A Study of Causes and Consequences of Client-Side JavaScript Bugs

Frolin S. Ocariza, Jr., *Student Member, IEEE,* Kartik Bajaj, *Student Member, IEEE,* Karthik Pattabiraman, *Senior Member, IEEE,* and Ali Mesbah, *Member, IEEE*

Abstract—Client-side JavaScript is widely used in web applications to improve user-interactivity and minimize client-server communications. Unfortunately, JavaScript is known to be error-prone. While prior studies have demonstrated the prevalence of JavaScript faults, no attempts have been made to determine their causes and consequences. The goal of our study is to understand the root causes and impact of JavaScript faults and how the results can impact JavaScript programmers, testers and tool developers. We perform an empirical study of 502 bug reports from 19 bug repositories. The bug reports are thoroughly examined to classify and extract information about each bug's cause (the *error*) and consequence (the *failure* and *impact*). Our results show that the majority (68%) of JavaScript faults are *DOM-related*, meaning they are caused by faulty interactions of the JavaScript code with the *Document Object Model (DOM)*. Further, 80% of the highest impact JavaScript faults are DOM-related. Finally, most JavaScript faults originate from programmer mistakes committed in the JavaScript code itself, as opposed to other web application components. These results indicate that JavaScript programmers and testers need tools that can help them reason about the DOM. Additionally, developers can use the error patterns we found to design more powerful static analysis tools for JavaScript.

Index Terms—Faults, JavaScript, Document Object Model (DOM), bug reports, empirical study.

1 Introduction

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

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

While the above features allow web applications to be highly interactive, they also introduce additional avenues for faults in the JavaScript code. In a previous study [1], we collected JavaScript console messages (i.e., exceptions) from fifty popular web applications to understand how prone web applications are to JavaScript faults and what kinds of JavaScript faults appear in these applications. While the study pointed to the prevalence of JavaScript faults, it did not explore their impact or root cause, nor did it analyze the kinds of failures they caused. Understanding the root cause and impact of the faults is vital for developers, testers, as well as tool builders to increase the reliability of web applications.

In this paper, our goal is to discover the causes of JavaScript faults (the *error*) in web applications, and analyze their consequences (the *failure* and *impact*). Towards this goal, we conduct an empirical study of over 500 publicly available JavaScript bug reports. We choose bug reports as they typically have detailed information about a JavaScript fault and also reveal how a web application is expected to behave; this is information that would be difficult to extract from JavaScript console messages or static analysis. Further, we confine our search to bug reports that are marked "fixed", which eliminates spurious or superfluous bug reports.

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

Our work makes the following main contributions:

- We collect and systematically categorize a total of **502 bug reports**, from 15 web applications and four JavaScript libraries, and put the reports in a standard format;
- We classify the JavaScript faults into multiple categories. We find that one category dominates the others, namely DOM-related JavaScript faults (more details below);

• *F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah are with the Department of Electrical and Computer Engineering, University of British Columbia, 2332 Main Mall, Vancouver, BC, Canada, V6T 1Z4. E-mail: {frolino, kbajaj, karthikp, amesbah}@ece.ubc.ca*

- We analyze how many of the bugs can be classified as type faults, which helps us assess the usefulness of programming languages that add strict typing systems to JavaScript, such as TypeScript [3] and Dart [4];
- We quantitatively analyze the nature (i.e., cause and consequences) and the impact of JavaScript faults;
- Where appropriate, we perform a temporal analysis of each bug report characteristic that we analyze. The results of this analysis will indicate how technological changes over the years have set the trend for these characteristics, enabling us to see if we are moving towards the right direction in improving the reliability of client-side JavaScript;
- We analyze the implications of the results on developers, testers, and tool developers for JavaScript code.

Our results show that around 68% of JavaScript faults are *DOM-related* faults, which occur as a result of a faulty interaction between the JavaScript code and the DOM. A simple example is the retrieval of a DOM element using an incorrect ID, which can lead to a null exception. Further, we find that DOM-related faults account for about 80% of the highest impact faults in the web application. Finally, we find that the majority of faults arise due to the JavaScript code rather than server side code/HTML, and that there are a few recurring programming patterns that lead to these bugs.

In addition to the results from the bug report study described in our ESEM'13 paper [5], of which this current work is an extension, we also find that a small – but non-negligible – percentage (33%) of the bug reports are *type* faults, which we describe in Section 2. Furthermore, in our temporal analysis, we observed both downward trends in certain metrics (e.g., browser specificity) and upward trends in others (e.g., number of errors committed in the JavaScript code).

2 Background and Motivation

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

2.1 Web Applications

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

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

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

2.2 JavaScript Bugs

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

JavaScript Bug Sequence. In this paper, we use the term *bug* as a catch-all term that pertains to an undesired behaviour of the web application's functionality. The following sequence describes the progression of a JavaScript bug, and the terms we use to describe this sequence:

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

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

1. JavaScript is a scripting language based on the ECMAScript standard, and it is used in other applications such as desktop widgets and even web servers. However, we restrict ourselves to JavaScript used on the client-side of web applications.

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

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

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

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

failure model to classify JavaScript bugs, as described in Section 3.

DOM-Related Faults. We define a DOM-related fault as follows:

Definition 1 (DOM-Related Fault) *A JavaScript bug $\mathcal{B}$ is considered to have propagated into a* DOM-related fault *if the corresponding error causes a DOM API method $\mathcal{DA}_m$ to be called (or causes an assignment to a DOM API property $\mathcal{DA}_p$ to be made), such that a parameter $\mathcal{P}$ passed to $\mathcal{DA}_m$ (or a value $\mathcal{A}$ assigned to $\mathcal{DA}_p$) is incorrect.*

In other words, if a JavaScript error propagates into the parameter of a DOM access/update method or to the assignment value for a DOM access/update property – thereby causing an incorrect retrieval or an incorrect update of a DOM element – then the error is said to have propagated into a ***DOM-related fault***. For example, if an error eventually causes the parameter of the DOM access method `getElementById()` to represent a nonexistent ID, and this method is called during execution with the erroneous parameter, then the error has propagated into a DOM-related fault. However, if the error does not propagate into a DOM access/update method/property, the error is considered to have propagated into a ***non-DOM-related fault***. Note that based on this definition, the presence of a large number of DOM interactions in the JavaScript code does not necessarily imply the presence of a large percentage of DOM-related faults, as not all errors would necessarily propagate to any DOM API method calls in the code.

Type Faults. We are also interested in determining the prevalence of *type faults*, which we define as follows:

Definition 2 (Type Fault) *A JavaScript bug $\mathcal{B}$ is considered to have propagated into a* type fault *if there exists a statement $\mathcal{L}$ in the JavaScript code such that, during the execution of the code that reproduces $\mathcal{B}$, the statement $\mathcal{L}$ references an expression or variable $\mathcal{E}$ that it expects to be of type t_1, but $\mathcal{E}$'s actual type at runtime is t_2 (with $t_2 \neq t_1$).*

In other words, a type fault occurs if the JavaScript code erroneously assumes during execution that a certain value is of a certain type. Note that our comparison of types bears some similarities to Pradel et al.'s definition of consistent types [6]. In particular, we both make a distinction between different "type categories", such as primitive types and custom types; we describe this in further detail in Section 3.4.

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

2.3 Goal and Motivation

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

Other work has studied JavaScript faults through console messages or through static analysis [7], [8], [9], [10]. However, bug reports contain detailed information about the root cause of the faults and the intended behaviour of the application, which is missing in these techniques. Further, they typically contain the fix associated with the fault, which is useful in further understanding it, for example, to determine fix times.

3 Experimental Methodology

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

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

3.1 Research Questions

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

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

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

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

RQ4: Do JavaScript faults exhibit browser-specific behaviour?

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

RQ6: How prevalent are type faults among the reported bugs?

RQ7: How have the characteristics of JavaScript faults – particularly those analyzed in the above research questions – varied over time?

3.2 Experimental Objects

To ensure representativeness, we collect and categorize bug reports from a wide variety of web applications and libraries. Each object is classified as either a web application or a JavaScript library. In the conference version of this paper [5], we initially made this distinction to see if there are any differences between JavaScript faults in web applications and those in libraries. We did not, however, find any substantial differences after performing our analysis; therefore, our selection of additional experimental objects compared to the conference version was not influenced by this distinction, and we do not report them separately. In total, we collected and analyzed 502 bug reports from 15 web applications and four libraries.

Table 1 lists the range of the software versions considered for each experimental object. The web applications and libraries were chosen based on several factors, including their popularity, their prominent use of client-side JavaScript, and the descriptiveness of their bug reports (i.e., the more information its bug reports convey, the better). Another contributing factor is the availability of a bug repository for the web application or library, as such repositories were not always made public. In fact, finding web applications and libraries that satisfied these criteria was a major challenge in this study.

3.3 Collecting the Bug Reports

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

Step 1 Use the filter/search tool available in the bug repository to narrow down the list of bug reports. The filters and search keywords used in each bug repository are listed in Table 1. In general, where appropriate, we used "javascript" and "js" as keywords to narrow down the list of bug reports (in some bug repositories, the keyword "jQuery" was also used to narrow down the list even further). Further, to reduce spurious or superfluous reports, we only considered bug reports with resolution "fixed", and type "bug" or "defect" (i.e., bug reports marked as "enhancements" were neglected). Table 1 also lists the number of search results after applying the filters in each bug repository. The bug report repositories were examined between January 30, 2013 and March 13, 2013 (applications #1-8, 16-19), and between February 30, 2015 and March 25, 2015 (applications #9-15).

