An Empirical Study of Client-Side JavaScript Bugs

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

***Abstract*—*Context*: Client-side JavaScript is widely used in web applications to improve user-interactivity and minimize client-server communications. Unfortunately, web applications are prone to JavaScript faults. While prior studies have demonstrated the prevalence of these faults, no attempts have been made to determine their root causes and consequences.**
***Objective*: The goal of our study is to understand the root causes and impact of JavaScript faults and how the results can impact JavaScript programmers, testers and tool developers.**
***Method*: We perform an empirical study of 317 bug reports from 12 bug repositories. The bug reports are thoroughly examined to classify and extract information about the fault's cause (the *error*) and consequence (the *failure* and *impact*).**
***Result*: The majority (65%) of JavaScript faults are *DOM-related*, meaning they are caused by faulty interactions of the JavaScript code with the *Document Object Model (DOM)*. Further, 80% of the highest impact JavaScript faults are DOM-related. Finally, most JavaScript faults originate from programmer mistakes committed in the JavaScript code itself, as opposed to other web application components such as the server-side or HTML code.**
***Conclusion*: Given the prevalence of DOM-related faults, JavaScript programmers need development tools that can help them reason about the DOM. Also, testers should prioritize detection of DOM-related faults as most high impact faults belong to this category. Finally, developers can use the error patterns we found to design more powerful static analysis tools for JavaScript.**

***Index Terms*—JavaScript, Document Object Model (DOM), empirical study**

I. Introduction

Web developers often rely on JavaScript to enhance the interactivity of a web application. For instance, JavaScript is used to assign event handlers to different web application components, such as buttons, links, and input boxes, effectively defining the functionality of the web application when the user interacts with its components. In addition, JavaScript can be used to send HTTP requests to the server, and update the web page's contents with the resulting response.

Client-side JavaScript contains several features that set it apart from other traditional languages. First of all, JavaScript code executes under an asynchronous model. This allows event handlers to execute on demand, as the user interacts with the web application components. Secondly, much of JavaScript is designed to interact with an external entity known as the *Document Object Model* (DOM). This entity is a dynamic tree-like structure that includes the components in the web application and how they are organized. Using DOM API calls, JavaScript can be used to access or manipulate the components stored in the DOM, thereby allowing the web page to change without requiring a page reload.

While the above features allow web applications to be highly interactive, they also introduce additional avenues for faults in the JavaScript code. In a previous study [1], we collected JavaScript console messages from fifty popular web applications to understand how prone web applications are to JavaScript faults and what kinds of JavaScript faults appear in these applications. We found that an average of four JavaScript console messages appear even in popular production web applications, and that most JavaScript console messages fall under five different categories. While the study pointed to the prevalence of JavaScript faults, it did not explore their impact or root cause, nor did it analyze the type of failures they caused. Understanding the root cause and impact of the faults is vital for developers, testers, as well as static and dynamic analysis tool builders to increase the reliability of web applications.

In this study, our goal is to discover the causes of JavaScript faults (the *error*) in web applications, and analyze their consequences (the *failure* and *impact*). Towards this goal, we conduct an empirical study of over 300 publicly available JavaScript bug reports. We choose bug reports as they typically have detailed information about a JavaScript fault and also reveal how a web application is expected to behave; this is information that would be difficult to extract from JavaScript console messages or static analysis. Further, we confine our search to bug reports that are marked "fixed", which further eliminates spurious or superfluous bug reports.

A major challenge with studying bug reports, however, is that few web applications make their bug repositories publicly available. Even those that do, often classify the reports in ad-hoc ways, which makes it challenging to extract the relevant details from the report [2].

Our work makes the following main contributions:

- We collect and systematically categorize a total of **317 bug reports**, from eight web applications and four JavaScript libraries;
- We categorize the JavaScript faults into multiple classes. We find that one class dominates the others, namely DOM-related JavaScript faults (more details below);
- We quantitatively analyze the nature (i.e., cause and consequences) and the impact of JavaScript faults; and
- We analyze the implications of the results on developers, testers, and tool developers for JavaScript code.

Our results show that around 65% of JavaScript faults are *DOM-related faults*, which occur as a result of a faulty interaction between the JavaScript code and the DOM. A

simple example is the retrieval of a DOM element using an incorrect ID, which can lead to a null exception. Further, we find that DOM-related faults account for about 80% of the highest impact faults in the web application. Finally, we find that the majority of faults arise due to the JavaScript code rather than server side code/HTML, and that there are a few recurring programming patterns that lead to these bugs.

II. Background and Motivation

This section provides background information on the structure of modern web applications, and how JavaScript is used in such applications. We also define terms used throughout this paper such as JavaScript *error*, *fault*, *failure*, and *impact*. Finally, we describe the goal and motivation of our study.

A. Web Applications

Modern web applications – commonly known as Web 2.0 applications – contain three client-side components: (1) HTML code, which defines the webpage's initial elements and its structure; (2) CSS code, which defines these elements' initial styles; and (3) JavaScript[1] code, which defines client-side functionality in the web application. These client-side components can either be written manually by the programmer, or generated automatically by the server-side (e.g., PHP) code.

The *Document Object Model* (DOM) is a dynamic tree data structure that defines the elements in the web application, their properties including their styling information, and how the elements are structured. Initially, the DOM contains the elements defined in the HTML code, and these elements are assigned the styling information defined in the CSS code. However, JavaScript can be used to manipulate this initial state of the DOM through the use of DOM API calls. For example, an element in the DOM can be accessed through its ID by calling the `getElementById()` method. The attributes of this retrieved DOM element can then be modified using the `setAttribute()` method. In addition, elements can be added to or removed from the DOM by the JavaScript code.

In general, a JavaScript method or property that retrieves elements or attributes from the DOM is called a ***DOM access method/property***. Examples of these methods/properties include `getElementById()`, `getElementsByTagName()`, and `parentNode`. Similarly, a JavaScript method or property that is used to update values in the DOM (e.g., its structure, its elements' properties, etc.) is called a ***DOM update method-/property***. Examples include `setAttribute()`, `innerHTML`, and `replaceChild()`. Together, the access and update methods/properties constitute the DOM API.

