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 web applications to improve functionality, increase responsiveness, and decrease load times. However, it is challenging to build reliable applications using JavaScript. This paper presents an empirical characterization of the error messages printed by JavaScript code in web applications, and attempts to understand their root causes. We find that JavaScript errors occur in production web applications, 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 in it in order to understand the root causes of the errors.**

***Index Terms*—JavaScript, Reliability, Error Messages, Alexa top 100, Benchmarking**

I. Introduction

Modern web applications, or Web 2.0 applications, retrieve information asynchronously without reloading the page or navigating to a new one. They are hence much more interactive than traditional web applications. This interactivity is accomplished through the use of JavaScript in the client, which allows for the creation, modification, and deletion of nodes in the application's Document Object Model (DOM). Today, as many as 97 of the top 100 most visited websites use client-side JavaScript[1], and each consists of thousands of lines of JavaScript code. Therefore, in this paper, we use the words website and web application interchangeably.

JavaScript is weakly typed, and allows the creation and execution of new code at runtime. It is widely believed that these factors make 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. This leads to subtle bugs that are difficult to find during testing [1].

The main goal of this paper is to empirically study the errors in JavaScript-based web 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 characteristics, such as the number of calls to *eval*. Another goal is to formulate design guidelines and principles to help developers and testers improve their web applications' reliability.

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 *errors* encountered in JavaScript-based web applications.

An error in a JavaScript-based web application can have severe consequences, including loss of its functionality. For example, one of the errors we found in our study prevented the header and navigational items of ifeng.com from displaying in one of its pages. Instead, these items were replaced by a warning at the top of the page indicating that an error has occurred. While this particular error may be caught by traditional testing techniques, there may be much more subtle errors that are not[2]. Therefore, it is important to study the reliability of JavaScript code in the wild, to understand the errors that occur in them. This is the goal of our study.

We base our study on error messages printed to the JavaScript console by web applications executing in the wild. 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 web-application's characteristics with the types and frequencies of error messages to understand their relationship.

In this study, we assume that an error message corresponds to an actual error, i.e., a software defect. Although this assumption may not hold for every error message, we believe that an error message is an indication of a potential problem in the application. Further, a benign error message may have a

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

[2]Indeed, most errors do not result in such explicit warnings/alerts.

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

[4]http://getfirebug.com

serious, unforeseen consequence after a code update. However, we do not consider the consequences of errors in this study.

Static analysis is an alternative technique to using error messages for identifying errors, and has been successfully applied to 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 testing methods. 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 typically model the DOM. In the extreme case, some static analyzers such as JSLint[5] and Closure Compiler[6] only analyze the code's syntax, disregarding semantics, and hence exhibit false positives and false negatives. 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 web applications do not confine themselves to this subset.

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

- We develop a systematic methodology to execute web applications in multiple testing modes, and categorize their error messages. The methodology has been implemented using both existing tools and tools we have developed.
- We run the tools on 50 of the top 100 most visited web applications to study the characteristics of their errors. We further correlate the messages with the static and dynamic characteristics of the web application. We make all our experimental data freely available for reproducibility[7].
- 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.

The main results from the study are as follows:

- **JavaScript errors occur in web applications**: Even production web applications that are mature and well-engineered exhibit errors (average of 4 distinct error messages per web application).
- **Errors fall into well-defined categories**: About 93% of the errors fall into one of four categories: Permission Denied (52%), Undefined Symbol (28%), Null Exception (9%), and Syntax Errors (4%).
- **Effect of testing mode**: The frequencies of errors depend on the speed of interaction with the web application (i.e., fast, medium or slow).
- **Presence of non-deterministic errors**: About 72% of the errors are non-deterministic (i.e., vary across executions).
- **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 the size of the code or the number of dynamic *eval* calls.

[5]www.jslint.com

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

[7]http://ece.ubc.ca/~frolino/projects/jser/

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 (e.g., through the *eval* construct). Therefore, it is believed to be prone to programming errors [1].

A web application[8] consists of three main components in the client side. First, there is the HTML code, which forms the basic building block of its webpages. Second, there are cascading style sheets (CSS), which are used to control the layout of elements in a webpage. Finally, there is the JavaScript code, which is either embedded in the webpages, or is imported as separate files. Unlike CSS and HTML, JavaScript is used for the web application's core functionality, and not only for display of elements. As a result, errors in JavaScript can have substantial impact, and may even be exploited by attackers [4].

