M. Weiser, "Program slicing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352–357, 1984.

[28] F. Tip, "A survey of program slicing techniques," *J. Prog. Lang.*, vol. 3, no. 3, 1995. [Online]. Available: http://compscinet.dcs.kcl.ac.uk/JP/jp030301.abs.html

[29] J. Patra, P. N. Dixit, and M. Pradel, "Conflictjs: Finding and understanding conflicts between javascript libraries," in *ICSE*, 2018.

[30] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," in *PLDI*, 2013, pp. 165–174.

[31] M. Selakovic, T. Glaser, and M. Pradel, "An actionable performance profiler for optimizing the order of evaluations," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2017, pp. 170–180.

[32] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," in *Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[33] V. Raychev, M. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[34] S. Hong, Y. Park, and M. Kim, "Detecting concurrency errors in client-side java script web applications." in *ICST*, 2014, pp. 61–70.

[35] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C. alexandru Staicu, "A survey of dynamic analysis and test generation for javascript," *ACM Computing Surveys*, 2017.

[36] 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, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, 2015, pp. 505–519.

[37] F. Logozzo and H. Venter, "RATA: Rapid atomic type analysis by abstract interpretation—application to JavaScript optimization," in *CC*, 2010, pp. 66–83.

[38] R. Chugh, D. Herman, and R. Jhala, "Dependent types for JavaScript," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012, pp. 587–606.

[39] A. Guha, C. Saftoiu, and S. Krishnamurthi, "Typing local control and state using flow analysis," in *European Symposium on Programming (ESOP)*, 2011, pp. 256–275.

[40] P. Thiemann, "Towards a type system for analyzing JavaScript programs," in *European Symposium on Programming (ESOP)*, 2005, pp. 408–422.

[41] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of JavaScript applications in the presence of frameworks and libraries," in *ESEC/SIGSOFT FSE*, 2013, pp. 499–509.

[42] A. Feldthaus and A. Møller, "Semi-automatic rename refactoring for JavaScript," in *OOPSLA*, 2013.

[43] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of javascript," in *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, 2012, pp. 435–458.

[44] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *Symposium on Static Analysis (SAS)*. Springer, 2009, pp. 238–255.

[45] B. Hackett and S. Guo, "Fast and precise hybrid type inference for JavaScript," in *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 239–250.