JavaScript: The (Un)covered Parts

Amin Milani Fard
University of British Columbia
Vancouver, BC, Canada
aminmf@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

***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 [32], [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], [32], [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* [15] and 52 JavaScript repositories from *GitHub Showcases*[1] [13]. 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).

[1]GitHub Showcases include popular and trending open source repositories organized around different topics.

TABLE I: Our JavaScript subject systems (60K files, 3.7 M production SLOC, 1.7 M test SLOC, and 100K test cases).

ID	Category	# Subject systems	Ave # JS files	Ave Prod SLOC	Ave Test SLOC	Ave # tests	Ave # assertions	Ave # stars
C1	UI Components, Widgets, and Frameworks	52	41	4.7K	2.8K	235	641	9.8K
C2	Visualization, Graphics, and Animation Libraries	48	53	10.2K	3.8K	425	926	7.5K
C3	Web Applications and Games	33	61	10.6K	1.4K	61	119	4K
C4	Software Development Tools	29	67	12.7K	7.8K	227	578	6.9K
C5	Web and Mobile App Design and Frameworks	25	91	22.3K	6.9K	277	850	14.4K
C6	Parsers, Code Editors, and Compilers	22	167	27K	9.5K	701	1142	5.5K
C7	Editors, String Processors, and Templating Engines	19	26	4.3K	1.9K	102	221	6.5K
C8	Touch, Drag&Drop, Sliders, and Galleries	19	10	1.9K	408	52	72	7.9K
C9	Other Tools and Libraries	17	93	9.1K	7.6K	180	453	8.5K
C10	Network, Communication, and Async Utilities	16	19	4.1K	7.6K	279	354	7.6K
C11	Game Engines and Frameworks	13	86	17K	1.2K	115	293	3.5K
C12	I/O, Stream, and Keyboard Utilities	13	8	0.6K	1K	40	61	1.5K
C13	Package Managers, Build Utilities, and Loaders	11	47	3.4K	5.4K	200	300	8.5K
C14	Storage Tools and Libraries	10	19	4K	7K	222	317	5.5K
C15	Testing Frameworks and Libraries	10	28	2.8K	3.6K	271	632	5.7K
C16	Browser and DOM Utilities	9	45	5.6K	7.1K	76	179	5.2K
C17	Command-line Interface and Shell Tools	9	9	2.8K	1K	26	244	2.6K
C18	Multimedia Utilities	9	11	1.6K	760	17	97	6.2K
C19	MVC Frameworks	9	174	40.1K	15.2K	657	1401	14.2K
	Client-side	128	39	8.2K	3.2K	343	798	7.9K
	Server-side	130	63	9.4K	7.2K	231	505	6.7K
	Client and server-side	115	73	12.7K	4.7K	221	402	7.4K
	Total	373	57	10.1K	4.5K	263	644	7.3K

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[2], 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), C19 (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.

[2]https://nodejs.org

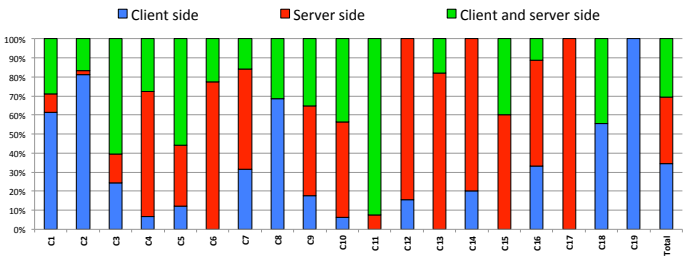


Fig. 1: Distribution of studied subject systems.

B. 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 namely *tests*, *specs*[3], 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.