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 asynchronous 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.

JavaScript code may be loaded in the web browser either by statically including it in the web page, or by dynamically creating it at runtime (e.g., through eval). In both cases, the code must be parsed before it is executed, and errors in this process would manifest as *syntax errors*.

JavaScript is weakly typed, which means that there is no constraint on the types of objects 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 browser and is a tree-like 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 the DOM contains a certain element and attempt to access the element. A *NullException*

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

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.

III. Experimental Methodology

In this section, we list the research questions we are seeking to answer in our experiments. We then describe the web applications used in our evaluation. Finally, we explain how we generate test suites, capture JavaScript errors in the web applications, and study their characteristics.

A. Research Questions

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

Question 2: Does the speed of interaction affect the frequency of JavaScript errors? An interaction refers to 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 applications? An error is considered *non-deterministic* if its frequency differs from one execution to another.

Question 4: Are there any correlations between a web application's static and dynamic characteristics and the number of errors in that web application?

Question 5: Are there inter-category correlations among the different error categories in web applications?

Question 6: Is the number of errors in a web application affected by the frameworks used in its construction?

B. Web applications

For our evaluation, we chose fifty web applications from the Alexa Top 100 (see Table I) (as of January 7, 2011). Further, we ensured that the chosen web applications formed a representative set with sufficient variety. For example, the Alexa Top 100 has many country-specific Google based websites; since these websites have similar characteristics, only the main Google website was chosen. We also excluded adult websites and sites containing no JavaScript code from the study. The chosen websites often contain several kilobytes of JavaScript code (minimum of 506 bytes, maximum of 1.56 megabytes, and average of 315 kilobytes), and some span multiple domains (up to 18, average is 6). Table I provides more details on the websites' characteristics.

C. Overview of Experiment

In this section, we describe the steps in our experiment. The tools used in the experiment are described in Section III-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 emulating this interaction would consist only of one event (i.e., clicking the link). In contrast, using a search engine would consist of two events, namely typing the keyword and clicking the search button. Fifteen test cases are created for each web application using the Selenium tool (see Section III-D); this group of fifteen test cases makes up one *test suite*. We created the test cases based on normal interactions with the web application i.e., no attempts were made to break the web applications to cause them to produce errors. On average, each test case consisted of 2.66 events and each test suite visited 29.46 webpages in our experiment.

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 answer Question 2. Note that the testing modes are consistent across all the web applications in the study. In slow mode, there is a delay of 1000 ms between each event (i.e., a delay of 1000 ms beyond the delay already present between each event due to processing); in medium mode, 500 ms; and in fast mode, 0 ms (or rather, negligible delay). To determine if JavaScript errors 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 a total of nine times in our experiment. We use the Selenium tool to replay the test suites (Section III-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 III-D) to parse the error files and count the number of *distinct errors*. Figure 1 shows the three attributes of a message. Two error messages are considered distinct if any one of their three attributes are different, namely (1) their text descriptions, (2) the JavaScript files that triggered the errors, or (3) the lines of code that triggered the errors. Note that it may be possible for identical error messages to represent different errors due to JavaScript minification in web pages; thus, the number of errors we report is a conservative estimate of the actual number of errors. For each distinct error, the parser counts the *actual* number of times the error occurred in each test suite run (because an error may occur 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 applications.

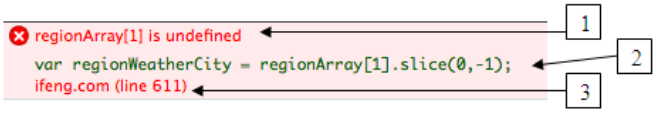

Fig. 1: A screenshot of a JavaScript error message as shown in the Firebug console. The message consists of (1) the text description, (2) the line where the error occurred, and (3) the JavaScript file containing the erroneous line. We assume that two error messages containing identical values for each attribute map to the same error.

D. Tools and Datasets

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

Once a test suite is created, Selenium can replay it 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 in a test suite at the chosen speed, for each application. Each test suite is run three times in each testing mode. The testing modes are identical across all the web applications.

The Firebug 1.6.1 debugger 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.

A 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 III-C.