B. JavaScript Bugs

JavaScript is particularly prone to faults, as it is a weakly typed language, which makes the language flexible but also opens the possibility for untyped variables to be (mis)used in important operations. In addition, JavaScript code can be dynamically created during execution (e.g., by using `eval`), which can lead to faults that are only detected at runtime. Further, JavaScript code interacts extensively with the DOM, which makes it challenging to test/debug, and this leads to many faults as we find in our study.

[1]JavaScript is a scripting language based on the ECMAScript standard, and it is used in other applications such as desktop widgets and even web servers.

Error: The programmer forgets to initialize the value of the `cmi.evaluation.comments` variable.

Fault: The `cmi.evaluation.comments` variable – which is uninitialized and hence has the value `null` – is used to access a property `X` (i.e., `cmi.evaluation.comments.X`) during JavaScript execution.

Failure: Since `cmi.evaluation.comments` is `null`, the code attempting to access a property through it leads to a null exception, which terminates JavaScript execution.

Fig. 1. Example that describes the error, fault, and failure of a JavaScript bug reported in Moodle.

JavaScript Bug Sequence. The following sequence describes the progression of a JavaScript *bug*, and the terms we use to describe this sequence:

1) The programmer makes a mistake at some point in the code being written or generated. These ***errors*** can range from simple mistakes such as typographical errors or syntax errors, to more complicated mistakes such as errors in logic or semantics. The error can be committed in the JavaScript code, or in other locations such as the HTML code or server-side code (e.g., PHP).
2) The error can propagate, for instance, into a JavaScript variable, the parameter or assignment value of a JavaScript method or property, or the return value of a JavaScript method during JavaScript code execution. Hence, by this point, the error has propagated into a ***fault***.
3) The fault either directly causes a JavaScript exception (***code-terminating failure***) or a corruption in the output (***output-related failure***). This is called the ***failure***.

Figure 1 shows a real-world example of the error, fault, and failure associated with a JavaScript bug report from the Moodle web application. Note that for output-related failures, the pertinent output can be one or a combination of many things, including the DOM, server data, or important JavaScript variables. We will be using the above error-fault-failure model to classify JavaScript bugs, as described in Section III.

If a JavaScript error propagates into the parameter of a DOM access/update method or to the assignment value for a DOM access/update property – thereby causing an incorrect retrieval or an incorrect update of a DOM element – then the error is said to have propagated into a ***DOM-related fault***.

For example, if an error eventually causes the parameter of the DOM access method `getElementById()` to represent a nonexistent ID, and this method is called during execution with the erroneous parameter, then the error has propagated into a DOM-related fault. However, if the error does not propagate into a DOM access/update method/property, the error is considered to have propagated into a ***non-DOM-related fault***.

Severity. While the appearance of a failure is clear-cut and mostly objective (i.e., either an exception is thrown or not; either an output contains a correct value or not), the severity of the failure is subjective, and depends on the context in which the web application is being used. For example, an exception may be tolerable and non-severe if it happens in a "news ticker" web application widget; but if the news ticker is used for something important – say, stocks data – the same exception may now be classified as severe. In this paper, we will refer to the severity as the ***impact*** of the failure.

C. Goal and Motivation

Our overall goal in this work is *to understand the sources and the impact of JavaScript faults in web applications*. To this end, we conduct an empirical study of JavaScript bug reports in deployed web applications. There are several factors that motivated us to pursue this goal. First, understanding the root cause of JavaScript faults could help make developers aware of programming pitfalls to be avoided, and the results could pave the way for better JavaScript debugging techniques. Second, analyzing the impact could steer developers' and testers' attention towards the highest impact faults, thereby allowing these faults to be detected early. Finally, we have reason to believe that JavaScript faults' root causes and impacts differ from those of traditional languages because of JavaScript's permissive nature and its many distinctive features (e.g., event-driven model; interaction with the DOM; dynamic code creation; etc.)

Other work has studied JavaScript faults through console messages or through static analysis [3], [4], [5], [6]. However, bug reports contain detailed information about the root cause of the faults and the intended behaviour of the application, which is missing in these techniques. Further, they typically contain the fix associated with the fault, which is useful in further understanding it.

III. Experimental Methodology

We describe our methodology for the empirical study on JavaScript faults. First, we enumerate the research questions that we want to answer. Then we describe the web applications we study and how we collected their bug reports. All our collected empirical data is available for download.[2]

A. Research Questions

To achieve our goal, we address the following research questions through our bug report study:

[2]http://ece.ubc.ca/~frolino/projects/js-bugs-study/

RQ1: What types of faults exist among reported JavaScript faults, and how prevalent are these fault types?

RQ2: What is the nature of failures stemming from JavaScript faults? What is the impact of the failures on the web applications?

RQ3: What is the root-cause of JavaScript faults? Are there specific programming practices that lead to JavaScript faults?

RQ4: Do JavaScript faults exhibit browser-specific behaviour?

RQ5: How long does it take programmers to triage a JavaScript fault reported in a bug report to a developer? How long does it take programmers to fix these JavaScript faults?

B. Experimental Objects

To ensure representativeness, we collect and categorize bug reports from a wide variety of web applications and libraries. Each object is classified as either a web application or a JavaScript library. We made this distinction to see if there are any differences between JavaScript faults in web applications and those in libraries; we did not, however, find any substantial differences after performing our analysis. In total, we collected and analyzed 317 bug reports from 8 web applications and 4 libraries.

Table I lists the range of the software versions considered for each experimental object. The web applications and libraries were chosen based on several factors, including their popularity, their prominent use of client-side JavaScript, and the descriptiveness of their bug reports (i.e., the more information its bug reports convey, the better). Another contributing factor is the availability of a bug repository for the web application or library, as such repositories ware not always made public. In fact, finding web applications and libraries that satisfied these criteria was a major challenge in this study.

