Abstract—Testing JavaScript code is important. JavaScript has grown to be among the most popular programming languages and it is extensively used to create web applications both on the client and server. We present the first empirical study of JavaScript tests to characterize their prevalence, quality metrics (e.g. code coverage), and shortcomings. We perform our study across a representative corpus of 373 JavaScript projects, with over 5.4 million lines of JavaScript code. Our results show that 22% of the studied subjects do not have test code. About 40% of projects with JavaScript at client-side do not have a test, while this is only about 3% for the purely server-side JavaScript projects. Also tests for server-side code have high quality (in terms of code coverage, test code ratio, test commit ratio, and average number of assertions per test), while tests for client-side code have moderate to low quality. In general, tests written in Mocha, Tape, Tap, and Nodeunit frameworks have high quality and those written without using any framework have low quality. We scrutinize the (un)covered parts of the code under test to find out root causes for the uncovered code.

Our results show that JavaScript tests lack proper coverage for event-dependent callbacks (36%), asynchronous callbacks (53%), and DOM-related code (63%). We believe that it is worthwhile for the developer and research community to focus on testing techniques and tools to achieve better coverage for difficult to cover JavaScript code.

Keywords—JavaScript applications; testing; empirical study; test quality; code coverage.

I. INTRODUCTION

JavaScript is currently the most widely used programming language according to a recent survey of more than 56K developers conducted by Stack Overflow [43], and also exploration of the programming languages used across GitHub repositories [22]. JavaScript is extensively used to build responsive modern web applications, and is also used to create desktop and mobile applications, as well as server-side network programs. Consequently, testing JavaScript applications and modules is important. However, JavaScript is quite challenging to test and analyze due to some of its specific features. For instance, the complex and dynamic interactions between JavaScript and the Document Object Model (DOM), makes it hard for developers to test effectively [22, 29, 19].

To assist developers with writing tests, there exist number of JavaScript unit testing frameworks, such as Mocha [6], Jasmine [4], QUnit [9], and Nodeunit [8], each having its own advantages [10]. Also the research community have proposed some automated testing tools and test generation techniques for JavaScript programs [33, 29, 22, 19, 23], though they are not considerably used by testers and developers yet.

Some JavaScript features, such as DOM interactions, event-dependent callbacks, asynchronous callbacks, and closures (hidden scopes), are considered to be harder to test [1, 2], [12], [11], [29], [44]. However, there is no evidence that to what extent this is true in real-world practice.

In this work, we study JavaScript (unit) tests in the wild from different angles. The results of this study reveal some of the shortcomings and difficulties of manual testing, which provide insights on how to improve existing JavaScript test generation tools and techniques. We perform our study across a representative corpus of 373 popular JavaScript projects, with over 5.4 million lines of JavaScript code. To the best of our knowledge, this work is the first study on JavaScript tests. The main contributions of our work include:

• A large-scale study to investigate the prevalence of JavaScript tests in the wild;
• A tool, called TESTSCANNER, which statically extracts different metrics in our study and is publicly available [18];
• An evaluation of the quality of JavaScript tests in terms of code coverage, average number of assertions per test, test code ratio, and test commit ratio;
• An analysis of the uncovered parts of the code under test to understand which parts are difficult to cover and why.

II. METHODOLOGY

The goal of this work is to study and characterize JavaScript tests in practice. We conduct quantitative and qualitative analyses to address the following research questions:

RQ1: How prevalent are JavaScript tests?
RQ2: What is the quality of JavaScript tests?
RQ3: Which part of the code is mainly uncovered by tests and why?

A. Subject Systems

We study 373 popular open source JavaScript projects. 138 of these subject systems are the ones used in a study for JavaScript callbacks [21] including 86 of the most depended-on modules in the NPM repository [13] and 52 JavaScript repositories from GitHub Showcases [18]. Moreover, we added 234 JavaScript repositories from Github with over 4000 stars. The complete list of these subjects and our analysis results, are available for download [18]. We believe that this corpus of 373 projects is representative of real-world JavaScript projects as they differ in domain (category), size (SLOC), maturity (number of commits and contributors), and popularity (number of stars and watchers).

1GitHub Showcases include popular and trending open source repositories organized around different topics.
We categorize our subjects into 19 categories using topics from *JSter* JavaScript Libraries Catalog [14] and *GitHub Showcases* [13] for the same or similar projects. Table I presents these categories with average values for the number of JavaScript files (production code), source lines of code (SLOC) for production and test code, number of test cases, and number of stars in Github repository for each category. We used SLOC [17] to count lines of source code excluding libraries. Overall, we study over 5.4 million (3.7 M production and 1.7 M test) source lines of JavaScript code.

Figure 1 depicts the distribution of our subject systems with respect to the client or server side code. Those systems that contain server-side components are written in Node.js, a popular server-side JavaScript framework. We apply the same categorization approach as explained in [21]. Some projects such as MVC frameworks, e.g. Angular, are purely client-side, while most NPM modules are purely server-side. We assume that client-side code is stored in directories such as *www*, *public*, *static*, or *client*. We also use code annotations such as `/** jshint browser:true, jquery:true */` to identify client-side code.

The 373 studied projects include 128 client-side, 130 server-side, and 115 client&server-side code. While distributions in total have almost the same size, they differ per project category. For instance subject systems in categories C1 (UI components), C2 (visualization), C8 (touch and drag&drop), C9 (MVC frameworks), and C18 (multimedia) are mainly client-side and those in categories C4 (software dev tools), C6 (parsers and compilers), C12 (I/O), C13 (package and build managers), C14 (storage), C16 (browser utils), and C17 (CLI and shell) are mainly server-side.