To help us answer Question 4, 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]. However, for our dynamic analysis, we only considered the web applications studied by Richards et al.; only 29 of the 50 applications overlap between the studies.

The dynamic characteristics considered in our study from Richards et al. are (1) number of function calls, (2) number of calls to *eval*, (3) properties deleted, and (4) object inheritance over-riding. The first two are self-explanatory. 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 metric measures the amount of polymorphism present in the application.

Finally, for Question 6, the frameworks were determined using the Library Detector[14] plugin available for Firefox.

IV. Results

The subsections in the results section parallel the research questions in Section III-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*). We compiled our results data in a spreadsheet available online[15].

A. Distribution of Error Categories

Table I presents the total number of distinct errors encountered across all nine runs of each web application's test suite. We make the following observations based on the table.

Observation 1: JavaScript errors occur in production web applications, with an average of around 4 distinct error messages per web application.

Data: JavaScript Errors column in Table I

Explanation: Table I shows that one or more JavaScript errors occurred in 49 of the 50 web applications in our experiment (Google was the only exception, perhaps due to its simplicity). The maximum distinct error count was 16 (CNET). On average, 3.88 distinct errors occurred in each application, with a standard deviation of 3.02.

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

Data: Permission Denied, Null Exception, Undefined Symbol, and Syntax Error columns in Table I

Explanation: The error messages were found to belong to the following categories: As explained in Section III-C, we determined these based on the results of a pilot study.

Permission Denied - These errors occur when JavaScript code from one domain attempts to access an object or variable belonging to a different domain, thereby violating the same-origin policy (SOP). In our study, these errors are often caused

[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/
[15]http://ece.ubc.ca/~frolino/projects/jser/

TABLE I: Website Error Data and Static Characteristics. The error frequency columns refer to the total number of distinct errors across all nine runs (slow-mode-only data in parentheses). Note that extensions for the web applications are *.com* unless specified otherwise.

