JavaScript Errors in the Wild: An Empirical Study

Frolin S. Ocariza, Jr., Karthik Pattabiraman
University of British Columbia
Vancouver, British Columbia, Canada
Email: {frolino, karthikp}@ece.ubc.ca

Benjamin Zorn
Microsoft Research
Redmond, WA, USA
Email: zorn@microsoft.com

Abstract—**Client-side JavaScript is being widely used in popular websites to improve functionality, increase responsiveness, and decrease load times. However, the use of JavaScript can decrease the reliability of these websites. This paper presents an empirical characterization of the error messages printed by JavaScript as it executes in these popular websites, and attempts to understand their root causes. We find that JavaScript errors abound in real websites, and that the errors fall into a small number of categories. We further find that both non-deterministic and deterministic errors occur in the applications, and that the speed of testing plays an important role in exposing errors. Finally, we study the correlations among the static and dynamic properties of the application and the frequency of errors exhibited by it in an attempt to understand the root causes of the errors. The study is carried out using the Firefox web browser.**

I. Introduction

Web 2.0 applications can retrieve information asynchronously without reloading the page or navigating to a new one, and are hence much more interactive than traditional web applications. The interactivity of Web 2.0 applications is made possible by the use of client-side JavaScript, which allows for the creation, modification, and deletion of nodes in the application's Document Object Model (DOM) [1]. Today, as many as 97 of the top 100 most visited websites use client-side JavaScript[2], consisting of thousands of lines of code.

However, JavaScript is weakly typed, and allows the creation and execution of new code at runtime, which makes it prone to programming errors. Further, web browsers are typically tolerant of errors in JavaScript code, although they differ in their handling of the errors. For example, web browsers do not stop executing a web application when it throws an exception; rather they continue to execute (other parts of the application) in response to user events and web browser notifications. As a result, an error in one part of the application can propagate to other parts of the application, and compromise its correctness. This leads to subtle bugs that are hard to find during testing [1].

The main goal of this paper is to empirically study the reliability of JavaScript-based Web 2.0 applications and to identify common error categories in these applications. We also seek to understand the sources of these errors, by studying their correlation with the application's static and dynamic features. In the process, our goal is to formulate design guidelines and principles to help developers and testers make their websites more reliable. Understanding the characteristics of errors in websites can also help developers of static and dynamic analysis tools refine their techniques to focus on the most commonly encountered errors in practice.

Although JavaScript was designed in 1995 by Brendan Eich and was part of the Netscape 2.0 browser, it became popular only in the last five years with the advent of applications such as Gmail and Google Docs. As a result, there have been few academic papers on JavaScript, and fewer still on empirical studies of the behaviour of JavaScript-based Web 2.0 applications. Recent work has studied the performance and runtime behaviour of JavaScript [2], [3], and the security and privacy of JavaScript-based websites [4], [5]. However, to our knowledge, there has been no study on characterizing the *reliability* of JavaScript-based Web 2.0 applications.

We base our study on error messages printed to the JavaScript console in the web browser when executing web applications. Whenever the JavaScript code throws an exception, an error message is printed to the JavaScript console [3]. We use FireBug[4], an add-on to the Firefox web browser, to capture the messages. Our evaluation set consists of fifty websites from the Alexa top 100 most visited websites, which we interact with in a "normal" manner. We analyze the error messages, categorize them and determine if they are non-deterministic, i.e., occur only in a subset of the executions. We also correlate the website's features with the types and frequencies of error messages to understand their relationship.

Static analysis is an alternative technique to using error messages for finding bugs, and has been used with great success for large code bases such as the Linux Kernel [6]. However, error messages have several advantages over static analysis. First, console messages represent errors in real settings after the web application has been released to the public, and hence these errors likely escaped traditional methods of testing. Further, the messages capture erroneous interactions between the web application and the DOM, which static analysis tools are likely to miss, as they do not model the DOM. In the extreme case, some static analysis tools such as JSLint[5] and Closure Compiler[6] only analyze the code's syntax, without any regard for the semantics, and are hence likely

[1]The DOM is a hierarchical representation of the elements in a webpage and is maintained by the web browser.

[2]From www.alexa.com, April 2011.

[3]This console is hidden from the user, but can be enabled on demand.

[4]http://getfirebug.com

[5]www.jslint.com

[6]code.google.com/closure/compiler/

to exhibit false positives. Finally, JavaScript is a challenging language to analyze statically, and hence many such analyzers confine themselves to a "sane" subset of the language [7], [8]. However, as recent studies have shown [2], many websites do not confine themselves to this subset.

A potential disadvantage of error messages is that it is often not possible to determine if the message represents a real bug. This problem may be alleviated by studying bug reports of websites, but unfortunately, a vast majority of real-world websites do not make their bug databases publicly available. Nonetheless, we believe that any user-visible error message is a sign of the website's unreliability and hence assume that every error message is a bug in this study.

We make three main contributions in this paper. They are:

- We develop a systematic methodology to execute web applications in multiple testing modes, and gather their error messages. The methodology has been implemented in the form of automated tools that we have developed [7].
- We observe the following results by executing the tools on popular websites from the Alexa top 100:
  - **JavaScript errors abound in websites**: Even production websites that are mature and well-engineered exhibit errors (average of 4 messages per website).
  - **Errors fall into well-defined categories**: About 94% of the errors fall into one of four categories: Permission Denied (52%), Undefined Symbol (28%), Null Exception (9%), and Syntax Errors (4%).
  - **Presence of non-deterministic errors**: About 72% of the errors are non-deterministic (i.e., vary from one execution to another) and depend on the speed of interaction with the web application (i.e., fast, medium or slow).
  - **Correlation with static and dynamic characteristics**: Error frequencies are positively correlated with some of the static and dynamic characteristics of the applications such as Alexa rank, the number of domains containing JavaScript, the number of function calls, the number of property deletions, the number of object inheritance overridings, but not with others such as size of the code or the number of dynamic *eval* calls in the web application.
- We consider the implications of the findings for web application programmers, testers and tool developers. In a sense, our paper is a "call to arms" for improving the reliability of JavaScript-based Web 2.0 applications.

II. JavaScript Background

JavaScript has gained prominence as the de-facto client-side programming language of the Web. In many respects, JavaScript is similar to languages such as C and Java. However, it differs from them in important ways. For example, JavaScript is dynamically typed, and allows code to be created and executed at runtime (through the *eval* construct, for example). Therefore, it is particularly prone to errors[1].

Note that JavaScript use is not confined to the web. For example, Windows 7 allows the use of JavaScript in sidebar widgets, and Firefox allows JavaScript code in its plugins. Similarly, web applications may use JavaScript in the client-side or the server-side. In this paper, we focus on web applications [8] that use JavaScript in the client-side, which represents the primary use case of the language today.