We used SLOC [17] to count lines of source code excluding libraries. Overall, we study over 5.4 million (3.7 M production and 1.7 M test) source lines of JavaScript code.

![Distribution of studied subject systems.](https://nodejs.org)

### Analysis

To address our research questions, we statically and dynamically analyze test suites of our subject programs. To extract some of the metrics in our study, we develop a static analyzer tool, called TESTSCANNER [18], which parses production and test code into an abstract syntax tree using Mozilla Rhino [7]. In the rest of this section we explain details of our analysis for each research question.

#### 1) Prevalence of tests (RQ1): To answer RQ1, we look for presence of JavaScript tests written in any framework (e.g. Mocha, Jasmine, or QUnit). Tests are usually located at folders named `tests`, `spec`, or similar names.

We further investigate the prevalence of JavaScript tests with respect to subject categories, client/server-side code, popularity (number of stars and watchers), maturity (number of commits and contributors), project size (production SLOC), and testing frameworks. To distinguish testing frameworks, we analyze package management files (such as `package.json`), task runner and build files (such as `grunt.js` and `gulpfile.js`), and test files themselves.

For instance Jasmine and Mocha tests are written as specs and are usually located at folders with similar names.
2) **Quality of tests (RQ2):** To address RQ2, for each subject with test we compute four quality metrics as following:

**Code coverage.** Coverage is generally known as an indicator of test quality. We compute statement, branch, and function coverage for JavaScript code using JSCover [5] (for tests that run in the browser), and Istanbul [3]. To calculate coverage of the minified JavaScript code, we beautify them prior to executing tests. We also exclude dependencies, such as files under the `node_modules` directory, and libraries (unless the subject system is itself a library).

**Average number of assertions per test.** Code coverage does not directly imply a test suite effectiveness [24], while assertions have been shown to be strongly correlated with it [49]. Thus, TESTSCANNER also computes average number of assertions per test case as a test suite quality metric. Our analysis tools detects usage of well-known assertion libraries such as assert.js, should.js, expect.js, and chai.

**Test code ratio.** This metric is defined as the ratio of test SLOC to production and test SLOC. A program with a high test code ratio may have a higher quality test suite.

**Test commit ratio.** This metric is the ratio of test commits to total commits. Higher test commit ratio may indicate more mature and higher quality tests. We assume that every commit that touches at least one file in a folder named `test`, `tests`, `spec`, or `specs` is a test commit. In rare cases that tests are stored elsewhere, such as the root folder, we manually extract number of test commits by looking at its Github repository page and counting commits on test files.

We investigate these quality metrics with respect to subject categories, client/server-side code, and testing frameworks.

3) **(Un)**covered code (RQ3): Code coverage is a widely accepted test quality indicator, thus finding the root cause of why a particular statement is not covered by a test suite, can help in writing higher quality tests. Some generic possible cases for an uncovered (missed) statement $s$, are as following:

1) $s$ belongs to an uncovered function $f$, where
   a) $f$ has no calling site in both the production and the test code. In this case, $f$ could be (1) a callback function sent to a callback-accepting function (e.g., `setTimeout()` that was never invoked, or (2) an unused utility function that was meant to be used in previous or future releases. Such unused code can be considered as code smells [28]. Consequently we cannot pinpoint such an uncovered function to a particular reason.
   b) the calling site for $f$ in the production code was never executed by a test. Possible root causes can be that (1) $f$ is used as a callback (e.g. event-dependent or asynchronous) that was never invoked, (2) the call to $f$ statement was never reached because of an earlier `return` statement or an exception, or the function call falls in a never met condition branch.
   c) $f$ is an anonymous function. Possible reasons that $f$ was not covered can be that (1) $f$ is used as a callback that was never invoked (e.g. an event-dependent callback while the required event was not triggered, or an asynchronous callback while did not wait for the response), (2) $f$ is a self-invoking function that was not executed to be invoked, or (3) $f$ is set to a variable and that variable was never used or its usage was not executed.

2) $s$ belongs to a covered function $f$, where
   a) the execution of $f$ was terminated, by a `return` statement or an exception, prior to reaching $s$.
   b) $s$ falls in a never met condition in $f$ (e.g. browser or DOM dependent statements).

3) The test case responsible for covering $s$ was not executed due to a test execution failure.
4) $s$ is a dead (unreachable) code.

**Uncovered statement in uncovered function ratio.** If an uncovered statement $s$ belongs to an uncovered function $f$, making $f$ called could possibly cover $s$ as well. This is important specially if $f$ needs to be called in a particular way, such as through triggering an event.

In this regard, our tool uses coverage report information (in json or lcov format) to calculate the ratio of the uncovered statements that fall within uncovered functions over the total number of uncovered statements. If this value is large it indicates that the majority of uncovered statements belong to uncovered functions, and thus code coverage could be increased to a high extent if the enclosing function is called by a test case.

**Hard-to-test JavaScript code.** Some JavaScript features, such as DOM interactions, event-dependent callbacks, asynchronous callbacks, and closures (hidden scopes), are considered to be harder to test [1], [2], [12], [11], [29], [44]. In this section we explain four main hard-to-test code with an example code snippet depicted in Figure 2. Also we fine-grain statement and function coverage metrics to investigate these hard-to-test code separately in detail. To measure these coverage metrics, TESTSCANNER maps a given coverage report to the locations of hard-to-test code.