C. Collecting the Bug Reports

For each web application bug repository, we collect a total of **min**{30, *NumJSReports*} JavaScript bug reports, where *NumJSReports* is the total number of JavaScript bug reports in the repository. We chose 30 as the maximum threshold for each repository to balance analysis time with representativeness. To collect the bug reports for each repository, we perform the following steps:

Step 1 Use the filter/search tool available in the bug repository to narrow down the list of bug reports. The filters and search keywords used in each bug repository are listed in Table I. In general, where appropriate, we used "javascript" and "js" as keywords to narrow down the list of bug reports (in some bug repositories, the keyword "jQuery" was also used to narrow down the list even further). Further, to reduce spurious or superfluous reports, we only considered bug reports with resolution "fixed", and type "bug" or "defect" (i.e., bug reports marked as "enhancements" were neglected). Table I also lists the number of search results after applying the filters in each bug repository. The bug report repositories were examined between January 30, 2013 and March 13, 2013.

TABLE I
EXPERIMENTAL OBJECTS FROM WHICH BUG REPORTS WERE COLLECTED.

Application	Version Range	Type	Description	Size of JS Code (KB)	Bug Report Search Filter	# of Reports Collected
Moodle	1.9-2.3.3	Web Application	Learning Management	352	(**Text** contains *javascript* OR *js* OR *jquery*) AND (**Issue type** is *bug*) AND (**Status** is *closed*) - Number of Results: 1209	30
Joomla	3.x	Web Application	Content Management	434	(**Category** is *JavaScript*) AND (**Status** is *Fixed*) - Number of Results: 62	11
WordPress	2.0.6-3.6	Web Application	Blogging	197	((**Description** contains *javascript* OR *js*) OR (**Keywords** contains *javascript* OR *js*)) AND (**status** is *closed*) - Number of Results: 875	30
Drupal	6.x-7.x	Web Application	Content Management	213	(**Text** contains *javascript* OR *js* OR *jQuery*) AND (**Category** is *bug report*) AND (**Status** is *closed(fixed)*) - Number of Results: 608	30
Roundcube	0.1-0.9	Web Application	Webmail	729	((**Description** contains *javascript* OR *js*) OR (**Keywords** contains *javascript* OR *js*)) AND (**status** is *closed*) - Number of Results: 234	30
WikiMedia	1.16-1.20	Web Application	Wiki Software	160	(**Summary** contains *javascript*) AND (**Status** is *resolved*) AND (**Resolution** is *fixed*) - Number of Results: 49	30
TYPO3	1.0-6.0	Web Application	Content Management	2252	(**Status** is *resolved*) AND (**Tracker** is *bug*) AND (**Subject** contains *javascript*) (Only one keyword allowed) - Number of Results: 81	30
TaskFreak	0.6.x	Web Application	Task Organizer	74	(**Search keywords** contain *javascript* OR *js*) AND (**User** is *any user*) - Number of Results: 57	6
jQuery	1.0-1.9	Library	—	94	(**Type** is *bug*) AND (**Resolution** is *fixed*) - Number of Results: 2421	30
Prototype.js	1.6.0-1.7.0	Library	—	164	(**State** is *resolved*) - Number of Results: 142	30
MooTools	1.1-1.4	Library	—	101	(**Label** is *bug*) AND (**State** is *closed*) - Number of Results: 52	30
Ember.js	1.0-1.1	Library	—	745	(**Label** is *bug*) AND (**State** is *closed*) - Number of Results: 347	30

Step 2 Once we have the narrowed-down list of bug reports from Step 1, we manually examine each report in the order in which it was retrieved. Since the filter/search features of some bug tracking systems were not as descriptive (e.g., the TYPO3 bug repository only allowed the user to search for bug reports marked "resolved", but not "fixed"), we also had to manually check whether the bug report satisfied the conditions described in Step 1. If the conditions are satisfied, the bug report is analyzed. Otherwise, the bug report is discarded. A bug report is also discarded if its fault is found to *not* be JavaScript-related – that is, the error does not propagate into any JavaScript code in the web application. This step is repeated until **min**{30, *NumJSReports*} reports have been collected in the repository. The number of bug reports we ended up collecting for each bug repository is shown in Table I. Note that Joomla and TaskFreak had only 11 and 6 reports, respectively, which satisfied the above criteria. For all remaining applications, we collected 30 bug reports each.

Step 3 For each report, we created an XML file that describes and classifies, the error, fault, failure, and impact of the JavaScript bug reported. The XML file also describes the fix applied for the bug. Typically this data is presented in raw form in the original bug report, based on the bug descriptions, developer discussions, patches, and supplementary data; hence, we needed to thoroughly read through, understand, and interpret each bug report in order to extract all the information included in the corresponding XML file. We also include data regarding the date and time of each bug being assigned and fixed in the XML file. We have made these bug report XML files publicly available for reproducibility.[2]

D. Analyzing the Collected Bug Reports

The collected bug report data, captured in the XML files, enable us to qualitatively and quantitatively analyze the nature of JavaScript bugs.

Fault Categories. To address RQ1, we classify the bug reports according to the following fault categories that were identified through an initial pilot study:

- **Undefined/Null Variable Usage**: A JavaScript variable that has a `null` or `undefined` value – either because the variable has not been defined or has not been assigned a value – is used to access an object property or method. *Example*: The variable `x`, which has not been defined in the JavaScript code, is used to access the property `bar` via `x.bar`.
- **Undefined Method**: A call is made in the JavaScript code to a method that has not been defined. *Example*: The undefined function `foo()` is called in the JavaScript code.
- **Incorrect Method Parameter**: An unexpected or invalid value is passed to a native JavaScript method, or assigned to a native JavaScript property. *Example*: A string value is passed to the JavaScript Date object's `setDate()` method, which expects an integer. Another example is passing an ID string to the DOM method `getElementById()` that does not correspond to any IDs in the DOM. Note that this latter example is a type of *DOM-related fault*, which is a subcategory of Incorrect

TABLE II
IMPACT TYPES.

