AutoFLox: An Automatic Fault Localizer for Client-Side JavaScript

Frolin S. Ocariza, Jr., Karthik Pattabiraman Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
{*frolino, karthikp, amesbah*}*@ece.ubc.ca*

***Abstract*—JAVASCRIPT is a scripting language that plays a prominent role in modern web applications today. It is dynamic, loosely typed, and asynchronous. In addition, it is extensively used to interact with the DOM at runtime. All these characteristics make JAVASCRIPT code error-prone and challenging to debug. JAVASCRIPT fault localization is currently a tedious and mainly manual task. Despite these challenges, the problem has received very limited attention from the research community. We propose an automated technique to localize JAVASCRIPT faults based on dynamic analysis of the web application, tracing, and backward slicing of JAVASCRIPT code. Our fault localization approach is implemented in an open source tool called AUTOFLOX. The results of our empirical evaluation indicate that (1) DOM-related errors are prominent in web applications, i.e., they form at least 79% of reported JAVASCRIPT bugs, (2) our approach is capable of automatically localizing DOM-related JAVASCRIPT errors with a high degree of accuracy (over 90%) and no false-positives, and (3) our approach is capable of isolating JAVASCRIPT errors in a production web application, viz., Tumblr.**

***Index Terms*—JavaScript, fault localization, dynamic slicing, Document Object Model (DOM)**

I. INTRODUCTION

Client-side JAVASCRIPT is increasingly used in web applications to increase their interactivity and responsiveness. JAVASCRIPT-based applications suffer from multiple dependability problems due to 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], [2], [3], [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 an error in a web program either spotted manually or through automated testing techniques. The developers then try to understand the root cause of the error by looking at the JAVASCRIPT code, examining the Document Object Model (DOM)[1] 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 error'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 error detection semantics. Thus, an error may propagate undetected in the application for a long time before finally triggering an exception. Further, errors may arise due to subtle asynchronous and dynamic interactions at runtime between the JAVASCRIPT code and the DOM tree, which make it challenging to understand their root causes. Finally, errors may arise in third-party code (e.g., libraries, widgets, advertisements), and may be outside the expertise of the web application's developer.

Although fault localization in general has been an active research topic [6], [7], [8], [9], automatically localizing web faults has received very limited attention from the research community. To the best of our 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, in this paper, we propose an automated technique based on *dynamic backward slicing* of the web application to localize JAVASCRIPT faults. Our fault localization approach is implemented in a tool called AUTOFLOX. We have empirically evaluated AUTOFLOX on three open-source web applications and a production web application, viz., Tumblr. The main contributions of this paper include:

- A discussion of the challenges surrounding JAVASCRIPT fault localization, highlighting the real-world relevance of the problem and identifying DOM-related JAVASCRIPT errors as an important sub-class 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;
- An open-source tool, called AUTOFLOX, implementing the fault localization technique;
- An empirical study to validate the proposed technique, demonstrating its efficacy and real-world relevance. Our examination of four bug-tracking systems indicates that DOM-related errors form the majority of reported JAVASCRIPT errors, i.e., *at least 79%* of the reported JAVASCRIPT errors were DOM-related. The results of our study show that our approach is capable of successfully

[1] 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.

```
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);
```

Fig. 1. Example JAVASCRIPT code fragment based on *tumblr.com*.

localizing DOM-related faults with a high degree of accuracy (over 90%) and *no false positives*.

II. CHALLENGES AND MOTIVATION

In this section, we describe how JAVASCRIPT[2] differs from other traditional programming languages and discuss the challenges involved in localizing faults in JAVASCRIPT code. First, we present a JAVASCRIPT code fragment that we will use as a running example throughout the paper.

A. 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 the code of a real-world web application.[3]

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 milliseconds). 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 above 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 seconds (i.e., 5000 milliseconds). After each execution of the `changeBanner` function, the timeout is cleared and reset so that the image is changed again after 5 seconds.