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

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

3.4 Analyzing the Collected Bug Reports

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

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

- **Undefined/Null Variable Usage**: A JavaScript variable that has a `null` or `undefined` value – either because the variable has not been defined or has not been assigned a value – is used to access an object property

TABLE 1: Experimental subjects from which bug reports were collected.

Application ID	Application Name	Version Range	Type	Description	Size of JS Code (KB)	Bug Report Search Filter	# of Reports Collected
1	Moodle	1.9-2.3.3	Web Application	Learning Management	352	(**Text** contains *javascript* OR *js* OR *jquery*) AND (**Issue type** is *bug*) AND (**Status** is *closed*) - Number of Results: 1209	30
2	Joomla	3.x	Web Application	Content Management	434	(**Category** is *JavaScript*) AND (**Status** is *Fixed*) - Number of Results: 62	11
3	WordPress	2.0.6-3.6	Web Application	Blogging	197	((**Description** contains *javascript* OR *js*) OR (**Keywords** contains *javascript* OR *js*)) AND (**status** is *closed*) - Number of Results: 875	30
4	Drupal	6.x-7.x	Web Application	Content Management	213	(**Text** contains *javascript* OR *js* OR *jQuery*) AND (**Category** is *bug report*) AND (**Status** is *closed(fixed)*) - Number of Results: 608	30
5	Roundcube	0.1-0.9	Web Application	Webmail	729	((**Description** contains *javascript* OR *js*) OR (**Keywords** contains *javascript* OR *js*)) AND (**status** is *closed*) - Number of Results: 234	30
6	WikiMedia	1.16-1.20	Web Application	Wiki Software	160	(**Summary** contains *javascript*) AND (**Status** is *resolved*) AND (**Resolution** is *fixed*) - Number of Results: 49	30
7	TYPO3	1.0-6.0	Web Application	Content Management	2252	(**Status** is *resolved*) AND (**Tracker** is *bug*) AND (**Subject** contains *javascript*) (Only one keyword allowed) - Number of Results: 81	30
8	TaskFreak	0.6.x	Web Application	Task Organizer	74	(**Search keywords** contain *javascript* OR *js*) AND (**User** is *any user*) - Number of Results: 57	6
9	Horde	1.1.2-2.0.3	Web Application	Webmail	238	(**Type** is *bug*) AND (**State** is *resolved (bug)*) AND ((**Summary** contains javascript) OR (**Comments** contain *javascript*)) - Number of Results: 300	30
10	FluxBB	1.4.3-1.4.6	Web Application	Forum System	8	(**Type** is *bug*) AND (**Status** is *fixed*) AND (**Search keywords** contain *javascript*) - Number of Results: 8	5
11	LimeSurvey	1.9.x-2.0.x	Web Application	Survey Maker	442	(**Status** is *closed*) AND (**Resolution** is *fixed*) AND (**Text** contains *javascript*) - Number of Results: 252	30
12	DokuWiki	2009-2014	Web Application	Wiki Software	446	(**Status** is *all closed tasks*) AND (**Task type** is *bug report*) AND (**Text** contains *javascript* OR *js*) - Number of Results: 159	30
13	phpBB	3.0.x	Web Application	Forum System	176	(**Status** is *closed*) AND (**Resolution** is *fixed*) AND (**Text** contains *javascript* OR *js*) - Number of Results: 112	30
14	MODx	1.0.3-2.3.0	Web Application	Content Management	1229	(**Type** is *issue*) AND (**Status** is *closed*) AND (**Text** contains *javascript* OR *js*) - Number of Results: 219	30
15	EZ Systems	3.5-5.3	Web Application	Content Management	180	(**Issue type** is *bug*) AND (**Status** is *closed*) AND (**Resolution** is *fixed*) AND (**Text** contains *javascript* OR *js*) - Number of Results: 229	30
16	jQuery	1.0-1.9	Library	—	94	(**Type** is *bug*) AND (**Resolution** is *fixed*) - Number of Results: 2421	30
17	Prototype.js	1.6.0-1.7.0	Library	—	164	(**State** is *resolved*) - Number of Results: 142	30
18	MooTools	1.1-1.4	Library	—	101	(**Label** is *bug*) AND (**State** is *closed*) - Number of Results: 52	30
19	Ember.js	1.0-1.1	Library	—	745	(**Label** is *bug*) AND (**State** is *closed*) - Number of Results: 347	30

or method. *Example*: The variable `x`, which has not been defined in the JavaScript code, is used to access the property `bar` via `x.bar`.

- **Undefined Method**: A call is made in the JavaScript code to a method that has not been defined. *Example*: The undefined function `foo()` is called in the JavaScript code.
- **Incorrect Method Parameter**: An unexpected or invalid value is passed to a native JavaScript method, or assigned to a native JavaScript property. *Example*: A string value is passed to the JavaScript Date object's `setDate()` method, which expects an integer. Another example is passing an ID string to the DOM method `getElementById()` that does not correspond to any IDs in the DOM. Note that this latter example is a type of *DOM-related fault*, which is a subcategory of Incorrect Method Parameter faults where the method/property is a DOM API method/property (as defined in Section 2.2).
- **Incorrect Return Value**: A user-defined method is returning an incorrect return value even though the parameter(s) is/are valid. *Example*: The user-defined method `factorial(3)` returns 2 instead of 6.
- **Syntax-Based Fault**: There is a syntax error in the

TABLE 2: Impact Categories.

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

JavaScript code. *Example*: There is an unescaped apostrophe character in a string literal that is defined using single quotes.

- **Other**: Errors that do not fall into the above categories. *Example*: There is a naming conflict between methods or variables in the JavaScript code.

Note that we do not find instances where a bug report belongs to multiple fault categories, and hence, we disregard the "Multiple" category in our description of the results.

Failure Categories. The failure category refers to the observable consequence of the fault. For each bug report, we marked the failure category as either *Code-terminating* or *Output-related*, as defined in Section 2.2. This categorization helps us answer RQ2.

Impact Categories. We manually classify the impact of a JavaScript bug according to the classification scheme used by Bugzilla.[3] This scheme is applicable to any software application, and has also been used in other studies [11], [12]. Table 2 shows the categories. This categorization helps us answer RQ2.

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

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

Time for Fixing. To answer RQ5, we define the ***triage time*** as the time it took a bug to get assigned to a developer, from the time it was reported (or, if there is no "assigned" marking, the time until the first comment is posted in the report). We also define ***fix time*** as the time it took the corresponding JavaScript fault to get marked as "fixed", from the time it was triaged. We recorded the time taken for each JavaScript bug report to be triaged, and for the report to be fixed. Other studies have classified bugs on a similar basis [13], [14]. Further, we calculate times based on the calendar date; hence, if a bug report was triaged on the same date as it was reported, the triage time is recorded as 0.

Type Faults. Programming languages such as TypeScript [3] and Dart [4] aim to minimize JavaScript faults through a stricter typing system for JavaScript; hence, the main fault model targeted by these languages are type faults. In our work, we assess the usefulness of strong typing in such languages by examining the prevalence of type faults among the bug reports, which addresses RQ6.

In our analysis, we consider four different "type categories", listed below.

- **Primitive types [P]**: Values of type `string`, `boolean`, or `number`
- **Null/undefined types [Nu]**: `null` or `undefined`
- **Native "class" types [Nc]**: Objects native to client-side JavaScript (e.g., `Function`, `Element`, etc.)
- **Custom "class" types [C]**: User-defined objects

We first categorize a bug report as either a *type fault* or *not a type fault*, based on the definition provided in Section 2.2. For each bug report categorized as a type fault, we further categorize it as belonging to one of 16 subcategories, each of the form, "*<type category> expected, but <type category> actual*," which, for simplicity, we abbreviate as *<type category>E<type category>A*. For example, a type fault belongs to the *PEPA* category if a value is expected to be of a certain primitive type in the JavaScript code – say `boolean` – but its actual type at runtime is a different primitive type – say `string`. Similarly, an *NcENuA* type fault occurs if a value is expected to be of a native "class" type, but its actual type at runtime is `null` or `undefined`.

Note that type comparisons are made in a way similar to Pradel et al.'s method for detecting inconsistent types [6]. Finally, note that we classify a bug report as "not a type fault" if we do not find a statement $\mathcal{L}$ that satisfies the definition given in Section 2.2.

Temporal Analysis. Where appropriate, for every data point we collect to answer the first six research questions, we analyze how these data have changed over time. The purpose of this analysis – which answers RQ7 – is to help us understand and speculate how historical factors such as browser improvements, the appearance of new JavaScript frameworks, and the rise in popularity of "Q&A" websites (e.g., StackOverflow), among others, have helped in the improvement of the reliability of client-side JavaScript, or degraded it.