Type	Description	Examples
1	Cosmetic	Table is not centred; header is too small
2	Minor functionality loss	Cannot create e-mail addresses containing apostrophe characters, which are often only used by spammers
3	Some functionality loss	Cannot use delete button to delete e-mails, but delete key works fine
4	Major functionality loss	Cannot delete e-mails at all; cannot create new posts
5	Data loss, crash, or security issue	Browser crashes/hangs; entire application unusable; save button does not work and prevents user from saving considerable amount of data; information leakage

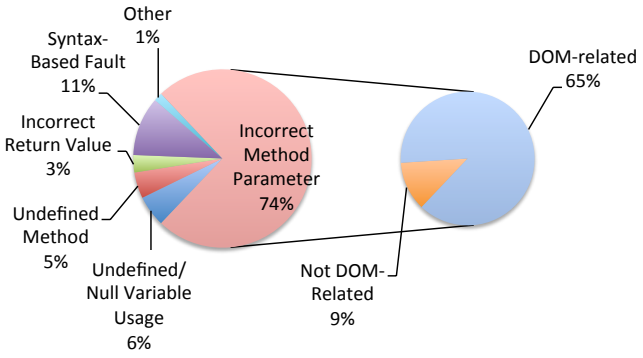

Fig. 2. Pie chart of the distribution of fault categories.

Method Parameter faults where the method/property is a DOM API method/property (as defined in Section II-B).

- **Incorrect Return Value**: A user-defined method is returning an incorrect return value even though the parameter(s) is/are valid. *Example*: The user-defined method `factorial(3)` returns 2 instead of 6.
- **Syntax-Based Fault**: There is a syntax error in the JavaScript code. *Example*: There is an unescaped apostrophe character in a string literal that is defined using single quotes.
- **Other**: Errors that do not fall into the above categories. *Example*: There is a naming conflict between methods or variables in the JavaScript code.

Failure Categories. The failure category refers to the observable consequence of the fault. For each bug report, we marked the failure category as either *Code-terminating* or *Output-related*, as defined in Section II-B. This categorization helps us answer RQ2.

Impact Types. To classify the impact of a fault, we use the classification scheme used by Bugzilla.[3] This scheme is applicable to any software application, and has been also used in other studies [7], [8]. Table II shows the categories. This categorization helps us answer RQ2.

Error Locations. The error location refers to the code unit or file where the error was made (either by the programmer or the server-side program generating the JavaScript code). For each bug report, we marked the error location as one of the following: (1) *JavaScript code (JS)*; (2) *HTML Code (HTML)*; (3) *Server-side code (SSC)*; (4) *Server configuration file (SCF)*; (5) *Other (OTH)*; and (6) *Multiple error locations (MEL)*. In cases where the error location is marked as either OTH or MEL, the location(s) is/are specified in the error description. This categorization helps us answer RQ3.

Browser Specificity. In addition, we also noted whether a certain bug report is browser-specific – that is, the fault described in the report only occurs in one or two browsers, but not in others – to help us answer RQ4.

Time for Fixing. To answer RQ5, we define the ***triage time*** as the time it took a bug to get assigned to a developer, from the time it was reported (or, if there is no "assigned" marking, the time until the first comment is posted in the report). We also define ***fix time*** as the time it took the corresponding JavaScript fault to get marked as "fixed", from the time it was triaged. We recorded the time taken for each JavaScript bug report to be triaged, and for the report to be fixed. Other studies have classified bugs on a similar basis [9], [10]. Further, we calculate times based on the calendar date; hence, if a bug report was triaged on the same date as it was reported, the triage time is recorded as 0.

[3]http://www.bugzilla.org/

IV. RESULTS

In this section, we present the results of our empirical study on JavaScript bug reports. The subsections are organized according to the research questions in Section III-A.

A. Fault Categories

Table III shows the breakdown of the fault categories in our experimental objects. The pie chart in Figure 2 shows the overall percentages. As seen from the table and the figure, approximately 74% of JavaScript faults belong to the "Incorrect Method Parameter" category. This suggests that most JavaScript faults result from errors related to setting up the parameters of native JavaScript methods, or the values assigned to native JavaScript properties.

> ***Finding #1***: *"Incorrect Method Parameter" faults account for around 74% of JavaScript faults.*

In our earlier studies of JavaScript console messages [1] and fault-localization of JavaScript bugs [11], we also noticed many "Incorrect Method Parameter" faults, but their prevalence was not quantified. Interestingly, we also observed in these earlier studies that many of the methods and properties affected by these faults are DOM methods/properties – in other words, DOM-related faults, as defined in Section II. Based on these prior observations, we became curious as to how many of these "Incorrect Method Parameter" faults are DOM-related.

TABLE III
FAULT TYPES OF THE BUG REPORTS ANALYZED. LIBRARY DATA ARE SHOWN IN ITALICS.

Application	Undefined/Null Variable Usage	Undefined Method	Incorrect Return Value	Syntax-Based Fault	Other	Incorrect Method Parameter			Percent DOM-related
						DOM-related	Not DOM-related	Total	
Moodle	3	3	0	7	0	15	2	17	50%
Joomla	1	0	0	3	0	6	1	7	55%
WordPress	1	2	0	3	1	21	2	23	70%
Drupal	0	1	0	5	0	23	1	24	77%
Roundcube	3	0	0	4	0	22	1	23	73%
WikiMedia	2	4	0	5	0	15	4	19	50%
TYPO3	2	2	0	7	1	18	0	18	60%
TaskFreak	1	0	0	0	0	4	1	5	67%
jQuery	*0*	*0*	*1*	*0*	*0*	*26*	*3*	*29*	*87%*
Prototype.js	*0*	*1*	*2*	*0*	*0*	*22*	*5*	*27*	*73%*
MooTools	*3*	*1*	*3*	*0*	*1*	*19*	*3*	*22*	*63%*
Ember.js	*2*	*1*	*4*	*0*	*2*	*16*	*5*	*21*	*53%*
Overall	18	15	10	34	5	207	28	235	65%

We further classified the "Incorrect Method Parameter" faults based on the methods/properties in which the incorrect values propagated, and found that 88% of these faults are DOM-related faults. This indicates that among *all* JavaScript faults, approximately 65% are DOM-related faults (see rightmost pie chart in Figure 2). We find that DOM-related faults range from 50 to 87% of the total JavaScript faults across applications, as seen on the last column of Table III.

> ***Finding #2****: DOM-related faults account for 88% of "Incorrect Method Parameter" faults. Hence, the majority – around 65% – of JavaScript faults are DOM-related.*

B. Consequences of JavaScript Faults

We now show the failure categories of the bug reports we collected, as well as the impact of the JavaScript faults that correspond to the reports.