**DOM related code coverage.** In order to unit test a JavaScript code with DOM read/write operations, a DOM instance has to be provided as a test fixture in the exact structure expected by the code under test. Otherwise, the test case can terminate prematurely due to a null exception. Writing such DOM based fixtures can be challenging due to the dynamic nature of JavaScript and the hierarchical structure of the DOM [29]. For example, to cover the `if` branch at line 29 in Figure 2 one needs to provide a DOM instance such as `<div id="checkList"><input type="checkbox" checked></input></div>`. To cover the `else` branch, a DOM instance such as `<div id="checkList"><input type="checkbox"></input></div>` is required. If such fixtures are not provided, `!($("checkList")).return null as the expected element is not available, and thus checkList.children returns null` causes a null exception and the test case terminates.

**DOM related code coverage** is defined as the fraction of number of covered over total number of DOM related statements. A DOM related statement is a statement that can affect or be affected by DOM interactions such as a DOM API usage. To detect DOM related statements TESTSCANNER extracts all DOM API usages in the code (e.g. `getElementById`, `createElement`, `appendChild`, `appendChild`
addEventListener, $, and innerHTML) and their forward slices. Forward slicing is applied on the variables that were assigned with a DOM element/attribute. For example the forward slice of checkList at line 26 in Figure 2 are lines 27–34. A DOM API could be located in a (1) return statement of a function $f$, (2) conditional statement, (3) function call (as an argument), (4) an assignment statement, or (5) other parts within a scope. In case (1), all statements that has a call to $f$ are considered DOM related. In case (2), the whole conditional statements (condition and the body of condition) are considered DOM related. In case (3) the statements in the called function, which use that DOM input will be considered DOM related. In other cases, the statement with DOM API is DOM related.

Event-dependent callback coverage. The execution of some JavaScript code may require triggering an event such as clicking on a particular DOM element. For instance it is very common in JavaScript client-side code to have an (anonymous) function bound to an element’s event, e.g. a click, which has to be simulated. The anonymous function in lines 17–24 is an event-dependent callback function. Such callback functions would only be passed and invoked if the corresponding event is triggered. In order to trigger an event, testers can use methods such as jQuery’s .trigger(event, data, ...) or .emit(event, data, ...) of Node.js EventEmitter. Note that if an event needs to be triggered on a DOM element, a proper fixture is required otherwise the callback function cannot be executed.

Event-dependent callback coverage is defined as the fraction of number of covered over total number of event-dependent callbacks. In order to detect event-dependent call-
the percentage of subjects in that category. In total, among the 373 studied subjects, 83 of them (i.e., 22%) do not have JavaScript tests. The majority (78%) of subjects have at least one test case.

**Finding 1:** 22% of the subject systems that we studied do not have any JavaScript test, and 78% have at least one test case.

As shown in figure 3(b) amongst subjects with test, the majority of tests are written in Mocha (38%), Jasmine (19%), and QUnit (18%). 6% does not follow any particular framework and have their own tests. Minor used frameworks are Tap (5%), Tape (4%), Nodeunit (3%), Vows (3%) and others (4%) including Jest, Evidence.js, Doh, CasperJS, Ava, UTent, TAD, and Lab. We also observe that 3 repositories have tests written in two testing frameworks: 2 projects (server and client-server) with Nodeunit+Mocha test, and one (client-server) with Jasmine+QUnit test.

**Finding 2:** The most prevalent used test frameworks for JavaScript unit testing are Mocha (38%), Jasmine (19%), and QUnit (18%).

We also investigate the prevalence of UI tests and observe that only 12 projects (i.e., 3%) among all 373 ones have UI tests for which 9 are written using Webdriverio and Selenium webdriver, and 3 uses CasperJS. 7 of these projects are client and server side, 3 are client-side, and 2 are server-side. One of these subjects does not have any JavaScript test.

**Finding 3:** Only 3% of the studied repositories have functional UI tests.

Almost all (95%) of purely server-side JavaScript projects have tests, while this is 61% for client-side and 76% for client&server-side ones. Note that the number of subjects in each category are not very different (i.e., 128 client-side, 130 server-side, and 115 client and server-side code). Interestingly the distribution of test frameworks looks very similar for client-side and client-server side projects.

As shown in Figure 3(a) all subjects systems in categories C6 (parsers and compilers), C12 (I/O), C13 (package and build managers), C14 (storage), C19 (MVC frameworks), and C17 (CLI and shell), have JavaScript unit tests. Projects in all of these categories, except for C19, are mainly server-side as depicted in Figure 1. In contrast, many of subjects in categories C1 (UI components), C3 (web apps), C8 (touch and drag&drop), and C18 (multimedia) do not have tests, which are mainly client-side. Thus we can deduce that JavaScript tests are written more for server-side code than client-side, or client and server-side code.

**Finding 4:** While almost all subjects (95%) in the server-side category have tests, about 40% of subjects in client-side and client-server side categories do not have tests.

We believe the more prevalence of tests for server-side code can be attributed to (1) the difficulties in testing client-side code, such as writing proper DOM fixtures or triggering events on DOM elements, and (2) using time-saving test scripts for most Node.js based projects, such as npm test that is included by default when initializing a new package.json file. This pattern is advocated in the Node.js community.
and thus many server-side JavaScript code, such as NPM modules, have test code.

