SOFTWARE TESTING, VERIFICATION AND RELIABILITY
*Softw. Test. Verif. Reliab.* (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1576

# Automatic fault localization for client-side JavaScript

Frolin S. Ocariza, Jr.[*,†], Guanpeng Li, Karthik Pattabiraman and Ali Mesbah

*Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC, Canada*

SUMMARY

JAVASCRIPT is a scripting language that plays a prominent role in web applications today. It is dynamic, loosely typed and asynchronous and is extensively used to interact with the Document Object Model (DOM) at runtime. All these characteristics make JAVASCRIPT code error-prone; unfortunately, JAVASCRIPT fault localization remains a tedious and mainly manual task. Despite these challenges, the problem has received very limited research attention. This paper proposes an automated technique to localize JAVASCRIPT faults based on dynamic analysis, tracing and backward slicing of JAVASCRIPT code. This technique is capable of handling features of JAVASCRIPT code that have traditionally been difficult to analyse, including `eval`, anonymous functions and minified code. The approach is implemented in an open source tool called AUTOFLOX, and evaluation results indicate that it is capable of (1) automatically localizing DOM-related JAVASCRIPT faults with high accuracy (over 96%) and no false-positives and (2) isolating JAVASCRIPT faults in production websites and actual bugs from real-world web applications. Copyright © 2015 John Wiley & Sons, Ltd.

Received 1 December 2014; Revised 10 March 2015; Accepted 10 March 2015

KEY WORDS: JavaScript; fault localization; dynamic slicing; Document Object Model (DOM)

## 1. INTRODUCTION

Client-side JAVASCRIPT is increasingly used in web applications to increase their interactivity and responsiveness. JAVASCRIPT-based applications suffer from multiple dependability problems because of their distributed, dynamic nature, as well as the loosely typed semantics of JAVASCRIPT. A common way of gaining confidence in software dependability is through *testing*. Although testing of modern web applications has received increasing attention in the recent past [1–4], there has been little work on what happens after a test reveals an error. Debugging of web applications is still an expensive and mostly manual task. Of all debugging activities, locating the faults or *fault localization* is known to be the most expensive [5, 6].

The fault-localization process usually begins when the developers observe a failure in a web program spotted either manually or through automated testing techniques. The developers then try to understand the root cause of the failure by looking at the JAVASCRIPT code, examining the Document Object Model (DOM)[‡] tree, modifying the code (e.g. with alerts or tracing statements), running the application again and manually going through the initial series of navigational actions that led to the faulty state or running the corresponding test case.

Manually isolating a JAVASCRIPT fault's root cause requires considerable time and effort on the part of the developer. This is partly due to the fact that the language is not type-safe and has loose fault-detection semantics. Thus, a fault may propagate undetected in the application for

*Correspondence to: Frolin S. Ocariza, Jr., Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC, Canada.

†E-mail: frolino@ece.ubc.ca

‡DOM is a standard object model representing HTML at runtime. It is used for dynamically accessing, traversing and updating the content, structure and style of HTML documents.

a long time before finally triggering an exception. Additionally, faults may arise in third-party code (e.g. libraries, widgets and advertisements) [7] and may be outside the expertise of the web application's developer.

Further, faults may arise because of subtle asynchronous and dynamic interactions at runtime between the JAVASCRIPT code and the DOM tree, which make it challenging to understand their root causes. Indeed, from the authors' recent large-scale study of over 300 JAVASCRIPT bug reports, faulty interactions between the JAVASCRIPT code and the DOM—which are called *DOM-related faults*—comprise over 65% of all JAVASCRIPT bugs [8]. From the same study, it was also found that these DOM-related faults take longer to fix, on average, compared with all other fault types; hence, these DOM–JAVASCRIPT interactions are of particularly great concern when localizing faults. For these reasons, this paper focuses on the localization of DOM-related faults.

Although fault localization in general has been an active research topic [6, 9–11], automatically localizing web faults has received very limited attention from the research community. To the best of the authors' knowledge, automated fault localization for JAVASCRIPT-based web applications has not been addressed in the literature yet.

To alleviate the difficulties with manual web fault localization, this paper proposes an automated technique based on *dynamic backward slicing* of the web application to localize DOM-related JAVASCRIPT faults. The proposed fault localization approach is implemented in a tool called AUTOFLOX. In addition, AUTOFLOX has been empirically evaluated on six open-source web applications and three production web applications, along with seven web applications containing real bugs. The main contributions of this paper include the following:

- A discussion of the challenges surrounding JAVASCRIPT fault-localization, highlighting the real-world relevance of the problem and identifying DOM-related JAVASCRIPT faults as an important subclass of problems in this space;
- A fully automated technique for localizing DOM-related JAVASCRIPT faults, based on dynamic analysis and backward slicing of JAVASCRIPT code. Unlike the previous version of the technique described in the authors' ICST'12 paper [12] on which this work is based, the current technique can localize faults in the presence of the `eval` function, anonymous functions and minified JAVASCRIPT code. In addition, the current technique is capable of localizing multiple faults;
- An open-source tool, called AUTOFLOX, implementing the fault localization technique. AUTOFLOX has been implemented both as a stand-alone program that runs with the CRAWLJAX tool, as well as an Eclipse plugin;
- An empirical study to validate the proposed technique, demonstrating its efficacy and real-world relevance. The results of this study show that the proposed approach is capable of successfully localizing DOM-related faults with a high degree of accuracy (over 96%) and *no false positives*. In addition, AUTOFLOX is able to localize JAVASCRIPT faults in production websites, as well as 20 actual, reported bugs from seven real-world web applications.

## 2. CHALLENGES AND MOTIVATION

This section describes how JAVASCRIPT[§] differs from other traditional programming languages and discusses the challenges involved in localizing faults in JAVASCRIPT code. First, a JAVASCRIPT code fragment that is used as a running example throughout the paper is presented.

### 2.1. Running example

Figure 1 presents an example JAVASCRIPT code fragment to illustrate some of the challenges in JAVASCRIPT fault-localization. This code fragment is based on a fault in a real-world web application.[¶]

[§]In this paper, the term JAVASCRIPT is used to mean client-side JAVASCRIPT.

[¶]https://www.tumblr.com.

```
1  function changeBanner(bannerID) {
2    clearTimeout(changeTimer);
3    changeTimer = setTimeout(changeBanner, 5000);
4
5    prefix = "banner_";
6    currBannerElem = document.getElementById(prefix + currentBannerID);
7    bannerToChange = document.getElementById(prefix + bannerID);
8    currBannerElem.removeClassName("active");
9    bannerToChange.addClassName("active");
10   currentBannerID = bannerID;
11 }
12 currentBannerID = 1;
13 changeTimer = setTimeout(changeBanner, 5000);
```

Figure 1. Example JavaScript code fragment based on *tumblr.com*.

The web application pertaining to the code fragment in Figure 1 consists of a banner at the top of the page. The image shown on the banner cycles through four images periodically (every 5000 ms). The four images are each wrapped in `div` elements with DOM IDs `banner_1` through `banner_4`. The `div` element wrapping the image being shown is identified as 'active' via its `class` attribute.

In the preceding code, the `changeBanner` function (Lines 1 to 10) updates the banner image to the next one in the sequence by updating the DOM. Lines 12 and 13, which are outside the function, are executed at load time. Line 12 sets the value of variable `currentBannerID` to 1, indicating that the current image being shown is `banner_1`. Line 13 sets a timer that will asynchronously call the `changeBanner` function after 5 s (i.e. 5000 ms). After each execution of the `changeBanner` function, the timeout function is cleared and reset so that the image is changed again after 5 s.

The JavaScript code in Figure 1 will throw a `null` exception in Line 9 when executed. Specifically, in the `setTimeout` calls, `changeBanner` is invoked without being passed a parameter, even though the function is expecting an argument, referenced by `bannerID`. Omitting the argument will not lead to an interpretation-time exception; rather, the `bannerID` will be set to `undefined` when `changeBanner` executes. As a result, the second `getElementById` call will look for the ID '`banner_undefined`' in the DOM; because this ID does not exist, a `null` will be returned. Hence, accessing the `addClassName` method via `bannerToChange` in Line 9 will lead to a `null` exception.

Note that this error arises because of the loose typing and permissive error semantics of JavaScript. Further, to understand the root cause of the error, one needs to analyse the execution of both the JavaScript code and the DOM. However, once the fault has been identified, the fix is relatively straightforward, viz. modify the `setTimeout` call in Line 13 to pass a valid value to the `changeBanner` function.

### 2.2. JavaScript *fault localization*

Although JavaScript is syntactically similar to languages such as Java and C++, it differs from them in two important ways, which makes fault localization challenging.