Web Application	Alexa Rank	Bytes of JavaScript Code	Total Number of Domains	Number of Domains with JavaScript	Errors in each category					Total JavaScript Errors	Non -Deterministic Errors
					Permission Denied	Null Exception	Undefined Symbol	Syntax Errors	Misc- -ellaneous		
Google	1	164089	1	1	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
YouTube	3	420894	2	1	2 (2)	2 (2)	0 (0)	0 (0)	0 (0)	4 (4)	4 (4)
Yahoo	4	504503	4	3	1 (0)	1 (1)	1 (1)	0 (0)	1 (1)	4 (3)	3 (2)
Baidu	6	12759	1	1	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)	1 (1)	0 (0)
QQ	9	210324	7	6	0 (0)	1 (1)	0 (0)	0 (0)	0 (0)	1 (1)	1 (1)
MSN	11	122143	7	5	4 (3)	0 (0)	1 (1)	0 (0)	0 (0)	5 (4)	4 (3)
Amazon	13	225149	3	2	0 (0)	1 (1)	1 (1)	0 (0)	0 (0)	2 (2)	0 (0)
Sina.com.cn	16	512392	18	17	4 (4)	0 (0)	2 (2)	0 (0)	0 (0)	6 (6)	5 (5)
WordPress	19	151959	8	7	0 (0)	0 (0)	1 (1)	0 (0)	0 (0)	1 (1)	1 (1)
Ebay	20	263615	3	2	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)	1 (1)	0 (0)
LinkedIn	22	289599	6	5	0 (0)	0 (0)	0 (0)	0 (0)	2 (2)	2 (2)	2 (2)
Bing	23	28678	1	1	3 (3)	0 (0)	0 (0)	0 (0)	0 (0)	3 (3)	2 (2)
Microsoft	24	276465	9	9	1 (1)	0 (0)	0 (0)	2 (2)	0 (0)	3 (3)	1 (1)
Yandex.ru	25	221566	3	2	0 (0)	0 (0)	1 (1)	0 (0)	0 (0)	1 (1)	0 (0)
163	28	438689	12	11	2 (2)	0 (0)	1 (1)	0 (0)	1 (1)	4 (4)	2 (2)
mail.ru	30	201063	3	2	0 (0)	1 (0)	0 (0)	1 (1)	0 (0)	2 (1)	1 (0)
PayPal	31	258071	2	1	0 (0)	0 (0)	2 (2)	1 (1)	0 (0)	3 (3)	0 (0)
FC2	32	91775	6	5	2 (1)	0 (0)	0 (0)	0 (0)	0 (0)	2 (1)	2 (1)
Flickr	36	8736	3	1	7 (4)	0 (0)	0 (0)	0 (0)	0 (0)	7 (4)	7 (3)
IMDb	37	380061	7	6	4 (2)	0 (0)	0 (0)	0 (0)	0 (0)	4 (2)	4 (2)
Apple	38	416295	2	1	0 (0)	2 (0)	1 (1)	0 (0)	1 (1)	4 (2)	3 (1)
BBC	43	557137	11	11	0 (0)	0 (0)	1 (1)	0 (0)	0 (0)	1 (1)	0 (0)
Sohu	44	224148	12	12	2 (2)	0 (0)	1 (1)	1 (1)	0 (0)	4 (4)	3 (3)
Go	45	83512	6	6	4 (3)	0 (0)	0 (0)	0 (0)	0 (0)	4 (3)	4 (3)
Soso	46	40439	2	1	1 (1)	0 (0)	0 (0)	0 (0)	3 (3)	4 (4)	0 (0)
Youku	50	298149	6	5	0 (0)	0 (0)	1 (1)	0 (0)	0 (0)	1 (1)	1 (1)
AOL	51	301306	6	5	1 (1)	1 (1)	1 (1)	0 (0)	0 (0)	3 (3)	2 (2)
CNN	54	892169	11	11	4 (4)	0 (0)	5 (3)	0 (0)	0 (0)	9 (7)	9 (7)
MediaFire	59	485692	3	2	0 (0)	0 (0)	1 (1)	0 (0)	0 (0)	1 (1)	0 (0)
ESPN	61	628953	9	8	3 (2)	0 (0)	2 (1)	0 (0)	0 (0)	5 (3)	5 (3)
MySpace	62	720027	8	6	4 (3)	0 (0)	1 (1)	0 (0)	0 (0)	5 (4)	4 (3)
MegaUpload	63	139857	3	2	6 (6)	0 (0)	0 (0)	0 (0)	0 (0)	6 (6)	6 (6)
Mozilla	64	138855	2	1	0 (0)	0 (0)	1 (1)	0 (0)	0 (0)	1 (1)	0 (0)
4shared	66	233052	5	4	2 (2)	0 (0)	2 (2)	0 (0)	1 (1)	5 (5)	2 (2)
Adobe	67	591191	4	3	0 (0)	0 (0)	3 (3)	0 (0)	2 (2)	5 (5)	0 (0)
About	68	147027	2	2	3 (1)	0 (0)	2 (2)	1 (1)	0 (0)	6 (4)	5 (3)
LiveJournal	74	343701	7	6	4 (3)	0 (0)	0 (0)	0 (0)	0 (0)	4 (3)	4 (3)
Tumblr	75	247224	4	3	0 (0)	1 (1)	0 (0)	0 (0)	0 (0)	1 (1)	1 (1)
GoDaddy	77	317264	4	2	0 (0)	0 (0)	1 (1)	0 (0)	0 (0)	1 (1)	0 (0)
CNET	78	987612	13	12	12 (8)	3 (3)	0 (0)	1 (1)	0 (0)	16 (12)	11 (7)
YieldManager	82	164512	1	1	0 (0)	0 (0)	1 (1)	0 (0)	0 (0)	1 (1)	0 (0)
Sogou	83	8436	1	1	0 (0)	0 (0)	3 (3)	0 (0)	0 (0)	3 (3)	0 (0)
Zedo	84	96504	4	4	1 (1)	0 (0)	1 (1)	0 (0)	0 (0)	2 (2)	0 (0)
Ifeng	85	101255	11	10	2 (2)	3 (3)	3 (3)	1 (1)	0 (0)	9 (9)	8 (4)
ThePirateBay.org	86	506	2	1	2 (2)	0 (0)	0 (0)	0 (0)	0 (0)	2 (2)	2 (2)
ImageShack.us	88	425050	10	10	6 (5)	1 (1)	1 (1)	0 (0)	0 (0)	8 (7)	6 (5)
Livedoor	91	143131	3	3	2 (1)	0 (0)	2 (2)	0 (0)	0 (0)	4 (3)	4 (2)
Weather	94	1637291	8	8	4 (4)	0 (0)	1 (1)	0 (0)	0 (0)	5 (5)	4 (4)
NYTimes	95	762306	12	11	6 (6)	1 (1)	0 (0)	0 (0)	0 (0)	7 (7)	6 (6)
Netflix	97	208821	2	2	0 (0)	0 (0)	10 (4)	0 (0)	1 (1)	11 (5)	10 (3)
Total					101 (81)	18 (15)	55 (46)	8 (8)	12 (12)	194 (162)	139 (100)

