SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. (2020)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1751

BUGSJS: a benchmark and taxonomy of JavaScript bugs

Péter Gyimesi[1,*,†], Béla Vancsics[1], Andrea Stocco[2], Davood Mazinanian[3], Árpád Beszédes[1], Rudolf Ferenc[1] and Ali Mesbah[3]

[1]*Department of Software Engineering, University of Szeged, Szeged, Hungary*
[2]*Software Institute, Università della Svizzera italiana, Lugano, Switzerland*
[3]*Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC Canada*

SUMMARY

JavaScript is a popular programming language that is also error-prone due to its asynchronous, dynamic, and loosely typed nature. In recent years, numerous techniques have been proposed for analyzing and testing JavaScript applications. However, our survey of the literature in this area revealed that the proposed techniques are often evaluated on different datasets of programs and bugs. The lack of a commonly used benchmark limits the ability to perform fair and unbiased comparisons for assessing the efficacy of new techniques. To fill this gap, we propose BUGSJS, a benchmark of 453 real, manually validated JavaScript bugs from 10 popular JavaScript server-side programs, comprising 444k lines of code (LOC) in total. Each bug is accompanied by its bug report, the test cases that expose it, as well as the patch that fixes it. We extended BUGSJS with a rich web interface for visualizing and dissecting the bugs' information, as well as a programmable API to access the faulty and fixed versions of the programs and to execute the corresponding test cases, which facilitates conducting highly reproducible empirical studies and comparisons of JavaScript analysis and testing tools. Moreover, following a rigorous procedure, we performed a classification of the bugs according to their nature. Our internal validation shows that our taxonomy is adequate for characterizing the bugs in BUGSJS. We discuss several ways in which the resulting taxonomy and the benchmark can help direct researchers interested in automated testing of JavaScript applications.

Received 11 November 2019; Revised 30 April 2020; Accepted 21 July 2020

KEY WORDS: JavaScript; bug database; benchmark; reproducibility; bug taxonomy; BugsJS

1. INTRODUCTION

JavaScript (JS) is the de-facto web programming language globally,[‡] and the most adopted language on GitHub.[§] JavaScript is massively used in the client-side of web applications to achieve high responsiveness and user friendliness. In recent years, due to its flexibility and effectiveness, it has been increasingly adopted also for server-side development, leading to full-stack web applications [1]. Platforms such as Node.js[¶] allow developers to conveniently develop both the front-end and back-end of the applications entirely in JS.

Despite its popularity, the intrinsic characteristics of JS—such as weak typing, prototypal inheritance, and run-time evaluation—make it one of the most error-prone programming languages. As such, a large body of software engineering research has focused on the analysis and testing of JS web applications [2-9].

[‡]https://insights.stackoverflow.com/survey/2019
[§]https://octoverse.github.com
[¶]https://nodejs.org/en/

*Correspondence to: Péter Gyimesi, Department of Software Engineering, University of Szeged, Szeged, Hungary.
[†]E-mail: pgyimesi@inf.u-szeged.hu

Existing research techniques are typically evaluated through empirical methods (e.g., controlled experiments), which need software-related artifacts, such as source code, test suites, and descriptive bug reports. To date, however, most of the empirical works and tools for JS have been evaluated on different datasets of subjects. Additionally, subject programs or accompanying experimental data are rarely made available in a detailed, descriptive, curated, and coherent manner. This not only hampers the reproducibility of the studies themselves but also makes it difficult for researchers to assess the state-of-the-art of related research and to compare existing solutions.

Specifically, testing techniques are typically evaluated with respect to their effectiveness at detecting faults in existing programs. However, real bugs are hard to isolate, reproduce, and characterize. Therefore, the common practice relies on manually seeded faults or mutation testing [10]. Each of these solutions has limitations. Manually injected faults can be biased toward researchers' expectations, undermining the representativeness of the studies that use them. Mutation techniques, on the other hand, allow generating a large number of 'artificial' faults. Although research has shown that mutants are quite representative of real bugs [11-13], mutation testing is computationally expensive to use in practice. For these reasons, benchmarks of manually validated bugs are of paramount importance for devising novel debugging, fault localization, or program repair approaches.

Several benchmarks of bugs have been proposed and largely utilized by researchers to advance testing research. Notable instances are the Software-artifact Infrastructure Repository (SIR) [14], Defects4J [15], ManyBugs [16], and BugSwarm [17]. Purpose-specific test and bug datasets also exist to support studies in program repair [18], test generation [19], and security [20]. However, to date, a well-organized repository of labeled JS bugs is still missing. The plethora of different JS implementations available (e.g., V8, JavaScriptCore, Rhino) further makes devising a cohesive bugs benchmark nontrivial.

In our previous work [21], we presented BUGSJS, a benchmark of 453 JS-related bugs from 10 open-source JS projects, based on Node.js and the Mocha testing framework. BUGSJS features an infrastructure containing detailed reports about the bugs, the faulty versions of programs, the test cases exposing them, as well as the patches that fix them.