To perform this temporal analysis, we mark each bug report with the year in which it was reported. In our specific case, the bug reports we collected were reported over a period of 13 calendar years, from 2003 to 2015. However, we neglect the years 2003 and 2015 in our analysis, since fewer than 3 bug reports were marked with each of these years; hence, we only consider 11 calendar years in our analysis (i.e., 2004 to 2014), each of which corresponds to at least

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

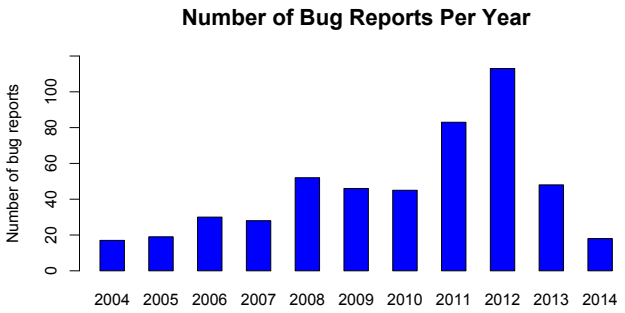


Fig. 2: Bug reports per calendar year. Note that there was one additional bug report from 2003, and two additional bug reports from 2015.

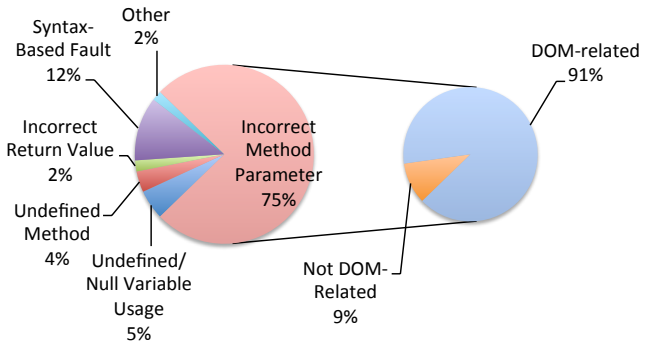


Fig. 3: Pie chart of the distribution of fault categories.

17 bug reports. The number of bug reports in each of these calendar years is shown in Figure 2.

4 Results

In this section, we present the results of our empirical study on JavaScript bug reports. The subsections are organized according to the research questions in Section 3.1.

4.1 Fault Categories

Table 3 shows the breakdown of the fault categories in our experimental objects. The pie chart in Figure 3 shows the overall percentages. As seen from the table and the figure, approximately 75% of JavaScript faults belong to the "Incorrect Method Parameter" category. This suggests that most JavaScript faults result from errors related to setting up the parameters of native JavaScript methods, or the values assigned to native JavaScript properties.

***Finding 1**: "Incorrect Method Parameter" faults account for around 75% of JavaScript faults.*

In our earlier studies of JavaScript console messages [1] and fault-localization of JavaScript bugs [15], we also noticed many "Incorrect Method Parameter" faults, but their prevalence was not quantified. Interestingly, we also observed in these earlier studies that many of the methods and properties affected by these faults are DOM methods/properties – in other words, DOM-related faults, as defined in Section 2. Based on these prior observations, we became curious as to how many of these "Incorrect Method Parameter" faults are DOM-related.

We further classified the "Incorrect Method Parameter" faults based on the methods/properties in which the incorrect values propagated, and found that 91% of these faults are DOM-related faults. This indicates that among *all* JavaScript faults, approximately 68% are DOM-related faults (see right-most pie chart in Figure 3). We find that DOM-related faults range from 50 to 100% of the total JavaScript faults across applications, as seen on the last column of Table 3.

Lastly, in order to assess how many of the DOM-related faults result from developers' erroneous understanding of the DOM, we make a distinction between *strong DOM-related faults* and *weak DOM-related faults*. A DOM-related fault is classified as *strong* if the incorrect parameter passed to the DOM method/property represents an inconsistency with the DOM; this includes, for example, cases where an incorrect or non-existent selector/ID is passed to a DOM method. Otherwise, the DOM-related fault is classified as weak; this includes, for example, cases where the wrong text value is assigned to `innerHTML`, or the wrong attribute value is assigned to an attribute. Overall, we find that strong DOM-related faults make up 88% of all DOM-related faults. This result therefore strongly indicates that *most DOM-related faults occur as a result of an inconsistency between what the developers think is the DOM contents, and what actually is the DOM's contents.*

***Finding 2**: DOM-related faults account for 91% of "Incorrect Method Parameter" faults. Hence, the majority – around 68% – of JavaScript faults are DOM-related, and the majority – around 88% – of these DOM-related faults are of the "strong" variety.*

Temporal Analysis. Figure 4a shows a scatter plot of the percentage of bug reports marked as DOM-related per calendar year. The linear regression line has a downward slope, indicating an overall decrease in the percentage of DOM-related faults over the years. However, this decrease is very small, i.e., a decrease of 7 percentage points, which corresponds to a 10% decrease. Hence, the percentage of DOM-related faults reported in the repositories we analyzed has remained rather consistent over the years.

***Finding 3**: The percentage of DOM-related faults among all JavaScript faults has experienced only a small decrease (10%) over the past ten years.*

4.2 Consequences of JavaScript Faults

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

TABLE 3: Fault categories of the bug reports analyzed. Library data are shown in italics.

Application	Undefined/Null Variable Usage	Undefined Method	Incorrect Return Value	Syntax-Based Fault	Other	Incorrect Method Parameter			Percent DOM-related
						DOM-related	Not DOM-related	Total	
Moodle	3	3	0	7	0	15	2	17	50%
Joomla	1	0	0	3	0	6	1	7	55%
WordPress	1	2	0	3	1	21	2	23	70%
Drupal	0	1	0	5	0	23	1	24	77%
Roundcube	3	0	0	4	0	22	1	23	73%
WikiMedia	2	4	0	5	0	15	4	19	50%
TYPO3	2	2	0	7	1	18	0	18	60%
TaskFreak	1	0	0	0	0	4	1	5	67%
Horde	1	0	0	4	0	24	1	25	80%
FluxBB	0	0	0	0	0	5	0	5	100%
LimeSurvey	0	0	0	4	0	24	2	26	80%
DokuWiki	2	0	0	3	2	20	3	23	67%
phpBB	3	1	0	3	0	23	0	23	77%
MODx	1	1	0	3	1	21	3	24	70%
EZ Systems	2	2	0	7	1	17	1	18	57%
jQuery	*0*	*0*	*1*	*0*	*0*	*26*	*3*	*29*	*87%*
Prototype.js	*0*	*1*	*2*	*0*	*0*	*22*	*5*	*27*	*73%*
MooTools	*3*	*1*	*3*	*0*	*1*	*19*	*3*	*22*	*63%*
Ember.js	*2*	*1*	*4*	*0*	*2*	*16*	*5*	*21*	*53%*
Overall	27	19	10	58	9	341	38	379	68%

Failure Categories. Table 4 shows the distribution of failure categories amongst the collected reports; all faults are classified as either leading to a code-terminating failure or an output-related failure (these terms are defined in Section 3.4). Note that the goal of this classification is *not* to assess the severity of the bugs, but rather, to determine the nature of the failure (i.e., whether there is a corresponding error message or not). Making this distinction will be helpful for developers of fault localization tools, for example, as these error messages can naturally act as a starting point for analysis of the bug.

As the table shows, around 57% of JavaScript faults are code-terminating, which means that in these cases, an exception is thrown. Faults that lead to code-termination are generally easier to detect, since the exceptions have one or more corresponding JavaScript error message(s) (provided the error can be reproduced during testing). On the other hand, output-related failures do not have such messages; they are typically only detected if the user observes an abnormality in the behaviour of the application.

Since the majority of JavaScript faults are DOM-related, we explored how these failure categories apply to DOM-related faults. Interestingly, we found that for DOM-related faults, most failures are output-related (at 57%), while for non-DOM-related faults, most failures are code-terminating (at 86%). This result suggests that DOM-related faults may be more difficult to detect than non-DOM-related faults, as most of them do not lead to exceptions or error messages.

***Finding 4**: While most non-DOM-related JavaScript faults lead to exceptions (around 86%), only a relatively small percentage (43%) of DOM-related faults lead to such exceptions.*

TABLE 4: Number of code-terminating failures compared to output-related failures. Library data are shown in italics. Data for DOM-related faults only are shown in parentheses.

Application	Code-terminating	Output-related
Moodle	21 (8)	9 (7)
Joomla	8 (3)	3 (3)
WordPress	11 (3)	19 (18)
Drupal	12 (5)	18 (18)
Roundcube	18 (11)	12 (11)
WikiMedia	19 (4)	11 (11)
TYPO3	21 (9)	9 (9)
TaskFreak	3 (2)	3 (2)
Horde	17 (11)	13 (13)
FluxBB	3 (3)	2 (2)
LimeSurvey	11 (7)	19 (17)
DokuWiki	9 (4)	21 (16)
phpBB	19 (12)	11 (11)
MODx	21 (14)	9 (7)
EZ Systems	27 (15)	3 (2)
jQuery	*17 (13)*	*13 (13)*
Prototype.js	*10 (7)*	*20 (15)*
MooTools	*21 (10)*	*9 (9)*
Ember.js	*16 (5)*	*14 (11)*
Overall	284 (146)	218 (195)

Temporal Analysis. A scatter plot of the percentage of code-terminating failures per year is shown in Figure 4b. This figure shows the percentage of code-terminating failures, over time, for each fault category (i.e., DOM-related vs non-DOM-related). In both cases, there is a net decrease in the number of code-terminating failures, with a slightly larger decrease for non-DOM-related faults. The overall decrease in code-terminating failures may be explained in part by the improvements in error consoles as well as the introduction of tools such as Firebug,[4] both of which facilitate the process of debugging code-terminating failures within the web browser.