[3]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")` returns `null` as the expected element is not available, and thus `checkList.children` 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`,

```
1 function setFontSize(size) {
2     return function() {
3         // this is an anonymous closure
4         document.body.style.fontSize = size + 'px';
5     };
6 }
7 var small = setFontSize(12);
8 var large = setFontSize(16);
9 ...
10 function showMsg() {
11     // this is an async callback
12     alert("Some message goes here!");
13 }
14 ...
15 $("#smallBtn").on("click", small);
16 $("#largeBtn").on("click", large);
17 $("#showBtn").on("click", function() {
18     // this is an event-dependent anonymous callback
19     setTimeout(showMsg, 2000);
20     $("#photo").fadeIn("slow", function() {
21       // this is an anonymous callback
22       alert("Photo animation complete!");
23     });
24 });
25 ...
26 checkList = $("#checkList");
27 checkList.children("input").each(function () {
28     // this is an DOM-related code
29     if (this.is(':checked')) {
30     ...
31     }else{
32     ...
33     }
34 });
```

Fig. 2: A hard to test JavaScript code snippet.

`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 callback functions. In order to detect event-dependent callbacks, our tool checks if a callback function is an event method such as `bind`, `click`, `focus`, `hover`, `keypress`, `emit`, `addEventListener`, `onclick`, `onmouseover`, and `onload`.

Asynchronous callback coverage. Callbacks are functions passed as an argument to another function to be invoked either immediately (synchronous) or at some point in the future (asynchronous) after the enclosing function returns. Callbacks are particularly useful to perform non-blocking operations. Function `showMsg` in lines 10–13 is an *asynchronous callback* function as it was passed to the `setTimeout()` asynchronous API call. Testing *asynchronous callbacks* requires waiting until the callback is called, otherwise the test would probably finish unsuccessfully before the callback is invoked. For instance QUnit's `asyncTest` allows tests to wait for asynchronous callbacks to be called.

Asynchronous callback coverage is defined as the fraction of number of covered over total number of asynchronous callback functions. Similar to a study of callbacks in JavaScript [21], if a callback argument is passed into a known deferring API call we count it as as an asynchronous callback. TESTSCANNER detects some asynchronous APIs including network calls (e.g. `XMLHTTPRequest.open`), DOM events (e.g. `onclick`), timers (`setImmediate`, `setTimeout`, `setInterval`, and `process.nextTick`), and I/O (e.g. APIs of `fs`, `http`, and net).

Closure function coverage. Closures are nested functions that make it possible to create hidden scope to privatize variables and functions from the global scope in JavaScript. A closure function, i.e., the inner function, has access to all parameters and variables – except for `this` and `argument` variables – of the outer function, even after the outer function has returned [20]. The anonymous function in lines 2–5 is an instance of a *closure*.

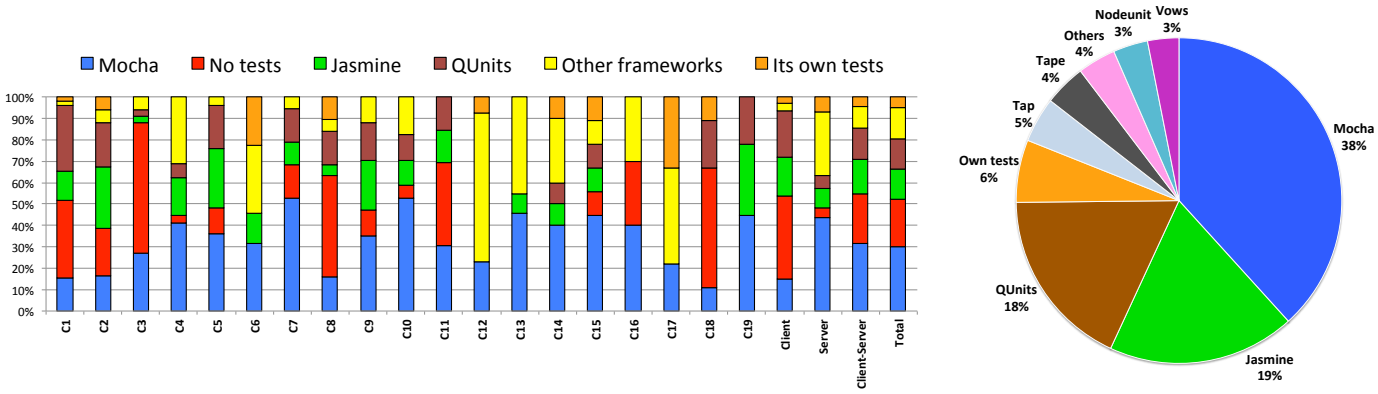
Such hidden functions cannot be called directly in a test case and thus testing them is challenging. In fact writing a unit test for a closure function without code modification is impossible. Simple solutions such as making them public or putting the test code inside the closure are not good software engineering practices. One approach to test such private functions is adding code inside the closure to store references to its local variables inside objects and return it to the outer scope [2]. *Closure function coverage* is defined as the fraction of number of covered over total number of closure functions.