**Asynchronous execution**: JavaScript code is executed asynchronously and is triggered by the occurrence of user-triggered events (e.g. click and mouse over), load events or events resulting from asynchronous function calls. These events may occur in different orders; although JavaScript follows a sequential execution model, it does not provide deterministic ordering. In Figure 1, the execution of the lines outside the `changeBanner` function is triggered by the load event, while the execution of the `changeBanner` itself is triggered asynchronously by a timeout event via the `setTimeout` call. Thus, each of these events triggered the execution of two different sequences of JavaScript code. In particular, the execution sequence corresponding to the load event is Line 12 → Line 13, while the execution sequence corresponding to the asynchronous event is Line 2 → Line 3 → Line 5 → Line 6 → Line 7 → Line 8 → Line 9.

In traditional programming languages, the goal of fault localization is to find the *erroneous lines* of code. For JavaScript, its asynchronous characteristic presents an additional challenge. The programmer will not only need to find the erroneous lines, but she will also have to map each executed sequence to the event that triggered their execution in order to understand the root cause of the fault. In addition, event handlers may overlap, as a particular piece of JAVASCRIPT code may be used by multiple event handlers. Thus, manual fault localization in client-side JAVASCRIPT is a tedious process, especially when many events are triggered.

**DOM interactions**: In a web application, JAVASCRIPT code frequently interacts with the DOM, which characterizes the dynamic HTML structure and elements present in the web page. As a result, the origin of a JAVASCRIPT fault is not limited to the JAVASCRIPT code; the JAVASCRIPT fault may also result from an error in the DOM. With regards to fault localization, the notion of an 'erroneous line' of code may not apply to JAVASCRIPT because it is possible that the error is in the DOM rather than the code. This is particularly true for *DOM-related* JAVASCRIPT *faults* (described in further detail in Section 3), which are defined as JAVASCRIPT faults that lead to either exceptions or incorrect DOM element outputs as a result of a DOM access or update. As a result, for such faults, one needs to formulate the goal of fault localization to isolate the first line of JAVASCRIPT code containing a call to a DOM access function (e.g. `getAttribute()`, `getElementById()`) or a DOM update function/property (e.g. `setAttribute()` and `innerHTML`) that directly causes JAVASCRIPT code to throw an exception or to update a DOM element incorrectly. This line is referred to as the *direct DOM interaction*.

For the example in Figure 1, the JAVASCRIPT exception occurs in Line 9, when the `addClassName` function is called on `bannerToChange`, which is `null`. The `null` value originated from Line 7, when the DOM access function `getElementById` returned `null`; thus, the direct DOM interaction is actually at Line 7. Note that even though this direct DOM interaction does not represent the actual 'erroneous' lines, which contain the missing parameter to the `changeBanner` function (Lines 3 and 13), knowing that `getElementById` in Line 7 returned `null` provides a hint that the value of either '`prefix`' or '`bannerID`' (or both) is incorrect. Using this knowledge, the programmer can isolate the erroneous line of code as she has to track the values of only these two variables. While in this simple example, the direct DOM interaction line is relatively easy to find; in more complex code, the `null` value could propagate to many more locations, and the number of DOM interactions to consider could be much higher, making it challenging to identify the direct DOM interaction. This is the challenge addressed in this paper.

### 2.3. *Challenges in analysing* JAVASCRIPT *code*

In addition to the challenges described in the previous subsection, JAVASCRIPT also contains several features that complicate the process of analysing JAVASCRIPT code for fault localization. These are described subsequently.

**Eval**: JAVASCRIPT allows programmers to dynamically create code through the use of the `eval` method. This method takes a string value as a parameter, where the string evaluates into JAVASCRIPT code. Although alternatives to certain uses of `eval` have been introduced in the language (e.g. the `JSON` API), studies show that `eval` use remains pervasive among web developers [13].

The presence of `eval` poses a challenge to both manual and automatic analysis of JAVASCRIPT code. The reason is twofold. First, the string parameter to `eval` is typically not just a simple string literal, but rather, a concatenation of multiple string values whose value cannot be determined based on a simple source-code level inspection; hence, it is difficult to infer the JAVASCRIPT code generated by `eval` at runtime. Second, the scope of variables introduced in `eval` code is directly linked to the scope in which the `eval` call is made; hence, `eval` code cannot be analysed in isolation, but must be analysed in relation to where the `eval` call is made. These, in turn, make fault localization more difficult, because the developer cannot easily keep track of the values that are created or modified through `eval`.

**Anonymous functions**: Because JAVASCRIPT treats functions as first-class citizens in the form of `Function` literals, programmers can define functions without providing them with a name; these unnamed functions are known as *anonymous functions*. Hence, when tracking the propagation of a JAVASCRIPT fault, it does not suffice to identify the lines of code involved in the propagation solely based on the function name, particularly if a JAVASCRIPT fault originates from or propagates through an anonymous function.

**Minified code**: Before deploying a web application, it is common practice for web developers to *minify* their JAVASCRIPT code, which compresses the code into one line. While this minification process reduces the size of JAVASCRIPT files, it also makes JAVASCRIPT code more difficult to read and analyse. This makes it very difficult to localize faults in minified code, as the developer will have a hard time keeping track of the relevant lines of code.

## 3. SCOPE OF THE PAPER

In prior work by the authors [7], it was found that deployed web applications experience on average four JAVASCRIPT exceptions (manifested as error messages) during execution. Further, in a follow-up study [8], it was found that over 65% of these JAVASCRIPT faults experienced by web applications are *DOM-related faults*. Formally, a fault is considered *DOM-related* if the corresponding error propagates into the parameter value of a DOM API method, such as `getElementById` and `querySelector`. In addition, these DOM-related faults comprise 80% of the highest impact JAVASCRIPT faults, according to the same study. Because of their prominence and severity, *this current paper focuses on the study of DOM-related faults*.

DOM-related JAVASCRIPT faults can be further divided into two classes, listed subsequently:

1. **Code-terminating DOM-related JAVASCRIPT faults**: A DOM access function returns a `null`, `undefined` or incorrect value, which then propagates into several variables and eventually causes an exception.
2. **Output DOM-related JAVASCRIPT faults**: A DOM update function sets the value of a DOM element property to an incorrect value without causing the code to halt.

The fault localization approach described in this paper can localize code-terminating DOM-related JAVASCRIPT faults automatically, requiring only the URL of the web application and the DOM elements needed to reproduce the failure as input from the user. Hence, in the sections that follow, it is assumed that the fault being localized is a code-terminating fault. However, note that the proposed approach can also support output DOM-related JAVASCRIPT faults, but the approach would only be semiautomatic, as the user must also provide the location of the failing line of code to initiate the localization process.

For code-terminating DOM-related JAVASCRIPT faults, the direct DOM interaction is the DOM access function that returned the `null`, `undefined` or incorrect value and is referred to as the *direct DOM access*.

## 4. APPROACH

The proposed fault localization approach consists of two phases: (1) *trace collection*, and (2) *trace analysis*. The trace collection phase involves crawling the web application and gathering traces of executed JAVASCRIPT statements until the occurrence of the failure that halts the execution. After the traces are collected, they are parsed in the trace analysis phase to find the direct DOM access. The two phases are described in detail in this section. A block diagram of the approach is shown in Figure 2. The usage model of the proposed approach is first described.

### *4.1. Usage model*

Because the focus is on fault localization, it is assumed that the failure whose corresponding fault needs to be localized has been detected before the deployment of the proposed technique. Further, it is also assumed that the user is able to replicate the failure during the localization process, either through a test case or by knowing the sequence of user events that would trigger the failure.

The approach is designed to automate the fault localization process. The only manual intervention required from the user is at the very beginning, where the user would have to specify which elements in the web application to click (during the trace collection phase) in order for the failure to occur.