We also consider how popularity (number of stars and watchers) and maturity (number of commits and contributors) of subject systems are related to the prevalence of unit tests. Figure 4(a) shows the percentage of subjects with tests in each quartile. As popularity and maturity increase, the percentage of subjects with test increases as well.

B. Quality of Tests (RQ2)

Code coverage. Calculating the code coverage requires executing tests on a properly deployed project. In our study, however, we faced number of projects with failure in build/deployment or running tests. We tried to resolve such problems by quick changes in build/task configuration files or by retrieving a later version (i.e., some days after fetching the previous release). In most cases build failure was due to errors in dependent packages or their absence. We could finally calculate coverage for 231 out of 290 (about 80%) subjects with tests. We could not properly deploy or run tests for 44 subject systems (41 with test run failure, freeze, or break, and 3 build and deployment error), and could not get coverage report for 15 projects with complex test configurations.

Boxplots in Figure 5 show that in total tests have a median of 83% statement coverage, 84% function coverage, and 69% branch coverage. Tests for server-side code have higher coverage in all aspects compared to those for client-side code.

We narrow down our coverage analysis into different subject categories. As depicted in Table II subjects in categories C6 (parsers and compilers), C10 (Network and Async), C12 (I/O), C13 (package and build managers), C14 (testing frameworks), and C19 (MVC frameworks) on average have higher code coverage. Projects in these categories are mainly server-side. In contrast, subjects in categories C2 (visualization), C3 (web apps), C8 (touch and drag&drop), C11 (game engines), C17 (CLI and shell), and C18 (multimedia), have lower code coverage. Note that subjects under these categories are mainly client-side.

Finding 5: The studied JavaScript tests have a median of 83% statement coverage, 84% function coverage, and 69% branch coverage. Tests for server-side code have higher coverage in all aspects compared to those for client-side code.

Average number of assertions per test. Figure 6 depicts boxplots of average number of assertions per test case. While median values are very similar (about 2.2) for all cases, server-side code has a slightly higher mean value (3.16) compared to client-side (2.71). As shown in Table II subjects in categories C3 (web apps), C11 (game engines), C15 (testing frameworks), C17 (CLI and shell), C18 (multimedia), and C19 (MVC frameworks) on average have higher average number of assertions per test compared to others. Interestingly among these categories only for C15 and C19 code coverage is also high while it is low for the rest.

Table II also depicts the achieved coverage per testing framework. Tests written in Tape, Tap, and Mocha have generally higher code coverage. The majority of server-side JavaScript projects are tested using these frameworks. On the other hand, tests written in QUnit, which is used more often for the client-side than the server-side, has generally lower code coverage. Developers that used their own style of testing without using popular frameworks write tests with the poorest coverage.

Finding 6: Tests written in Tape, Tap, and Mocha frameworks, generally have higher coverage compared to those written in QUnit, Nodeunit, and those without using any test framework.

Finding 7: The studied test suites have a median of 2.19 and a mean of 2.96 for the average number of assertions per test. These values do not differ much among server-side and client-side code.

Also results shown in Table II indicate that tests written in QUnit, Tape, Nodeunit, other frameworks (e.g. Jest, CasperJS,
TABLE II: Test quality metrics average values.