Average number of function calls per test. Some code functionalities depend on the execution of a sequence of function calls. For instance in a shopping application, one needs to add items to the cart prior to check out. We perform a correlation analysis between average number of unique function calls per test and code coverage. We also investigate whether JavaScript unit tests are mostly written at single function level or they execute sequence of function calls.

III. RESULTS

A. Prevalence of Tests (RQ1)

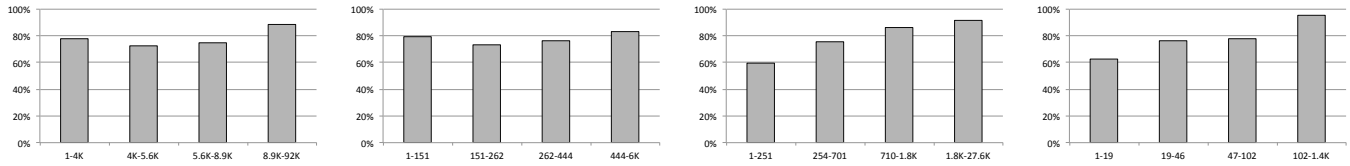
The stacked bar charts in Figure 3(a) depicts the percentage of JavaScript tests, per system category (Table I), per client/server side, and in aggregate. The height of each bar indicates



(a) Distribution within all subjects.

(b) Testing frameworks distribution.

Fig. 3: Distribution of JavaScript tests.



(a) Number of stars

(b) Number of watchers

(c) Number of commits

(d) Number of contributors

Fig. 4: Percentage of subjects with test per each quartile with respect to popularity (number of stars and watchers) and maturity (number of commits and contributors).

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*, *UTest*, *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.

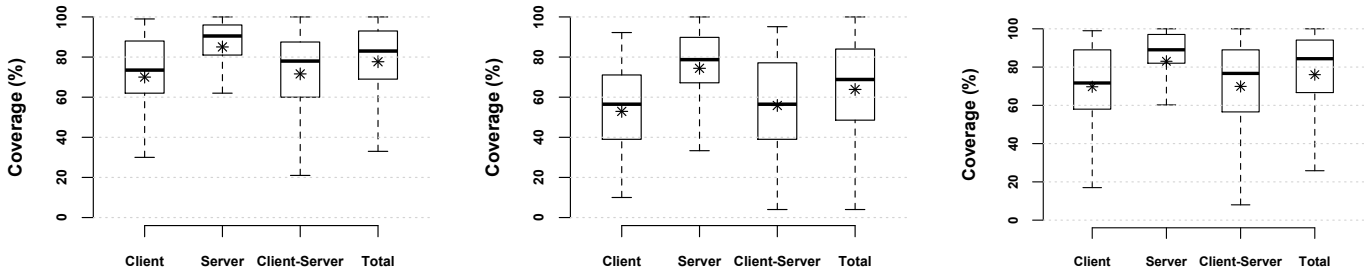
***Finding 3**: Only 3% of the studied repositories have functional UI tests.*

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 [16]



(a) Statement coverage (b) Branch coverage (c) Function coverage

Fig. 5: Boxplots of the code coverage of the executed JavaScript tests. Mean values are shown with (*).