Failure Categories. Table IV shows the distribution of failure categories amongst the collected reports; all faults are classified as either leading to a code-terminating failure or an output-related failure (these terms are defined in Section III-D). As the table shows, around 56% of JavaScript faults are code-terminating, which means that in these cases, an exception is thrown. Faults that lead to code-termination are generally easy to detect, since the exceptions have one or more corresponding JavaScript error message(s) (provided the error can be reproduced during testing). On the other hand, output-related failures do not have such messages; they are typically only detected once the user observes an abnormality in the behaviour or appearance of the application.

Since the majority of JavaScript faults are DOM-related, we explored how these failure categories apply to these DOM-related faults. Interestingly, we found that for DOM-related faults, most failures are output-related (at 61%), while for non-DOM-related faults, most failures are code-terminating (at 88%). This result suggests that DOM-related faults may be more difficult to detect than non-DOM-related faults, as most of them do not have error messages.

TABLE IV
NUMBER OF CODE-TERMINATING FAILURES COMPARED TO OUTPUT-RELATED FAILURES. LIBRARY DATA ARE SHOWN IN ITALICS.

Application	Code-terminating	Output-related
Moodle	21	9
Joomla	8	3
WordPress	11	19
Drupal	12	18
Roundcube	18	12
WikiMedia	19	11
TYPO3	21	9
TaskFreak	3	3
jQuery	*17*	*13*
Prototype.js	*10*	*20*
MooTools	*21*	*9*
Ember.js	*16*	*14*
Overall	177	140

> ***Finding #3****: While most non-DOM-related JavaScript faults lead to exceptions (around 88%), only a small percentage (39%) of DOM-related faults lead to such exceptions.*

Impact Types. The impact indicates the severity of the failure; Hence, we also classify bug reports based on impact types as defined in Section III-D (i.e., Type 1 has lowest severity, and Type 5 has highest severity).

The impact type distribution for each web application and library is shown in Table V. Most of the bug reports were classified as having Type 3 impact (i.e., some functionality loss). Type 1 and Type 5 impact faults are the fewest, with around 30 bug reports each. Finally, Type 2 and Type 4 impact faults are represented by 92 and 43 bug reports, respectively. The average impact of the collected JavaScript bug reports is close to the middle, at 2.83.

Table V also shows the impact distribution for DOM-related faults in parentheses. As seen in the table, each impact type is comprised primarily of DOM-related faults. Further, almost 80% (23 out of 29) of the highest severity faults (i.e., Type 5 faults) are DOM-related. Additionally, all but two of the experimental objects contain at least one DOM-related fault

TABLE V
IMPACT TYPES OF THE BUG REPORTS ANALYZED. LIBRARY DATA ARE SHOWN IN ITALICS. IMPACT TYPES DATA FOR DOM-RELATED FAULTS ONLY ARE SHOWN IN PARENTHESES.

Application	Type 1	Type 2	Type 3	Type 4	Type 5
Moodle	10 (5)	12 (5)	0 (0)	6 (3)	2 (2)
Joomla	2 (2)	2 (0)	4 (2)	2 (1)	1 (1)
WordPress	4 (4)	7 (3)	12 (9)	3 (2)	4 (3)
Drupal	3 (3)	2 (1)	17 (12)	1 (1)	7 (6)
Roundcube	2 (2)	5 (4)	14 (9)	5 (3)	4 (4)
WikiMedia	2 (1)	8 (6)	15 (6)	1 (0)	4 (2)
TYPO3	0 (0)	4 (2)	20 (13)	5 (2)	1 (1)
TaskFreak	2 (1)	1 (1)	1 (0)	2 (2)	0 (0)
jQuery	*3 (3)*	*13 (13)*	*1 (1)*	*11 (8)*	*2 (1)*
Prototype.js	*0 (0)*	*7 (6)*	*19 (12)*	*2 (2)*	*2 (2)*
MooTools	*0 (0)*	*16 (8)*	*10 (8)*	*3 (3)*	*1 (0)*
Ember.js	*2 (0)*	*15 (10)*	*10 (4)*	*2 (1)*	*1 (1)*
Overall	30 (21)	92 (59)	123 (76)	43 (28)	29 (23)

TABLE VI
ERROR LOCATIONS OF THE BUG REPORTS ANALYZED. LIBRARY DATA ARE SHOWN IN ITALICS.

Legend: JS = JavaScript code, HTML = HTML code, SSC = Server-side code, SCF = Server configuration file, OTH = Other, MEL = Multiple error locations

Application	JS	HTML	SSC	SCF	OTH	MEL
Moodle	22	2	6	0	0	0
Joomla	9	0	1	0	0	1
WordPress	24	0	6	0	0	0
Drupal	29	0	1	0	0	0
Roundcube	26	0	4	0	0	0
WikiMedia	25	0	5	0	0	0
TYPO3	18	1	9	2	0	0
TaskFreak	6	0	0	0	0	0
jQuery	*30*	–	–	–	*0*	*0*
Prototype.js	*25*	–	–	–	*4*	*1*
MooTools	*30*	–	–	–	*0*	*0*
Ember.js	*30*	–	–	–	*0*	*0*
Overall	274	3	32	2	4	2

with Type 5 impact. This result suggests that *high severity failures often result from DOM-related faults*. We find that these high-impact faults broadly fall into three categories.

1) **Application/library becomes unusable**. This occurs because an erroneous feature is preventing the user from using the rest of the application, particularly in DOM-related faults. For example, one of the faults in Drupal prevented users from logging in (due to incorrect attribute values assigned to the username and password elements), so the application could not even be accessed.
2) **Data loss**. Once again, this is particularly true for DOM-related faults, which account for 9 out of the 10 data-loss-causing faults that we encountered. One example comes from Roundcube; in one of the bug reports, the fault causes an empty e-mail to be sent, which causes the e-mail written by the user to be lost. As another example, a fault in WordPress causes server data (containing posts) to be deleted automatically without confirmation.
3) **Browser hangs and information leakage**. Hangs often occur as a result of a bug in the browser; the type 5 faults leading to browser hangs that we encountered are all browser-specific. Information leakage only occurred once, as a result of a JavaScript fault in TYPO3 that caused potentially security-sensitive code from the server to be displayed on the page.