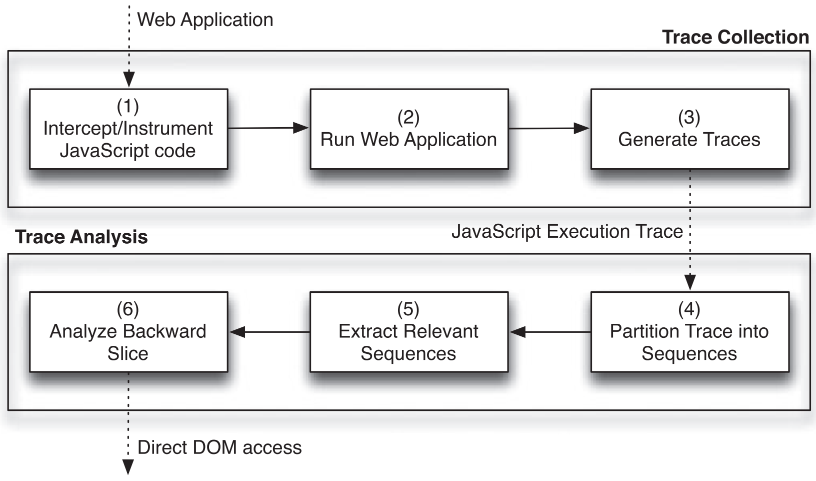

Web Application
Trace Collection
(1) Intercept/Instrument JavaScript code
(2) Run Web Application
(3) Generate Traces
JavaScript Execution Trace
Trace Analysis
(6) Analyze Backward Slice
(5) Extract Relevant Sequences
(4) Partition Trace into Sequences
Direct DOM access

Figure 2. Block diagram illustrating the proposed fault localization approach.

```
Trace Record Prefix:
  changeBanner:::4
Variables:
  currentBannerID (global): 1
  changeTimer (global): 2
  bannerID (local): none
  prefix (local): none
  currBannerElem (local): none
  bannerToChange (local): none
```

Figure 3. Example trace record for Line 5 of the running example from Figure 1.

The output of the approach is the direct DOM access corresponding to the fault being localized and specifies (1) the function containing the direct DOM access, (2) the line number of the direct DOM access relative to this function and (3) the JAVASCRIPT file containing the direct DOM access.

### 4.2. *Trace collection*

In the trace collection phase, the web application is crawled (by systematically emulating the user actions and page loads) to collect the trace of executed JAVASCRIPT statements that eventually lead to the failure. This trace is generated through on-the-fly instrumentation of *each line* of client-side JAVASCRIPT code before it is passed on to and loaded by the browser (box 1, Figure 2). Thus, for every line $l$ of JAVASCRIPT code executed, the following information is written to the trace: (1) the function containing the line, (2) the line number relative to the function to which it belongs, (3) the names and scopes (global or local) of all the variables within the scope of the function and (4) the values of these variables prior to the execution of the line. In the example in Figure 1, the order of the first execution is as follows: Line 12 → Line 13 → Line 2 → Line 3 → Line 5 → Line 6 → Line 7 → Line 8 → Line 9. Thus, each of these executed lines will have an entry in the trace corresponding to it. The trace record for Line 5 is shown in Figure 3. Note that in this figure, the trace record prefix contains the name of the function and the line number relative to this function; the variable names, scopes and values are also shown, and other variables, which have not been assigned values up to the current line, are marked with 'none'. In the figure, `bannerID`'s value is recorded as 'none' because this parameter is unspecified in the `setTimeout` call.

In addition to the trace entries corresponding to the executed lines of JAVASCRIPT code, three special markers, called FAILURE, ASYNCCALL and ASYNC, are added to the trace. The FAILURE marker is used in the trace analysis phase to determine at which line of JAVASCRIPT code the exception was thrown; if a line $l$ is marked with the FAILURE marker, then the value $l.failure$ is set to `true`. The ASYNCCALL and ASYNC markers address the asynchronous nature of JAVASCRIPT execution as described in Section 2. In particular, these two markers are used to determine the points in the program where asynchronous function calls have been made, thereby simplifying the process of mapping each execution trace to its corresponding event. If a line $l$ is marked with the ASYNC or ASYNCCALL marker, then the values $l.async$ or $l.asynccall$, respectively, are set to `true`.

The FAILURE marker is added when a failure is detected (the mechanism to detect failures is discussed in Section 5). It contains information about the exception thrown and its characteristics. In the example in Figure 1, the FAILURE marker is placed in the trace after the entry corresponding to Line 9, as the null exception is thrown at this line.

The second marker, ASYNCCALL, is placed after an asynchronous call to a function (e.g. via the `setTimeout` function). Each ASYNCCALL marker contains information about the caller function and a unique identifier that distinguishes it from other asynchronous calls. Every ASYNC-CALL marker also has a corresponding ASYNC marker, which is placed at the beginning of the asynchronous function's execution and contains the name of the function and the identifier of the asynchronous call. In the example in Figure 1, an ASYNCCALL marker is placed in the trace after the execution of Line 13, which has an asynchronous call to `changeBanner`. The corresponding ASYNC marker is placed before the execution of Line 2, at the beginning of the asynchronously called function `changeBanner`.

To insert the ASYNCCALL and ASYNC markers, the known asynchronous functions in JAVASCRIPT are overridden by a trampoline function that sets up and writes the ASYNCCALL marker to the trace. The trampoline function then calls the original function with an additional parameter indicating the identifier of the asynchronous call. This parameter is written to the trace within the called function along with the ASYNC marker to uniquely identify the asynchronous call.

## *4.3. Trace analysis*

Once the trace of executed statements has been collected, the trace analysis phase begins. The goal of this phase is to analyse the trace entries and find the direct DOM access responsible for the JAVASCRIPT failure. First, the approach partitions the trace into *sequences*, where a sequence $(l_1, l_2, \ldots, l_n)$ represents the series of JAVASCRIPT statements $l_1, l_2, \ldots, l_n$ that were triggered by the same event (e.g. a page load). Each sequence corresponds to exactly one event. This step corresponds to box 4 in Figure 2. As mentioned in the previous section, the executed JAVASCRIPT program in the example in Figure 1 consists of two sequences: one corresponding to the load event and the other corresponding to the time-out event.

After partitioning the trace into sequences, the algorithm looks for the sequence that contains the direct DOM access (box 5 in Figure 2). This is called the *relevant sequence*. The relevant sequence $\rho$ is initially chosen to be the sequence that contains the FAILURE marker,[‖] that is, at the beginning of the algorithm, $\rho$ is initialized as follows:

$$\rho \leftarrow (l_1, l_2, \ldots, l_n) \iff \exists l_i \in \{l_1, l_2, \ldots, l_n\},\ l_i.failure = \texttt{true} \tag{1}$$

This marker will always be the last element of the relevant sequence, because the execution of the sequence must have halted once the failure occurred; hence, it suffices to check if $l_n.failure = \texttt{true}$ in expression (1). The direct DOM access will be found within the initial relevant sequence provided the sequence was not triggered by an asynchronous function call but rather by the page load or user-triggered event. However, if the relevant sequence was triggered asynchronously, that is, it begins with an ASYNC marker, then the sequence containing the corresponding asynchronous call (i.e. with the ASYNCCALL marker) is prepended to the relevant sequence to create the new relevant sequence. This process is continued recursively until the top of the trace is reached or the sequence does not begin with an ASYNC marker.

In the running example, the relevant sequence is initially set to the one corresponding to the timeout event and consists of Lines 2, 3, 5, 6, 7, 8 and 9 (see Sequence 2 in Figure 4). Because the relevant sequence begins with an ASYNC marker, the sequence containing the asynchronous call (see Sequence 1 in Figure 4) is prepended to it to create the new, final relevant sequence. However,

[‖]For output-related DOM-related JAVASCRIPT faults, the FAILURE marker is replaced by an analogous marker that represents the failure line identified by the user.

```
Sequence 1:
  root:::12 (Line 12)
  root:::13 (Line 13)
  root:::ASYNC_CALL - ID = 1
Sequence 2:
  changeBanner:::ASYNC - ID = 1
  changeBanner:::1 (Line 2)
  changeBanner:::2 (Line 3)
  changeBanner:::4 (Line 5)
  changeBanner:::5 (Line 6)
  changeBanner:::6 (Line 7)
  changeBanner:::7 (Line 8)
  changeBanner:::8 (Line 9) - FAILURE
Relevant Sequence:
  root:::12 (Line 12)
  root:::13 (Line 13)
  changeBanner:::1 (Line 2)
  changeBanner:::2 (Line 3)
  changeBanner:::4 (Line 5)
  changeBanner:::5 (Line 6)
  changeBanner:::6 (Line 7) **
  changeBanner:::7 (Line 8)
  changeBanner:::8 (Line 9) - FAILURE
```

Figure 4. Abridged execution trace for the running example showing the two sequences and the relevant sequence. Each trace record is appended with either a marker or the line number relative to the function. Numbers in parentheses refer to the line numbers relative to the entire JAVASCRIPT file. `root` refers to code outside a function. The line marked with a (**) is the direct DOM access, and the goal of this design is to correctly identify this line as the direct DOM access.

there are no more sequences left in the trace, and the process terminates. Although in this example, the relevant sequence consists of all executed statements, this will not always be the case, especially in complex web applications where many events are triggered.