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 (storage), C15 (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.*

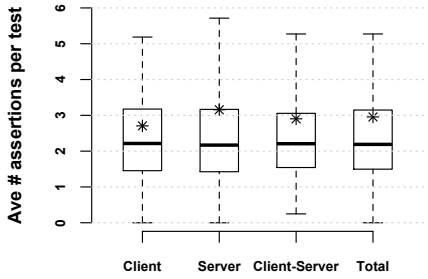


Fig. 6: Average number of assertions per test.

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.*

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.

> ***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.

		Statement coverage	Branch coverage	Function coverage	Ave # assertions per test	Test code ratio	Test commit ratio
Subject category	C1	77%	57%	76%	2.83	0.41	0.16
	C2	67%	52%	65%	2.72	0.28	0.14
	C3	60%	38%	58%	3.75	0.88	0.14
	C4	79%	68%	78%	2.50	0.58	0.24
	C5	75%	63%	75%	2.53	0.52	0.21
	C6	87%	79%	88%	2.53	0.47	0.24
	C7	80%	67%	72%	2.51	0.46	0.22
	C8	64%	47%	60%	2.04	0.35	0.12
	C9	73%	58%	69%	2.67	0.49	0.23
	C10	91%	79%	90%	2.73	0.72	0.24
	C11	64%	45%	57%	3.41	0.18	0.11
	C12	90%	77%	89%	2.36	0.59	0.20
	C13	86%	67%	84%	2.27	0.59	0.18
	C14	88%	77%	87%	2.74	0.62	0.26
	C15	81%	69%	79%	5.79	0.59	0.25
	C16	78%	67%	79%	1.67	0.49	0.29
	C17	67%	54%	63%	8.32	0.47	0.21
	C18	60%	31%	62%	4.42	0.31	0.16
	C19	81%	67%	80%	3.58	0.53	0.21
Testing framework	Mocha	82%	70%	79%	2.39	0.49	0.20
	Jasmine	74%	60%	75%	1.93	0.41	0.21
	QUnit	71%	54%	71%	3.93	0.41	0.16
	Own test	61%	41%	58%	5.99	0.30	0.16
	Tap	89%	80%	89%	1.56	0.58	0.21
	Tape	93%	81%	94%	2.93	0.70	0.18
	Others	80%	65%	77%	5.60	0.46	0.24
	Nodeunit	74%	63%	72%	6.20	0.57	0.24
	Vows	74%	66%	72%	1.92	0.55	0.27
	Client	70%	53%	70%	2.71	0.36	0.16
	Server	85%	74%	83%	3.16	0.58	0.23
	C&S	72%	56%	70%	2.9	0.4	0.18
	Total	78%	64%	76%	2.96	0.46	0.2

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[4]. The result shows that there exists a moderate to strong correlation (ρ = 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 (ρ = 0.68, p = 0) between test code ratio and code coverage.*

[4]The 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.

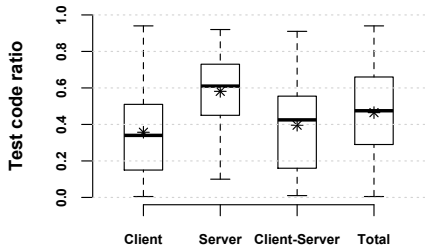


Fig. 7: Test to total code ratio.

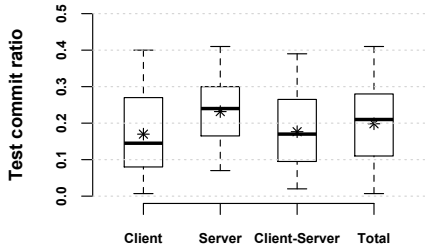


Fig. 8: Test to total commits ratio.