***Finding 5**: The percentage of code-terminating failures experienced a net decrease for both DOM-related faults (17% decrease) and non-DOM-related faults (21% decrease) over the past ten years.*

Impact Categories. The impact indicates the severity of the failure. Hence, we also classify bug reports based on impact categories as defined in Section 3.4 (i.e., Type 1 has lowest severity, and Type 5 has highest severity).

4. http://getfirebug.com/

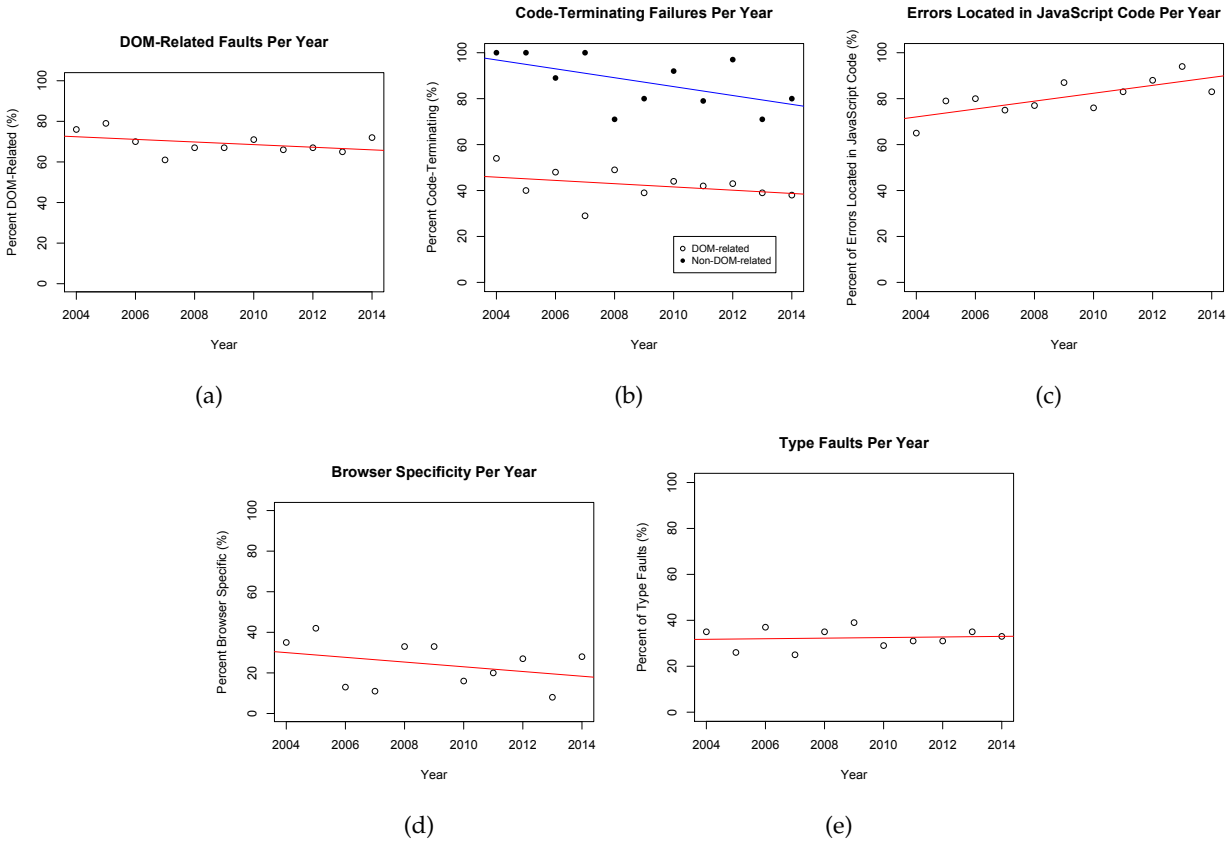


Fig. 4: Temporal graphs showing (a) the percent of DOM-related faults per year; (b) the percent of code-terminating failures per year. The red regression line represents DOM-related faults, while the blue regression line represents non-DOM-related faults; (c) the percent of bug reports whose error is located in the JavaScript code, per year; (d) the percent of browser-specific bugs per year; and (e) the percent of type faults per year

The impact category distribution for each web application and library is shown in Table 5. Most of the bug reports were classified as having Type 3 impact (i.e., some functionality loss). Type 1 and Type 5 impact faults are the fewest, with 53 and 34 bug reports, respectively. Finally, Type 2 and Type 4 impact faults are represented by 123 and 60 bug reports, respectively. The average impact of the collected JavaScript bug reports is close to the middle, at 2.80, which is in line with other studies [16].

Table 5 also shows the impact distribution for DOM-related faults in parentheses. As seen in the table, each impact category is comprised primarily of DOM-related faults. Further, almost 80% (27 out of 34) of the highest severity faults (i.e., Type 5 faults) are DOM-related. Additionally, 13 of the 19 experimental subjects contain at least one DOM-related fault with Type 5 impact. This result suggests that *high severity failures often result from DOM-related faults*. We find that these high-impact faults broadly fall into three categories.

1) **Application/library becomes unusable**. This occurs because an erroneous feature is preventing the user from using the rest of the application, particularly in DOM-related faults, which make up 11 of the 15 faults in this category. For example, one of the faults in Drupal prevented users from logging in (due to incorrect attribute values assigned to the username and password elements), so the application could not even be accessed.
2) **Data loss**. Once again, this is particularly true for DOM-related faults, which account for 13 out of the 14 data-loss-causing faults that we encountered. One example

TABLE 5: Impact categories of the bug reports analyzed. Library data are shown in italics. Impact categories data for DOM-related faults only are shown in parentheses.

Application	Type 1	Type 2	Type 3	Type 4	Type 5
Moodle	10 (5)	12 (5)	0 (0)	6 (3)	2 (2)
Joomla	2 (2)	2 (0)	4 (2)	2 (1)	1 (1)
WordPress	4 (4)	7 (3)	12 (9)	3 (2)	4 (3)
Drupal	3 (3)	2 (1)	17 (12)	1 (1)	7 (6)
Roundcube	2 (2)	5 (4)	14 (9)	5 (3)	4 (4)
WikiMedia	2 (1)	8 (6)	15 (6)	1 (0)	4 (2)
TYPO3	0 (0)	4 (2)	20 (13)	5 (2)	1 (1)
TaskFreak	2 (1)	1 (1)	1 (0)	2 (2)	0 (0)
Horde	6 (3)	7 (6)	13 (11)	2 (2)	2 (2)
FluxBB	1 (1)	1 (1)	2 (2)	1 (1)	0
LimeSurvey	5 (4)	4 (2)	19 (16)	1 (1)	1 (1)
DokuWiki	5 (3)	7 (3)	16 (13)	2 (1)	0 (0)
phpBB	3 (0)	5 (4)	16 (14)	6 (5)	0 (0)
MODx	2 (1)	5 (3)	18 (14)	3 (2)	2 (1)
EZ Systems	1 (1)	2 (0)	25 (15)	2 (1)	0 (0)
jQuery	*3 (3)*	*13 (13)*	*1 (1)*	*11 (8)*	*2 (1)*
Prototype.js	*0 (0)*	*7 (6)*	*19 (12)*	*2 (2)*	*2 (2)*
MooTools	*0 (0)*	*16 (8)*	*10 (8)*	*3 (3)*	*1 (0)*
Ember.js	*2 (0)*	*15 (10)*	*10 (4)*	*2 (1)*	*1 (1)*
Overall	53 (34)	123 (78)	232 (161)	60 (41)	34 (27)

TABLE 6: Error locations of the bug reports analyzed. Library data are shown in italics.

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

Application	JS	HTML	SSC	SCF	OTH	MEL
Moodle	22	2	6	0	0	0
Joomla	9	0	1	0	0	1
WordPress	24	0	6	0	0	0
Drupal	29	0	1	0	0	0
Roundcube	26	0	4	0	0	0
WikiMedia	25	0	5	0	0	0
TYPO3	18	1	9	2	0	0
TaskFreak	6	0	0	0	0	0
Horde	22	1	4	2	1	0
FluxBB	5	0	0	0	0	0
LimeSurvey	25	1	3	0	1	0
DokuWiki	26	1	2	0	1	0
phpBB	23	5	0	1	1	0
MODx	23	1	6	0	0	0
EZ Systems	19	5	6	0	0	0
jQuery	*30*	–	–	–	*0*	*0*
Prototype.js	*25*	–	–	–	*4*	*1*
MooTools	*30*	–	–	–	*0*	*0*
Ember.js	*30*	–	–	–	*0*	*0*
Overall	417	17	53	5	8	2

comes from Roundcube; in one of the bug reports, the fault causes an empty e-mail to be sent, which causes the e-mail written by the user to be lost. As another example, a fault in WordPress causes server data (containing posts) to be deleted automatically without confirmation.

3) **Browser hangs and information leakage**. Hangs often occur as a result of a bug in the browser; the Type 5 faults leading to browser hangs that we encountered are all browser-specific. Information leakage occurred twice – in TYPO3 and MODx – as a result of JavaScript faults that caused potentially security-sensitive code from the server to be displayed on the page; one of these bugs leading to information leakage is DOM-related.