<table>
<thead>
<tr>
<th>Subject category</th>
<th>Statement coverage</th>
<th>Branch coverage</th>
<th>Function coverage</th>
<th>Ave # assertions</th>
<th>Test code ratio</th>
<th>Test commit ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>77%</td>
<td>57%</td>
<td>76%</td>
<td>2.83</td>
<td>0.41</td>
<td>0.16</td>
</tr>
<tr>
<td>C2</td>
<td>67%</td>
<td>52%</td>
<td>65%</td>
<td>2.72</td>
<td>0.28</td>
<td>0.14</td>
</tr>
<tr>
<td>C3</td>
<td>60%</td>
<td>58%</td>
<td>58%</td>
<td>5.73</td>
<td>0.88</td>
<td>0.14</td>
</tr>
<tr>
<td>C4</td>
<td>79%</td>
<td>68%</td>
<td>78%</td>
<td>2.50</td>
<td>0.38</td>
<td>0.24</td>
</tr>
<tr>
<td>C5</td>
<td>75%</td>
<td>63%</td>
<td>75%</td>
<td>2.53</td>
<td>0.32</td>
<td>0.21</td>
</tr>
<tr>
<td>C6</td>
<td>87%</td>
<td>79%</td>
<td>88%</td>
<td>2.53</td>
<td>0.47</td>
<td>0.24</td>
</tr>
<tr>
<td>C7</td>
<td>80%</td>
<td>67%</td>
<td>72%</td>
<td>2.51</td>
<td>0.46</td>
<td>0.22</td>
</tr>
<tr>
<td>C8</td>
<td>64%</td>
<td>47%</td>
<td>60%</td>
<td>2.04</td>
<td>0.35</td>
<td>0.12</td>
</tr>
<tr>
<td>C9</td>
<td>75%</td>
<td>63%</td>
<td>75%</td>
<td>2.53</td>
<td>0.52</td>
<td>0.21</td>
</tr>
<tr>
<td>C10</td>
<td>87%</td>
<td>79%</td>
<td>88%</td>
<td>2.53</td>
<td>0.47</td>
<td>0.24</td>
</tr>
<tr>
<td>C11</td>
<td>91%</td>
<td>79%</td>
<td>90%</td>
<td>2.73</td>
<td>0.72</td>
<td>0.24</td>
</tr>
<tr>
<td>C12</td>
<td>64%</td>
<td>45%</td>
<td>57%</td>
<td>3.41</td>
<td>0.18</td>
<td>0.11</td>
</tr>
<tr>
<td>C13</td>
<td>90%</td>
<td>77%</td>
<td>89%</td>
<td>2.36</td>
<td>0.39</td>
<td>0.20</td>
</tr>
<tr>
<td>C14</td>
<td>88%</td>
<td>67%</td>
<td>84%</td>
<td>2.27</td>
<td>0.39</td>
<td>0.20</td>
</tr>
<tr>
<td>C15</td>
<td>81%</td>
<td>69%</td>
<td>79%</td>
<td>5.79</td>
<td>0.39</td>
<td>0.25</td>
</tr>
<tr>
<td>C16</td>
<td>78%</td>
<td>67%</td>
<td>79%</td>
<td>1.67</td>
<td>0.49</td>
<td>0.29</td>
</tr>
<tr>
<td>C17</td>
<td>67%</td>
<td>54%</td>
<td>63%</td>
<td>8.32</td>
<td>0.47</td>
<td>0.21</td>
</tr>
<tr>
<td>C18</td>
<td>60%</td>
<td>51%</td>
<td>62%</td>
<td>4.42</td>
<td>0.31</td>
<td>0.16</td>
</tr>
<tr>
<td>C19</td>
<td>81%</td>
<td>67%</td>
<td>80%</td>
<td>3.20</td>
<td>0.53</td>
<td>0.24</td>
</tr>
<tr>
<td>Mocha</td>
<td>82%</td>
<td>70%</td>
<td>79%</td>
<td>2.39</td>
<td>0.49</td>
<td>0.20</td>
</tr>
<tr>
<td>Jasmine</td>
<td>74%</td>
<td>60%</td>
<td>75%</td>
<td>1.93</td>
<td>0.41</td>
<td>0.21</td>
</tr>
<tr>
<td>QUnit</td>
<td>71%</td>
<td>54%</td>
<td>71%</td>
<td>5.93</td>
<td>0.41</td>
<td>0.16</td>
</tr>
<tr>
<td>Own test</td>
<td>61%</td>
<td>41%</td>
<td>58%</td>
<td>5.99</td>
<td>0.30</td>
<td>0.16</td>
</tr>
<tr>
<td>Tape</td>
<td>89%</td>
<td>80%</td>
<td>89%</td>
<td>2.74</td>
<td>0.62</td>
<td>0.26</td>
</tr>
<tr>
<td>Others</td>
<td>81%</td>
<td>69%</td>
<td>79%</td>
<td>5.79</td>
<td>0.39</td>
<td>0.25</td>
</tr>
<tr>
<td>Nodeunit</td>
<td>74%</td>
<td>63%</td>
<td>72%</td>
<td>9.61</td>
<td>0.46</td>
<td>0.24</td>
</tr>
<tr>
<td>Vows</td>
<td>74%</td>
<td>66%</td>
<td>72%</td>
<td>0.20</td>
<td>0.57</td>
<td>0.24</td>
</tr>
<tr>
<td>C&amp;S</td>
<td>72%</td>
<td>56%</td>
<td>70%</td>
<td>2.91</td>
<td>0.49</td>
<td>0.24</td>
</tr>
<tr>
<td>Total</td>
<td>78%</td>
<td>64%</td>
<td>76%</td>
<td>2.96</td>
<td>0.46</td>
<td>0.2</td>
</tr>
</tbody>
</table>

and UTest), and those without using a framework, have on average more assertions per test. The majority of server-side JavaScript projects are tested using these frameworks. Again we observe that only for tests written in Tape framework code coverage is also high while it is low for the rest.

Test code ratio. Figure 7 shows test to total (production and test) code ratio comparison. The median and mean of this ratio is about 0.6 for server-side projects and about 0.35 for client-side ones. As shown in Table II on average subjects with higher test code ratio belongs to categories C3, C4, C5, C10, C12, C13, C14, C15, and C19 while those in C2, C8, C11, and C18 have lower test code ratio. Also tests written in Tap, Tape, Nodeunit, and Vows have higher test code ratio while tests written without using any framework have lower test code ratio.

We further study the relationship between test code ratio and total code coverage (average of statement, branch, and function coverage) through the Spearman’s correlation analysis. The result shows that there exists a moderate to strong correlation ($\rho = 0.68$, $p = 0$) between test code ratio and code coverage.

Finding 8: Tests for server-side code have higher test code ratio (median and mean of about 0.6) compared to client-side code (median and mean of about 0.35). Also there exists a moderate to strong correlation ($\rho = 0.68$, $p = 0$) between test code ratio and code coverage.

4The non-parametric Spearman’s correlation coefficient measures the monotonic relationship between two continuous random variables and does not require the data to be normally distributed.

Test commit ratio. Figure 8 depicts test to total (production and test) commit ratio comparison. The median and mean of this ratio is about 0.25 for server-side projects and about 0.15 for client-side ones. As shown in Table II on average subjects with higher test commit ratio belongs to categories C4, C6, C9, C10, C14, C15, and C16 while those in C1, C2, C3, C8,
TABLE III: Statistics for analyzing uncovered code. The "-" sign indicates no instance of a particular code.