A website consists of three main components in its web-pages. First, there is the HTML code, which is the basic building block of the page. Second, there are cascading style sheets (CSS), which are used to control the layout of elements in a page. Finally, there is the JavaScript code, which is either embedded in the webpages, or is imported as separate files. The JavaScript code comprises what we call the web application. Unlike CSS and HTML, JavaScript is used for more than just the visual display of the page. As a result, errors in JavaScript can have substantial impact, and may be potentially exploited by attackers.

Typical web applications are structured as a set of event handlers that are triggered by specific actions on the webpage, or by the loading of the page. For example, an 'on click' event handler will be executed whenever a certain element in the webpage is clicked, if the developer has specified a handler for the element. In addition, handlers may be triggered by the expiration of timers and the receipt of AJAX messages from the server. Because of this structure, a JavaScript-based web application may continue execution even if one of the handlers fails. As a result, the application may throw multiple exceptions in a single execution. We consider some exceptions that may be thrown below.

JavaScript code may be included in a web application either when the web browser loads a page for the first time, or when it is dynamically created at runtime (e.g., through eval). In both cases, the code must be parsed before it is executed, and errors in this process can also show up as *syntax errors*.

JavaScript is weakly typed, which means that there is no constraint on the types of variables that a JavaScript variable can refer to. Therefore, before invoking a method on an object or accessing its field, programmers need to ensure that the object has a member function or field by that name. Otherwise the code will throw an *Undefined Symbol* exception.

JavaScript code typically interacts with the elements of the webpage through a data structure called the Document Object Model (DOM). The DOM is an internal representation of the webpage within the web browsers and is organized in a hierarchical structure. The JavaScript code often makes certain assumptions about the DOM, which, if violated, can lead to its failure. For example, an event handler may assume that a certain DOM element is present and attempt to access it. A *NullException* is thrown if the element is not present.

Finally, Web browsers enforce the Same-Origin Policy (SOP), which ensures that JavaScript code from one domain cannot access methods or properties from another domain. Violations of the SOP result in a *Permission Denied* exception.

[7]We will make these tools freely available on the web

[8]We use the term web application to mean Web 2.0 application.

III. Anecdotal Examples

Prior to performing a full experiment on fifty websites, we conducted an initial pilot study on five websites — CNN, IMDb, Yahoo, Amazon, and YouTube. This pilot study allowed us to formulate hypotheses with regards to the research questions we would like to answer. In addition, since only a few websites were considered, we were able to conduct a detailed source code-level analysis of the errors that appeared in these websites, helping us better explain the results from the full experiment. One of our main findings from this pilot study was that errors that appear in production websites could be classified into at least three different categories: *Permission Denied Errors*, *Null Exception Errors*, and *Undefined Symbol Errors*.

In this section, we present some sample error messages from each of the three error categories encountered in the pilot study and explain how they came about. These examples illustrate the subtle nature of these errors, and the potentially severe implications such errors may have.

Permission Denied: Permission denied errors occur when JavaScript code from one domain attempts to access JavaScript components from another domain, violating the same-origin policy. For example, the error message "Permission denied for http://view.atdmt.com to call method Location.toString on http://www.imdb.com." appeared in the IMDb website. It turns out, in this case, that view.atdmt.com is the domain used to set up advertisements in the website.

This error illustrates an important point: JavaScript errors are not solely caused by code written by the developer of the web application, but may also be caused by code written by others. The usage of JavaScript code coming from different domains and the usage of JavaScript frameworks developed separately by other programmers makes this intermingling of JavaScript code even more prominent. Current web development practices could therefore make JavaScript code prone to errors if developers are not made aware of the potential pitfalls associated with this intermingling of JavaScript code.

Null Exception: These errors occur when a property or method is accessed via a null object. As an example, the error message "C is null" was encountered in the Yahoo website. We analyzed the root cause of this error message by tracing through the JavaScript code that lead to the error. After doing so, we found that the error was caused by a typographical error in the value of the "id" attribute of a *div* element in the DOM. The incorrect id caused the getElementById method to return a null value, which, in this case, was assigned to the variable "C". The variable "C" was later used to update the class name of the div element, causing a null exception to be thrown (because "C" has been assigned the value null).

The root cause of this error suggests two important things. First, the structure of the DOM could have an impact on the reliability of a web application. In this example, the error occurred because the div element's ID — which is an attribute found in the DOM — was incorrectly typed. Thus, the reliability of JavaScript based web applications not only relies on the semantic correctness of JavaScript code, but also on correct DOM structure and correct interaction with the DOM.

Second, the appearance of even a small error could prevent subsequent JavaScript code (which could potentially be important) from being executed. In the example provided, the error prevented the styling rules of three additional menu elements from being updated by JavaScript code, since the execution of the JavaScript file containing this code was halted once the error appeared. Although this particular error only affected the appearance of the website in a small way, such a subtle error has the potential to lead to more severe consequences.

Consider, for example, a hypothetical scenario in which a web application allows the user to modify his work online. In this application, the user clicks on a Save button to save his work, and JavaScript is used to specify the file name of the server-side script that would handle the actual saving. If the JavaScript code that specifies the file name is preceded by JavaScript code that sets up other buttons in the application, a null exception error caused by this preceding code would prevent the server-side script for the "Save" button from being specified. This is because the JavaScript code that sets up the file name of the server-side script for the "Save" button has been prevented from executing. As a result, clicking on the "Save" button would do nothing, as it has no server-side function associated with it.

Unfortunately, the user will not be aware of this error and would likely assume that the button is working, so he will simply click on this button every time he wishes to save his work. Thus, by the time the user exits, his work will have been completely lost. The user will not be aware that his work has been lost until after he tries to modify the document again. This hypothetical example goes to show that a fault that initially looks benign could have potentially severe consequences by preventing the execution of subsequent JavaScript code.

Another Null Exception error was encountered in Amazon, where the JavaScript error message "document.getElementById("inappDiv") is null" appeared. This error appears only when the user is not signed in. From the source code, inappDiv is the ID of a div element corresponding to a form for reporting inappropriate content. This form is available only for registered users of Amazon; thus, for users who are not signed in, the inappDiv element does not exist. This error also suggests the subtle nature of JavaScript errors. In addition, it shows that the environment in which the web application is executed (e.g., the interaction speed, whether the user is signed in or not, etc.) could have an effect on the appearance of JavaScript errors.

Undefined Symbol: These errors occur when JavaScript code attempts to call a function or access a variable that has not been previously defined. An example of this error occurred in the Amazon website, where the error message "gbEnableTwisterJS is not defined" appeared.

After doing the source code analysis, we found that the variable "gbEnableTwisterJS" was used as a condition for an *if* statement. Since this variable has not previously been defined, an error took place. We did some additional research and determined that prior versions of this JavaScript code did in fact include the statement “gbEnableTwisterJS = 0”, defining the gbEnableTwisterJS variable. This finding suggests that gbEnableTwisterJS *was* initially used as a variable for the page in which the error appeared, but was later discarded. Unfortunately, not all references to gbEnableTwisterJS got removed from the JavaScript code, leading to the error.