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 a random value when `changeBanner` executes. If the random value of `bannerID` falls outside the allowed range (i.e., 1, 2, 3, or 4), the call to `getElementById` in Line 7 will return `null`. For example, if `bannerID` is -11, the second `getElementById` call will look for the ID "`banner_-11`" in the DOM; since 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.

We note that this error arises due to the loose typing and permissive error semantics of JAVASCRIPT. Further, to understand the root cause of the error, one needs to analyze 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.

B. 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 events such as user-triggered ones (e.g., click, mouseover), page load, or 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 $\rightarrow$ Line 13, while the execution sequence corresponding to the asynchronous event is Line 2 $\rightarrow$ Line 3 $\rightarrow$ Line 5 $\rightarrow$ Line 6 $\rightarrow$ Line 7 $\rightarrow$ Line 8 $\rightarrow$ Line 9.

In traditional programming languages, the goal of fault localization is to find the *faulty lines* of code. For JAVASCRIPT, its asynchronous characteristic presents an additional challenge. The programmer will not only need to find the faulty 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 error. 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 error is not limited to the JAVASCRIPT code; the JAVASCRIPT error may also result from a fault in the DOM. With regards to fault localization, the notion of a "faulty line" of code may not apply to JAVASCRIPT because it is possible that the fault is

[2] In this paper, we use JAVASCRIPT to mean client-side JAVASCRIPT.
[3] https://www.tumblr.com

in the DOM rather than the code. This is particularly true for *DOM-related* JAVASCRIPT *errors*, which are defined as JAVASCRIPT errors that lead to either exceptions or incorrect DOM element outputs as a result of a DOM access or update. As a result, for such errors, we need 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()`, `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 "faulty" 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 faulty 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.

C. Scope of the Paper

In prior work [10], we found that deployed web applications experience on average four JAVASCRIPT exceptions (manifested as error messages) during execution. Further analysis revealed that many of these errors were related to the DOM. Therefore, we choose to study DOM-related JAVASCRIPT errors in this paper. DOM-related JAVASCRIPT errors can be further divided into two classes, listed below:

1) **Code-terminating DOM-related JAVASCRIPT errors**: 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 errors**: A DOM update function sets the value of a DOM element property to an incorrect value without causing the code to halt.

In this paper, we focus on performing fault localization of code-terminating DOM-related JAVASCRIPT errors as we have observed (see Section V) that they are the more prominent of the two classes. In the next section, we describe our fault localization approach.

For code-terminating DOM-related JAVASCRIPT errors, 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*.

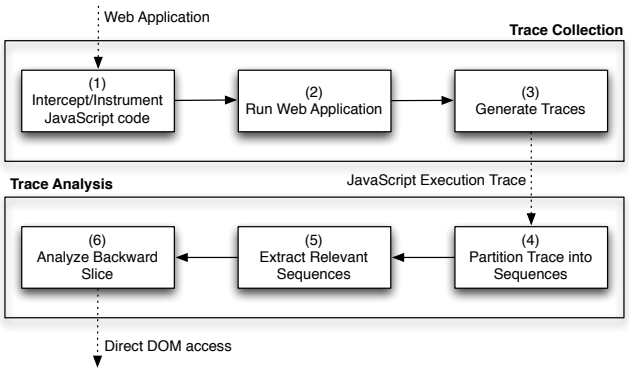

Fig. 2. Block diagram illustrating our fault localization approach.

III. APPROACH

Our 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 error 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 after we describe the usage model of the approach in the next subsection. A block diagram of the approach is shown in Figure 2.

A. Usage Model

Because we focus on fault localization, we assume that the error whose corresponding fault needs to be localized has been detected before the deployment of our technique. Further, we also assume that the user is able to replicate the error during the localization process.

Our 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 error to occur.

The output of our approach is the direct DOM access corresponding to the error 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.