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.

		Function coverage				Statement coverage			
		All	Async callback	Event dependent callback	Closure	All	DOM related	Ave # func calls per test	USUF ratio
Subject category	C1	76%	65%	33%	79%	77%	73%	2.91	0.59
	C2	65%	43%	17%	62%	67%	61%	2.82	0.73
	C3	58%	21%	10%	38%	60%	27%	3.94	0.82
	C4	79%	49%	48%	70%	81%	75%	2.89	0.53
	C5	75%	52%	33%	65%	75%	62%	3.05	0.72
	C6	88%	60%	32%	87%	87%	57%	3.30	0.33
	C7	72%	34%	28%	81%	80%	52%	3.11	0.52
	C8	60%	40%	39%	80%	64%	78%	2.18	0.77
	C9	69%	14%	8%	80%	73%	23%	2.89	0.59
	C10	90%	65%	60%	95%	91%	81%	4.98	0.5
	C11	57%	33%	7%	68%	64%	51%	2.79	0.85
	C12	89%	85%	68%	98%	90%	85%	3.56	0.32
	C13	84%	71%	74%	60%	86%	85%	2.86	0.49
	C14	87%	66%	36%	89%	88%	–	2.98	0.62
	C15	79%	70%	39%	62%	81%	58%	2.16	0.59
	C16	79%	40%	5%	43%	78%	48%	2.69	0.39
	C17	63%	7%	5%	56%	67%	–	2.42	0.65
	C18	62%	–	0%	89%	60%	40%	2.19	0.86
	C19	81%	61%	47%	76%	82%	62%	2.92	0.53
Testing framework	Mocha	79%	50%	34%	71%	82%	58%	3.62	0.56
	Jasmine	75%	65%	34%	69%	74%	62%	2.28	0.71
	QUnit	71%	53%	28%	76%	71%	68%	3.35	0.66
	Own test	58%	45%	26%	66%	61%	51%	1.78	0.63
	Tap	89%	68%	87%	94%	89%	–	2.52	0.24
	Tape	94%	79%	65%	92%	93%	88%	3.19	0.22
	Others	77%	33%	30%	66%	80%	79%	2.14	0.48
	Nodeunit	72%	53%	63%	74%	74%	52%	4.08	0.62
	Vows	72%	60%	38%	79%	74%	0%	1.60	0.6
	Client	70%	46%	25%	69%	70%	66%	2.96	0.68
	Server	83%	64%	48%	82%	85%	67%	3.19	0.45
	C&S	70%	48%	29%	69%	72%	57%	2.93	0.69
	Total	76%	53%	36%	74%	78%	63%	3.05	0.57

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 (ρ = 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 (ρ = 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 indicate 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-B3. 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 async tests on asynchronous callback coverage through correlation analysis. The results show that there exists a weak correlation (ρ = 0.22) between number of event triggers and event-dependent callback coverage, and a very weak correlation (ρ = 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 ρ = 0.4, p = 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 (ρ = 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 (ρ = 0.13, p = 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 [37] 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 [18]. 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-sideJavaScript 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

[1] Examples of hard to test JavaScript. https://www.pluralsight.com/blog/software-development/6-examples-of-hard-to-test-javascript.
[2] How to unit test private functions in JavaScript. https://philipwalton.com/articles/how-to-unit-test-private-functions-in-javascript/.
[3] Istanbul - a JS code coverage tool written in JS. https://github.com/gotwarlost/istanbul.
[4] Jasmine. https://github.com/pivotal/jasmine.