***Finding 6**: About 80% of the highest severity JavaScript faults are DOM-related.*

4.3 Causes of JavaScript Faults

Locations. Before we can determine the causes, we first need to know *where* the programmers committed the programming errors. To this end, we marked the error locations of each bug report; the error location categories are listed in Section 3.4. The results are shown in Table 6. As the results show, the vast majority (83%) of the JavaScript faults occur as a result of programming errors in the JavaScript code itself. When only DOM-related faults were considered, a similar distribution of fault locations was observed; in fact, the majority is even larger for DOM-related faults that originated from the JavaScript code, at 89%. Although JavaScript code could be automatically written by external tools, we observed that the fix for these bugs involved the *manual* modification of the JavaScript file(s) where the error is located. This observation provides a good indication that JavaScript faults typically occur because the programmer herself writes erroneous code, as opposed to server-side code automatically generating erroneous JavaScript code, or HTML.

***Finding 7**: Most JavaScript faults (83%) originate from manually-written JavaScript code as opposed to code automatically generated by the server.*

Patterns. To understand the programmer mistakes associated with JavaScript errors, we manually examined the bug reports for errors committed in JavaScript code (which were the dominant category). We found that 55% of the errors fell into the following common patterns (the remaining 45% of the errors followed miscellaneous patterns):

1) **Erroneous input validation**. Around 16% of the bugs occurred because inputs passed to the JavaScript code (i.e., user input from the DOM or inputs to JavaScript functions) are not being validated or sanitized. The most common mistake made by programmers in this case is neglecting valid input cases. For example, in the jQuery library, the `replaceWith()` method is allowed to take an empty string as input; however, the implementation of this method does not take this possibility into account, thereby causing the call to be ignored.
2) **Error in writing a string literal**. Approximately 13% of the bugs were caused by a mistake in writing a string literal in the JavaScript code. These include forgetting prefixes and/or suffixes, typographical errors, and including wrong character encodings. About half of these errors relate to writing a syntactically valid but incorrect CSS selector (which is used to retrieve DOM elements) or regular expression.
3) **Forgetting null/undefined check**. Around 10% of the bugs resulted from missing null/undefined checks for a particular variable, assuming that the variable is *allowed* to have a value of `null` or `undefined`.
4) **Neglecting differences in browser behaviour**. Around 9% of the bugs were caused by differences in how browsers treat certain methods, properties or operators in JavaScript. Of these, around 60% pertain to differences in how browsers implement native JavaScript methods. For example, a fault occurred in WikiMedia in Internet Explorer 7 and 8 because of the different way those browsers expect the `history.go()` method to be used.
5) **Error in syntax**. Interestingly, around 7% of bugs resulted from syntax errors in the JavaScript code that were made by the programmer. Note, also, that we found instances where server-side code generated syntactically incorrect JavaScript code, though this is not accounted for here.

***Finding 8**: There are several recurring error patterns – causing JavaScript faults – that arise from JavaScript code.*

Temporal Analysis. When plotted per year, the percentage of bug reports whose corresponding error is located in the JavaScript code results in a regression line that has a positive slope, as seen in Figure 4c. In other words, the percentage

TABLE 7: Browser specificity of the bug reports analyzed. Library data are shown in italics.

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

Application	IE	FF	CHR	SAF	OPE	OTH	NBS	MUL
Moodle	4	0	0	0	0	0	25	1
Joomla	1	0	0	0	0	0	10	0
WordPress	1	0	0	0	0	1	28	0
Drupal	2	0	1	1	0	0	26	0
Roundcube	5	0	1	0	1	1	22	0
WikiMedia	6	0	0	0	0	0	24	0
TYPO3	7	1	0	0	1	0	20	1
TaskFreak	1	0	0	1	0	0	4	0
Horde	8	1	0	3	0	0	18	0
FluxBB	0	0	0	0	0	0	5	0
LimeSurvey	5	1	0	0	0	0	24	0
DokuWiki	2	1	2	0	1	0	23	1
phpBB	2	2	0	1	2	1	22	0
MODx	3	0	0	1	0	0	26	0
EZ Systems	1	0	1	0	0	0	27	1
jQuery	*7*	*0*	*0*	*0*	*0*	*0*	*22*	*1*
Prototype.js	*8*	*1*	*1*	*2*	*1*	*0*	*14*	*3*
MooTools	*10*	*2*	*0*	*0*	*1*	*0*	*17*	*0*
Ember.js	*2*	*0*	*0*	*0*	*0*	*0*	*28*	*0*
Overall	75	9	6	9	7	3	385	8

of bug reports whose errors are located in the JavaScript code generally increased from the year 2004 to 2014; in this case, based on the endpoints of the regression line, there was an increase of around 25%. We believe this trend is a product of client-side scripting gaining more prominence in web application development as the years went by. In particular, during this time period, new and richer web standards for ECMAScript [17] and XMLHttpRequest [18] were being introduced, giving way for the rise in popularity of Ajax, which developers could use to offload server-side functionality to the client-side. As a result, since JavaScript is being used more frequently, more errors are being committed in JavaScript code. In addition, the overall increase in the trend may also be attributable to JavaScript code becoming more complex as JavaScript gradually rose in popularity over time.

***Finding 9**: Among JavaScript bugs, the percentage of errors committed in the JavaScript code has experienced a 25% increase over the past ten years.*

4.4 Browser Specificity

We analyzed the browser specificity of the bug reports we collected. A bug is browser specific if it occurs in at most two web browsers. As Table 7 shows, most JavaScript faults (77%) are non-browser specific (the same percentage is acquired when only DOM-related faults are considered). However, among the browser-specific faults, about 64% are specific to Internet Explorer (IE).

After analyzing the IE-specific faults, we found that most of them (56%) were due to the use of methods and properties that were not supported in that browser (particularly in earlier versions, pre-Internet Explorer 8). This is likely because the use of browser-specific method and property names (which may not be standards-compliant) is more prevalent in IE than in other browsers. In addition, IE has low tolerance of small errors in the JavaScript code. For example, 21% of the IE-specific faults occurred because IE could not handle trailing commas in object-creation code; while these trailing commas are syntax errors as per the ECMAScript standard, other browsers can detect their presence and remove them.

***Finding 10**: Most JavaScript faults (77%) are not browser-specific.*

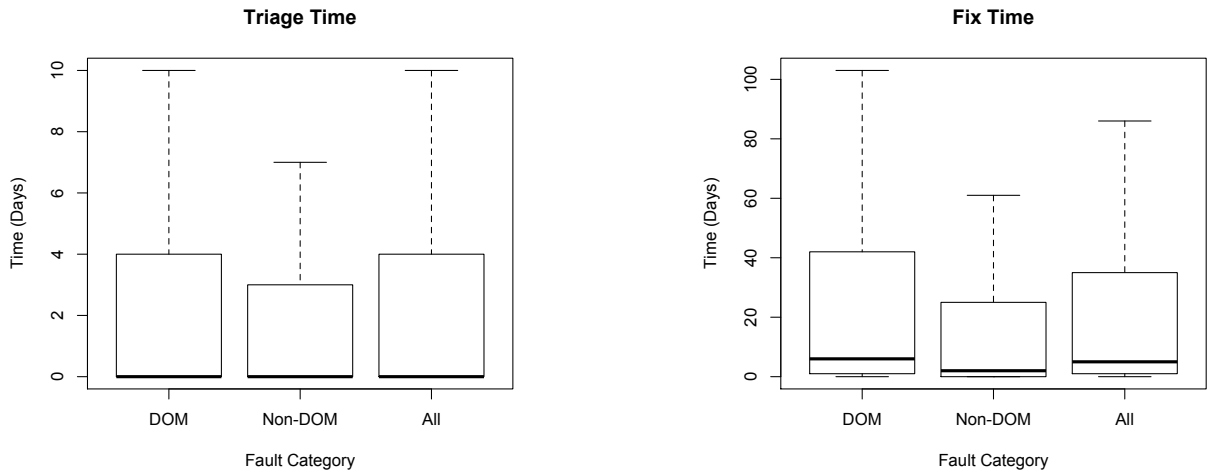
Temporal Analysis. Figure 4d shows the scatter plot for the percentage of browser specific bug reports per year. Here, the regression line has a negative slope; more specifically, the regression line shows the browser specificity decreasing by around 50% (i.e., from 32% in 2004 to 17% in 2014). This decrease in browser specificity is consistent with results found by three of the authors in a recent study, which show an overall decrease in cross-browser-related questions in StackOverflow from 2009 to 2012 [19]; this work posits that the decrease may have been caused by the maturation of browser support for HTML5, which was the focus of the study. Other factors that may have contributed to this decline include the introduction of JavaScript libraries designed to eliminate cross-browser incompatibilities, some of which are used in the web applications we studied, as well as extensive research done recently on ways to mitigate cross-browser compatibility issues [20], [21], [22], [23].