Once the relevant sequence has been found, the algorithm starts locating the direct DOM access within that sequence (box 6 in Figure 2). To do so, it analyses the backward slice of the variable in the line marked with the FAILURE marker, that is, the line $l$ such that $l.failure = \texttt{true}$. If the line $l$ itself contains the direct DOM access, the process is halted, and the line is identified as the direct DOM access. If not, a variable called `null_var` is introduced to keep track of the most recent variable to have held the `null` value.

The initial value of `null_var` is inferred from the error message contained in the FAILURE marker. The message is typically of the form ***x** is null*, where **x** is the identifier of a variable; in this case, the initial value of `null_var` is set to the identifier **x**. The relevant sequence is traversed backward and `null_var` is updated based on the statement encountered:

1. If the statement is an assignment of the form `null_var = new_var`, `null_var` is set to the identifier of `new_var`.
2. If it is a return statement of the form `return ret_var;`, where the return value is assigned to the current `null_var` in the calling function, `null_var` is set to the identifier of `ret_var`.
3. If it is a function call of the form `foo(..., arg_var ,...)` where `foo()` is a function with `arg_var` as one of the values passed, and the current `null_var` is the parameter to which `arg_var` corresponds in the declaration of `foo()`, `null_var` is set to the identifier of `arg_var`.

If the line does not fall into any of the preceding three forms, it is ignored, and the algorithm moves to the previous line. Note that although syntactically valid, an assignment of the form `null_var = new_var1 op new_var2 op ...`, where `op` is a binary operator, makes little semantic sense as these operations are not usually performed on DOM element nodes (For instance, it makes no sense to add two DOM element nodes together). Hence, it is assumed that such assignments will not appear in the JAVASCRIPT code. Therefore, at every statement in the code, `null_var` takes a unique value. In addition, this implies that there can only be one possible direct DOM access along the null propagation path.

The algorithm ends when `new_var`, `ret_var` or `arg_var` is a call to a DOM access function. The line containing this DOM access is then identified as the direct DOM access.

In the example in Figure 1, the `null_var` is initialized to `bannerToChange`. The trace analyser begins at Line 9, where the FAILURE marker is placed; this is also the last line in the relevant sequence, as seen in Figure 4. Because this line does not contain any DOM access functions, the algorithm moves to the previous line in the relevant sequence, which is Line 8. It then determines that Line 8 does not take on any of the earlier three forms and moves to Line 7. The algorithm then determines that Line 7 is of the first form listed earlier. It checks the `new_var` expression and finds that it is a DOM access function. Therefore, the algorithm terminates and identifies Line 7 as the direct DOM access.

### 4.4. *Support for challenging cases*

As explained in Section 2.3, programmers typically use features of the JAVASCRIPT language that complicate the process of analysing JAVASCRIPT code. This subsection describes how the approach was extended to handle these features.

*4.4.1. Eval.* As described in Section 4.2, the approach instruments each line of JAVASCRIPT code to retrieve three pieces of information, namely the containing function, the line number and an array of the names and values of all in-scope variables. The function that is responsible for adding this information to the execution trace is as follows, where the parameters correspond to the retrieved information.

$$\texttt{send(functionName, lineNo, variableArray)} \tag{2}$$

The `send()` function is included prior to every line of JAVASCRIPT code, which is useful for retrieving trace records for statically loaded code; however, the `send()` function does not collect trace records for JAVASCRIPT code generated through `eval`.

A naïve approach for extending the approach to handle `eval` would be to simply add a call to `send()` prior to every line in the string passed to the `eval` call. The problem with this approach is that the `eval` parameter is not necessarily a string literal; hence, its value may not be known until the `eval` call is made at runtime. To make the approach more general, every call to `eval` in the JAVASCRIPT code is replaced with a call to a wrapper function called `processEval()`. This function first evaluates the *string value* of the expression passed to `eval`. Thereafter, the function parses this string value and adds a call to the `send()` function prior to each expression statement in the parsed string; this generates a new string, which comprises of the original parameter to `eval`, but with a call to `send()` prior to each statement. Finally, this new string is passed to `eval` in order for the corresponding code to execute. The approach described is illustrated in Figure 5.

**New variables.** Note that it is possible for new variables to be declared in `eval` code. Hence, the approach needs a way to update the array of variables that is passed to the `send()` function. To do this, an array object is created at the beginning of every function in the JAVASCRIPT code.

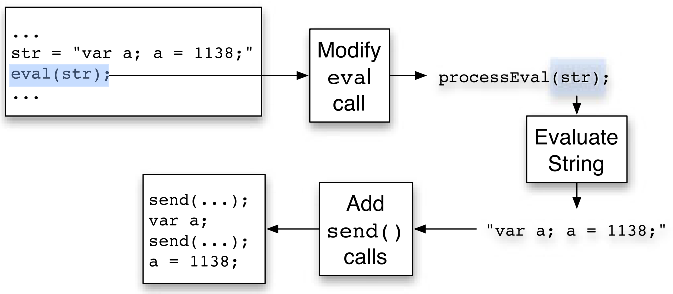


Figure 5. Example illustrating the approach for supporting `eval`.

The array is initialized with the names and scopes of all the variables declared within the function and is updated with new variables whenever a call to `processEval()` is made.

Because the string value passed to `eval` is parsed separately from the rest of the JAVASCRIPT code (i.e. the code outside `eval`), `processEval()` may inaccurately label the scope of a variable defined in the `eval` code as 'global'. In order to make sure the variables declared in `eval` are marked with the correct scope (i.e. local or global), a *scope marker* is passed to `processEval()` as a parameter. This scope marker contains the value `root` if the `eval` call is made outside a function; otherwise, the scope marker is assigned the name of the function. Thus, if the scope marker has value `root`, the 'global' markings are retained for the new variables; otherwise, the 'global' markings are changed to 'local'. This ensures that variable scopes are accurately recorded.

*4.4.2. Anonymous functions.* The initial approach relied on the names of the functions containing the line numbers to determine the dynamic backward slice. More specifically, during the trace collection phase, the technique records the name of the function, which is then included in the corresponding trace record. During the trace analysis phase, the technique then fetches the name of the function from the trace record so that it knows where to find the line of JAVASCRIPT code that needs to be analysed.

Unfortunately, this approach does not work for anonymous functions, because they are not given a name. To account for this limitation, the trace collection scheme was modified so that it assigns a unique name for every anonymous function encountered during the crawling. In particular, each anonymous function is assigned a name of the form *anonymous-file-script-line*, where *file* is the file name; *script* is the index of the script tag containing the function (i.e. the order that the script tag appears in the file); and *line* is the line number of the function relative to the script tag in which it is defined. Note that the script tag is only applicable to JAVASCRIPT code embedded in `.html` files. If the code is included in a `.js` file, *script* is simply assigned zero.

In the trace analysis phase, if a trace whose corresponding function name is of the form *anonymous-file-script-line* is encountered, then the trace must correspond to a line of code located in an anonymous function. The location of the line of code is determined by taking the *file*, *script* and *line* portions of the function name, and the line of code is fetched once found.

*4.4.3. Minified code.* In order to handle minified code, the approach first 'beautifies' (i.e. unminifies) this code. The trace collection and trace analysis phases will then both proceed as before, but this time, operating on the beautified version of the code. The problem is that by operating on the beautified version, the approach will output the line number of the direct DOM access *in the beautified version*; because this beautified version is transparent to the developer, this line number will not be very meaningful.

Therefore, the challenge, in this case, is in mapping every line number in the beautified version with the *column* number in the minified version. In most cases, this mapping is achieved by performing a string match with the minified version and identifying the starting column of the matching string. However, this approach will not always work because of the possibility of identical lines; for instance, if the running example were originally minified, and Figure 1 is the beautified version, then Lines 3 and 13—which are identical—will lead to multiple matches with the minified version. To account for this possibility, a regular expression is used to identify all the identical lines in the beautified version. The identical lines in the beautified version are then sequentially assigned an index, and a regular expression is used to find *all* the matches in the minified version. In this case, the line with the $n$th index is mapped to the column where the $n$th match is located. This is illustrated in Figure 6.

## 4.5. Assumptions

The described approach makes a few simplifying assumptions, listed subsequently. In the evaluation described in Section 6, the correctness of the approach will be assessed on various open-source web applications, thus evaluating the reasonableness of these assumptions in the real world.