A similar error message which reads “E(“set_hp_firefox_instructions”) is not defined” appeared in Yahoo. In this case, E is an associative array containing the element “set_hp_firefox_instructions”. This element was defined in E in previous versions of the source code, but not in the current version, leading to the error.

This simple error illustrates the difficulties associated with the web application development process — in particular when writing JavaScript code. JavaScript coding typically involves the creation of different JavaScript files which interact with one another during execution, and involves the creation of many functions and variables; keeping track of these different components to ensure consistency can be very complex. In the example given, JavaScript code that was supposed to be taken out has accidentally been left behind, showing that JavaScript code can be difficult to manage, making the web application prone to more programmer errors.

Further evidence of the difficulty of managing JavaScript code could be found in CNN, where the error message “cnn_onMemFBInit() is undefined” appeared. From the source code, it turns out that cnn_onMemFBInit() is a function called only when the condition “CNN_IsMemInit && CNN_IsFBInit” is satisfied. In this case, both Boolean variables were set to true, causing the condition to be satisfied. Interestingly, a comment could be found right after the statement setting CNN_IsMemInit to true that says, “this probably isn’t needed anymore”. Thus, based on this comment, it seems like the Boolean condition was no longer meant to be satisfied and the cnn_onMemFBInit() function was no longer meant to execute; nonetheless, CNN_IsMemInit was still set to true, causing the function to get executed.

IV. Experimental Setup

In this section, we list the research questions we are seeking to answer in our experiments. We then describe the websites we consider in our evaluation. Finally, we explain how we generate test suites and capture JavaScript errors in the websites.

A. Research Questions

Question 1: Are JavaScript errors prevalent in web applications, and if so, do these errors share common traits across websites?

Question 2: Does the speed of interaction affect the frequency of JavaScript errors? An interaction refers to the clicks, *mouseouts*, *mouseovers* and other events triggered by the user when visiting a web application. The speed of interaction refers to how quickly a user performs these interactions.

Question 3: Do non-deterministic JavaScript errors occur in Web 2.0 applications? An error is considered *non-deterministic* if its frequency differs from one execution to another.

Question 4: Are there correlations between the type of content a web application serves and the number of errors in that web application?

Question 5: Are there any correlations between a web application’s static and dynamic characteristics and the number of errors in that web application?

Question 6: Are there inter-category correlations among the different error categories in Web 2.0 applications?

Question 7: Is the reliability of a web application affected by the frameworks used in its construction?

B. Chosen Websites

The websites considered in this experiment belong to the set of top 100 most visited websites as ranked by the web traffic reporting website www.alexa.com on January 7, 2011. The websites are relatively mature and well-engineered. We choose fifty websites from the Alexa Top 100 (see Table I). However, we also ensured that the chosen websites formed a representative set with sufficient variety. For example, multiple country-based Google websites appear in the Alexa Top 100; since these websites have similar characteristics, only one Google website was chosen. We also excluded adult websites and sites that do not contain JavaScript code from the study. The websites all contain several lines of JavaScript code (ranging from 16 to 77722, average is 13856), and some span multiple domains (up to 18, average is 6).

C. Overview of Experiment

In this section, we describe the steps in our experiment. The tools used in the experiment are described in Section IV-D.

Our experiment consists of the following steps.

Step 1: Create *test cases* for each web application. A test case represents an “interaction” with a web application, which may consist of one or more events, depending on the context of use of the web application. For example, opening a webpage involves only a single click, so a test case imitating this interaction would consist only of one event (i.e., clicking the link). In contrast, using a search engine would consist of two events — typing the keyword and clicking the search button. Fifteen test cases are created for each web application using the Selenium tool (see Section IV-D); this group of fifteen test cases makes up one *test suite*. We created the test cases based on normal interactions with the website — i.e., no attempts were made to break the websites to cause them to produce errors.

Step 2: Replay the test suites corresponding to each web application multiple times. Each test suite is replayed in three *testing modes* — fast, medium, and slow — representing the speed of interaction (i.e., the speed at which events in the test suite are replayed in sequence), to determine the effect of the speed of interaction on the frequency of errors (Question 2). To determine if JavaScript errors in websites are non-deterministic (Question 3), each test suite is replayed three times in each of the three testing modes. Thus, each test suite is executed

TABLE I: Error Data and Static Characteristics. The PD, NE, US, SE, and Misc. columns refer to the total number of distinct errors in each error category across all nine runs of the experiment for each website. The JavaScript Errors column refers to the totals. Note that the extensions for the websites are *.com* unless specified otherwise.

Website	Alexa Rank	Bytes of JavaScript Code	Number of Domains	Number of Domains with JavaScript	PD	NE	US	SE	Misc.	Java-Script Errors	Non-Deterministic Errors
Google	1	164089	1	1	0	0	0	0	0	0	0
YouTube	3	420894	2	1	2	2	0	0	0	4	4
Yahoo	4	504503	4	3	1	1	1	0	1	4	3
Baidu	6	12759	1	1	1	0	0	0	0	1	0
QQ	9	210324	7	6	0	1	0	0	0	1	1
MSN	11	122143	7	5	4	0	1	0	0	5	4
Amazon	13	225149	3	2	0	1	1	0	0	2	0
Sina.com.cn	16	512392	18	17	4	0	2	0	0	6	5
WordPress	19	151959	8	7	0	0	1	0	0	1	1
Ebay	20	263615	3	2	1	0	0	0	0	1	0
LinkedIn	22	289599	6	5	0	0	0	0	2	2	2
Bing	23	28678	1	1	3	0	0	0	0	3	2
Microsoft	24	276465	9	9	1	0	0	2	0	3	1
Yandex.ru	25	221566	3	2	0	0	1	0	0	1	0
163	28	438689	12	11	2	0	1	0	1	4	2
mail.ru	30	201063	3	2	0	1	0	1	0	2	1
PayPal	31	258071	2	1	0	0	2	1	0	3	0
FC2	32	91775	6	5	2	0	0	0	0	2	2
Flickr	36	8736	3	1	7	0	0	0	0	7	7
IMDb	37	380061	7	6	4	0	0	0	0	4	4
Apple	38	416295	2	1	0	2	1	0	1	4	3
BBC	43	557137	11	11	0	0	1	0	0	1	0
Sohu	44	224148	12	12	2	0	1	1	0	4	3
Go	45	83512	6	6	4	0	0	0	0	4	4
Soso	46	40439	2	1	1	0	0	0	3	4	0
Youku	50	298149	6	5	0	0	1	0	0	1	1
AOL	51	301306	6	5	1	1	1	0	0	3	2
CNN	54	892169	11	11	4	0	5	0	0	9	9
MediaFire	59	485692	3	2	0	0	1	0	0	1	0
ESPN	61	628953	9	8	3	0	2	0	0	5	5
MySpace	62	720027	8	6	4	0	1	0	0	5	4
MegaUpload	63	139857	3	2	6	0	0	0	0	6	6
Mozilla	64	138855	2	1	0	0	1	0	0	1	0
4shared	66	233052	5	4	2	0	2	0	1	5	2
Adobe	67	591191	4	3	0	0	3	0	2	5	0
About	68	147027	2	2	3	0	2	1	0	6	5
LiveJournal	74	343701	7	6	4	0	0	0	0	4	4
Tumblr	75	247224	4	3	0	1	0	0	0	1	1
GoDaddy	77	317264	4	2	0	0	1	0	0	1	0
CNET	78	987612	13	12	12	3	0	1	0	16	11
YieldManager	82	164512	1	1	0	0	1	0	0	1	0
Sogou	83	8436	1	1	0	0	3	0	0	3	0
Zedo	84	96504	4	4	1	0	1	0	0	2	0
Ifeng	85	101255	11	10	2	3	3	1	0	9	8
ThePirateBay.org	86	506	2	1	2	0	0	0	0	2	2
ImageShack.us	88	425050	10	10	6	1	1	0	0	8	6
Livedoor	91	143131	3	3	2	0	2	0	0	4	4
Weather	94	1637291	8	8	4	0	1	0	0	5	4
NYTimes	95	762306	12	11	6	1	0	0	0	7	6
Netflix	97	208821	2	2	0	0	10	0	1	11	10
Total					101	18	55	8	12	194	139