B. 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 error. 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 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

```
1 Trace Record Prefix:
2   changeBanner:::4
3 Variables:
4   currentBannerID (global): 1
5   changeTimer (global): 2
6   bannerID (local): -11
7   prefix (local): none
8   currBannerElem (local): none
9   bannerToChange (local): none
```

Fig. 3. Example trace record for Line 5 of the running example from Figure 1. 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. Here, `bannerID` has randomly been assigned the value -11 because this parameter is unspecified in the `setTimeout` call. Other variables which have not been assigned values up to the current line are marked with "none".

(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.

In addition to the trace entries corresponding to the executed lines of JAVASCRIPT code, three special markers, called ERROR, ASYNCCALL and ASYNC, are added to the trace. The ERROR marker is used in the trace analysis phase to determine at which line of JAVASCRIPT code the exception was thrown. The ASYNCCALL and ASYNC markers address the asynchronous nature of JAVASCRIPT execution as described in Section II. 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.

The ERROR marker is added when an error is detected (the mechanism to detect errors is discussed in Section IV). It contains information about the exception thrown and its characteristics. In the example in Figure 1, the ERROR 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 ASYNCCALL 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 as well as 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.

C. Trace Analysis

Once the trace of executed statements has been collected, the trace analysis phase begins. The goal of this phase is to analyze the trace entries and find the direct DOM access responsible for the JAVASCRIPT error. First, we partition the trace into *sequences*, where a sequence represents the series of JAVASCRIPT statements 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 in the example in Figure 1 consists of two sequences: one corresponding to the load event, and the other corresponding to the timeout event.

After partitioning the trace into sequences, the algorithm looks for the sequence that contains the direct DOM access (box 5 in Figure 2). We call this the **relevant sequence**. The relevant sequence is initially chosen to be the sequence that contains the ERROR marker. This marker will always be the last element of the relevant sequence, since the execution of the sequence must have halted once the error occurred[4]. 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, i.e., 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 our running example, the relevant sequence is initially set to the one corresponding to the timeout event and consists of Line 2 → Line 3 → Line 5 → Line 6 → Line 7 → Line 8 → Line 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, 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 analyzes the backward slice of the variable in the ERROR marker. If the line corresponding

[4]Recall that we consider only code-terminating errors in this paper.

```
 1 Sequence 1:
 2   root:::12 (Line 12)
 3   root:::13 (Line 13)
 4   root:::ASYNC_CALL - ID = 1
 5 Sequence 2:
 6   changeBanner:::ASYNC - ID = 1
 7   changeBanner:::1 (Line 2)
 8   changeBanner:::2 (Line 3)
 9   changeBanner:::4 (Line 5)
10   changeBanner:::5 (Line 6)
11   changeBanner:::6 (Line 7)
12   changeBanner:::7 (Line 8)
13   changeBanner:::8 (Line 9) - ERROR
14 Relevant Sequence:
15   root:::12 (Line 12)
16   root:::13 (Line 13)
17   changeBanner:::1 (Line 2)
18   changeBanner:::2 (Line 3)
19   changeBanner:::4 (Line 5)
20   changeBanner:::5 (Line 6)
21   changeBanner:::6 (Line 7) **
22   changeBanner:::7 (Line 8)
23   changeBanner:::8 (Line 9) - ERROR
```

Fig. 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.

to the ERROR marker 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 ERROR 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 above 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, we assume 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 analyzer begins at Line 9 where the ERROR 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 above three forms and moves to Line 7. The algorithm then determines that Line 7 is of the first form listed above. 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.

IV. TOOL IMPLEMENTATION

We have implemented the approach described in Section III in an automated tool called AUTOFLOX[5] using the Java programming language. In addition, a number of existing tools are used to assist in the trace collection phase.