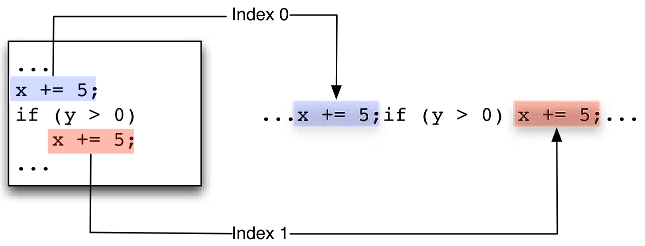


Figure 6. Example illustrating the mapping from the beautified version to the minified version, in the approach for supporting minified code.

1. The JAVASCRIPT error is manifested in a null exception, where the null value is originated from a call to a DOM access function.
2. There are no calls to recursive functions in the relevant sequence. More specifically, the approach relies on the (*name*, *file* and *script*) tuple—where *name* is either the function name or, in the case of anonymous functions, some uniquely assigned name—to distinguish functions from each other. Because traces that point to a recursive function map to the same tuple, the approach cannot distinguish between calls to the same line from different recursion levels.
3. There are no object property accesses in the null propagation path. In other words, the approach assumes that `null_var` will only be a single identifier and not a series of identifiers connected by the dot operator (e.g. `a.property` and `this.x`)

## 5. TOOL IMPLEMENTATION

The approach described in Section 4 has been implemented in an automated tool called AUTOFLOX[**] using the Java programming language. In addition, a number of existing tools are used to assist in the trace collection phase, including RHINO [14] for parsing and instrumenting the JAVASCRIPT code and jsbeautifier [15] for beautifying minified code.

AUTOFLOX has been implemented in two different interfaces.

**CRAWLJAX interface.** In the first interface, AUTOFLOX prompts the user for the URL of the web application containing the fault and crawls this application to perform trace collection. Here, the CRAWLJAX [16] tool is used to systematically crawl the web application and trigger the execution of JAVASCRIPT code corresponding to user events. Other tools such as WaRR [17], Mugshot [18] and Selenium [19] can aid in the reproduction phase. However, those tools require manually or programmatically interacting with the web application at hand. Thus, CRAWLJAX was used because of the level of automation and flexibility it provides. Prior to crawling the web application, the AUTOFLOX user can specify which elements in the web application the crawler should examine during the crawling process (otherwise the default settings are used). These elements should be chosen so that the JAVASCRIPT error is highly likely to be reproduced.[††] In this mode, only one fault can be localized at a time.

**Eclipse interface.** In the second interface, AUTOFLOX runs as an Eclipse IDE [20] plugin. Here, the programmer can develop her web application project on Eclipse; with the project files open, she can subsequently click the 'Run AUTOFLOX' button to run the tool. Doing so will open the Firefox web browser, which allows the user to replicate the fault by either interacting with the application or running a test case (e.g. a Selenium test case). Where applicable, AUTOFLOX will then output the direct DOM access each time a null exception is thrown. Note that in this interface, AUTOFLOX is able to localize multiple faults by assigning a unique ID to each exception encountered.

[**]http://ece.ubc.ca/~frolino/projects/autoflox/.

[††]While nondeterministic errors can be localized with AUTOFLOX, they may require multiple runs to reproduce the error (i.e. until the error appears).

The JAVASCRIPT code instrumentation and tracing technique used in the proposed approach are based on an extension of the INVARSCOPE [21] plugin to CRAWLJAX. The following modifications were made to INVARSCOPE in order to facilitate the trace collection process:

1. While the original INVARSCOPE tool only collects traces at the function entry and exit points, the modified version collects traces at every line of JAVASCRIPT code to ensure that the complete execution history can be analysed in the trace analysis phase.
2. The original INVARSCOPE does not place information on the scope of each variable in the trace; thus, it has been modified to retrieve this information and include it in the trace.
3. The modifications allow asynchronous function calls to be overridden and to place extra instrumentation at the beginning of each function to keep track of asynchronous calls (i.e. to write the ASYNCCALL and ASYNC markers in the trace).
4. Finally, `Try-Catch` handlers are placed around each function call in the JAVASCRIPT code in order to catch exceptions and write FAILURE markers to the trace in the event of an exception.

Note that the tool allows the user to exclude specific JAVASCRIPT files from being instrumented. This can speed up the trace collection process, especially if the user is certain that the code in those files does not contain the direct DOM access.

Finally, the trace analysis phase has also been added as a part of the AUTOFLOX tool implementation and requires no other external tools.

## 6. EMPIRICAL EVALUATION

### 6.1. *Goals and research questions*

An empirical study has been conducted to evaluate the accuracy and real-world relevance of the proposed fault localization approach.

The research questions that are answered in the evaluation are as follows:

**RQ1**: What is the fault localization accuracy of AUTOFLOX? Are the implementation assumptions reasonable?
**RQ2**: Is AUTOFLOX capable of localizing bugs from real-world web applications?
**RQ3**: What is the performance overhead of AUTOFLOX on real-world web applications?

### 6.2. *Methodology*

The subsequent subsections address each of the preceding questions. An overview of the evaluation methodology used to answer each research question is shown subsequently.

To answer **RQ1**, AUTOFLOX is run on six open-source web applications and three production websites. DOM-related JAVASCRIPT faults are injected into the applications and AUTOFLOX is run to localize the direct DOM accesses corresponding to the faults.

To address **RQ2**, AUTOFLOX is subjected to 20 bugs (which satisfy the fault model) that have previously been observed and reported for seven open-source web applications. Most of these bugs are from the authors' previous study [8].

The performance (**RQ3**) is measured by calculating the overhead incurred by the instrumentation and the time it takes for the tool to find the direct DOM access.

Note that the experiments were performed on an Ubuntu 12.04 platform using the Firefox v. 33.0.2 web browser. The machine used was a 2.66 GHz Intel Core 2 Duo, with 4 GB of RAM.

### 6.3. *Accuracy of* AUTOFLOX

To answer RQ1, a fault injection experiment was performed on six open-source web applications, shown in Table I. As seen in this table, the applications consist of thousands of lines of JAVASCRIPT code each. Fault-injection was used to establish the ground truth for measurement of the accuracy of AUTOFLOX. However, the fault injection process was not automated. Rather, a search was first made for calls to DOM access functions—either from the DOM API or from popular JavaScript libraries—that return `null`, such as `getElementById()`, `getAttribute()` and `$()`. The

Table I. Results of the experiment on open-source web applications, assessing the accuracy of AUTOFLOX.

| JAVASCRIPT web applications | Lines of JS code | # of mutations | # of direct DOM accesses identified | `eval` support increase | Anonymous support increase | Percentage identified |
|---|---|---|---|---|---|---|
| TASKFREAK | 3,044 | 39 | 39 (38) | +1 | — | 100% (97.4%) |
| TUDU | 11,653 | 9 | 9 (0) | — | +9 | 100% (0%) |
| WORDPRESS | 8,366 | 17 | 14 (9) | — | +5 | 82.4% (52.9%) |
| CHATJAVASCRIPT | 1,372 | 10 | 10 (10) | — | — | 100% (100%) |
| JSSCRAMBLE | 131 | 6 | 6 (6) | — | — | 100% (100%) |
| JS TODO | 241 | 2 | 2 (1) | — | +1 | 100% (50%) |
| Overall | | 83 | 80 (64) | +1 | +15 | 96.4% (77.1%) |

DOM, Document Object Model; JS, JavaScript.

faults were then manually injected by mutating the parameter of the DOM access function; this parameter mutation will ensure that the call to the function will return a `null` value, thereby leading to a null exception in a later usage. In order to demonstrate the effect of adding support for anonymous functions, `eval` and minified code, the fault injection experiment was run twice—once with these new modules enabled and once with the modules disabled.

Only one mutation is performed in each run of the application to ensure controllability. For each injection, the direct DOM access is the mutated line of JAVASCRIPT code. Thus, the goal is for AUTOFLOX to successfully identify this mutated line as the direct DOM access, based on the message printed due to the exception.

Furthermore, localization was performed on injected faults rather than actual faults because no known code-terminating DOM-related faults existed in these web applications at the time of the experiment. However, AUTOFLOX is also used to localize real faults that appear in seven other web applications, which is described in further detail in Section 6.4.

Table I shows the results of the experiments; the results for the case where the new modules are disabled are shown in parentheses. As shown in the table, with the new modules enabled, AUTOFLOX was able to identify the direct DOM access for all mutations performed in five of the six applications, garnering *100% accuracy* in these applications; when all six applications are considered, the overall accuracy was 96.4%. In contrast, when the new modules are disabled, only two of the applications (CHATJAVASCRIPT and JSSCRAMBLE) had perfect accuracies, and the overall accuracy was significantly lower, at 77.1%.