This article is a revised and expanded version of our conference paper [21]. We provide details on the differences between the prior paper and this article. From the *technical* standpoint, first we added a port 'dissection' to BUGSJS, that is, a web interface to inspect/query information about the bugs. Second, we enriched the API with the possibility of conducting more fine-grained analysis with an optimized command that retrieves the coverage for each individual test (per-test coverage). We also added more precomputed data from the source code, test cases, and executions to better facilitate related research. Concerning the *intellectual* contributions, we performed a classification of the bugs in BUGSJS, which was missing in the initial conference paper. We constructed our taxonomy using faceted classification [22], that is, we created the categories/sub-categories of our taxonomy in a bottom-up fashion, by analyzing different sources of information about the bugs.

This article makes the following contributions:

Survey.	A survey of the previous work on analysis and testing of JS applications, revealing the lack of a comprehensive benchmark of JS programs and bugs to support empirical evaluation of the proposed techniques.
Dataset.	BUGSJS, a benchmark of 453 manually selected and validated JS bugs from 10 JS Node.js programs pertaining to the Mocha testing framework.
Framework.	A Docker-based infrastructure to download, analyze, and run test cases exposing each bug in BUGSJS and the corresponding real fixes implemented by developers. The infrastructure includes a web-based dashboard and a set of precomputed data from the subjects and tests as well.
Taxonomy.	A qualitative analysis of BUGSJS resulting in a bug taxonomy of server-side JS bugs, which, to our knowledge, is the first of this kind.
Evaluation.	A quantitative and qualitative analysis of the bug fixes related to the BUGSJS bugs in relation to existing classification schemes.

2. STUDIES ON JAVASCRIPT ANALYSIS AND TESTING

To motivate the need for a novel benchmark for JS bugs, we surveyed the works related to software analysis and testing in the JS domain. Our review of the literature also allowed us to gain insights about the most active research areas in which our benchmark should aim to be useful.

In the JS domain, the term benchmark commonly refers to collections of programs used to measure and test the *performance* of web browsers with respect to the latest JS features and engines. Instances of such performance benchmarks are JetStream,[‖] Kraken,[**] Dromaeo,[††] Octane,[‡‡] and V8.[§§] In this work, however, we refer to *benchmark* as a collection of JS programs and artifacts (e.g., test cases or bug reports) used to support empirical studies (e.g., controlled experiments or user studies) related to one or more research areas in software analysis and testing.

We used the databases of scientific academic publishers and popular search engines to look for papers related to different software analysis and testing topics for JS. We adopted various combinations of keywords: `JavaScript`, `testing` (including code coverage measurement, mutation testing, test generation, unit testing, test automation, regression testing), `bugs` and `debugging` (including fault localization, bug, and error classification), and `web`. We also performed a lightweight forward and backward snowballing [23] to mitigate the risk of omitting relevant literature. Last, we examined the evaluation section of each paper. We retained only papers in which real-world, open-source JS projects were used, whose repositories and versions could be clearly identified. This yielded 25 final papers. Nine of these studies are related to bugs, in which 670 subjects were used in total. The remaining 16 papers are related to other testing fields, comprising 494 subjects in total.

In presenting the results of our survey of the literature, we distinguish (i) studies containing specific bug information and other artifacts (such as source code and test cases), and (ii) studies containing only JS programs and other artifacts not necessarily related to bugs.

2.1. Bug-related studies for JavaScript

We analyzed papers using JS systems that include bug data in greater detail, because these works can provide us important insights about the kind of analysis researchers used the subjects for, and thus, the requirements that a new benchmark of bugs should adhere to.

We found nine studies in this category. Ocariza et al. [9] present an analysis and classification of bug reports to understand the root causes of client-side JS faults. This study includes 502 bugs from 19 projects with over 2MLOC. The results of the study highlight that the majority (68%) of JS faults are caused by faulty interactions of the JS code with the Document Object Model (DOM). Moreover, most JS faults originate from programmer mistakes committed in the JS code itself, as opposed to other web application components.

Another bug classification presented by Gao et al. [24] focuses on type system-related issues in JS (which is a dynamically typed language). The study includes about 400 bug reports from 398 projects with over 7MLOC. The authors ran a static type checker such as Facebook's Flow[‖‖] or Microsoft's TypeScript[***] on the faulty versions of the programs. On average, 60 out of 400 bugs were detected (15%), meaning they may have been avoided in the first place if a static type checker were used to warn the developer about the type-related bug.

Hanam et al. [25] present a study of cross-project bug patterns in server-side JS code, using 134 Node.js projects of about 2.5MLOC. They propose a technique called BugAID for discovering such bug patterns. BugAID builds on an unsupervised machine learning technique that learns the most common code changes obtained through AST differencing. Their study revealed 219 bug

[‖]https://browserbench.org/JetStream
[**]https://wiki.mozilla.org/Kraken
[††]https://wiki.mozilla.org/Dromaeo
[‡‡]https://developers.google.com/octane
[§§]https://github.com/hakobera/node-v8-benchmark-suite
[‖‖]https://flow.org/
[***]https://www.typescriptlang.org/

fixing change types and 13 pervasive bug patterns that occur across multiple projects. In our evaluation, we conduct a thorough comparison with Hanam et al.'s taxonomy.