***Finding 11**: The percentage of browser-specific faults among all JavaScript faults has experienced a 50% decrease over the past ten years.*

4.5 Triage and Fix Time for JavaScript Faults

We calculated the triage time and fix time for each bug report. The results are shown in Figure 5a (triage time) and Figure 5b (fix time) as box plots (the outliers are not shown). Most of the bug reports were triaged on the same day they were reported, which explains why the median of the triage time for all bug reports, as seen in Figure 5a, is 0. Also, as seen in Figure 5b, the median of the fix time for all bug reports is 5 days. Finally, in addition to the box plots, we calculated the mean triage and fix times for each bug report. We found that, on average, the triage time for JavaScript faults is approximately 29 days, while the average fix time is approximately 65 days (see Table 8).

As before, we made the same calculations for DOM-related faults and non-DOM-related faults. The comparisons are shown in Figures 5a and 5b, as well as in Table 8. From the box plots, we find that both DOM-related faults and non-DOM-related faults have a median triage time of 0 days, which indicates that the majority of either of these faults gets triaged on the same day as they are reported. For the fix times, DOM-related faults have a median fix time of 6 days, compared to 2 days for non-DOM-related faults. With respect to the means (Table 8), we found that DOM-related faults have an average triage time of 26 days, compared to 37 days for non-DOM-related faults. On the other hand,



(a) (b)

Fig. 5: Box plot for (a) triage time, and (b) fix time

TABLE 8: Average triage times (T) and fix times (F) for each experimental object, rounded to the nearest whole number. Library data are shown in italics.

Application	All Faults		DOM-Related Faults Only		Non-DOM-Related Faults Only	
	T	F	T	F	T	F
Moodle	248	10	205	12	292	8
Joomla	4	57	1	67	8	46
WordPress	1	138	2	150	0	108
Drupal	7	66	3	47	22	130
Roundcube	18	118	25	160	0	9
WikiMedia	18	26	36	44	1	8
TYPO3	7	55	7	67	6	36
TaskFreak	23	17	32	23	6	5
Horde	4	7	5	8	0	1
FluxBB	1	5	1	5	–	–
LimeSurvey	10	47	5	58	28	5
DokuWiki	11	13	10	15	13	9
phpBB	3	60	3	64	4	46
MODx	86	46	116	64	16	4
EZ Systems	35	41	22	41	52	42
jQuery	*1*	*33*	*1*	*36*	*1*	*10*
Prototype.js	*28*	*343*	*33*	*294*	*14*	*478*
MooTools	*10*	*48*	*10*	*49*	*9*	*47*
Ember.js	*0*	*11*	*1*	*14*	*0*	*8*
Overall	29	65	26	71	37	52

DOM-related faults have an average fix time of 71 days, compared to 52 days for non-DOM-related faults.

Taking the above results into account, it appears that DOM-related faults generally have lower triage times than non-DOM-related faults, while DOM-related faults have higher fix times than non-DOM-related faults. This suggests that developers find DOM-related faults important enough to be triaged more promptly than non-DOM-related faults. However, DOM-related faults take longer to fix, perhaps because of their inherent complexity.

***Finding 12**: On average, DOM-related faults get triaged more promptly than non-DOM-related faults (26 days vs. 37 days); however, DOM-related faults take longer to fix than non-DOM-related faults (71 days vs. 52 days).*

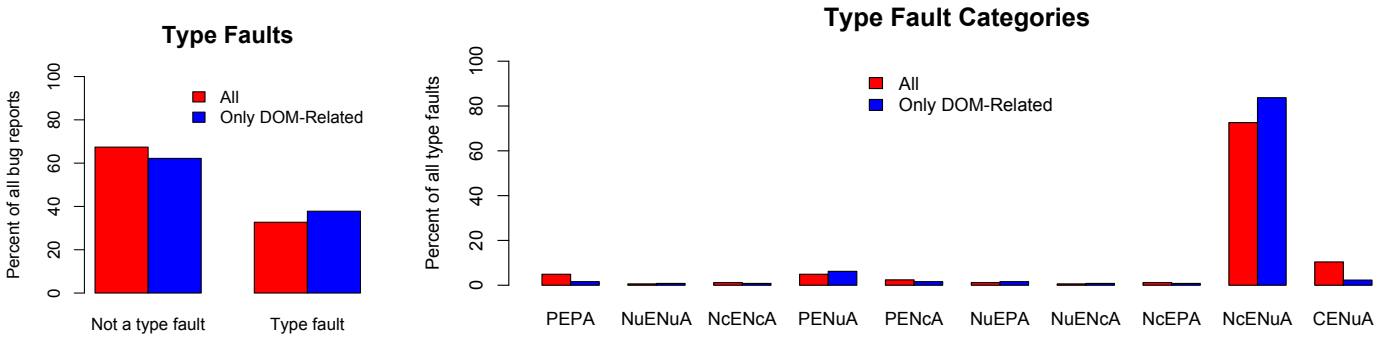
4.6 Prevalence of Type Faults

We now discuss our findings regarding the prevalence of type faults in the bug reports that we analyzed. As discussed in Section 3.4, each type fault category is identified as "__E__A", where the first blank is represented by the abbreviation of the expected type category, and the second blank is represented by the abbreviation of the actual type category.

The results are shown in Figure 6. Overall, as seen in Figure 6a, only about 33% of the bug reports in our study were classified as type faults. Further, of all the type faults, 72% belong to the *NcENuA* category, in which a native "class" type is expected, but the actual type at runtime is `null` or `undefined` (see Figure 6b). These results show that in the subject systems, the vast majority of the bugs are *not* type faults; thus, programming languages introducing stronger typing systems to JavaScript, such as TypeScript and Dart, as well as type checkers, may not eliminate most JavaScript faults. It is worth noting, however, that these languages have other advantages apart from strong typing, including program comprehension and support for class-based object-oriented programming.

We also studied the DOM-related faults, to determine how many of them are type faults. The results for DOM-related faults are also shown in Figure 6. Overall, we found that 38% of DOM-related faults are also type faults. Most of these type faults are also of the *NcENuA* variety; in this case, the expected native "class" type is, for the most part, `Element`. This finding suggests that stronger typing systems and type checkers may also not eliminate most DOM-related faults.

We also analyzed the severity of type faults, based on the impact types assigned to each bug report. We found that out of all the Type 4 and Type 5 impact bug reports, which are the highest severity bugs, about 30% are type faults; considering Type 5 impact bug reports alone, about 18% are type faults. Therefore, based on our results, these languages have limited ability in eliminating the majority of the *highest impact* JavaScript bugs in web applications.



(a) (b)

Fig. 6: Bar graphs showing (a) the percentage of bug reports that are type faults versus the percentage of bug reports that are not type faults and (b) the distribution of type fault categories

***Finding 13**: The majority (67%) of JavaScript faults are not type faults, and the majority (62%) of DOM-related faults are also not type faults. In addition, the majority of Type 4 and Type 5 impact bugs are not type faults, at 70%.*

Temporal Analysis. As seen in Figure 4e, the regression line for the percentage of type faults is relatively flat (with a slight 3% increase from 2004 to 2014). Therefore, despite all the technology that has been developed to eliminate these type faults, the percentage of type faults has remained more or less constant over the years. This may be caused by the fact that most of the type faults we found in our study belong to the *NcENuA* category, where a native "class" type is expected, but the actual type at runtime is `null` or `undefined`. In particular, current type checkers normally cannot predict if the return value of a native JavaScript method is `null` or `undefined`; for example, they cannot predict if the return value of `getElementById()` is `null` without looking at the DOM – which almost none of them do – so that particular type fault will be missed.

***Finding 14**: The percentage of type faults among all JavaScript faults has remained constant (with only a 3% increase) over the past ten years.*

4.7 Threats to Validity

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

Another internal threat is in our analysis of type faults, in which we assumed that a bug report does not correspond to a type fault if the existence of a statement $\mathcal{L}$ from the definition given in Section 2.2 could not be established, based on our qualitative reading of the bug report. Hence, the percentage of type faults we presented in Section 4.6 is technically a lower bound on the actual number of type faults. Nonetheless, the lack of any indication in a bug report that a statement $\mathcal{L}$ exists strongly suggests that inconsistencies in types are not an issue with the bug.

In terms of external threats, our results are based on bug reports from a limited number of experimental subjects, from a limited time duration of ten years, which calls into question the representativeness; unfortunately, public bug repositories for web applications are not abundant, as previously mentioned. We mitigated this by choosing web applications that are used for different purposes, including content management, webmail, and wiki. Further, prior to 2004, few websites used JavaScript, since Ajax programming – which is often credited for popularizing JavaScript – did not become widespread until around 2005 when Google Maps was introduced [18].

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

For the triage and fix times, we did not account for possible delays in marking a bug report as "assigned" or "fixed", which may skew the results. In addition, the triage time is computed as the time until the first developer comment, when there is no "assigned" marking; although we find this approximation reasonable, the developer may not have started fixing until some days after the first comment was posted. Finally, the triage and fix times may be influenced by external factors apart from the complexity of the bugs (e.g., bug replication, vacations, etc.). These are likewise construct validity threats. Nonetheless, note that other studies

have similarly used time to estimate the difficulty of a fix, including Weiss et al. [24] and Kim and Whitehead [25]. In the former, the authors point out that the time it takes to fix a bug is indicative of the effort required to fix it; the difference between our estimation and theirs is that they use the fix time *reported* by developers assigned to the bug, which is unavailable in the bug repositories we studied.