by advertisements from domains attempting to access the Location.toString method in the domain of the web application. 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 application. In this case, the view.atdmt.com is the domain used to serve advertisements in the IMDb application, and is attempting to call a function in the IMDb domain, which violates the SOP.

Null Exception - These errors occur when a null value is used to access properties or methods. In our study, this error often arises due to missing or mistyped DOM elements. For example, the error message "C is null" was encountered in the Yahoo application. Subsequent analysis revealed 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.

Undefined Symbol - 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. An example of this error occurs in Amazon, where the error message "gbEnableTwisterJS is not defined" occurred. We found that the variable "gbEnableTwisterJS" was used as a condition for an *if* statement, but that the variable had not been defined in the code. Further investigation revealed that prior versions of the code did in fact include the statement "gbEnableTwisterJS = 0", defining the gbEnableTwisterJS variable. This finding suggests that gbEnableTwisterJS *was* initially defined in the code, but was later removed. However, not all references to gbEnableTwisterJS were removed from the code, thus leading to the error.

Syntax Errors – These errors occur due to syntactic violations in JavaScript code. Examples include missing end brackets, missing semi-colons, and unterminated string literals. An example of a syntax error is "missing ; before statement" in mail.ru. Syntax errors can occur either in static JavaScript code or in dynamic code created at runtime. However, all eight syntax errors found in our study came from static code, with six occurring in the main application, and two occurring in advertisements.

Miscellaneous Errors – 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 occurred in the LinkedIn application, but did not occur in other applications, and is therefore

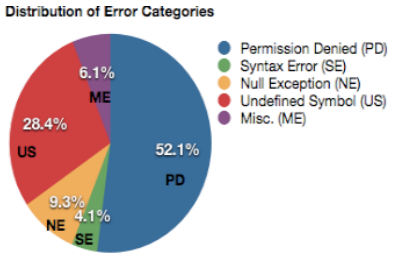

Fig. 2: Percent distribution of each error category

classified under the "Miscellaneous" category.

Distribution of errors: Figure 2, which is based on Table I data, shows the distribution of error categories across all applications. 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%. Together these encompass 93% of the errors. 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 web applications, 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 one application — CNN — to illustrate the trends we observe across all web applications. Table II shows the occurrences of a subset of the error message that occurred in CNN for each testing mode. Similar trends hold for the other applications studied, but space constraints prevent us from reporting them.

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

Data: Table II (Average columns)

Explanation: Looking at the "Average" columns in Table II, it becomes apparent that some error messages occur in one mode, but not in another. For example, the error "targetWindow.cnnad_showAd is not a function" occurs in fast mode, but does not occur in the other two modes. Similarly, the error "window.parent.CSIManager is undefined" occurs in slow mode, but not in other 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" occurs an average of 12.33 times and 10.00 times in fast and slow mode, respectively, but only occurs an average of 4.33 times in medium mode.

C. Occurrence of Non-deterministic Errors

In this section, we study the occurrence of non-deterministic errors in web applications. Recall that a non-deterministic error is one whose frequency varies across multiple executions of the web application in the *same* testing mode. In other words, a non-deterministic error occurs different number of times in each execution of the application.

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

Data: Table I, Table II

Explanation: Table II shows the actual number of occurrences of several errors in different runs of the CNN application. 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, occurs 25 times in the first run, 20 times in the second run, and 16 times in the third 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 transitioning among pages (i.e., navigating to a new webpage) in the middle of accessing a member of the "parent" or "window" objects. During the transition, 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 transition happens *while* a member of these objects is being accessed by JavaScript code in the previous page, the objects will be undefined during the transition, leading to the error. Such errors occur in the event handler of the old page only if the transition happens *during* execution of the specific line of code that uses "parent " or "window", and are hence non-deterministic in nature. These page transition errors may lead to undesirable consequences; for example, if a document is in the midst of being saved, transitioning to a new page will cause the save to abort in the event of an error.