<table>
<thead>
<tr>
<th>Subject category</th>
<th>Function coverage</th>
<th>Statement coverage</th>
<th>Ave # func calls per test</th>
<th>USUF ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>Async</td>
<td>Event dependent</td>
<td>Closure</td>
</tr>
<tr>
<td>C1</td>
<td>76%</td>
<td>65%</td>
<td>33%</td>
<td>79%</td>
</tr>
<tr>
<td>C2</td>
<td>65%</td>
<td>45%</td>
<td>17%</td>
<td>62%</td>
</tr>
<tr>
<td>C3</td>
<td>58%</td>
<td>21%</td>
<td>10%</td>
<td>58%</td>
</tr>
<tr>
<td>C4</td>
<td>79%</td>
<td>49%</td>
<td>48%</td>
<td>70%</td>
</tr>
<tr>
<td>C5</td>
<td>75%</td>
<td>52%</td>
<td>33%</td>
<td>65%</td>
</tr>
<tr>
<td>C6</td>
<td>88%</td>
<td>60%</td>
<td>32%</td>
<td>87%</td>
</tr>
<tr>
<td>C7</td>
<td>/2%</td>
<td>34%</td>
<td>28%</td>
<td>81%</td>
</tr>
<tr>
<td>C8</td>
<td>60%</td>
<td>40%</td>
<td>39%</td>
<td>80%</td>
</tr>
<tr>
<td>C9</td>
<td>69%</td>
<td>14%</td>
<td>8%</td>
<td>80%</td>
</tr>
<tr>
<td>C10</td>
<td>90%</td>
<td>65%</td>
<td>60%</td>
<td>95%</td>
</tr>
<tr>
<td>C11</td>
<td>57%</td>
<td>35%</td>
<td>7%</td>
<td>68%</td>
</tr>
<tr>
<td>C12</td>
<td>89%</td>
<td>83%</td>
<td>68%</td>
<td>98%</td>
</tr>
<tr>
<td>C13</td>
<td>84%</td>
<td>71%</td>
<td>74%</td>
<td>60%</td>
</tr>
<tr>
<td>C14</td>
<td>81%</td>
<td>66%</td>
<td>46%</td>
<td>89%</td>
</tr>
<tr>
<td>C15</td>
<td>79%</td>
<td>70%</td>
<td>39%</td>
<td>62%</td>
</tr>
<tr>
<td>C16</td>
<td>79%</td>
<td>40%</td>
<td>5%</td>
<td>43%</td>
</tr>
<tr>
<td>C17</td>
<td>63%</td>
<td>7%</td>
<td>5%</td>
<td>36%</td>
</tr>
<tr>
<td>C18</td>
<td>62%</td>
<td>–</td>
<td>10%</td>
<td>89%</td>
</tr>
<tr>
<td>C19</td>
<td>81%</td>
<td>61%</td>
<td>47%</td>
<td>76%</td>
</tr>
<tr>
<td>Mocha</td>
<td>79%</td>
<td>50%</td>
<td>34%</td>
<td>71%</td>
</tr>
<tr>
<td>Jasmine</td>
<td>75%</td>
<td>65%</td>
<td>34%</td>
<td>69%</td>
</tr>
<tr>
<td>QUnit</td>
<td>11%</td>
<td>53%</td>
<td>28%</td>
<td>76%</td>
</tr>
<tr>
<td>Own test</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>/0%</td>
</tr>
<tr>
<td>Tap</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>/0%</td>
</tr>
<tr>
<td>Tape</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>/0%</td>
</tr>
<tr>
<td>Others</td>
<td>77%</td>
<td>33%</td>
<td>30%</td>
<td>66%</td>
</tr>
<tr>
<td>Nodeunit</td>
<td>72%</td>
<td>55%</td>
<td>63%</td>
<td>74%</td>
</tr>
<tr>
<td>Vows</td>
<td>72%</td>
<td>35%</td>
<td>63%</td>
<td>74%</td>
</tr>
<tr>
<td>Client</td>
<td>70%</td>
<td>46%</td>
<td>25%</td>
<td>69%</td>
</tr>
<tr>
<td>Server</td>
<td>83%</td>
<td>64%</td>
<td>48%</td>
<td>82%</td>
</tr>
<tr>
<td>C&amp;S</td>
<td>70%</td>
<td>48%</td>
<td>29%</td>
<td>69%</td>
</tr>
<tr>
<td>Total</td>
<td>76%</td>
<td>53%</td>
<td>36%</td>
<td>74%</td>
</tr>
</tbody>
</table>

C11, and C18 have lower test commit ratio. Also tests written in Nodeunit, Vows, and other frameworks (e.g. Jest, CasperJS, and UTest) have higher test commit ratio while tests written in QUnit or without using any framework have lower test commit ratio.

Similar to the correlation analysis for test code ratio, we study the relationship between test commit ratio and total code coverage. The result indicates that there exists a moderate to low correlation ($\rho = 0.49, p = 0$) between test commit ratio and code coverage.

Finding 9: While test commit ratio is relatively high for server-side projects (median and mean of about 0.25), it is moderate in total and relatively low for client-side projects (median and mean of about 0.15). Also there exists a moderate to low correlation ($\rho = 0.49, p = 0$) between test commit ratio and code coverage.

C. (Un)covered Code (RQ3)

As explained earlier in Section II-B3 one possible root cause for uncovered code is that the responsible test code was not executed. In our evaluation, however, we observed that for almost all the studied subjects, test code had very high coverage meaning that almost all statements in test code were executed properly. Thus the test code coverage does not contribute in the low coverage of production code.

Uncovered statement in uncovered function (USUF) ratio. If an uncovered code $c$ belongs to an uncovered function $f$, making $f$ called could possibly cover $c$ as well. As described in Section II-B3 we calculate the ratio of uncovered statements that fall within uncovered functions over the total number of uncovered statements.