Taking a closer look at the unsuccessful cases when the new modules are disabled, it was found that AUTOFLOX was not able to accurately pinpoint the direct DOM access because in these cases, the dynamic backward slice included lines from anonymous function code and `eval` code. In particular, 15 of the unsuccessful cases resulted from the presence of anonymous function code, while four of the unsuccessful cases resulted from the presence of `eval` code. This result demonstrates that these features are commonly used in JAVASCRIPT code, and it is therefore important to add support for them, as has been done.

For the case where the new modules are enabled, the only application for which AUTOFLOX had imperfect accuracy was WORDPRESS, where it failed to detect three direct DOM access lines; in all three cases, AUTOFLOX generated an error message stating that the direct DOM access could not be found. Further analysis of the JAVASCRIPT code in WORDPRESS revealed that in these three unsuccessful cases, the dynamic backward slice included calls to the `setTimeout()` method, where the first parameter passed to the method is a function literal; this is currently not supported by AUTOFLOX.[‡‡] Note, however, that this is an implementation issue which does not fundamentally limit the design; one possible way to overcome this problem is by integrating the parameter of `setTimeout()` as part of the `eval`-handling module.

Overall, this experiment demonstrates that AUTOFLOX has an accuracy of 96.4% across the six open-source web applications. Note that AUTOFLOX had *no* false positives, that is, there were no

[‡‡]Note that in the running example, the `setTimeout()` call is passed a function *identifier*, which is different from a function *literal*.

Table II. Results of the experiment on production websites, assessing the robustness of AUTOFLOX (in particular, how well it works in production settings).

| Production website | Total number of faults | Number of direct DOM accesses identified | Anonymous support increase | Minified support increase |
|---|---|---|---|---|
| ubuntu.com | 1 | 1 (0) | +1 | — |
| Hacker News | 2 | 2 (2) | — | — |
| W3C School | 1 | 1 (0) | — | +1 |
| Overall | 4 | 4 (2) | +1 | +1 |

DOM, Document Object Model.

cases where the tool incorrectly localized a fault or said that a fault had been localized when it had not.

**Production website**: AUTOFLOX was also used to localize faults on three production websites, listed in Table II. For these websites, the faults are injected in the homepage, in a way similar to what was done in the fault injection experiment described earlier.[§§] As before, the experiment was run twice, enabling the new modules in one case and disabling them in the other.

As Table II shows, AUTOFLOX was able to identify all four of the direct DOM accesses, with the new modules enabled. In contrast, with the new modules disabled (whose results are shown in parentheses in Table II), the tool only identified two of the four direct DOM accesses. One of the unsuccessful cases (ubuntu.com) resulted from the presence of an anonymous function, while the other unsuccessful case (W3C School) resulted from the presence of minified code (which is a common practice in many production websites).

*The overall implication of this study is that the assumptions made by* AUTOFLOX *are reasonable as they were followed both in the open-source applications and in the production websites*. Further, *the new features added to* AUTOFLOX *significantly boosted its accuracy*. Later in Section 7, the broader implications of the assumptions made by AUTOFLOX are discussed.

## *6.4. Real bugs*

In order to assess how well AUTOFLOX works on actual bugs that have appeared in real-world web applications, the authors collected 19 bug reports from the applications in a prior study [8] and subjected AUTOFLOX to them to see if it can successfully identify the direct DOM access. In addition, the authors found a real bug in the production website 'tumblr', which was also included in the experiment; this is the running example described earlier. These 20 bugs come from seven open-source web applications and libraries, shown in Table III and have all been fixed by their respective developers. Here, ground truth is established by comparing the direct DOM access output by AUTOFLOX with the actual direct DOM access, which is identified by the developers in the corresponding bug report (or, in the case of tumblr, by analysing the code to determine the fix).

In the end, AUTOFLOX was able to successfully identify the direct DOM access for *all 20 of the bugs*. Table III also shows the distance of the backward slice from the direct DOM access to the line where the `null` exception takes place (second–last column), and the number of functions spanned by the backward slice (last column). Note that finding the direct DOM access is nontrivial for some of these bugs. For example, one of the bugs in Moodle (MD2) had a dynamic backward slice that spanned multiple lines within the same function, and in fact, the variable to which the DOM element is being assigned is constantly being reused in that function to refer to other elements. In addition, the dynamic backward slice for a bug in WordPress (WP1) spanned over 40 lines, spread out across multiple functions; manually tracing through these lines and functions can evidently be a time-consuming process. This demonstrates the usefulness of AUTOFLOX, as well as its robustness, as it is capable of localizing real bugs in real web applications.

[§§]Because the authors did not have write access to the JAVASCRIPT source code for these websites, the mutation was performed by manually modifying the code intercepted by the proxy.

Table III. Web applications and libraries in which the real bugs to which AUTOFLOX is subjected appear.

| Application/Library | Number of bugs | Bug identifier | Identified by AUTOFLOX? | Backward slice Length (LOC) | Number of functions in backward slice |
|---|---|---|---|---|---|
| Joomla | 3 | JM1 | ✓ | 2 | 1 |
| | | JM2 | ✓ | 2 | 1 |
| | | JM3 | ✓ | 8 | 2 |
| Moodle | 6 | MD1 | ✓ | 2 | 1 |
| | | MD2 | ✓ | 5 | 1 |
| | | MD3 | ✓ | 7 | 2 |
| | | MD4 | ✓ | 8 | 2 |
| | | MD5 | ✓ | 2 | 1 |
| | | MD6 | ✓ | 1 | 1 |
| MooTools | 2 | MT1 | ✓ | 1 | 1 |
| | | MT2 | ✓ | 1 | 1 |
| Prototype | 1 | PT1 | ✓ | 1 | 1 |
| Tumblr | 1 | TB1 | ✓ | 1 | 1 |
| WikiMedia | 4 | WM1 | ✓ | 2 | 1 |
| | | WM2 | ✓ | 1 | 1 |
| | | WM3 | ✓ | 2 | 1 |
| | | WM4 | ✓ | 1 | 1 |
| WordPress | 3 | WP1 | ✓ | 44 | 4 |
| | | WP2 | ✓ | 5 | 1 |
| | | WP3 | ✓ | 10 | 2 |

LOC, lines of code.

Table IV. Performance results.

| Production website | Trace collection overhead (%) | Total time (s) |
|---|---|---|
| ubuntu.com | 63.3 | 50.3 |
| Hacker News | 30.9 | 28.8 |
| W3C School | 119.3 | 56.8 |
| Tumblr | 35.0 | 33.3 |

## 6.5. *Performance*

The performance overhead of AUTOFLOX is reported in this section. The measurements are performed on the production websites because production code is more complex than development code (such as the ones in the open-source web applications tested earlier) and, hence, incurs higher performance overheads. The following metrics are measured: (1) performance overhead due to instrumentation in the trace collection phase, and (2) time taken by the trace analyser to find the direct DOM access. To measure (1), the production websites are crawled using CRAWLJAX both with instrumentation and without instrumentation; the baseline is the case where the web application is run only with CRAWLJAX. For measuring (2), AUTOFLOX was run on the collected trace. Note that the Eclipse interface experiences similar overheads.

Table IV shows the performance measurements. As the table shows, the overhead incurred by the trace collection phase (average of three runs) ranges from 30.9% in Hacker News to 119.3% in W3C School. Also, on average, the trace analysis phase ran for 0.1 s in all four websites. Note that AUTOFLOX's trace collection module is only intended to be turned on when a fault needs to be localized—when interacting with a website as normal, the module will be off; hence, the high overheads in some websites (e.g. W3C School) are not expected to be problematic. Indeed, AUTOFLOX does not run for more than a minute in any of the websites, from trace collection to fault localization.

## 7. DISCUSSION

Some issues relating to the limitations of AUTOFLOX and some threats to the validity of the evaluation are now discussed.

### 7.1. *Limitations*

Currently, AUTOFLOX requires the user to specify the elements that will be clicked during the web application run to replicate the failure. This process can be tedious for the programmer if she is not aware of all the DOM elements (and their corresponding IDs) present in the web application and will often require the programmer to search for these elements in the source code of the web application. The Eclipse plugin version of AUTOFLOX mitigates this problem to a certain extent, by asking the user to replicate the failure by manually interacting with the web application; however, doing this for a large set of bugs may be tedious, and ways to automate this process without sacrificing accuracy are currently being explored.

One way to simplify the preceding task and effectively automate the process of identifying all the DOM IDs is to do a preliminary run of the web application that detects all the DOM elements —where all elements are considered clickable—and present this list of DOM elements to the user. However, this approach would have the disadvantage of having to run the web application multiple times, which would slow down the fault localization process. In addition, this approach may not be able to detect DOM elements created dynamically by the JAVASCRIPT code if only a subset of the web application is crawled.