> ***Finding #4****: About 80% of the highest severity JavaScript faults are DOM-related.*

C. Causes of JavaScript Faults

Locations. Before we can determine the causes, we first need to know *where* the programmers committed the programming errors. To this end, we marked the error locations of each bug report; the error location categories are listed in Section III-D. The results are shown in Table VI. As the results show, the vast majority (86%) of the JavaScript faults occur as a result of programming errors in the JavaScript code itself. If only DOM-related faults were considered, a similar distribution of fault locations was observed; in fact, the majority is even larger for DOM-related faults that originated from the JavaScript code, at 92%. For these bug reports where the error location is in the JavaScript code itself, the fix involved the *manual* modification of the corresponding JavaScript file(s). This observation suggests that JavaScript faults typically occur because the programmer herself writes erroneous code, as opposed to server-side code automatically generating erroneous JavaScript code, or HTML.

> ***Finding #5****: Most JavaScript faults (86%) originate from manually-written JavaScript code as opposed to code automatically generated by the server.*

Patterns. To understand the programmer mistakes associated with JavaScript errors, we manually examined the bug reports for errors committed in JavaScript code (which were the dominant category). We found that errors fell into the following common patterns:

1) **Erroneous input validation**. Around 18% of the bugs occurred because inputs passed to the JavaScript code (i.e., user input from the DOM or inputs to JavaScript functions) are not being validated or sanitized. The most common mistake made by programmers in this case is neglecting valid input cases. For example, in the jQuery library, the `replaceWith()` method is allowed to take an empty string as input; however, the implementation of this method does not take this possibility into account, thereby causing the call to be ignored.
2) **Error in writing a string literal**. Approximately 14% of the bugs were caused by a mistake in writing a string literal in the JavaScript code. These include forgetting prefixes and/or suffixes, typographical errors, and including wrong character encodings. Half of these errors relate to writing a syntactically valid but incorrect CSS selector (which is used to retrieve DOM elements) or regular expression.
3) **Neglecting differences in browser behaviour**. Around

TABLE VII
BROWSER SPECIFICITY OF THE BUG REPORTS ANALYZED. LIBRARY DATA ARE SHOWN IN ITALICS.

Legend: IE = Internet Explorer, FF = Firefox, CHR = Chrome, SAF = Safari, OPE = Opera, OTH = Other, NBS = Not browser-specific, MUL = Multiple

Application	IE	FF	CHR	SAF	OPE	OTH	NBS	MUL
Moodle	4	0	0	0	0	0	25	1
Joomla	1	0	0	0	0	0	10	0
WordPress	1	0	0	0	0	1	28	0
Drupal	2	0	1	1	0	0	26	0
Roundcube	5	0	1	0	1	1	22	0
WikiMedia	6	0	0	0	0	0	24	0
TYPO3	7	1	0	0	1	0	20	1
TaskFreak	1	0	0	1	0	0	4	0
jQuery	*7*	*0*	*0*	*0*	*0*	*0*	*22*	*1*
Prototype.js	*8*	*1*	*1*	*2*	*1*	*0*	*14*	*3*
MooTools	*10*	*2*	*0*	*0*	*1*	*0*	*17*	*0*
Ember.js	*2*	*0*	*0*	*0*	*0*	*0*	*28*	*0*
Overall	54	4	3	4	4	2	240	6

TABLE VIII
AVERAGE TRIAGE TIMES (T) AND FIX TIMES (F) FOR EACH EXPERIMENTAL OBJECT. LIBRARY DATA ARE SHOWN IN ITALICS.

Application	All Faults		DOM-Related Faults Only		Non-DOM-Related Faults Only	
	T	F	T	F	T	F
Moodle	248	9.5	204.5	11.5	291.5	7.5
Joomla	4.1	57.1	0.7	66.7	8.2	45.6
WordPress	1.1	137.5	1.5	150.3	0.3	107.7
Drupal	7.0	66.2	2.6	46.7	21.7	130.1
Roundcube	18.4	118.3	25.1	160.0	0	9.3
WikiMedia	18.4	25.9	35.5	44.3	1.3	7.5
TYPO3	6.9	54.7	7.2	67.2	6.3	35.8
TaskFreak	22.8	16.5	31.5	22.5	5.5	4.5
jQuery	*1.2*	*32.5*	*1.3*	*36.0*	*0.5*	*10.3*
Prototype.js	*28.0*	*343.4*	*33.2*	*294.3*	*13.6*	*478.4*
MooTools	*9.5*	*48.1*	*9.9*	*48.5*	*8.8*	*47.4*
Ember.js	*0.4*	*11.4*	*0.7*	*14.3*	*0.1*	*8.1*
Overall	32.7	82.5	26.4	90.8	44.4	66.8

9% of the bugs were caused by differences in how browsers treat certain methods, properties or operators in JavaScript. Of these, around 60% pertain to differences in how browsers implement native JavaScript methods. For example, a fault occurred in WikiMedia in Internet Explorer 7 and 8 because of the different way those browsers expect the `history.go()` method to be used.

4) **Forgetting null/undefined check**. Around 9% of the bugs resulted from missing null/undefined checks for a particular variable, assuming that the variable is *allowed* to have a value of `null` or `undefined`.
5) **Error in syntax**. Interestingly, around 7% of bugs resulted from syntax errors in the JavaScript code that were made by the programmer. Note, also, that we found instances where server-side code generated syntactically incorrect JavaScript code, though this is not accounted for here.

> ***Finding #6****: There are several recurring error patterns – causing JavaScript faults – that arise from JavaScript code.*

D. Browser Specificity

We analyzed the browser specificity of the bug reports we collected. A bug is browser specific if it occurs only in a certain browser. As Table VII shows, most JavaScript faults (74%) are non-browser specific. However, among the browser-specific faults, about 69% are specific to Internet Explorer (IE).

After analyzing the IE-specific faults, we found that most of them (44%) were due to the use of methods and properties that were not supported in that browser (particularly in earlier versions, pre-Internet Explorer 8). This is likely because the use of browser-specific method and property names (which may not be standards-compliant) is more prevalent in IE than in other browsers. In addition, IE has low tolerance of small errors in the JavaScript code. For example, 24% of the IE-specific faults occurred because IE could not handle trailing commas in object-creation code; while these trailing commas are technically syntax errors, other browsers can detect their presence and remove them.