We use the CRAWLJAX [11] tool to systematically crawl the web application and trigger the execution of JAVASCRIPT code corresponding to user events. Other tools such as WaRR [12], Mugshot [13], and Selenium[6] can aid in the reproduction phase. However, we have decided to use CRAWLJAX because of the flexibility that it provides in allowing users to specify clickable elements and text box inputs, and because of its easy-to-use plugin development framework. 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.[7]

Our JAVASCRIPT code instrumentation and tracing is based on an extension of the INVARSCOPE[8] [14] plugin to CRAWLJAX. We have made the following modifications in order to facilitate the trace collection process:

1) While the original INVARSCOPE tool only collects traces at the function entry and exit points, our modified version collects traces at every line of JAVASCRIPT code to ensure that the complete execution history can be analyzed in the trace analysis phase.
2) The original INVARSCOPE does not place information on the scope of each variable in the trace; thus, we have

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

[6] http://seleniumhq.org

[7] While non-deterministic errors can be localized with AUTOFLOX, they may require multiple runs to reproduce the error

[8] http://www.crawljax.com/plugins/invarscope-plugins/

modified it to retrieve this information and include it in the trace.

3) Our 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 ASYNC_CALL and ASYNC markers in the trace).
4) Finally, we place `Try-Catch` handlers around each function call in the JAVASCRIPT code in order to catch exceptions and write ERROR 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.

A. Assumptions

AUTOFLOX makes a few simplifying assumptions, listed below. In our evaluation of the tool, we will assess the correctness of AUTOFLOX on various open-source web applications, thus evaluating the reasonableness of these assumptions.

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.
3) The null variable does not propagate through *anonymous* JAVASCRIPT function.
4) There are no object property accesses in the null propagation path. In other words, our tool 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`, `this.x`, etc.)

V. EMPIRICAL EVALUATION

A. Goals and Research Questions

We have conducted an empirical study to (1) assess the prominence of DOM-related JAVASCRIPT errors in web applications, and (2) evaluate the accuracy and real-world relevance of our fault localization approach.

The research questions we wish to answer in our evaluation are as follows:

RQ1*: How prominent are DOM-related* JAVASCRIPT *errors in web applications?*

RQ2*: What is the fault localization accuracy of* AUTOFLOX*? Are the implementation assumptions reasonable?*

RQ3*: What is the performance overhead of* AUTOFLOX*?*

B. Methodology

The subsections that follow address each of the above questions. An overview of the evaluation methodology we have used to answer each research question is shown below.

RQ1 Approach: We answer this question by conducting a study of 29 publicly available JAVASCRIPT bug reports from four web applications. We characterize these bug reports as either DOM-related or non-DOM-related, based on keyword matching and manual analysis.

RQ2 Approach: To answer this question, we run AUTOFLOX on three open-source web applications and a production website. We inject DOM-related JAVASCRIPT faults into the open-source applications and run AUTOFLOX to localize the direct DOM accesses corresponding to the faults. For the production website, we use AUTOFLOX to perform fault localization on a real JAVASCRIPT error we encountered.

RQ3 Approach: We measure the performance by calculating the overhead incurred by the instrumentation and the time it takes for the tool to find the direct DOM access.

C. Prominence of DOM-related Errors

To answer RQ1, we conducted a study of 29 openly available JAVASCRIPT bug reports from four open-source web applications: TUDU,[9] TASKFREAK,[10] WORDPRESS,[11] and Google.[12] Barring the above four applications, very few web applications publicize their bug databases.

We searched the bug databases of the applications using the keywords "javascript", "js", and "console". In total, these keywords gave us a list of 135 bug reports to consider. To ensure that the reports correspond to real bugs, we restricted ourselves to those that have subsequently been fixed — specifically, those that have been marked as *fixed* or *valid*.[13] Furthermore, to ensure that we consider only JAVASCRIPT errors, we included only the bug reports whose fix involved JAVASCRIPT. This includes bugs where the fix involved a modification in the JAVASCRIPT code, or bugs that involved a change in the server code because not applying such a fix would result in a JAVASCRIPT exception. After applying these inclusion criteria, we were left with a total of 29 JAVASCRIPT-related bug reports.

The reports were classified into five categories, as follows. The number of reports in each category are shown in Table I.