As seen from the accuracy results in the evaluation, although AUTOFLOX is capable of handling calls to `eval` (or `eval`-like functions such as `setTimeout`) where a string (or a concatenation of strings) is passed as the parameter, it currently does not support the case where a function literal is passed as a parameter to this function. Based on the applications that were evaluated, passing function literals to `eval`-like functions does not seem to be a common practice among developers. Most parameters to `setTimeout`, for instance, are in the form of function identifiers, similar to the running example.

### 7.2. *Threats to validity*

An external threat to the validity of the evaluation is that only a limited number of web applications are considered to assess the correctness of AUTOFLOX. However, these applications have been chosen as they contain many lines of JAVASCRIPT code, thereby allowing multiple fault injections to be performed per application.

In terms of internal validity, a fault injection approach was used to emulate the DOM-related faults in the evaluation. The threat here is that the faults injected may not be completely representative of the types of faults that happen in the real world. Nonetheless, the bug report study from the authors' prior work [8] provides supportive evidence that the bugs that were injected are prominent and must therefore be considered. Further, AUTOFLOX was also tested on real bugs in one of the experiments, demonstrating its applicability in more realistic settings.

Finally, while AUTOFLOX is openly available, and the fault injection experiment on the six open-source web applications is replicable, the experiment on production websites is not guaranteed to be replicable, as the source code of these websites may change over time, and the authors do not have access to prior versions of the website.

## 8. RELATED WORK

Here, related work is classified into two broad categories: web application reliability and fault localization.

### *8.1. Web application reliability*

Web applications have been an active area of research for the past decade. This paper focuses on reliability techniques that pertain to JAVASCRIPT-based web applications, which are a more recent phenomenon.

**Static analysis.** There have been numerous studies to find errors and vulnerabilities in web applications through static analysis [22–25]. Because JAVASCRIPT is a difficult language to analyse statically, these techniques typically restrict themselves to a safe subset of the language. In particular, they do not model the DOM, or they oversimplify the DOM, which can lead to both false positives and false negatives. Jensen *et al.* [26] model the DOM as a set of abstract JAVASCRIPT objects. However, they acknowledge that there are substantial gaps in their static analysis, which can result in false-positives. In contrast, the proposed technique is based on dynamic execution and, as a result, does not suffer from false-positives.

**Testing and replay.** Automated testing of JAVASCRIPT-based web applications is an active area of research [2–4, 27]. ATUSA [2] is an automated technique for enumerating the state space of a JAVASCRIPT-based web application and finding errors or invariant violations specified by the programmer. JSart [28] and DODOM [3] dynamically derive invariants for the JAVASCRIPT code and the DOM, respectively. Finally, MUTANDIS [29] determines the adequacy of JAVASCRIPT test cases using mutation testing. However, none of these techniques focus on fault localization. Alimadadi *et al.* recently introduced a program comprehension tool called CLEMATIS [30], which maps user events to JAVASCRIPT code; although this tool can help the developer narrow down the list of JAVASCRIPT lines to consider, it does not pinpoint a precise fault location, as AUTOFLOX does.

WaRR [17], Mugshot [18] and Jalangi [31]—among others [32, 33]—replay a web application's execution after a failure in order to reproduce the events that led to the failure. However, they do not provide any support for localizing the fault and leave it to the programmer to do so. As shown in Section 2, this is often a challenging task.

Finally, tools such as Firefox's Firebug [34] plug-in exist to help JAVASCRIPT programmers debug their code. However, such tools are useful only for the bug identification phase of the debugging process and not the fault localization phase.

### *8.2. Fault localization*

Fault localization techniques isolate the root cause of a fault based on the dynamic execution of the application. They can be classified into spectrum-based and slicing-based.

**Spectrum-based fault localization** techniques [35–37] include Pinpoint [38], Tarantula [6], Whither [39] and MLNDebugger [36]. Additionally, MUSE [40] and FIFL [41] also perform spectrum-based fault localization based on injected mutants in traditional programs, focusing primarily on regression bugs. These techniques execute the application with multiple inputs and gather the dynamic execution profile of the application for each input. They assume that the executions are classified as success or failure and look for differences in the profile between successful and failing runs. Based on the differences, they isolate the parts of the application, which are likely responsible for the failure. However, spectrum-based techniques are difficult to adapt to web applications, as web applications are rarely deterministic, and hence they may incur false positives. Also, it is not straightforward to classify a web application's execution as success or failure, as the results depend on its usage [42].

**Slicing-based fault localization** techniques have been proposed by Agrarwal *et al.* [10] and Gupta *et al.* [43]. These techniques isolate the fault based on the dynamic backward slice of the faulty statement in the code. AUTOFLOX is similar to this body of work in that it also extracts the dynamic backward slice of the JAVASCRIPT statement that throws an exception. However, it differs in two ways. First, it focuses on errors in the DOM–JAVASCRIPT interaction. The DOM is unique to web applications, and hence, the other fault-localization techniques do not consider it. Second, JAVASCRIPT code is often executed asynchronously in response to events such as mouse clicks and time-outs and does not follow a deterministic control-flow (see Section 2.2 for more details).

**Web fault localization.** As a complementary tool to AUTOFLOX, the authors developed VEJOVIS [44], which is a JAVASCRIPT fault repair suggestion tool. This tool is similar to AUTOFLOX in that it also targets DOM-related faults and uses a backward slicing approach. However, unlike AUTOFLOX, which starts with the line of code that leads to the failure and tries to find the direct DOM access by examining the dynamic backward slice, VEJOVIS starts with the direct DOM access and examines its parameters to see how it can be fixed to match the DOM.

To the best of the authors' knowledge, the only papers apart from the current work that has explored fault localization in the context of web applications are those by Artzi *et al.* [45] and Samimi *et al.* [46]. Like AUTOFLOX, the goal of the tools proposed in these papers is to automatically localize web application faults, achieving high accuracies. However, their work differs from the current one in various aspects: (1) they focus on the server-side code, that is, PHP, while the current work focuses on the client-side; and (2) they localize HTML validation errors, while the current work's proposed approach localizes JAVASCRIPT faults. In addition, Artzi *et al.* have opted for a spectrum-based approach based on Tarantula, while AUTOFLOX is a dynamic slicing-based approach. To the best of the authors' knowledge, automated fault localization for JAVASCRIPT-based web applications has not been addressed in the literature.

## 9. CONCLUSIONS AND FUTURE WORK

This paper introduces a fault-localization approach for JAVASCRIPT-based web applications. The approach is based on dynamic slicing and addresses the two main problems that inhibit JAVASCRIPT fault localization, namely asynchronous execution and DOM interactions. Here, the focus is on DOM-related JAVASCRIPT faults, which is the most prominent class of JAVASCRIPT faults. The proposed approach has been implemented as an automated tool, called AUTOFLOX, which is evaluated using six open-source web applications and four production websites. The results indicate that AUTOFLOX can successfully localize over 96% of the faults, with no false-positives.

There are several ways in which the work outlined in this paper will be extended. First, the current work focuses on code-terminating JAVASCRIPT faults, that is, faults that lead to an exception thrown by the web application. However, not all DOM-related faults belong to this category. The design will therefore be extended to include a more automated technique for localizing non-code-terminating JAVASCRIPT faults. In addition, the empirical evaluation will be extended to perform user studies of the AUTOFLOX tool, in order to measure its ease of use and efficacy in localizing faults. This is also an avenue for future work.

## ACKNOWLEDGEMENTS

This research was supported in part by NSERC Discovery and Strategic Projects grants and a research gift from Intel Corporation.

## REFERENCES

1. Marchetto A, Tonella P, Ricca F. State-based testing of AJAX web applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE Computer Society: Lillehammer, Norway, 2008; 121–130.
2. Mesbah A, van Deursen A, Roest D. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering (TSE)* 2012; **38**(1):35–53.
3. Pattabiraman K, Zorn B. DoDOM: leveraging DOM invariants for web 2.0 application robustness testing. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, IEEE Computer Society: San Jose, CA, USA, 2010; 191–200.
4. Artzi S, Dolby J, Jensen SH, Møller A, Tip F. A framework for automated testing of JavaScript web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM: Honolulu, HI, USA, 2011; 571–580.
5. Vessey I. Expertise in debugging computer programs: a process analysis. *International Journal of Man-Machine Studies* 1985; **23**(5):459–494.
6. Jones J, Harrold M. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, ACM: Long Beach, CA, USA, 2005; 273–282.