> ***Finding #7****: Most JavaScript faults (74%) are not browser-specific.*

E. Triage and Fix Time for JavaScript Faults

We calculated the triage time and fix time for each bug report and found that on average, the triage time for JavaScript faults is 32.7 days, while the average fix time is 82.5 days (see Table VIII).

As before, we made the same calculations for DOM-related faults and non-DOM-related faults. We found that DOM-related faults have an average triage time of 26.4 days, compared to 44.4 days for non-DOM-related faults. On the other hand, DOM-related faults have an average fix time of 90.8 days, compared to 66.8 days for non-DOM-related faults. This suggests that developers find DOM-related faults important enough to be triaged more promptly than non-DOM-related faults. However, DOM-related faults take longer to fix, perhaps because of their inherent complexity.

> ***Finding #8****: On average, DOM-related faults get triaged more promptly than non-DOM-related faults (26.4 days vs. 44.4 days); however, DOM-related faults take longer to fix than non-DOM-related faults (90.8 days vs. 66.8 days)*

F. Threats to Validity

An internal validity threat is that the classifications were made by multiple individuals (i.e., two of the co-authors), which may introduce inconsistencies and bias, particularly in the classification of the impacts. In order to mitigate any possibilities of bias, we conducted a review process in which each person reviews the classifications assigned by the other person. Any disagreements were discussed until a consensus on the classification was reached.

In terms of external threats, our results are based on bug reports from a limited number of experimental objects, which calls into question the representativeness; unfortunately, public bug repositories for web applications are not abundant, as previously mentioned. We mitigated this by choosing web applications that are used for different purposes, including content management, webmail, and wiki.

A construct validity threat is that the bug reports may not be fully representative of the JavaScript faults that occur in web applications. This is because certain types of faults – such as non-deterministic faults and faults with low visual impact – may go unreported. In addition, we focus exclusively on bug reports that were fixed. This decision was made since the root cause would be difficult to determine from open reports, which have no corresponding fix. Further, open reports may not be representative of real bugs, as they are not deemed important enough to fix.

For the triage and fix times, we did not account for possible delays in marking a bug report as "assigned" or "fixed", which may skew the results. In addition, the triage time is computed as the time until the first developer comment, when there is no "assigned" marking; although we find this approximation reasonable, the developer may not have started fixing until some days after the first comment was posted. These are likewise construct validity threats.

V. Discussion

In this section, we discuss the implications of our findings on web application developers, testers, developers of web analysis tools, and designers of web application development frameworks.

Findings 1 and 2 reveal the difficulties that web application developers have in setting up values passed or assigned to native JavaScript methods and properties – particularly DOM methods and properties. Many of these difficulties arise because the asynchronous, event-driven JavaScript code must deal with the highly dynamic nature of the DOM. This requirement forces the programmer to have to think about how the DOM is structured and what properties its elements possess at certain DOM interaction points in the JavaScript code; doing so can be difficult because (1) the DOM frequently changes at runtime and can have many states, and (2) there are many different ways a user can interact with the web application, which means there are many different orders in which JavaScript event handlers can execute. This suggests the need to equip these programmers with appropriate tools that would help them reason about the DOM, thereby simplifying these DOM-JavaScript interactions.

With regards to Findings 3, 4, and 8, these results suggest that web application testers should prioritize emulating DOM-related faults, as most high-impact faults belong to this category. One possible way to do this is to prioritize the creation of tests that cover DOM interaction points in the JavaScript code. By doing so, testers can immediately find most of the high-impact faults. This early detection is useful because, as Finding 3 suggests, DOM-related faults often have no accompanying error messages and can be more difficult to detect. Further, as Finding 8 suggests, DOM-related faults take longer to fix on average compared to non-DOM-related faults.

As for Findings 5 and 6, these results can be useful for developers of static analysis tools for JavaScript. Many of the current static analysis tools only address syntactic issues with the JavaScript code (e.g., JSLint,[4] Closure Compiler,[5] JSure[6]), which is useful since a few JavaScript faults occur as a result of syntax errors, as described in Section IV-C. However, the majority of JavaScript faults occur because of errors in semantics or logic. Some developers have already started looking into building static semantics checkers for JavaScript, including TAJS [12], which is a JavaScript type analyzer. However, the programming mistakes we encountered in the bug reports (e.g., erroneous input validations, erroneous CSS selectors, etc.) call for more powerful tools to improve JavaScript reliability.

Finally, while Finding 7 suggests that most JavaScript faults are non-browser specific, we did find a few (mostly IE-specific) faults that are browser-specific. Hence, it is useful to design JavaScript development tools that recognize cross-browser differences and alerts the programmer whenever she forgets to account for these. Some Integrated Development Environments (IDEs) for JavaScript have already implemented this feature, including NetBeans[7] and Aptana.[8]

VI. Related Work

There has been a large number of empirical studies conducted on faults that occur in various types of software applications [13], [14], [15], [16], [17]. Due to space constraints, we focus on those studies that pertain to web applications.

Server-Side Studies. In the past, researchers have studied the causes of web application faults at the server-side using session-based workloads [18], server logs [19], and website outage incidents [20]. Further, there have been studies on the control-flow integrity [21] and end-to-end availability [22], [23] of web applications. Our current study differs from these works in that we focus on web application faults that occur at the client-side, particularly ones that propagate into the JavaScript code.

Client-Side Studies. Several empirical studies on the characteristics of client-side JavaScript have been made. For instance, Ratanaworabhan et al. [24] used their JSMeter tool to analyze the dynamic behaviour of JavaScript in web applications. Similar work was conducted by Richards et al. [25] and Martinsen et al. [26]. A study of parallelism in JavaScript code was also undertaken by Fortuna et al. [27]. Finally, there have been empirical studies on the security of JavaScript. These include empirical studies on cross-site scripting (XSS) sanitization [28], privacy-violating information flows [29],

[4] http://www.jslint.com
[5] http://code.google.com/closure/compiler/
[6] https://github.com/berke/jsure
[7] http://netbeans.org/
[8] http://www.aptana.com/

and remote JavaScript inclusions [30], [31]. Unlike our work which studies functional JavaScript faults, these related works address non-functional properties such as security and performance.