a total of nine times in our experiment. We use the Selenium tool to replay the test suites (Section IV-D). JavaScript errors that occur during a test suite's replay are typically displayed on a console. For each run of the test suite, the error messages are redirected to a file (called an *error file*).

Step 3: Parse the error files to collect error statistics.We have written a parser (see Section IV-D) to parse the error files and count the number of *distinct errors*. Figure 1 shows the three attributes of a message. Two errors are considered distinct if any one of the three attributes are different: (1) their corresponding text descriptions, (2) the JavaScript files that triggered the errors, or (3) the lines of code that triggered the errors. For each distinct error, the parser counts the *actual* number of times the error appeared in each test suite run (because an error may appear multiple times in a run).

For each distinct error, the parser determines if the error is non-deterministic by comparing its frequencies in each run. An error is considered non-deterministic *in a given testing mode* if its frequency differs across the three runs of that testing mode (because this indicates that the error was triggered in some executions but not in others). We *count* an error as non-deterministic if it is non-deterministic in any of the three testing modes. Finally, the parser classifies each distinct error message into five mutually exclusive categories and counts the number of errors in each category. The error categories were determined based on an initial pilot study of five websites.

D. Tools

For this experiment, the Firefox v. 3.6.13 web browser is used under the Mac OS/X Snow Leopard (10.6.6) platform.

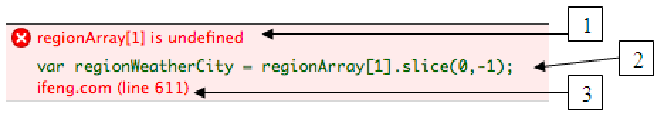
```
regionArray[1] is undefined
var regionWeatherCity = regionArray[1].slice(0,-1);
ifeng.com (line 611)
```

Fig. 1: A screenshot of a JavaScript error message as shown in the Firebug error console. The error message consists of (1) the text description, (2) the line where the error occurred, and (3) the JavaScript file containing the erroneous line.

The machine used for the experiments was a 2.66 GHz Intel Core 2 Duo, with 4 GB of RAM.

The Selenium[9] IDE (v. 1.0.10) is used to create test cases and group the test cases together into test suites. Selenium is an extension to the Firefox web browser that captures and records user interaction with a webpage and converts these interactions into events for later replay. Examples of events are clicks, *mouseouts*, *mouseovers*, and dropdown selections.

To create the test suites, we interact with each web site in reasonable ways to exercise its behavior. The Selenium IDE's recorder runs in the background and records this interaction to create the test case. In some cases, Selenium commands need to be entered manually due to limitations of the Selenium IDE. For example, the Selenium recorder currently does not support the recording of *mouseout* and *mouseover* events; however, commands for these events are added manually to the test case. Fifteen test cases for a given web application together constitute the test suite for this web application.

Once a test suite is created, Selenium can replay this test suite at a speed that can be set by the user. The replay speed is adjusted using a slider that ranges from "Slow" to "Fast". In our experiments, the testing modes correspond to three replay speeds — fast, medium, and slow. Selenium replays the fifteen test cases at the chosen speed, for each application. Each test suite is run three times in each testing mode.

The Firebug 1.6.1 debugger, which is an extension to Firefox, is used to capture JavaScript errors during replay. Although Firebug can capture other errors such as those in CSS and XML, we modify its settings to capture only JavaScript errors, which are this study's focus.

Firebug extension called ConsoleExport[10] is used to export the error messages to an error file. The error files are parsed to collect error statistics, as described in Section IV-C.

To help us answer Question 5, we collect each web application's static characteristics using two Firefox extensions: Web Developer[11] and Phoenix[12]. We use Web Developer to determine the JavaScript code size in the web application, and we use Phoenix to count the number of domains and the number of domains with JavaScript. The static characteristics are based on the initial loading of each website's homepage.

For the dynamic characteristics data, we use the traces collected by Richards et al. [2]. We downloaded the traces from the authors' website[13]. Note that for our dynamic analysis, we only considered websites studied by Richards et al.; only 29 of the 50 websites overlap between the studies. Also, note that the study done by Richards et al. is based on Safari, not Firefox, and they consider a different test suite for the websites. Nonetheless, we assume that their measurements represent the true dynamic characteristics of the websites.

Two of the dynamic parameters, properties deleted and object inheritance overriding, need further explanation [2]. Properties deleted refers to the number of object fields, object methods, or DOM elements that are deleted dynamically. Object inheritance overriding refers to the number of times a method belonging to a parent object is overriden by a child object. In other words, this measures the amount of polymorphism present in the application.

Finally, for Question 7, the frameworks were determined using the Library Detector[14] plugin available for Firefox.

V. Results

The results section is organized paralleling the research questions in Section IV-A. For each result, (1) we state our observation (*Observation*), (2) refer to the data from which we made this observation (*Data*), and (3) explain how we arrived at this observation and its possible causes (*Explanation*).

A. Distribution of Error Categories

Table I presents the total number of distinct errors encountered across all nine runs of each website's test suite. The following observations may be made based on the table.

Observation 1: JavaScript errors abound in production Web 2.0 applications, with an average of around 4 errors per website.

Data: JavaScript Errors column in Table I

Explanation: Table I shows that one or more JavaScript errors appeared in 49 of the 50 websites tested in our experiment (Google was the only website which did not have errors). The maximum distinct error count was 16 (CNET). On average, 3.88 distinct errors appeared in each web application, with a standard deviation of 3.02.

Observation 2: Errors predominantly fall into four distinct categories, which are described below.

Data: PD, NE, US, SE, and Misc. columns in Table I

Explanation: The error messages were found to belong to the following categories.