1) **Code-terminating DOM-related** (Type 1): An error caused by a DOM access function that returns a `null`, `undefined`, or incorrect value, which then propagates to one or more variables and eventually causes an exception.
2) **Output DOM-related** (Type 2): An error caused by a DOM update function that sets the value of a DOM element property to an incorrect value without throwing an exception.
3) **DOM-related error of unknown kind** (Type 3): A DOM-related JAVASCRIPT error whose fix involved a

[9] http://sourceforge.net/tracker/?group_id=131842

[10] http://forum.taskfreak.com/index.php?board=3.0

[11] http://core.trac.wordpress.org/report/1

[12] http://code.google.com/p/googlebugs/ - This is an unofficial bug tracker for Google services such as Docs and GMail.

[13] In the case of TASKFREAK, the bugs were reported via a forum, and did not include such markings; thus, fixed bugs were manually inferred by reading the discussion spawned by the bug report.

TABLE I
RESULTS OF THE STUDY ON 29 BUG REPORTS FROM FOUR OPEN-SOURCE WEB APPLICATIONS AND WEBSITE.

JAVASCRIPT Error Type	Type Number	Number of Bug Reports	% of Total	% of All Known DOM-related Errors
Code-terminating DOM-related	1	15	51.7	65.2
Output DOM-related	2	5	17.2	21.7
Unknown DOM-related	3	3	10.3	13.0
Non-DOM-related	4	5	17.2	—
Unknown Type	5	1	3.4	—
Total		29	100	

DOM function, but it was not possible to determine from the bug report whether it was type 1 or type 2.

4) **Non-DOM-related error** (Type 4): An error that does not result from a DOM interaction.
5) **Unknown JAVASCRIPT error** (Type 5): A JAVASCRIPT error where the bug report discussion does not make it apparent if the error is DOM-related or not.

As seen in Table I, DOM-related JAVASCRIPT errors (Types 1, 2, and 3) make up anywhere between 79% (if none of the Type 5 errors are DOM-related) and 83% (if all of the Type 5 errors are DOM-related) of the bug reports. This result suggests two important findings: (1) DOM-related JAVASCRIPT errors are abundant in web applications, and (2) users *want* these errors to be fixed judging by the fact that these bugs were reported by users (and fixed). In addition, Table I suggests that code-terminating DOM-related JAVASCRIPT errors (Type 1) make up anywhere between 65% to 78% of all DOM-related JAVASCRIPT errors, and are therefore more commonly reported than output DOM-related JAVASCRIPT errors (Type 2). This partially motivated our decision to focus on code-terminating JAVASCRIPT errors in this work.

D. Accuracy of AUTOFLOX

To answer RQ2, we performed a fault injection experiment on three open-source web applications: TUDU (v. 2.3), TASKFREAK (v. 0.6.4), and WORDPRESS (v. 3.2.1). The applications consist of thousands of lines of JAVASCRIPT code each. Note that we did not perform an experiment on Google as we had no write-access to its source code, preventing us from manually injecting faults. This experiment was performed on the Mac OS/X Snow Leopard (10.6.6) platform using the Firefox v. 3.6.22 web browser. The machine used was a 2.66 GHz Intel Core 2 Duo, with 4 GB of RAM. We use fault-injection to establish the ground truth for measurement of the accuracy of AUTOFLOX. However, we did not automate the fault injection process. Rather, we first searched for calls to three DOM access functions, `getElementById`, `getAttribute` and `getComputedStyle` in the JAVASCRIPT code of each of the three web applications.[14] We then manually injected the faults as follows:

- For `getElementById()` calls, the ID parameter is mutated either by switching the order of letters in the ID string or by adding an extra string at the end of the ID. This will ensure that the `getElementById()` call will return a null value, thereby leading to a null exception in a later usage.
- For `getAttribute()` calls, the attribute name parameter is mutated similarly, by switching the order of letters in the attribute name string or appending an extra string at the end of the attribute name. This will likewise ensure that `getAttribute()` will return a null value, and its later usage will lead to a null exception.
- No instances of `getComputedStyle()` were found in any of the web applications, and hence this function is not included in the results.

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 error message printed due to the exception.