5 Discussion

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

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

These first two findings also impact web application testers, as they reveal certain categories of JavaScript faults that users consider important enough to report, and hence, that testers should focus on. Currently, one of the most popular ways to test JavaScript code is through unit testing, in which modules are tested individually; when creating these unit tests, a mock DOM object is usually needed, in order to allow the DOM API method calls present in the module to function properly. While useful, this approach often does not take into account the changing states of the DOM when users are interacting with the web application in real settings, because testers often create these mock DOM objects simply to prevent the calls from failing. Finding 2, in contrast, suggests that testers need to be more "DOM-aware", in that they need to take extra care in ensuring that these mock objects emulate the actual DOM as closely as possible.

In addition to unit testing, web application testers also perform end-to-end (E2E) testing; here, user actions (e.g., clicks, hovers, etc.) are automatically applied to individual webpages to assert that certain conditions about the DOM are satisfied after carrying out these user actions. E2E testing can help detect DOM-related faults; however, the problem is that these tests often require the tester to know certain properties about the DOM in order to set up the user actions in the tests. As mentioned above, keeping track of these properties of the DOM is difficult to do, judging by the large percentage of strong DOM-related faults we observed in our study. This makes E2E tests themselves susceptible to DOM-related faults if written manually, which motivates the potential usefulness of automated tools for writing these tests, including record-replay and webpage crawling techniques.

With regards to Findings 4, 6, and 12, these results suggest that web application testers should also prioritize *emulating* DOM-related faults, as most high-impact faults belong to this category. One possible way to do this is to prioritize the creation of tests that cover DOM interaction points in the JavaScript code. By doing so, testers can immediately find most of the high-impact faults. This early detection is useful because, as Finding 4 suggests, DOM-related faults often have no accompanying error messages and can be more difficult to detect. Further, as Finding 12 suggests, DOM-related faults take longer to fix on average compared to non-DOM-related faults.

As mentioned previously, the presence of error messages in JavaScript bugs can be useful, as these messages can provide a natural starting point for analysis of these bugs. Indeed, our fault localization tool AutoFLox [26] uses error messages to automatically infer the line of code where the failure takes place – which, in this case, is the same as the exception point – as well as to determine the backward slice of the `null` or `undefined` values that led to the exception. However, as Finding 4 suggests, the majority of DOM-related faults do *not* lead to exceptions and hence, do not have accompanying error messages. This points to the need to devise alternative ways to automatically determine the line of code where the failure takes place when performing fault localization. One possibility is to give developers the ability to select, on the webpage itself, any DOM element that is observed to be incorrect. A static analyzer can then try to guess which lines of JavaScript code were the latest ones to update the element; these lines will therefore be the starting point for localization.

As for Findings 7 and 8, these results can be useful for developers of static analysis tools for JavaScript. Many of the current static analysis tools only address syntactic issues with the JavaScript code (e.g., JSLint,[5] Closure Compiler,[6] JSure[7]), which is useful since a few JavaScript faults occur as a result of syntax errors, as described in Section 4.3. However, the majority of JavaScript faults occur because of errors in semantics or logic. Some developers have already started looking into building static semantics checkers for JavaScript, including TAJS [27], which is a JavaScript type analyzer. However, the programming mistakes we encountered in the bug reports (e.g., erroneous input validations, erroneous CSS selectors, etc.) call for more powerful tools to improve JavaScript reliability.

The recurring patterns to which Finding 8 refers can be a starting point for devising a taxonomy for common JavaScript errors. This taxonomy can be helpful in two ways. First, it can facilitate the code review process, as the taxonomy helps web developers identify certain "hot spots"

5. http://www.jslint.com
6. http://code.google.com/closure/compiler/
7. https://github.com/berke/jsure

in the code that have historically been susceptible to error. In addition, tools such as FindBugs[8] use a taxonomy of common coding patterns and smells to automatically detect errors in Java code using static analysis; in the same vein, a taxonomy for JavaScript errors can help achieve the same automatic error detection task for JavaScript code.

While Finding 10 suggests that most JavaScript faults are non-browser specific, we did find a few (mostly IE-specific) faults that are browser-specific. Hence, it is useful to design JavaScript development tools that recognize cross-browser differences and alerts the programmer whenever she forgets to account for these. Some Integrated Development Environments (IDEs) for JavaScript have already implemented this feature, including NetBeans[9] and Aptana.[10]

Finding 13 shows that there is a significant number, though a minority, of type faults encountered in the subject systems, some of which have high impact. This provides motivation for the development of the strongly-typed languages mentioned earlier, as well as type checkers [27], [28], [29]. However, such tools and languages are far from being a panacea, as the vast majority of JavaScript faults are not type faults, according to our study. Therefore, tool developers should not focus exclusively on type checking when looking for ways to improve the reliability of JavaScript code because type checking, while useful, does not suffice in detecting or eliminating most JavaScript faults.

According to Findings 5 and 11, there was a significant decrease in the number of code-terminating failures and browser specific faults; this suggests that developers of browsers and browser-based tools are heading towards the right direction in terms of facilitating the debugging process for JavaScript faults and ensuring cross-browser compliance of JavaScript code. However, from Findings 3 and 14, we observed that the number of DOM-related faults and type faults has remained relatively constant from 2004 to 2014; hence, developers, testers, and tool designers must pay more careful attention towards these two kinds of faults, especially DOM-related faults, as they constitute almost 70% of all JavaScript faults.

Finding 9 shows that the percentage of errors located in the JavaScript code has increased from 2004 to 2014. This suggests that tools for improving the client-side reliability should consider performing an analysis of the client-side code itself – which is where the majority of JavaScript bugs arise as per Findings 7 and 8 – instead of simply looking at how server-side code generates malformed client-side code, as other tools have done [30], [31].

Finally, recall that this study focuses on *client-side* JavaScript; hence, the results may not directly be applicable to JavaScript developers at the server-side who use Node.js. For example, there is no DOM present at the server-side, which indicates that DOM-related faults will not be present there. Nonetheless, many of the error and fault patterns we found are not client-side-specific. A recent paper by Hanam et al. [32], for instance, sheds light on some of the pervasive JavaScript bug patterns that appear at the server-side. One of the bug patterns they found is "Dereferenced Non-Values", which corresponds to both the "Undefined/Null Variable Usage" fault category and the "Forgetting null/undefined check" error pattern. Hence, JavaScript developers at the server-side can still gather important takeaways from our results, even if our study was not specifically targeted towards server-side JavaScript.

8. http://findbugs.sourceforge.net/
9. http://netbeans.org/
10. http://www.aptana.com/

6 Related Work

There has been a large number of empirical studies conducted on faults that occur in various types of software applications [16], [33], [34], [35], [36], [37]. Due to space constraints, we focus on only those studies that pertain to web applications.

Server-Side Studies. In the past, researchers have studied the causes of web application faults at the server-side using session-based workloads [38], server logs [39], and website outage incidents [40]. Further, there have been studies on the control-flow integrity [41] and end-to-end availability [42], [43] of web applications. Finally, studies have been conducted which propose web application fault models and taxonomies [44], [45], [46]. Our current study differs from these papers in that we focus on web application faults that occur at the client-side, particularly ones that propagate into the JavaScript code.

Client-Side Studies. Several empirical studies on the characteristics of client-side JavaScript have been made. For instance, Ratanaworabhan et al. [47] used their JSMeter tool to analyze the dynamic behaviour of JavaScript in web applications. Similar work was conducted by Richards et al. [48] and Martinsen et al. [49]. A study of parallelism in JavaScript code was also undertaken by Fortuna et al. [50]. Finally, there have been empirical studies on the security of JavaScript. These include empirical studies on cross-site scripting (XSS) sanitization [51], privacy-violating information flows [52], and remote JavaScript inclusions [53], [54]. Unlike our work which studies functional JavaScript faults, these related papers address non-functional properties such as security and performance.

In recent work, Bajaj et al. [19] mined web application-related questions in StackOverflow to determine common difficulties that developers face when writing client-side code. This work is similar to the current one in that it attempts to infer reliability issues with JavaScript, using developers' questions as an indicator. However, unlike our current work, this study does not make any attempt to determine the characteristics of JavaScript faults.

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

Finally, in very recent work which followed the original paper on which this current work is based, Pradel et al. [6] and Bae et al. [55] proposed tools for detecting type inconsistencies and web API misuses in JavaScript code, respectively. These studies provide examples of common

type faults and common web API misuse patterns. However, they do not establish the prevalence of their respective fault categories, nor do they identify DOM-related faults as an important subclass of JavaScript faults.

7 Conclusions

Client-side JavaScript contains many features that are attractive to web application developers and is the basis for modern web applications. However, it is prone to errors that can impact functionality and user experience. In this paper, we perform an empirical study of over 500 bug reports from various web applications and JavaScript libraries to help us understand the nature of the errors that cause these faults, and the failures to which these faults lead. Our results show that (1) around 68% of JavaScript faults are DOM-related; (2) most (around 80%) high severity faults are DOM-related; (3) the vast majority (around 83%) of JavaScript faults are caused by errors manually introduced by JavaScript code programmers; (4) error patterns exist in JavaScript bug reports; (5) DOM-related faults take longer to fix than non-DOM-related faults; (6) only a small but non-negligible percentage of JavaScript faults are type faults; and (7) although the percentage of code-terminating failures and browser-specific faults has decreased over the past ten years, the percentage of DOM-related faults and type faults has remained relatively constant.