Permission Denied Errors (PD) – These errors occur when JavaScript code from one domain attempts to access an object or variable belonging to a different domain in the same webpage, thereby violating the same-origin policy. In our study, these errors are often caused by the inclusion of advertisements from domains attempting to access the Location.toString method in the domain of the website.

Null Exception Errors (NE) – These errors occur when a null value is used to access properties or methods.

[9]http://seleniumhq.org/
[10]http://www.softwareishard.com/blog/consoleexport/
[11]http://chrispederick.com/work/web-developer/
[12]https://addons.mozilla.org/en-us/firefox/addon/phoenix/
[13]http://sss.cs.purdue.edu/projects/dynJavaScript/
[14]https://addons.mozilla.org/en-US/firefox/addon/library-detector/

For example, if a variable X is assigned the value document.getElementById(”label”), but the id “label” does not exist in the DOM, X will be assigned the value null; when X.innerHTML is used in a subsequent line, a null exception error will occur.

Undefined Symbol Errors (US) – These errors occur when the JavaScript code (1) calls a function that has not been defined, (2) refers to a method or property that does not belong to a particular object, or (3) uses a variable that has either not been declared or assigned a value. For example, if an object A has the properties a1 and a2, while object B has the properties b1 and b2, attempting to access a2 via object B (i.e., B.a2) results in an undefined symbol error.

Syntax Errors (SE) – These errors occur due to syntactic violations in JavaScript code. Examples include missing end brackets, missing semi-colons, and unterminated string literals. They can also occur as a result of differences in parsing JavaScript among browsers. All syntax errors found in our study came from static code (as opposed to dynamic code generated by the *eval* function). Also, of the eight syntax errors encountered in our study, six came from JavaScript code written specifically for the website, while two came from external JavaScript code for advertisements.

Miscellaneous Errors (Misc.) – Errors that occur in only a single web application and do not fall under the above categories are categorized as “Miscellaneous” errors. For example, an “uncaught exception” error appeared in LinkedIn, but did not appear in other websites, and is therefore classified under the “Miscellaneous” category.

Distribution of errors: Errors belonging to the permission denied category are the most prominent. Based on the data from Table I, permission denied errors make up 52.1% of all errors; null exception errors make up 9.3%; undefined symbol errors make up 28.4%; and syntax errors make up 4.1%. The remaining errors are in the Miscellaneous category. As mentioned, permission denied errors are mostly caused by advertisements. We find that advertisements are present in over 30 of the 50 websites, and hence the dominance of this category.

B. Effect of Testing Mode

In the previous subsection, we studied the number of distinct error messages. In this section, we analyze the actual number of occurrences of the error messages in order to understand the effect of testing mode. Due to space constraints, we focus on two websites — CNN and mail.ru — to illustrate the trends we observe across all web applications. Tables II and III show the occurrences of a subset of the error message that appeared in CNN and mail.ru, respectively, for each testing mode.

Observation 3: The appearance of an error message depends on testing mode (i.e., the speed of interaction).

Data: Tables II and III (Average columns)

Looking at the “Average” columns in Tables II and III, it becomes apparent that some error messages appear in one mode, but not in another. For example, in CNN, the error “targetWindow.cnnad_showAd is not a function” appears in fast mode, but does not appear in the other two modes. Similarly, the error “window.parent.CSIManager is undefined” appears in slow mode, but not in other modes. For mail.ru, the error “b is null” appears in fast mode, but not in the medium and slow modes.

Further, the tables show that some errors are more frequent in one mode compared to others. For example, the error message “Permission Denied for ad.doubleclick.net to call method Location.toString on www.cnn.com” appears an average of 12.33 times and 10.00 times in fast and slow mode, respectively, but only appears an average of 4.33 times in medium mode.

C. Appearance of Non-deterministic Errors

In this section, we present observations regarding the appearance of non-deterministic errors in Web 2.0 applications. Recall that a non-deterministic error is one whose frequency varies across multiple executions of the web application in the *same* testing mode.

Observation 4: Non-deterministic errors occur in many Web 2.0 applications.

Data: Tables II and III

Explanation: Tables II and III show the actual number of appearances of several errors in different runs. From this data, it can be seen that for a given testing mode, the number of actual occurrences of some errors vary across different executions. These are classified as non-deterministic errors. For example, the error “Permission Denied for view.atdmt.com to call method Location.toString on www.cnn.com” in CNN (second row) in slow mode, appears 25 times in the first run, 20 times in the second run, and 16 times in the third run. Similarly, for mail.ru, in fast mode, the error “b is null” appears twice in the first and third runs, but appears three times in the second run.

Non-deterministic errors are caused by different factors in each of the modes. Non-deterministic errors in the fast and medium modes are typically caused by switching pages (i.e., navigating to a new webpage) in the middle of accessing a member of the “parent” or “window” objects. During the switch, the value of the “parent” and/or “window” object changes because the values of these objects are dependent on the page being visited. As a result, if the switch happens *while* a member of these objects is being accessed in the previous page, the objects will be undefined during the transition, leading to the error. Such errors only occur if the switch happens *during* execution of the specific line of code that uses “parent ” or “window”; since the switch does not always align with the execution, it is non-deterministic in nature.

To verify the above explanation, a subset of the test suites was run with pauses introduced between each page switch in both fast and medium modes. The non-deterministic errors no longer appeared, indicating that the sudden page switches were responsible for the non-deterministic errors in both modes.

In contrast, most of the non-deterministic errors in slow mode are caused by advertisements, mainly due to permission denied errors. In some runs, the advertisement appears

TABLE II: Actual number of appearances of errors in CNN across different runs (long error messages have been shortened to save space).

	Fast Mode				Medium Mode				Slow Mode			
Error Message	Run 1	Run 2	Run 3	Average	Run 1	Run 2	Run 3	Average	Run 1	Run 2	Run 3	Average
Permission Denied for view.atdmt.com to call method Location.toString on marquee.blogs.cnn.com	4	4	4	4.00	1	3	3	2.33	2	2	3	2.33
Permission Denied for view.atdmt.com to call method Location.toString on www.cnn.com	20	17	20	19.00	22	22	16	20.00	25	20	16	20.33
Permission Denied for ad.doubleclick.net to call method Location.toString on www.cnn.com	8	16	13	12.33	3	6	4	4.33	7	12	11	10.00
targetWindow.cnnad_showAd is not a function	0	2	5	2.33	0	0	0	0.00	0	0	0	0.00
window.parent.CSIManager is undefined	0	0	0	0.00	0	0	0	0.00	1	1	0	0.67

TABLE III: Actual number of appearances of errors in mail.ru across different runs (long error messages have been shortened to save space).