Furthermore, localization was performed on injected faults rather than actual faults because no known code-terminating DOM-related errors existed in these web applications at the time of the experiment. However we attempted to emulate the code-terminating DOM-related bugs we found in the study reported in Section V-C (which had since been fixed) in our injections.

Table II shows the results of our experiments. A total of 29 mutations were performed in TASKFREAK; 24 in TUDU; and 13 in WORDPRESS. Only TUDU contained calls to `getAttribute()`. As shown in the table, AUTOFLOX was able to identify the direct DOM access for all mutations performed in TUDU and TASKFREAK, garnering *100% accuracy* in these two applications. However, AUTOFLOX was only successful in identifying the direct DOM access in 7 of the 13 WORDPRESS mutations; for the other 6 mutations for WORDPRESS, 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 these six mutations were all applied to `getElementById()` calls that were within an anonymous JAVASCRIPT function. Since AUTOFLOX assumes that the null variable does not propagate through an anonymous function (see Section IV-A), AUTOFLOX was not able to correctly identify the direct DOM access in these cases. Note, however, that this is an implementation issue, and there is no fundamental reason why support for anonymous functions cannot

[14] To our knowledge, these are the only three DOM access functions in JAVASCRIPT that return `null` if the DOM element is not found.

TABLE II
RESULTS OF THE EXPERIMENT ON OPEN-SOURCE WEB APPLICATIONS, ASSESSING THE ACCURACY OF AUTOFLOX.

JAVASCRIPT Web Applications	Lines of JAVASCRIPT code	Number of `getElementById` mutations	Number of `getAttribute` mutations	Total Number of mutations	Number of direct DOM accesses identified	Percentage identified
TASKFREAK	3044	29	0	29	29	100%
TUDU	11653	21	3	24	24	100%
WORDPRESS	8366	13	0	13	7	53.8%
Overall	23063	63	3	66	60	90.9%

be integrated into the tool. One possible way of doing this is by assigning a unique name to each anonymous function encountered in the code to distinguish them from each other. The way function parameters are declared for anonymous functions also slightly differs from the way named functions are declared; hence, support for anonymous functions will require extending the JAVASCRIPT parser to support anonymous function declarations. These extensions are directions for future work.

Overall, this experiment demonstrates that AUTOFLOX has an accuracy of 90.9% across the three open-source web applications. Note that AUTOFLOX had *n*o false positives, i.e., there were no cases where the tool incorrectly localized a fault or said that a fault had been localized when it had not.

Production website: We also used AUTOFLOX to localize a fault on a production website — Tumblr.com. One of the pages in Tumblr contained erroneous JAVASCRIPT code that threw a null exception. The structure of the code is similar to the example in Figure 1, and also involved the cycling of different images in a banner.

Note that fault injection was not performed on Tumblr because unlike the three open-source web applications described above, we did not have write access to the website's JAVASCRIPT code, which would have allowed us to modify the code. In addition, JAVASCRIPT code mutators currently do not exist to the best of our knowledge, so mutating the JAVASCRIPT code in a website to which we have no write access is a technical challenge.

We performed manual analysis of the JAVASCRIPT code in Tumblr and found that the null value also originated from a call to `getElementById()`. Thereafter, we tested AUTOFLOX on Tumblr to see if it is able to identify this direct DOM access. Indeed, AUTOFLOX was able to pinpoint the correct direct DOM access in the JAVASCRIPT code of Tumblr.

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 web application*. Later in Section VI we discuss the broader implications of the assumptions made by AUTOFLOX.

E. Performance

We report the performance overhead of AUTOFLOX in this section. We choose the Tumblr web application for these measurements because the production code is more complex than development code (such as the ones in the open-source web applications tested above), and hence incurs higher performance overheads. We measured: (1) performance overhead due to instrumentation in the trace collection phase, and (2) time taken by the trace analyzer to find the direct DOM access.