Table III shows average values for this ratio (USUF). The mean value of USUF ratio is 0.57 in total, 0.45 for server-side projects, and about 0.7 for client-side ones. This indicates that the majority of uncovered statements in client-side code belong to uncovered functions, and thus code coverage could be increased to a high extent if the enclosing function could be called during test execution.

Finding 10: A large portion of uncovered statements fall in uncovered functions for client-side code (about 70%) compared to server-side code (45%).

Hard-to-test-function coverage. We measure coverage for hard-to-test functions as defined in Section II-B5. While the average function coverage in total is 76%, the average event-dependent callback coverage is 36% and the average asynchronous callback coverage is 53%. The average value of closure function coverage in total is 74% and for server-side subjects is 82% while it is 69% for client-side ones.

Finding 11: On average, JavaScript tests have low coverage for event-dependent callbacks (36%) and asynchronous callbacks (53%). Average values for client-side code are even worse (25% and 46% respectively). The average closure function coverage is 74%.
We measure the impact of tests with event triggering methods on event-dependent callback coverage, and writing asynchronous callback coverage through correlation analysis. The results show that there exists a weak correlation ($\rho = 0.22$) between average number of event triggers and event-dependent callback coverage, and a very weak correlation ($\rho = 0.1$) between number of asynchronous tests and asynchronous callback coverage.

**Finding 12:** There is no strong correlation between number of event triggers and event-dependent callback coverage. Also number of asynchronous tests and asynchronous callback coverage are not strongly correlated.

This was contrary to our expectation for higher correlations, however, we observed that in some cases asynchronous tests and tests that trigger events were written to merely target specific parts and functionalities of the production code without covering most asynchronous or event-dependent callbacks.

**DOM related code coverage.** On average, JavaScript tests have a moderately low coverage of 63% for DOM-related code. We also study the relationship of existence of DOM fixtures and DOM related code coverage through correlation analysis. The result shows that there exists a correlation of $\rho = 0.4$, $\rho = 0$ between having DOM fixtures in tests and DOM related code coverage. Similar to the cases for event-dependent and async callbacks, we also observed that DOM fixtures were mainly written for executing a subset of DOM related code.

**Finding 13:** On average, JavaScript tests lack proper coverage for DOM-related code (63%). Also there exists a moderately low correlation ($\rho = 0.4$) between having DOM fixtures in tests and DOM related code coverage.

**Average number of function calls per test.** As explained in Section II-B3 we investigate number of unique function calls per test. The average number of function calls per test has a mean value of about 3 in total and also across server-side and client-side code. We further perform a correlation analysis between the average number of function calls per test and total code coverage. The result shows that there exists a weak correlation ($\rho = 0.13$, $\rho = 0$) between average number of function calls per test and code coverage.

**Finding 14:** On average, there are about 3 function calls to production code per test case. The average number of function calls per test is not strongly correlated with code coverage.

### D. Discussion

**Implications.** Our findings regarding RQ1 indicate that the majority (78%) of studied JavaScript projects and in particular popular and trending ones have at least one test case. This indicates that JavaScript testing is getting attention, however, it seems that developers have less tendency to write tests for client-side code as they do for the server-side code. Possible reasons could be difficulties in writing proper DOM fixtures or triggering events on DOM elements. We also think that the high percentage of test for server-side JavaScript can be ascribed to the testing pattern that is advocated in the Node.js community [16]. To assist developers with testing their JavaScript code, we believe that it is worthwhile for the research community to invest on developing test generation techniques in particular for the client-side code, such as [33], [29], [32].

For RQ2, the results indicate that in general, tests written for mainly client-side subjects in categories C2 (visualization), C8 (touch and drag&drop), C11 (game engines), and C18 (multimedia) have lower quality. Compared to the client-side projects, tests written for the server-side have higher quality in terms of code coverage, test code ratio, and test commit ratio. The branch coverage in particular for client-side code is low, which can be ascribed to the challenges in writing tests for DOM related branches. We investigate reasons behind the code coverage difference in Section III-C. The higher values for test code ratio and test commit ratio can also be due to the fact that writing tests for server-side code is easier compared to client-side.

Developers and testers could possibly increase code coverage of their tests by using existing JavaScript test generator tools, such as Kudzu [41], ARTEMIS [19], JALANGI [42], SymJS [27], JSEFT [32], and CONFIX [29]. Tests written in Mocha, Tap, Tape, and Nodeunit generally have higher test quality compared to other frameworks and tests that do not use any testing framework. In fact developers that do not write their test by leveraging an existing testing framework write low quality tests almost in all aspects. Thus we recommend JavaScript developers community to use a well-maintained and mature testing framework to write their tests.

As far as RQ3 is concerned, our study shows that JavaScript tests lack proper coverage for event-dependent callbacks, asynchronous callbacks, and DOM-related code. Since these parts of code are hard to test they can be error prone and thus requires effective targeted tests. For instance a recent empirical study [36] reveals that the majority of reported JavaScript bugs and the highest impact faults are DOM-related.

It is expected that using event triggering methods in tests, increase coverage for event-dependent callbacks, asynchronous callbacks, and DOM-related statements. However, our results do not show a strong correlation to support this. Our manual analysis revealed that tests with event triggering methods, async behaviours, and DOM fixtures are mainly written to cover only particular instances of event-dependent callbacks, asynchronous callbacks, or DOM-related code. This again can imply difficulties in writing tests with high coverage for such hard-to-test code.