	Fast Mode				Medium Mode				Slow Mode			
Error Message	Run 1	Run 2	Run 3	Average	Run 1	Run 2	Run 3	Average	Run 1	Run 2	Run 3	Average
missing ; before statement	7	7	7	7.00	7	7	7	7.00	7	7	7	7.00
b is null	2	3	2	2.33	0	0	0	0.00	0	0	0	0.00

TABLE IV: Average number of distinct errors for each web application type

Website type	Average Errors	Average Non-deterministic errors	Number of sites
Search Engine	2.17	1.50	6
Media Download	4.25	3.50	8
News	5.35	4.00	17
Blogs	2.00	2.00	3
Shopping	4.67	3.33	3
Social Networking	3.50	3.00	2
Business	2.64	1.09	11

in a given page, but in other runs, the advertisement may not appear, explaining the non-deterministic behaviour. This behaviour is not as prominent in fast and medium modes because in these modes, the test suites are switching between pages so quickly that the erroneous JavaScript code is not triggered during testing.

Overall, when all distinct errors across all websites are considered, fast mode exposes a total of 82 non-deterministic errors; medium mode exposes 90, and slow mode exposes 100 non-deterministic errors (not shown in the table). Thus, counter-intuitively, slow mode actually exposes the most non-deterministic errors across the modes.

Observation 5: Non-deterministic errors are more prominent than deterministic errors in Web 2.0 applications, and constitute 72% of the total distinct errors.

Data: JavaScript Errors, Table I

Explanation: From Table I, the total number of distinct errors found across all websites is 194 (summation of the "JavaScript Errors" column). Of these 194 errors, 139 are non-deterministic in one or more of the three testing modes (the summation of the "non-deterministic errors" column).

D. Web Application Type

We categorized the fifty websites in our study based on the category of content they serve. The categories are roughly based on the the categorizations provided in CyberPatrol[15]. Table IV shows the number of websites in each category, the average number of errors in each category, and the average number of non-deterministic errors.

Observation 6: News websites have the highest average number of errors and non-deterministic errors across all application types.

Data: Table IV

Explanation: On average, news websites have 5.35 errors, which is higher than any other web application type. This is also true for non-deterministic errors. Although this behaviour requires further investigation, we believe that the prominence of errors in news websites has to do with the overall structure of these kinds of websites. News websites tend to be very dynamic in structure, containing advertisements and flashing content, and allows many interactive features such as polls. In other words, news websites look "busier" compared to other website types and hence are more likely to exhibit errors.

E. Correlation with Static and Dynamic Characteristics

In this section, we study the correlation of JavaScript errors with the static and dynamic characteristics of the tested websites. We use the Spearman rank correlation coefficient because it is non-parametric and hence does not require the data to be normally distributed [9]. Table V shows the Spearman coefficients between each error category and each static characteristic of the web application. Table VI shows the Spearman coefficients of the JavaScript errors with the dynamic web application characteristics. The higher the magnitude of the coefficient, the higher the correlation (a positive correlation means one value has the tendency to increase as the other

[15]www.cyberpatrol.com/research/sitereview.asp

TABLE V: Spearman rank correlation coefficients between error categories and static web application characteristics. Correlations at the 0.05 level are marked with a *, while those at the 0.01 level are marked with **.

	Correlations			
Error Category	Alexa Rank	JavaScript Size (Bytes)	Domains	Domains with JavaScript
PD	0.222	0.166	0.465**	0.450**
NE	0.213	0.401**	0.334*	0.312*
US	0.374**	0.246	0.152	0.200
SE	0.339*	0.305*	0.420**	0.435**
All Errors	0.375**	0.273	0.397**	0.396**

increases, while a negative correlation means one value has a tendency to decrease as the other increases).

As mentioned in Section IV-D, dynamic characteristics data were available for only 29 of the 50 websites. Of these 29 websites, only two incurred syntax errors. As a result, we do not report the values for this category. Further, we study the correlations with seven dynamic characteristics in their study.

As always, it is important to remember that correlation does not imply causation. However, correlations can still provide explanations as to the *possible* causes of the JavaScript errors, which can be verified through additional investigation. We have not investigated causations in this study, and hence our explanations for the results are anecdotal at this point.

We now report the significant trends in the correlation coefficients. We base our interpretation of the correlation coefficients on the guidelines suggested in [10], as follows: values of 0.00 to 0.35 represents no correlation or low correlation, 0.35 to 0.50 represents a medium correlation, and values higher than 0.50 represent a high correlation.

Observation 7: There is a medium correlation between the total number of JavaScript errors and the number of domains and the number of domains containing JavaScript code.

Data: Table V

Explanation: Table V indicates that JavaScript errors have a 0.397 correlation with the total number of domains, and a 0.396 correlation with the number of domains with JavaScript, suggesting that applications using more domains tend to be less reliable. Further, permission denied errors have a 0.465 correlation with the total number of domains, and a 0.450 correlation with the number of domains with JavaScript. The reason for this behavior is that permission denied errors occur when JavaScript code from one domain tries to access resources from another domain. Thus, the more domains there are (with or without JavaScript), the higher the chances of different domains trying to access resources from one another.

Observation 8: There is a low correlation between the total number of distinct JavaScript errors and the JavaScript code size (i.e., number of bytes).

Data: Table V

Explanation: The Spearman rank correlation coefficient between the total number of distinct JavaScript errors and the JavaScript code size (in bytes) is 0.273. Thus, there is a low correlation between these two parameters, suggesting that smaller code sizes will not necessarily lead to fewer errors because even with few lines, many functions may be called while the web application is running. However, null exception errors have a medium correlation of 0.401 with the JavaScript code size. A source code-level analysis of the JavaScript program may be required to explain this latter result.

Observation 9: There is a medium correlation between the total number of distinct JavaScript errors and the Alexa rank of the website.

Data: Table V

Explanation: The Spearman rank correlation coefficient between the total number of distinct JavaScript errors and the Alexa rank is 0.375. Thus, there is a medium correlation between these two parameters, suggesting that less popular websites may have higher number of errors. This result suggests that heavily visited websites are likely to be more reliable. Further investigation across a wider range of Alexa ranks is needed to substantiate this result.

Observation 10: There is a medium correlation between the total number of distinct null exception errors and the number of functions called dynamically by the web application.

Data: Table VI

Explanation: As shown in Table VI, the Spearman rank correlation coefficient between the total number of distinct null exception errors per web application and the number of functions called at runtime is 0.426. This medium correlation could be explained by the fact that null exception errors are often caused by failed accesses to the DOM of the web application (e.g., undefined DOM element ids causing a variable to be null after a call to getElementById). This is also supported by the next observation. Because DOM manipulation is one of the most common usages of JavaScript [11], an increase in the number of functions called at runtime would increase the number of DOM accesses, thereby increasing the likelihood of null exception errors.

Observation 11: There is medium correlation between the number of null exception errors and the average number of element/property deletions in the JavaScript code.

Data: Table VI

Explanation: The correlation coefficient between the number of null exception errors and the average number of property and element deletions is 0.448, signifying a medium correlation. We believe this behaviour is also due to the relationship between null exception errors and DOM accesses (see previous observation). Specifically, if a DOM element is deleted and the JavaScript code tries to subsequently access that element, the resulting value will be null, thus resulting in a null exception.