To measure (1), we crawled the Tumblr web application using CRAWLJAX both with instrumentation and without instrumentation; our baseline is the case where the web application is run only with CRAWLJAX. CRAWLJAX took 24.6 seconds to crawl the web application without instrumentation, and 33.2 seconds to crawl it with instrumentation (average of four runs). This means AUTOFLOX incurred a 35% overhead, on average, in the trace collection phase. For measuring (2), we ran AUTOFLOX on the collected trace. Averaged over four runs, AUTOFLOX took approximately 0.115 seconds to identify the direct DOM access in the trace analysis phase. Therefore, the performance overhead incurred by AUTOFLOX is relatively low, and does not affect the user experience of the web application.

VI. DISCUSSION

Here, we discuss some issues relating to the limitations of AUTOFLOX and some threats to the validity of our evaluation.

A. Limitations

Currently, AUTOFLOX requires the user to specify the elements that will be clicked during the web application run to replicate the error. 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. One way to simplify this 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.

Another limitation of AUTOFLOX is that it relies on two other tools — CRAWLJAX and INVARSCOPE— to crawl the web application and instrument the JAVASCRIPT code. Hence, when running the web application during the trace collection

phase, the appearance of the error is dependent on the clickable elements chosen by the user, and the error will only appear if the subset of crawled elements has been chosen appropriately.

Lastly, AUTOFLOX currently does not handle cases in which the web application throws multiple JAVASCRIPT errors in one execution, which is possible in JAVASCRIPT due to its permissive nature. In such cases, AUTOFLOX would only be able to localize the first error that occurred during the execution.

B. Threats to Validity

An external threat to the validity of our evaluation is that we only considered a limited number of web applications to assess the correctness of AUTOFLOX. However, we have chosen these applications as they contain many lines of JAVASCRIPT code, thereby allowing us to perform multiple fault injections per application. In addition, we have no reason to believe that the JAVASCRIPT code in these web applications have been developed in a significantly different way compared to other web applications.

With regards to the bug report study, the limited number of bug report repositories used also threatens the external validity. Unfortunately, few such repositories are publicly available for web applications, and even fewer contain client-side bug reports.

In terms of internal validity, we have chosen to use a fault injection approach to emulate the DOM-related errors in our 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 we conducted provides supportive evidence that the bugs we have injected are prominent and must therefore be considered.

Finally, while our fault injection experiment on the three open-source web applications is replicable, the experiment on Tumblr is not guaranteed to be replicable, as the Tumblr source code may change over time, and we do not have access to prior versions of the website.

VII. RELATED WORK

We classify related work into two broad categories: web application reliability and fault localization.

A. Web Application Reliability

Web applications have been an active area of research for the past decade. We focus 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 [15], [16], [17]. Because JAVASCRIPT is a difficult language to analyze 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. [18] 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, our 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 [1], [2], [3], [4]. 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. INVARSCOPE [14] and DODOM [3] dynamically derive invariants for the JAVASCRIPT code and the DOM respectively. However, none of these techniques focus on fault localization.

WaRR [12] and Mugshot [13] 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 II, this is often a challenging task.

Finally, tools such as Firefox's Firebug[15] 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.

B. 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 include Pinpoint [19], Tarantula [6] and Whither [20]. 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 [21].

Slicing-based fault localization techniques have been proposed by Agrarwal et al. [8] and Gupta et al. [22]. These techniques isolate the fault based on the dynamic backward slice of the faulty statement in the code. Our technique is similar to this body of work in that we also extract the dynamic backward slice of the JAVASCRIPT statement that throws an exception. However, our technique differs in two ways. First, we focus 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 timeouts, and does not follow a deterministic control-flow (see Section II-B for more details).