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 in a given page, but in other runs, a different advertisement may 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 transitioning between pages so quickly that the erroneous JavaScript code is not triggered, and hence they do not throw exceptions.

Summary: When all distinct errors across all web applications are considered, fast mode exposes a total of 82 non-deterministic errors, medium mode exposes 90, and slow mode exposes 100 (not shown in the table). Thus, counter-intuitively, slow mode actually exposes the maximum number of non-deterministic errors across the three modes. This is also reflected in Table I, in which the numbers of distinct errors exposed by slow mode are shown within parentheses. As shown in the table, slow mode exposes a total of 162 distinct errors, which corresponds to about 83% of the total errors.

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

TABLE II: Actual number of occurrences 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: Spearman coefficients between error categories and static web application characteristics. Correlations at the 0.05 level are marked with *, while those at the 0.01 level are marked with **.

	Correlations			
Error Category	Alexa Rank	JavaScript Size (Bytes)	Domains	Domains with JavaScript
Permission Denied	0.222	0.166	0.465**	0.450**
Null Exception	0.213	0.401**	0.334*	0.312*
Undefined Symbol	0.374**	0.246	0.152	0.200
Syntax Error	0.339*	0.305*	0.420**	0.435**
All Errors	0.375**	0.273	0.397**	0.396**

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

	Correlations			
Error Category	Function Calls	Eval Calls	Properties Deleted	Inheritance Overriding
Permission Denied	0.159	0.126	0.056	-0.070
Null Exception	0.426*	0.195	0.448*	0.269
Undefined Symbol	0.074	0.200	0.033	0.490**
Total	0.308	0.257	0.128	0.186

Data: JavaScript Errors, Table I

Explanation: From Table I, the total number of distinct errors found across all web applications 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. Correlation with Static and Dynamic Characteristics

In this section, we study the correlation of JavaScript errors with the static and dynamic characteristics of the web applications. 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 III shows the Spearman coefficients between the error categories and static characteristics of the web application. Table IV shows the Spearman coefficients of the JavaScript error categories with the application's dynamic characteristics. The higher the magnitude of the coefficient, the higher the correlation (a positive correlation means one value increases as the other increases, while a negative correlation means the opposite).

As mentioned in Section III-D, dynamic characteristics data were available for only 29 of the 50 applications. Of these 29 web applications, 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; however we report only the four interesting ones. Our technical report [10] has the full details.

Note: 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. Although we have performed a detailed study on the potential causes of null exception errors through manual analysis, we have not done so for other error categories. We leave a detailed investigation of causation to future work.

Significance: We now report the significant trends in the correlation coefficients. For most observations, we report the correlations for which $p < 0.05$ (i.e., significant at the 0.05 level). We call such correlations *significant*.

Observation 6: There is a significant correlation between the total number of JavaScript errors and the number of domains with and without JavaScript code.

Data: Table III

Explanation: Table III 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 have more errors. 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. A possible 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 7: There is no significant correlation between the total number of distinct JavaScript errors and the JavaScript code size (i.e., number of bytes of JavaScript).

Data: Table III

Explanation: The Spearman rank correlation coefficient between the total number of distinct JavaScript errors and the JavaScript code size (in bytes) is 0.273, which is not

significant at the 0.05 level. Thus, smaller code sizes will not necessarily lead to fewer errors. We use the code size instead of the number of lines of JavaScript code, because many web applications minify JavaScript by packing it in a single line.

Observation 8: There is a significant correlation between the total number of distinct JavaScript errors and the Alexa rank of the web application.

Data: Table III

Explanation: The Spearman rank correlation coefficient between the total number of distinct JavaScript errors and the Alexa rank is 0.375, suggesting that less popular applications may have higher number of errors (and vice versa). Further investigation across a wider range of Alexa ranks is needed to substantiate this result.

Observation 9: There is a significant correlation between the total number of distinct null exception errors and the number of functions called dynamically by the web application.

Data: Table IV

Explanation: As shown in Table IV, 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 significant correlation can 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 10: There is significant correlation between the number of null exception errors and the average number of element/property deletions in the JavaScript code.