Acknowledgments

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), a Four Year Fellowship (FYF) from UBC, a MITACS Graduate Fellowship, and a research gift from Intel Corporation.

References

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

[2] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2008, pp. 308–318.

[3] Microsoft, "TypeScript," http://www.typescriptlang.org/.

[4] L. Bak and K. Lund, "Dart," https://www.dartlang.org/.

[5] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side JavaScript bugs," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE Computer Society, 2013, pp. 55–64.

[6] M. Pradel, P. Schuh, and K. Sen, "TypeDevil: Dynamic type inconsistency analysis for JavaScript," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2015.

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

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

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

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

[11] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2012, pp. 419–429.

[12] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient JavaScript mutation testing," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2013.

[13] L. Marks, Y. Zou, and A. E. Hassan, "Studying the fix-time for bugs in large open source projects," in *Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE)*. ACM, 2011, p. 11.

[14] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2006, pp. 361–370.

[15] F. Ocariza, K. Pattabiraman, and A. Mesbah, "AutoFLox: An automatic fault localizer for client-side JavaScript," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2012, pp. 31–40.

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

[17] "ECMAScript documentation," http://www.ecmascript.org/docs.php.

[18] A. Swartz, "A brief history of Ajax," 2005, http://www.aaronsw.com/weblog/ajaxhistory.

[19] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Mining questions asked by web developers," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 112–121.

[20] S. R. Choudhary, M. R. Prasad, and A. Orso, "X-PERT: Accurate identification of cross-browser issues in web applications," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2013, pp. 702–711.

[21] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 561–570.

[22] S. R. Choudhary, M. R. Prasad, and A. Orso, "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2012, pp. 171–180.

[23] N. Semenenko, M. Dumas, and T. Saar, "Browserbite: Accurate cross-browser testing via machine learning over image features," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2013, pp. 528–531.

[24] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the International Workshop on Mining Software Repositories (MSR)*. IEEE Computer Society, 2007, pp. 1–8.

[25] S. Kim and E. J. Whitehead Jr, "How long did it take to fix bugs?" in *Proceedings of the International Workshop on Mining Software Repositories (MSR)*. ACM, 2006, pp. 173–174.

[26] F. Ocariza, G. Li, K. Pattabiraman, and A. Mesbah, "Automatic fault localization for client-side JavaScript," *Software Testing, Verification and Reliability (STVR)*, vol. 26, no. 1, pp. 69–88, 2016.

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

[28] B. Hackett and S.-y. Guo, "Fast and precise hybrid type inference for JavaScript," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 239–250.

[29] "Flow: a static type checker for JavaScript," http://flowtype.org/.

[30] H. Samimi, M. Schafer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, "Automated repair of HTML generation errors in PHP applications using string constraint solving," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2012, pp. 277–287.

[31] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds for web applications," in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2010, pp. 237–246.

[32] Q. Hanam, F. Brito, and A. Mesbah, "Discovering bug patterns in JavaScript," in *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, ACM. ACM, 2016, p. 11 pages.
[33] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: An empirical study of bug characteristics in modern open source software," in *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, 2006, pp. 25–33.
[34] M. Cinque, D. Cotroneo, Z. Kalbarczyk, and R. Iyer, "How do mobile phones fail?: A failure data analysis of Symbian OS smart phones," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 585–594.
[35] Z. Yin, M. Caesar, and Y. Zhou, "Towards understanding bugs in open source router software," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 3, pp. 34–40, 2010.
[36] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for Eclipse," in *Proceedings of the International Workshop on Predictor Models in Software Engineering (PROMISE)*. IEEE Computer Society, 2007, pp. 71–77.
[37] M. Erfani Joorabchi, M. Mirzaaghaei, and A. Mesbah, "Works for me! Characterizing non-reproducible bug reports," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 62–71.
[38] K. Goseva-Popstojanova, S. Mazimdar, and A. D. Singh, "Empirical study of session-based workload and reliability for web servers," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 2004, pp. 403–414.
[39] J. Tian, S. Rudraraju, and Z. Li, "Evaluating web software reliability based on workload and failure data extracted from server logs," *Transactions on Software Engineering (TSE)*, vol. 30, pp. 754–769, 2004.
[40] S. Pertet and P. Narasimhan, "Causes of failure in web applications," *Parallel Data Laboratory, Carnegie Mellon University, CMU-PDL-05-109*, 2005.
[41] B. Braun, P. Gemein, H. P. Reiser, and J. Posegga, "Control-flow integrity in web applications," in *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSOS)*. Springer, 2013, pp. 1–16.
[42] 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.
[43] V. N. Padmanabhan, S. Ramabhadran, S. Agarwal, and J. Padhye, "A study of end-to-end web access failures," in *Proceedings of the Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2006, pp. 15:1–15:13.
[44] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "Automated replay and failure detection for web applications," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2005, pp. 253–262.
[45] S. Elbaum, G. Rothermel, S. Karre, M. Fisher *et al.*, "Leveraging user-session data to support web application testing," *Transactions on Software Engineering (TSE)*, vol. 31, no. 3, pp. 187–202, 2005.
[46] A. Marchetto, F. Ricca, and P. Tonella, "An empirical validation of a web fault taxonomy and its usage for web testing," *Journal of Web Engineering (JWE)*, vol. 8, no. 4, pp. 316–345, 2009.
[47] P. Ratanaworabhan, B. Livshits, and B. Zorn, "JSMeter: comparing the behavior of JavaScript benchmarks with real web applications," in *Proceedings of the USENIX Conference on Web Application Development (WebApps)*, 2010.
[48] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 1–12.
[49] J. Martinsen, H. Grahn, and A. Isberg, "A comparative evaluation of JavaScript execution behavior," in *Proceedings of the International Conference on Web Engineering (ICWE)*. Springer, 2011, pp. 399–402.
[50] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers, "A limit study of JavaScript parallelism," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2010, pp. 1–10.
[51] 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.
[52] 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 ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 270–283.
[53] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: Large-scale evaluation of remote JavaScript inclusions," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012.
[54] C. Yue and H. Wang, "Characterizing insecure JavaScript practices on the web," in *Proceedings of the International Conference on the World Wide Web (WWW)*. ACM, 2009, pp. 961–970.
[55] S. Bae, H. Cho, I. Lim, and S. Ryu, "SAFEWAPI: web API misuse detector for web applications," in *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 507–517.

Frolin S. Ocariza, Jr. Frolin S. Ocariza, Jr. received his BASc degree (with honours) from the University of Toronto in 2010, and his MASc degree from the University of British Columbia (UBC) in 2012. Currently, he is working towards getting his PhD degree at the University of British Columbia. His main area of research is in software engineering, with emphasis on web applications, software code analysis, empirical software engineering, software fault localization and repair, and software fault detection. He received a nomination for the Best Paper Award at the IEEE International Conference on Software Testing, Verification and Validation (ICST), 2012. He was awarded the NSERC Canada Graduate Scholarship (CGS-D) in 2014, and he received two ACM CAPS-GRAD travel grants from 2014 to 2015 to attend and present his papers at the International Conference on Software Engineering (ICSE). He is a student member of the IEEE Computer Society.

Kartik Bajaj Kartik Bajaj received the Master's degree in Electrical and Computer Engineering from the University of British Columbia (UBC) in 2015. His research interests include machine learning, program synthesis and analysis, web applications, and software testing. He has published papers at the International Conference on Mining Software Repositories (MSR), 2013, and the International Conference on Automated Software Engineering (ASE), 2014-2015. He was awarded the MITACS Globalink Fellowship in 2013.

Karthik Pattabiraman Karthik Pattabiraman received his M.S and PhD. degrees from the University of Illinois at Urbana-Champaign (UIUC) in 2004 and 2009 respectively. After a post-doctoral stint at Microsoft Research (Redmond), Karthik joined the University of British Columbia (UBC) as an assistant professor of electrical and computer engineering. Karthik's research interests include dependable software systems and web applications. Karthik has won best paper or runner up awards at the IEEE International Conference on Dependable Systems and Networks (DSN), 2008, the IEEE International Conference on Software Testing (ICST), 2013 and the IEEE/ACM International Conference on Software Engineering (ICSE), 2014. He was also awarded the NSERC Discovery Accelerator Supplement award in Canada. Karthik is a member of the Steering Committee of the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), 2013 and the IFIP WG 10.4 on Dependable Computing and Fault Tolerance.

Ali Mesbah Ali Mesbah received the PhD degree in computer science from the Delft University of Technology in 2009. He is an assistant professor at the University of British Columbia (UBC) where he leads the Software Analysis and Testing (SALT) research lab. His main area of research is in software engineering and his research interests include software analysis and testing, web-based systems, software maintenance and evolution, fault localization and repair, and empirical software engineering. He has received two ACM Distinguished Paper Awards at the International Conference on Software Engineering (ICSE 2009 and ICSE 2014), a Best Paper Award at the International Symposium on Empirical Software Engineering and Measurement (ESEM 2015), and a Best Paper Award at the International Conference on Web Engineering (ICWE 2013). He was awarded the NSERC Discovery Accelerator Supplement (DAS) award in 2016. He serves on the program committees of software engineering conferences such as ICSE17, ISSTA17, and ICST17. He is a member of the IEEE Computer Society.