[5] Jscover. http://tntim96.github.io/JSCover/.
[6] Mocha. https://mochajs.org/.
[7] Mozilla Rhino. https://github.com/mozilla/rhino.
[8] Nodeunit. https://github.com/caolan/nodeunit.
[9] QUnit. http://qunitjs.com/.
[10] Which JavaScript test library should you use? http://www.techtalkdc.com/which-javascript-test-library-should-you-use-qunit-vs-jasmine-vs-mocha/.
[11] Writing testable code in JavaScript: A brief overview. https://www.toptal.com/javascript/writing-testable-code-in-javascript.
[12] Writing testable JavaScript. http://www.adequatelygood.com/Writing-Testable-JavaScript.html.
[13] Github Showcases. https://github.com/showcases, 2014.
[14] JSter JavaScript Libraries Catalog. http://jster.net/catalog, 2014.
[15] Most depended-upon NMP packages. https://www.npmjs.com/browse/depended, 2014.
[16] Testing and deploying with ordered npm run scripts. http://blog.npmjs.org/post/127671403050/testing-and-deploying-with-ordered-npm-run-scripts, 2015.
[17] SLOC (source lines of code) counter. https://github.com/flosse/sloc/, 2016.
[18] TestScanner. https://github.com/saltlab/testscanner, 2016.
[19] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 571–580. ACM, 2011.
[20] D. Crockford. *JavaScript: the good parts*. O'Reilly Media, Incorporated, 2008.
[21] K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 247–256. IEEE Computer Society, 2015.
[22] GitHut. A small place to discover languages in GitHub. http://githut.info, 2016.
[23] P. Heidegger and P. Thiemann. Contract-driven testing of Javascript code. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 154–172. Springer-Verlag, 2010.
[24] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014.
[25] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 270–283. ACM, 2010.
[26] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.
[27] G. Li, E. Andreasen, and I. Ghosh. SymJS: Automatic symbolic testing of JavaScript web applications. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, page 11 pages. ACM, 2014.
[28] A. Milani Fard and A. Mesbah. JSNose: Detecting JavaScript code smells. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE Computer Society, 2013.
[29] A. Milani Fard, A. Mesbah, and E. Wohlstadter. Generating fixtures for JavaScript unit testing. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 190–200. IEEE Computer Society, 2015.
[30] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 67–78. ACM, 2014.
[31] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient JavaScript mutation testing. In *Proc. of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2013.
[32] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Jseft: Automated JavaScript unit test generation. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, page 10 pages. IEEE Computer Society, 2015.
[33] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Atrina: Inferring unit oracles from GUI test cases. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, page 11 pages. IEEE Computer Society, 2016.
[34] H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mining interprocedural, data-oriented usage patterns in JavaScript web applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 791–802. ACM, 2014.
[35] 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 2012 ACM conference on Computer and communications security*, pages 736–747. ACM, 2012.
[36] 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)*, pages 55–64. IEEE Computer Society, 2013.
[37] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. A study of causes and consequences of client-side JavaScript bugs. *IEEE Transactions on Software Engineering (TSE)*, page 17 pages, 2017.
[38] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.
[39] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *ECOOP 2011–Object-Oriented Programming*, pages 52–78. Springer, 2011.
[40] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2010.
[41] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the Symposium on Security and Privacy*, pages 513–528. IEEE Computer Society, 2010.
[42] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 488–498. ACM, 2013.
[43] Stack Overflow. 2016 Developer Survey. http://stackoverflow.com/research/developer-survey-2016, 2016.
[44] M. E. Trostler. *Testable JavaScript*. O'Reilly Media, Incorporated, 2013.
[45] A. Vahabzadeh, A. Milani Fard, and A. Mesbah. An empirical study of bugs in test code. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110. IEEE Computer Society, 2015.
[46] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. An empirical analysis of XSS sanitization in web application frameworks. *Electrical Engineering and Computer Sciences University of California at Berkeley, Technical Report*, pages 1–17, 2011.
[47] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 961–970. ACM, 2009.
[48] A. Zaidman, B. van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production and test code. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 220–229, 2008.
[49] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 214–224. ACM, 2015.