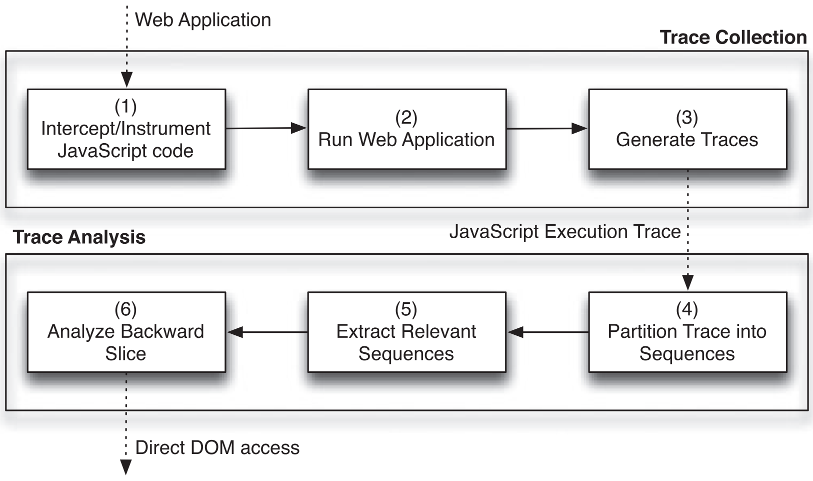
7. Ocariza F, Pattabiraman K, Zorn BG. JavaScript errors in the wild: an empirical study. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, IEEE Computer Society: Hiroshima, Japan, 2011; 100–109.
8. Ocariza F, Bajaj K, Pattabiraman K, Mesbah A. An empirical study of client-side JavaScript bugs. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE Computer Society: Baltimore, MD, USA, 2013; 55–64.
9. Abreu R, Zoeteweij P, Gemund AJC. Spectrum-based multiple fault localization. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society: Auckland, New Zealand, 2009; 88–99.
10. Agrawal H, Horgan JR, London S, Wong WE. Fault localization using execution slices and dataflow tests. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, IEEE: Toulouse, France, 1995; 143–151.
11. Cleve H, Zeller A. Locating causes of program failures. In *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM: St. Louis, MO, USA, 2005; 342–351.
12. Ocariza F, Pattabiraman K, Mesbah A. AutoFLox: an automatic fault localizer for client-side JavaScript. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE Computer Society: Montreal, QC, Canada, 2012; 31–40.
13. Richards G, Hammer C, Burg B, Vitek J. The eval that men do: a large-scale study of the use of eval in JavaScript applications. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP),* Lancaster, UK, 2011; 52–78.
14. Rhino. Available from: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino [last accessed 28 October 2011].
15. JSBeautifier. Available from: http://www.jsbeautifier.org/ [last accessed 1 November 2014].
16. Mesbah A, van Deursen A, Lenselink S. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)* 2012; **6**(1):53–82.
17. Andrica S, Candea G. WaRR: high fidelity web application recording and replaying. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN),* Hong Kong, China, 2011; 403–410.
18. Mickens J, Elson J, Howell J. Mugshot: deterministic capture and replay for JavaScript applications. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI),* San Jose, CA, USA, 2010; 159–174.
19. Selenium. Available from: http://seleniumhq.org [last accessed 28 October 2011].
20. Eclipse IDE. Available from: http://www.eclipse.org/ [last accessed 1 November 2014].
21. Groeneveld F, Mesbah A, van Deursen A. Automatic invariant detection in dynamic web applications. *Technical Report TUD-SERG-2010-037*, Delft University of Technology, 2010.
22. Guarnieri S, Livshits B. Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of the USENIX Security Symposium (SSYM)*, ACM: Montreal, QC, Canada, 2009; 151–168.
23. Guha A, Krishnamurthi S, Jim T. Using static analysis for AJAX intrusion detection. In *Proceedings of the International Conference on the World Wide Web (WWW),* Madrid, Spain, 2009; 561–570.
24. Zheng Y, Bao T, Zhang X. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the International Conference on the World Wide Web (WWW)*, ACM: Hyderabad, India, 2011; 805–814.
25. Bae S, Cho H, Lim I, Ryu S. SAFEWAPI: web API misuse detector for web applications. *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, ACM: Hong Kong, China, 2014; 507–517.
26. Jensen SH, Madsen M, Møller A. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ACM: Szeged, Hungary, 2011; 59–69.
27. Mirshokraie S, Mesbah A, Pattabiraman K. JSeft: automated JavaScript unit test generation. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE Computer Society: Graz, Austria, 2015.
28. Mirshokraie S, Mesbah A. JSART: JavaScript assertion-based regression testing. In *Proceedings of the International Conference on Web Engineering (ICWE)*, Springer: Berlin, Germany, 2012; 238–252.
29. Mirshokraie S, Mesbah A, Pattabiraman K. Efficient JavaScript mutation testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE Computer Society: Luxembourg, Luxembourg, 2013; 74–83.
30. Alimadadi S, Sequeira S, Mesbah A, Pattabiraman K. Understanding JavaScript event-based interactions. In *Proceedings of the International Conference on Software Engineering (ICSE),* Hyderabad, India, 2014; 367–377.
31. Sen K, Kalasapur S, Brutch T, Gibbs S. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, ACM: Saint Petersburg, Russia, 2013; 488–498.
32. Burg B, Bailey R, Ko AJ, Ernst MD. Interactive record/replay for web application debugging. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, ACM: St. Andrews, UK, 2013; 473–484.
33. Yildiz A, Aktemur B, Sozer H. Rumadai: a plug-in to record and replay client-side events of web sites with dynamic content. In *Proceedings of the Workshop on Developing Tools as Plug-ins (TOPI)*, IEEE: Zurich, Switzerland, 2012; 88–89.
34. Firebug. Available from: http://getfirebug.com [last accessed 28 October 2011].

35. Bandyopadhyay A, Ghosh S. Tester feedback driven fault localization. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE: Montreal, QC, Canada, 2012; 41–50.
36. Zhou J, Zhang H, Lo D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE: Zurich, Switzerland, 2012; 14–24.
37. Shu G, Sun B, Podgurski A, Cao F. Mfl: method-level fault localization with causal inference. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE: Luxembourg, Luxembourg, 2013; 124–133.
38. Chen MY, Kiciman E, Fratkin E, Fox A, Brewer E. Pinpoint: problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, IEEE Computer Society: Bethesda, MD, USA, 2002; 595–604.
39. Renieris M, Reiss SP. Fault localization with nearest neighbor queries. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society: Montreal, QC, Canada, 2003; 30–39.
40. Moon S, Kim Y, Kim M, Yoo S. Ask the mutants: mutating faulty programs for fault localization. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE: Cleveland, OH, USA, 2014; 153–162.
41. Zhang L, Zhang L, Khurshid S. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, ACM: Indianapolis, IN, USA, 2013; 765–784.
42. Dobolyi K, Weimer W. Modeling consumer-perceived web application fault severities for testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, ISSTA'10, ACM: Trento, Italy, 2010; 97–106.
43. Zhang X, He H, Gupta N, Gupta R. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the International Symposium on Automated Analysis-Driven Debuggin (AADEBUG)*, ACM: Monterey, CA, USA, 2005; 33–42.
44. Ocariza F, Pattabiraman K, Mesbah A. Vejovis: suggesting fixes for JavaScript faults. In *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM: Hyderabad, India, 2014; 837–847.
45. Artzi S, Dolby J, Tip F, Pistoia M. Practical fault localization for dynamic web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM: Cape Town, South Africa, 2010; 265–274.
46. Samimi H, Schäfer M, Artzi S, Millstein T, Tip F, Hendren L. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE: Zurich, Switzerland, 2012; 277–287.

**Research Article**

**Automatic fault localization for client-side JavaScript**

Frolin S. Ocariza, Jr., Guanpeng Li, Karthik Pattabiraman and Ali Mesbah


Web Application
Trace Collection
(1) Intercept/Instrument JavaScript code
(2) Run Web Application
(3) Generate Traces
JavaScript Execution Trace
Trace Analysis
(4) Partition Trace into Sequences
(5) Extract Relevant Sequences
(6) Analyze Backward Slice
Direct DOM access

This work proposes an approach for automatically localizing Document-Object-Model-related (i.e. DOM-related) JavaScript faults in AJAX-based web applications. The technique has been implemented in a tool called AutoFLox, and it is capable of localizing faults in a wide array of web applications, including minified applications, as well as those using anonymous functions and eval. An evaluation of the proposed approach shows that it is accurate, with a recall of 96% and no false positives, and can localize real bugs from real-world web applications. Copyright © 2015 John Wiley & Sons, Ltd.