Observation 12: There is medium correlation between the number of undefined symbol errors and the average number of object inheritance overridings in the code.

Data: Table VI

Explanation: The Spearman coefficient between the number of undefined symbol errors and the average number of object inheritance overridings is 0.490, signifying a medium correlation. If we assume that object inheritance overridings is a proxy for the amount of polymorphism in the application,

TABLE VI: Spearman rank correlation coefficients between error categories and dynamic web application characteristics. Correlations at the 0.05 level are marked with a *, while those at the 0.01 level are marked with **.

	Correlations					
Error Cat.	Function Calls	Variadic Function Calls	Eval Calls	Properties Added	Properties Deleted	Inheritance Overriding
PD	0.159	-0.053	0.126	-0.005	0.056	-0.070
NE	0.426*	0.235	0.195	0.152	0.448*	0.269
US	0.074	0.240	0.200	-0.069	0.033	0.490**
Total	0.308	-0.003	0.257	0.099	0.128	0.186

then this number reflects the fact that programmers are more likely to be confused about the identity of objects when the code has a lot of polymorphism. For example, if an object B inherits from object A, and object B has a method called b() which object A does not have, then, a call to A.b() would lead to an undefined symbol error.

Observation 13: There is a low correlation between the total number of distinct JavaScript errors and the number of calls to the *eval* function.

Data: Table VI

Explanation: The correlation coefficient between the total number of distinct JavaScript errors and the number of eval calls is 0.257. In previous studies [5], [4], it has been suggested that calls to *eval* tend to compromise the security and privacy of the web application. On the other hand, our study suggests that the number of *eval* calls does not correlate with the frequency of errors in the web application. A possible reason could be that *eval* is used primarily for JSON and other mundane reasons [12].

F. Inter-Category Correlations

In this section, we present the inter-category correlations we found in our study, namely those among different categories of errors identified in Section V-A. Due to space constraints, we do not report these results in the tables.

Observation 14: The correlations of each non-miscellaneous error category (permission denied, null exception, and undefined symbol) with syntax errors range from medium to high.

Data: Table I

Explanation: After calculating the Spearman rank correlation coefficients, the correlations between total syntax errors and total permission denied, null exception, and undefined symbol errors are 0.378 (medium), 0.635 (high), and 0.409 (medium), respectively. This result suggests that syntax errors often lead to errors belonging to other categories (except miscellaneous). Table I supports this observation — all the websites in which an syntax error appeared, one or more errors from the non-miscellaneous categories appeared.

Observation 15: There is a high correlation between the number of non-deterministic null exception errors and the number of non-deterministic undefined symbol errors.

Data: Table I

Explanation: We found the Spearman rank correlation coefficient between non-deterministic null exception errors and non-deterministic undefined symbol errors to be 0.560, suggesting that there is high correlation between these two error categories when it comes to non-deterministic errors. This behaviour requires further investigation.

TABLE VII: Average number of distinct errors for each framework

JavaScript Framework	Average Errors	Number of sites
jQuery	4.04	26
Yahoo UI	3.67	6
Prototype	3.00	3
Mixed	5.25	4
None	2.10	10

G. JavaScript Framework

We now analyze the relationship between errors in a web application and the JavaScript framework(s) they use. Table VII shows the classification of websites by framework. websites using multiple frameworks are classified as “Mixed”, while those using no frameworks are classified as “None”. Framework categories encompassing fewer than three websites are not shown as they may not be significant.

Observation 16: Websites using multiple JavaScript frameworks have a higher number of JavaScript errors compared to websites using only a single framework.

Data: Table VII

Explanation: Websites using multiple JavaScript frameworks had an average of 5.25 errors — higher than the average for websites using only a single framework. It has been suggested that using multiple frameworks for a single website can affect the performance of Web 2.0 applications, as it forces the client to download more JavaScript code[13]. Based on our result, another reason to avoid mixed framework usage is that it can cause the web application to become unreliable, perhaps due to inconsistencies between the frameworks, and the difficulty of maintaining code written using multiple frameworks. Another noteworthy result is that websites that do not deploy any framework do not have significantly different number of errors than those that use frameworks.

VI. Threats to Validity

An internal threat to the validity of our results is that the number of web applications considered in our study is limited. In addition, we restricted our study to the most visited websites according to Alexa. It is possible that some of our observations may not hold for websites with lower Alexa ranks.

An external threat to validity is that we performed our study on only one browser (Firefox). We believe the Firefox browser is a fitting choice to carry out our empirical study because of its popular usage. However, future work may have to consider the behaviour of web applications in other commonly-used browsers such as Internet Explorer, Safari, and Opera.

In our study, we have also assumed that all the JavaScript error messages displayed by the Firebug console are actual bugs, and that bugs are representative of the reliability of the web application. JavaScript bug reports may be used to confirm the nature of these error messages; unfortunately, such bug reports are often not available even for popular web applications such as the ones we studied. As such, we consider this a potential construct threat to our results' validity.

VII. Related Work

There have been several studies analyzing the causes of errors that occur at the server-side of web applications and their ensuing reliability [14], [15], [16]. Other studies have examined the end-to-end availability of internet applications [17], [18]. Our study differs in that we focus on errors in JavaScript code, which executes on the client (i.e., web browsers).

Recent work has investigated the client-side reliability of web applications. Methods such as user behaviour analysis [19], robustness testing [20], invariant extraction [21], and web fault taxonomy creation [22] have been used to improve the detection of client-side errors. Further, static analysis has been used to find errors in web applications [7], [8], [23]. Record and replay tools such as Mugshot [24] and WaRR [25] aid in the reproduction of client-side errors. However, unlike our work, these papers do not conduct an empirical study of JavaScript errors in web applications.

A number of papers have studied the dynamic behavior of JavaScript programs. For instance, Richards et al. [2] conduct an empirical study of dynamic JavaScript behaviour based on collected traces; similar work was done by Ratanaworabhan et al. [26] with their JSMeter tool. Fortuna et al. [27] perform a limit study on the parallelism available in JavaScript code. However, none of these papers investigate the reliability of web applications.

Empirical studies on the security [28], [4], [8] and privacy [5] of Web 2.0 applications have been conducted on the Alexa top websites and popular widgets. These papers also differ from our study in that they do not study web applications' errors, which may or may not lead to security vulnerabilities.

VIII. Summary and Implications

Summary: In this paper, we have performed an empirical study of JavaScript error messages logged by fifty of the top 100 websites. Our results show that errors: (1) abound in these web applications, (2) fall into well-defined categories, (3) are often non-deterministic in nature, and (4) exhibit correlations with both static and dynamic characteristics of the web application, and (5) depend on the speed of interaction.

The high frequencies of errors is surprising given that these are highly popular and mature, production websites, and that our test cases constitute normal interactions with them. One reason for this result may be that many of the errors occur due to the interaction of the JavaScript code with the the webpage's DOM, and because they are non-deterministic. This makes the errors difficult to find during the development and testing phases of the web application.