Our earlier work [1] looked at the characteristics of failures caused by JavaScript faults, based on console logs. However, we did not study the causes or impact of JavaScript faults, nor did we examine bug reports as we do in this study. To the best of our knowledge, we are the first to perform an empirical study on the characteristics of these real-world JavaScript faults, particularly their causes and impacts.

VII. Conclusions and Future Work

Client-side JavaScript contains many features that are attractive to web application developers and is the basis for modern web applications. However, it is prone to errors that can impact functionality and user experience. In this paper, we perform an empirical study of over 300 bug reports from various web applications and JavaScript libraries to help us understand the nature of the errors that cause these faults, and the failures to which these faults lead. Our results show that (1) around 65% of JavaScript faults are DOM-related; (2) most (around 80%) high severity faults are DOM-related; (3) the vast majority (around 86%) of JavaScript faults are caused by errors manually introduced by JavaScript code programmers; (4) error patterns exist in JavaScript bug reports; and (5) DOM-related faults take longer to fix than non-DOM-related faults.

For future work, we plan to use the results of this study to design static and dynamic analysis tools that would simplify how programmers write reliable JavaScript code.

Acknowledgment

This research was supported in part by NSERC Strategic Project Grants (Mesbah and Pattabiraman), a Four Year Fellowship (FYF) from UBC, a MITACS Graduate Fellowship, and a research gift from Intel Corporation.

References

[1] F. Ocariza, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *Proc. of the Intl. Symp. on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 2011, pp. 100–109.

[2] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proc. Intl. Symp. on Foundations of Softw. Eng. (SIGSOFT)*. ACM, 2008, pp. 308–318.

[3] S. Guarnieri and B. Livshits, "Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code," in *Proc. Conference on USENIX Security Symposium (SSYM)*, 2009, pp. 151–168.

[4] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for AJAX intrusion detection," in *Proc. Intl. Conference on World Wide Web (WWW)*, 2009, pp. 561–570.

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

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

[7] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proc. Intl. Conf. on Softw. Eng. (ICSE)*. IEEE Computer Society, 2012, pp. 419–429.

[8] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient JavaScript mutation testing," in *Proc. Intl. Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2013.

[9] L. Marks, Y. Zou, and A. E. Hassan, "Studying the fix-time for bugs in large open source projects," in *Proc. Intl. Conf. on Predictive Models in Softw. Eng. (PROMISE)*. ACM, 2011, p. 11.

[10] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proc. Intl. Conf. on Softw. Eng. (ICSE)*. ACM, 2006, pp. 361–370.

[11] F. Ocariza, K. Pattabiraman, and A. Mesbah, "AutoFLox: An automatic fault localizer for client-side JavaScript," in *Proc. Intl. Conf. on Softw. Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2012, pp. 31–40.

[12] S. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," *Static Analysis*, pp. 238–255, 2009.

[13] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? a fault study using open-source software," in *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, 2000, pp. 97–106.

[14] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, 2006, pp. 25–33.

[15] M. Cinque, D. Cotroneo, Z. Kalbarczyk, and R. Iyer, "How do mobile phones fail? a failure data analysis of symbian os smart phones," in *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 585–594.

[16] Z. Yin, M. Caesar, and Y. Zhou, "Towards understanding bugs in open source router software," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 3, pp. 34–40, 2010.

[17] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*. IEEE Computer Society, 2007, pp. 9–9.

[18] K. Goseva-Popstojanova, S. Mazimdar, and A. D. Singh, "Empirical study of session-based workload and reliability for web servers," in *Proc. Intl. Symp. on Softw. Reliability Eng. (ISSRE)*. IEEE Computer Society, 2004, pp. 403–414.

[19] J. Tian, S. Rudraraju, and Z. Li, "Evaluating web software reliability based on workload and failure data extracted from server logs," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 754–769, 2004.

[20] S. Pertet and P. Narasimhan, "Causes of failure in web applications," *Parallel Data Laboratory, Carnegie Mellon University, CMU-PDL-05-109*, 2005.

[21] B. Braun, P. Gemein, H. P. Reiser, and J. Posegga, "Control-flow integrity in web applications," in *Proc. of Intl. Symp. on Engineering Secure Software and Systems*. Springer, 2013, pp. 1–16.

[22] M. Kalyanakrishnan, R. Iyer, and J. Patel, "Reliability of internet hosts: a case study from the end user's perspective," *Computer Networks*, vol. 31, no. 1-2, pp. 47–57, 1999.

[23] V. N. Padmanabhan, S. Ramabhadran, S. Agarwal, and J. Padhye, "A study of end-to-end web access failures," in *CoNEXT conference*, 2006, pp. 15:1–15:13.

[24] P. Ratanaworabhan, B. Livshits, and B. Zorn, "JSMeter: comparing the behavior of JavaScript benchmarks with real web applications," in *USENIX conference on Web Application Development*, 2010.

[25] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *ACM Conf. on Programming Language Design and Implementation (PLDI)*, 2010, pp. 1–12.

[26] J. Martinsen, H. Grahn, and A. Isberg, "A comparative evaluation of JavaScript execution behavior," in *Proc. of Intl. Conf. on Web Engineering (ICWE)*. Springer, 2011, pp. 399–402.

[27] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers, "A limit study of JavaScript parallelism," in *Proc. Intl. Symposium on Workload Characterization (IISWC)*, 2010, pp. 1–10.

[28] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "An empirical analysis of XSS sanitization in web application frameworks," UC Berkeley, Tech. Rep. EECS-2011-11, 2011.

[29] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript web applications," in *ACM Conf. on Comp. and Communications Security*, 2010, pp. 270–283.

[30] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: Large-scale evaluation of remote JavaScript inclusions," in *Proc. of the Conf. on Computer and Communications Security*. ACM, 2012.

[31] C. Yue and H. Wang, "Characterizing insecure JavaScript practices on the web," in *Proc. Intl. Conf. on World Wide Web (WWW)*. ACM, 2009, pp. 961–970.