We believe that there is a research potential in this regard for proposing test generation techniques tailored to such uncovered parts. While most current test generation tools for JavaScript produce tests at single function level, in practice developers often write tests that invoke about 3 functions per test on average. It might also worth for researchers to develop test generation tools that produce tests with a sequence of function calls per test case.

Finally, we observed that UI tests are much less prevalent in the studied JavaScript projects. Our investigation of the coverage report did not show a significant coverage increase on the uncovered event-dependent callbacks or DOM-related code between UI and unit tests. Since UI tests do not need
DOM fixture generation, they should be able to trigger more of the UI events, compared to code level unit tests. It would be interesting to further investigate this in JavaScript applications with large UI tests.

**Test effectiveness.** Another test quality metric that is interesting to investigate is test effectiveness. An ideal effective test suite should fail if there is a defect in the code. *Mutation score*, i.e., the percentage of killed mutants over total non-equivalent mutants, is often used as an estimate of defect detection capability of a test suite. In fact it has been shown that there exists a significant correlation between mutant detection and real fault detection [26]. In this work, however, we did not consider mutation score as a quality metric as it was too costly to generate mutants for each subject and execute the tests on each of them. We believe that it is worthwhile to study the effectiveness of JavaScript tests using mutation testing techniques, such as Mutandis [31], which guides mutation generation towards parts of the code that are likely to affect the program output. This can help to find out which aspects of code are more error-prone and not well-tested. Apart from test quality evaluation based on mutation score, studying JavaScript bug reports [47] and investigating bug locations, can give us new insights for developing more effective test generation tools.

**Threats to validity.** With respect to reproducibility of the results, our tool and list of the studied subjects are publicly available [13]. Regarding the generalizability of the results to other JavaScript projects, we believe that the studied set of subjects is representative of real-world JavaScript projects as they differ in domain (category), size (SLOC), maturity (number of commits and contributors), and popularity (number of stars and watchers). With regards to the subject categorization, we used some existing categories proposed by JSter Catalog [14] and GitHub Showcases [13].

There might be case that TESTSCANNER cannot detect a desired pattern in the code as it performs complex static code analysis for detecting DOM-related statements, event-dependent callbacks, and asynchronous APIs. To mitigate this threat, we made a second pass of manual investigation through such code patterns using grep with regular expressions in command line and manually validated random cases. Such a textual search within JavaScript files through grep was especially done for a number of projects with parsing errors in their code for which TESTSCANNER cannot generate a report or the report would be incomplete. Since our tool statically analyzes test code to compute the number of function calls per test, it may not capture the correct number of calls that happen during execution. While dynamic analysis could help with this regard, it can not be used for the unexecuted code and thus is not helpful to analyze uncovered code.

**IV. RELATED WORK**

There are number of previous empirical studies on JavaScript. Ratanaworabhan et al. [38] and Richards et al. [40] studied JavaScript’s dynamic behavior and Richards et al. [39] analyzed security issues in JavaScript projects. Ocariza et al. [37] performed study to characterize root causes of client-side JavaScript bugs. Gallaba et al. [21] studied the use of callback in client and server-side JavaScript code. Security vulnerabilities in JavaScript have also been studied on remote JavaScript inclusions [35], [47], cross-site scripting (XSS) [46], and privacy violating information flows [25]. Milani Fard et al. [28] studied code smells in JavaScript code. Nguyen et al. [34] performed usage patterns mining in JavaScript web applications.

Researchers also studied test cases and mining test suites in the past. Inozemtseva et al. [24] found that code coverage does not directly imply the test suite effectiveness. Zhang et al. [49] analyzed test assertions and showed that existence of assertions is strongly correlated with test suite effectiveness. Vahabzadeh et al. [45] studied bugs in test code. Milani Fard et al. proposed Testilizer [30] that mines information from existing test cases to generate new tests. Zaidman et al. [48] investigated co-evolution of production and test code.

These work, however, did not study JavaScript tests. Related to our work, Mirshokraie et al. [31] presented a JavaScript mutation testing approach and as part of their evaluation, assessed mutation score for test suites of two JavaScript libraries. To the best of our knowledge, our work is the first (large scale) study on JavaScript tests and in particular their quality and shortcomings.

**V. CONCLUSIONS AND FUTURE WORK**

JavaScript is heavily used to build responsive client-side web applications as well as server-side projects. While some JavaScript features are known to be hard to test, no empirical study was done earlier towards measuring the quality and coverage of JavaScript tests. This work presents the first empirical study of JavaScript tests to characterize their prevalence, quality metrics, and shortcomings.

We found that a considerable number of JavaScript projects do not have any tests and this is in particular for projects with client-side JavaScript code. On the other hand, almost all purely server-side JavaScript projects have tests and the quality of those tests are higher compared to client-side tests. On average, JavaScript tests lack proper coverage for event-dependent callbacks, asynchronous callbacks, and DOM-related code.

The results of this study can be used to improve JavaScript test generation tools in producing more effective test cases that target hard-to-test portions of the code. We also plan to evaluate effectiveness of JavaScript test by measuring their mutation score, which reveals the quality of written assertions. Another possible direction could be designing automated JavaScript code refactoring techniques towards making the code more testable and maintainable.

**ACKNOWLEDGMENT**

This work was supported by the National Science and Engineering Research Council of Canada (NSERC) through its Strategic Project Grants programme and Alexander Graham Bell Canada Graduate Scholarship.

**References**