Data: Table IV

Explanation: The correlation coefficient between the number of null exception errors and the average number of property and element deletions is 0.448, which is significant at the 0.05 level. We believe this behaviour is also due to the relationship between null exception errors and DOM accesses (fourteen of the eighteen null exception errors in our study were due to DOM accesses, based on our analysis of the code). 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 11: There is significant correlation between the number of undefined symbol errors and the average number of object inheritance overridings in the code.

Data: Table IV

Explanation: The Spearman coefficient between the number of undefined symbol errors and the average number of object inheritance overridings is 0.490, which is significant at the 0.05 level. If we assume that object inheritance overridings is representative of the amount of polymorphism in the application, 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, and hence are more likely to make mistakes in accessing member functions or fields. 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 12: There is no significant correlation between the total number of distinct JavaScript errors and the number of calls to the *eval* construct.

Data: Table IV

Explanation: The correlation coefficient between the total number of distinct JavaScript errors and the number of *eval* calls is 0.257. Prior studies [12], [4] have suggested that calls to *eval* can compromise the reliability and security of the web application. However, we do not find evidence for the claim that they can compromise the reliability of the application, perhaps because *eval* is used primarily for JSON and other idiomatic purposes that are less prone to errors [2].

E. 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 IV-A. Due to space constraints, we do not report these results in the tables.

Observation 13: The correlations of each non-miscellaneous error category (permission denied, null exception, and undefined symbol) with syntax errors are significant.

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, 0.635, and 0.409, respectively, all of which are significant. This result suggests that syntax errors often lead to errors belonging to other categories (except miscellaneous). Table I supports this observation - all web applications with a syntax error had one or more errors in the other categories as well.

Observation 14: There is a significant 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 a significant correlation between these two error categories when it comes to non-deterministic errors. This behaviour requires further investigation.

F. JavaScript Framework

Table V shows the classification of applications by framework used in its construction. Applications using multiple frameworks are classified as “Mixed”, while those using no frameworks are classified as “None”. Frameworks encompassing fewer than three websites are not shown as they may not be significant. As can be seen from the table, the majority of web applications in the study were constructed using one or more frameworks, with jQuery being the most popular.

TABLE V: 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

Observation 15: Web applications using multiple JavaScript frameworks have a higher number of JavaScript errors compared to those using only a single framework.

Data: Table V

Explanation: Web applications using multiple JavaScript frameworks had an average of 5.25 errors — higher than the average for applications using only a single framework. It has been suggested that using multiple frameworks can increase the loading time of web applications, as it forces the client to download more JavaScript code [13]. Our result suggests that multiple frameworks also make the application more error-prone, perhaps due to inconsistencies between the frameworks, and the difficulty of maintaining the code.

Observation 16: Web applications using no frameworks have a lower number of JavaScript errors compared to those using at least one framework.

Data: Table V

Explanation: The average number of errors for web applications using no frameworks is 2.10, which is lower than the average for web applications using at least one framework. The reason for this behavior may be that frameworks abstract away details of the JavaScript code, making it more difficult for programmers to map back errors to source code. It is also possible that the library functions used in the frameworks are themselves responsible for the errors.

V. 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 100 most visited websites according to Alexa. It is possible that some of our observations may not hold for websites that are not as popular.

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 browsers.

Our results represent a snapshot of web applications at a specific point in time during which the experiments were performed, and may hence change over time. This is also an internal threat to validity of the results.

In our study, we assume that all the JavaScript error messages are actual bugs. 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.

The study by Richards et al. [2] is based on Safari, not Firefox, and they use different test suites for the web applications in their study. Further, their study precedes ours by at least two years; as a result, some of their data may be outdated.

VI. Related Work

Because of the vast number of studies on web applications' performance, security and reliability, we focus on those pertaining to client-side JavaScript.

Reliability: Dynamic analysis techniques have been proposed to detect client-side errors in web applications. Examples are user behaviour analysis [14], robustness testing [15], invariant-based testing [16], and web fault taxonomy creation [17]. Static analysis techniques have also been used to find errors in web applications [7], [8], [18]. Record and replay tools such as Mugshot [19] and WaRR [20] 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.

Performance: Recent work has studied the performance and parallelism 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. [3] with their JSMeter tool. Fortuna et al. [21] perform a limit study on the parallelism available in JavaScript code. However, none of these papers investigate the reliability of web applications.