We emphasize that this study is only a preliminary step. A more comprehensive study of websites beyond the most popular ones in Alexa may reveal other classes of errors beyond that found in our study. Such a study would reveal to what extent the reliability of a website correlates with its popularity (Observation 9 suggests an inverse correlation), which can quantify the benefits of reliability improvement.

We therefore believe that there needs to be a concerted effort on the part of programmers, testers and tool developers (static and dynamic analysis) to improve the reliability of JavaScript based Web 2.0 applications. We examine the implications of our results on these three groups.

Implications for Programmers: For programmers, our observations could act as guidelines that help them write more reliable JavaScript code. For instance, Observation 12, which states that object inheritance overriding correlates well with undefined symbol errors, suggests that inheritance in JavaScript is best avoided unless absolutely essential. Methods such as namespacing and reuse of methods across objects have been suggested as alternatives to inheritance, which is error-prone due to the complicated nature of Prototype-based inheritance in JavaScript [29]. In addition, Observation 7 suggests that using fewer domains could result in fewer errors. Finally, Observation 16 suggests that mixing of JavaScript frameworks should be avoided. Finally, the categorization of JavaScript errors (Observation 2) can also help programmers shift their attention towards preventing these common errors.

Implications for Testers: One of the most significant implications of our results is that JavaScript testing should be done in different testing speeds (i.e., testing modes). According to Observation 3, each testing mode exposes errors that are different from the others; thus, testing in only one mode would allow the tester to catch only a (small) subset of the errors.

In addition, we have shown in our results that non-deterministic errors are prominent in web applications (Observations 4 and 5). Thus, it is important to create testing schemes that specifically attempt to catch these kinds of errors. In our results, for example, we found that several non-deterministic errors were caused by advertisements, particularly in slow mode (see Observation 4). This observation calls for the need for more extensive integration testing, in which the JavaScript code is tested after the advertisements have been integrated. This could even be offered as a service by advertisement serving applications such as Google AdSense [16].

Traditionally, testers have used code coverage as a metric to measure the completeness of their tests. The inherent assumption in the use of this metric is that the amount of code in the application correlates with the number of errors. However, Observation 8 suggests that for web applications using JavaScript, there is a low correlation between these two parameters. Thus, when testing the reliability of web

[16]www.google.com/adsense/

applications, code coverage may not be the appropriate metric. Instead, *path coverage* may be a better choice.

Implications for Tool Developers: The dependence of the appearance of errors on testing mode (Observation 3) means that more emphasis should be placed on the speed of interaction when testing JavaScript code. Such tests could be simplified if JavaScript testing tools are developed to automatically perform tests in different testing modes. For example, in Selenium, testing mode needs to be adjusted manually using a slider, which means only one testing mode could be considered at each test suite run. It would, however, be much simpler for testers to specify multiple testing modes that should be considered prior to running the test suite.

Our observations also suggest the need for static analysis tools appropriate for JavaScript. Simple syntactic checks no longer suffice, as the execution of JavaScript code depends not only on the semantic correctness of individual event handlers, but also on the order events are triggered and the current state of the DOM. For example, Observation 11 suggests that the number of element or property deletions correlates with the number of null exception errors. Static analyzers of JavaScript code should therefore consider the DOM in their analysis. Further, knowledge of the common kinds of errors may enable targeted checking by static analysis tools.

References

[1] T. Mikkonen and A. Taivalsaari, "Using JavaScript as a Real Programming Language," *Sun Microsystems Laboratories Technical Report*, vol. 168, 2007.

[2] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10, 2010, pp. 1–12.

[3] P. Ratanaworabhan, B. Livshits, and B. Zorn, "JSMeter: comparing the behavior of JavaScript benchmarks with real web applications," in *Proceedings of the 2010 USENIX conference on Web application development*, 2010, pp. 3–3.

[4] C. Yue and H. Wang, "Characterizing insecure JavaScript practices on the web," in *Proceedings of the 18th international conference on World wide web*, ser. WWW '09, 2009, pp. 961–970.

[5] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript web applications," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10, 2010, pp. 270–283.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ser. SOSP '01, 2001, pp. 73–88.

[7] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for AJAX intrusion detection," in *Proceedings of the 18th international conference on World Wide Web*, ser. WWW '09, 2009, pp. 561–570.

[8] S. Guarnieri and B. Livshits, "Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code," in *Proceedings of the 18th conference on USENIX security symposium*, ser. SSYM'09, 2009, pp. 151–168.

[9] S. Kan, *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[10] J. Cohen, *Statistical power analysis for the behavioral sciences*. Lawrence Erlbaum, 1988.

[11] H. Bidgoli, *The Internet Encyclopedia*. John Wiley & Sons Inc, 2004.

[12] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do: A large-scale study of the use of eval in javascript applications," in *ECOOP*, 2011.

[13] Pingdom. (2008, Jun.) JavaScript framework usage among top websites. [Online]. Available: http://royal.pingdom.com/2008/06/11/javascript-framework-usage-among-top-websites/

[14] 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.

[15] K. Goseva-Popstojanova, S. Mazimdar, and A. D. Singh, "Empirical study of session-based workload and reliability for web servers," in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004, pp. 403–414.

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

[17] 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.

[18] V. N. Padmanabhan, S. Ramabhadran, S. Agarwal, and J. Padhye, "A study of end-to-end web access failures," in *Proceedings of the 2006 ACM CoNEXT conference*, ser. CoNEXT '06, 2006, pp. 15:1–15:13.

[19] W. Li, M. Harrold, and C. Gorg, "Detecting user-visible failures in AJAX web applications by analyzing users' interaction behaviors," in *Proceedings of the IEEE/ACM conference on Automated software engineering*, 2010, pp. 155–158.

[20] K. Pattabiraman and B. Zorn, "DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, 2010, pp. 191–200.

[21] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of AJAX user interfaces," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 210–220.

[22] A. Marchetto, F. Ricca, and P. Tonella, "Empirical Validation of a Web Fault Taxonomy and its usage for Fault Seeding," in *Proceedings of the 2007 9th IEEE International Workshop on Web Site Evolution-Volume 00*, 2007, pp. 31–38.

[23] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *WWW*, 2011, pp. 805–814.

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

[25] S. Andrica and G. Candea, "WaRR: High Fidelity Web Application Recording and Replaying," in *Proceedings of the 2011 International Conference on Dependable Systems and Networks*, 2011.

[26] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn, "JSMeter: Measuring JavaScript behavior in the wild," *Usenix Conference on Web Application Development (WebApps)*, 2010.

[27] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers, "A limit study of JavaScript parallelism," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, 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," 2011.

[29] R. Nyman. (2008, October) JavaScript namespacing—an alternative to JavaScript inheritance. [Online]. Available: http://robertnyman.com/2008/10/29/javascript-namespacing-an-alternative-to-javascript-inheritance/