Web Fault localization. To the best of our knowledge, the only paper that has explored fault localization in the context

[15] http://getfirebug.com

of web applications is that by Artzi et al. [23]. However, their work differs from ours in various aspects: (1) they focus on the server-side code, i.e., PHP, while we focus on the client-side; (2) they localize HTML validation errors, while our approach localizes JAVASCRIPT faults; and (3) they have opted for a spectrum-based approach based on Tarantula, while ours is a dynamic slicing-based approach. To the best of our knowledge, automated fault localization for JAVASCRIPT-based web applications has not been addressed in the literature.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we introduce a fault-localization approach for JAVASCRIPT-based web applications. Our approach is based on dynamic slicing, and addresses the two main problems that inhibit JAVASCRIPT fault localization, namely asynchronous execution and DOM interactions. We focus on DOM-related JAVASCRIPT errors as we find that about 80% of the JAVASCRIPT errors reported in online bug databases of web applications pertain to the DOM. We have implemented our approach as an automated tool, called AUTOFLOX, which we evaluate using three open-source web applications and one production application. We find that AUTOFLOX can successfully localize over 90% of the faults, with no false-positives.

We plan to extend this paper in a number of ways. First, we have focused on code-terminating JAVASCRIPT errors, i.e., errors where an exception was thrown by the web application. However, not all DOM-related errors belong to this category. We will extend our technique to include localization of non-code terminating JAVASCRIPT errors. Second, the AUTOFLOX tool makes certain assumptions about the JAVASCRIPT code in the web application, which could limit its application. Although we did not encounter any issues in deploying AUTOFLOX on the four web applications in this study, it is possible that the assumptions made by AUTOFLOX do not hold for other web applications. Relaxing the assumptions is a subject of future work. Finally, we will extend the empirical evaluation 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.

ACKNOWLEDGMENT

This research was supported in part by NSERC Discovery grants (Mesbah and Pattabiraman), the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC, and a research gift from Microsoft Corporation. We thank the anonymous reviewers of ICST 2012 for insightful comments, which have served to improve the presentation.

REFERENCES

[1] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of AJAX web applications," in *Intl. Conference on Software Testing, Verification, and Validation (ICST)*. IEEE Computer Society, 2008, pp. 121–130.

[2] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of AJAX user interfaces," in *Intl. Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2009, pp. 210–220.

[3] K. Pattabiraman and B. Zorn, "DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing," in *IEEE Intl. Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 2010, pp. 191–200.

[4] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip, "A Framework for Automated Testing of JavaScript Web Applications," in *Intl. Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 571–580.

[5] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[6] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 273–282.

[7] R. Abreu, P. Zoeteweij, and A. Gemund, "Spectrum-based multiple fault localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 88–99.

[8] H. Agrawal, J. Horgan, S. London, and W. Wong, "Fault localization using execution slices and dataflow tests," in *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*. IEEE, 1995, pp. 143–151.

[9] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 342–351.

[10] F. Ocariza, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, November 2011.

[11] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web (TWEB)*, 2012.

[12] S. Andrica and G. Candea, "WaRR: High Fidelity Web Application Recording and Replaying," in *IEEE Intl. Conference on Dependable Systems and Networks*, 2011.

[13] J. Mickens, J. Elson, and J. Howell, "Mugshot: deterministic capture and replay for JavaScript applications," in *7th USENIX conference on Networked systems design and implementation*, 2010, pp. 11–11.

[14] F. Groeneveld, A. Mesbah, and A. van Deursen, "Automatic invariant detection in dynamic web applications," Delft University of Technology, Tech. Rep. TUD-SERG-2010-037, 2010.

[15] S. Guarnieri and B. Livshits, "Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code," in *Conference on USENIX security symposium*, ser. SSYM'09, 2009, pp. 151–168.

[16] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for AJAX intrusion detection," in *Intl. conference on World Wide Web*, 2009, pp. 561–570.

[17] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *Intl. Conference on the World-Wide Web (WWW)*. ACM, 2011, pp. 805–814.

[18] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the HTML DOM and browser API in static analysis of JavaScript web applications," in *Proc. European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011, pp. 59–69.

[19] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proc. International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2002, pp. 595–604.

[20] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2003, pp. 30–39.

[21] K. Dobolyi and W. Weimer, "Modeling consumer-perceived web application fault severities for testing," in *19th Intl. symposium on Software testing and analysis*, ser. ISSTA'10. ACM, 2010, pp. 97–106.

[22] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, 2005, pp. 33–42.

[23] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 265–274.