Security: Empirical studies on the security [22], [4], [8] and privacy [5] of web applications focus 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.

VII. Implications

A surprising result of this study is the relatively high frequency of errors in highly popular, production web applications, especially because our test cases constitute normal interactions with them. This may be because 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. Hence, the errors are difficult to find during development and testing using current practices (e.g., unit testing).

While it may be argued that errors do not really matter as users continue to use web applications extensively, our experiments unearthed many instances of errors impairing a web application's key functionality. Further, we have shown that there is a correlation between the Alexa rank of a web application and the average number of errors in it (Observation 8). Finally, the rapid pace at which applications are being migrated to the web, and the fact that some of the errors we uncover are fairly straightforward to fix (e.g., syntax errors) suggests that not enough effort is being expended in making web applications reliable, perhaps due to the lack of established tools and practices in this domain. Therefore, there needs to be a concerted effort on the part of programmers, testers and tool developers (of static and dynamic analysis

tools) to improve the reliability of JavaScript web applications.

Implications for Programmers: For programmers, our observations could act as guidelines that help them write more reliable JavaScript code. For instance, Observation 11, which states that object inheritance overriding correlates 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 in JavaScript [23]. In addition, Observation 6 suggests that using fewer domains may result in fewer errors. Finally, Observation 15 suggests that mixing of JavaScript frameworks should be avoided.

An interesting implication of our results is that often errors arise from code over which the programmer may not have direct control. For example, we find that many errors are caused by advertisements, and from unwanted interactions with the DOM (which may or may not be under the programmer's control). These observations lead us to posit that web applications must be tolerant of errors that arise in other components in order to be reliable. Further, the application should have consistency checks and other mechanisms to ensure that errors are caught early and not allowed to propagate.

Implications for Testers: One of the most significant implications of our results is that testing should be done at multiple speeds (i.e., testing modes). Each testing mode exposes different kinds of errors (Observation 3); thus, testing in only one mode would catch only a subset of the errors.

We have also shown in our results that non-deterministic errors are prominent in web applications (Observations 4 and 5). Thus, it is important to develop 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 may even be offered as a service by advertisement serving applications such as Google AdSense.

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 speeds. Further, dynamic tools should execute the web application in realistic settings, with advertisements and other third-party code.

Our observations also suggest the need for advanced static analysis tools. 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 in which events are triggered and the current state of the DOM. For example, Observation 10 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.

VIII. Conclusions and Future Work

We have performed an empirical study of JavaScript error messages in the Alexa top 100 web applications. Our results show that JavaScript errors: (1) occur in these web applications, (2) fall into well-defined categories, (3) depend on the speed of testing, (4) are often non-deterministic in nature, and (5) exhibit correlations with static and dynamic characteristics of the application. Future work will focus on expanding the range of web applications considered in the study.

Acknowledgements: We thank Ali Mesbah and Paruj Ratanaworabhan for providing insightful comments about this work. This work was supported in part by an NSERC Discovery Grant and a research gift from Microsoft Corporation.

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 *ACM Conference on Programming Language Design and Implementation*, ser. PLDI '10, 2010, pp. 1–12.

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

[4] C. Yue and H. Wang, "Characterizing insecure JavaScript practices on the web," in *Intl. Conference on World Wide Web (WWW)*, 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 *ACM Conference on Computer and Communications Security*, 2010, pp. 270–283.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *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 *Intl. Conference on World Wide Web*, 2009, pp. 561–570.

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

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

[10] F. Ocariza, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," University of British Columbia (UBC), Tech. Rep. RADICAL-2011-08-01, May 2011.

[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 *European Conference on Object-Oriented Programming (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] W. Li, M. Harrold, and C. Gorg, "Detecting user-visible failures in AJAX web applications by analyzing users' interaction behaviors," in *IEEE/ACM Conference on Automated Software Engineering*, 2010, pp. 155–158.

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

[16] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of AJAX user interfaces," in *Intl. Conference on Software Engineering*, 2009, pp. 210–220.

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

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

[19] J. Mickens, J. Elson, and J. Howell, "Mugshot: deterministic capture and replay for JavaScript applications," in *7th USENIX Conference on Networked Systems Design and Implementation*, 2010, pp. 11–11.

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

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

[22] 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.

[23] 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/