Ocariza et al. [26] propose an inconsistency detection technique for model-view-controller-based JS applications which is evaluated on 18 bugs from 12 web applications (7k LOC). A related work [27] uses 15 bugs in 20 applications (nearly 1M LOC). They also present an automated technique to localize JS faults based on a combination of dynamic analysis, tracing, and backward slicing, which is evaluated on 20 bugs from 15 projects (14k LOC) [28]. Also, their technique for suggesting repairs for DOM-based JS faults is evaluated on 22 bugs from 11 applications (1M LOC) [29].

Wang et al. [3] present a study on 57 concurrency bugs in 53 Node.js applications (about 3.5M LOC). The paper proposes several different analyses pertaining to the retrieved bugs, such as bug patterns, root causes, and repair strategies. Davis et al. [30] propose a fuzzing technique for identifying concurrency bugs in server-side event-driven programs and evaluate their technique on 12 real-world programs (around 216k LOC) and 12 manually selected bugs.

2.2. Other analysis and testing studies for JavaScript

Empirical studies in software analysis and testing benefit from a large variety of software artifacts other than bugs, such as test cases, documentation, or code revision history. In this section, we briefly describe the remaining papers of our survey.

Milani Fard and Mesbah [31] characterize JS tests in 373 JS projects according to various metrics, for example, code coverage, test commits ratio, and number of assertions.

Mirshokraie et al. propose several approaches to JS automated testing. This includes an *automated regression testing* based on dynamic analysis, which is evaluated on nine web applications [32]. The authors also propose a *mutation testing* approach, which is evaluated on seven subjects [33] and on eight applications in a related work [34]. They also propose a technique to aid *test generation* based on program slicing [35], where unit-level assertions are automatically generated for testing JS functions. Seven open-source JS applications are used to evaluate their technique. The authors also present a related approach for JS unit test case generation, which is evaluated on 13 applications [36].

Adamsen et al. [37] present a hybrid static/dynamic program analysis method to check *code coverage-based properties* of test suites from 27 programs. Dynamic symbolic execution is used by Milani Fard et al. [38] to generate DOM-based *test fixtures and inputs* for unit testing JS functions, and four experimental subjects are used for evaluation. Ermuth and Pradel propose a GUI test generation approach [7], and evaluate it on four programs.

Artzi et al. [39] present a framework for *feedback-directed automated test generation* for JS web applications. In their study, the authors use 10 subjects. Mesbah et al. [40] present Atusa, a *test generation technique for Ajax-based applications*, which they evaluate on six web applications. A comprehensive survey of *dynamic analysis and test generation* for JS is presented by Andreasen et al. [41].

Billes et al. [8] present a black-box analysis technique for multi-client web applications to detect *concurrency errors* on three real-world web applications. Hong et al. [42] present a testing framework to detect concurrency errors in client-side web applications written in JS and use five real-world web applications.

Wang et al. [2] propose a modification to the *delta debugging* approach that reduces the event trace, which is evaluated on 10 real-world JS application failures. Dhok et al. [43] present a *concolic testing* approach for JS which is evaluated on 10 subjects.

2.3. Findings

In the surveyed papers, we observed that the proposed techniques were evaluated using different sets of programs, with little to no overlap.

Table I shows the program distribution per paper. In bug-related studies, 633 subject programs were adopted overall, with 607 of these programs (96%) were used in only one study, and no subject was used in more than four papers (Table I, columns 1 and 2). Other studies exhibit the same

Table I. Subject distribution among surveyed papers

Bug-related		All studies	
# Papers	# Subjects	# Papers	# Subjects
1	607	1	910
2	17	2	91
3	7	3	17
4	2	4	4
5	0	5	1

trend (Table I, columns 3 and 4): overall, 1,164 subjects were used in all the investigated papers, of which 1,023 were unique. From these, 910 (89%) were used in only one paper, and no subject was used in more than five papers.

In conclusion, we observe that the investigated studies involve different sets of programs, since no centralized benchmark is available to support reproducible experiments in analysis and testing related to JS bugs.

To devise a centralized benchmark for JS bugs that enables reproducibility studies in software analysis and testing, we considered the insights and guidelines provided by existing similar datasets (e.g., Defects4J [15]), as well as the knowledge gained from our study of the literature on empirical experiments using JS programs and bugs.

First, the considered subject systems should be real-world, publicly available open-source JS programs. To ensure representativeness of the benchmark, they should be diverse in terms of the application domain, size, development, testing, and maintenance practices (e.g., the use of continuous integration CI or code review process).

Second, the *buggy versions* of the programs must have one or more *test cases* available demonstrating each bug. The bugs must be reproducible under reasonable constraints; this excludes non-deterministic or flaky features.

Third, the versions of the programs in which the bugs were fixed by developers, that is, the *patches*, must be also available. Typically, when a bug is fixed, new unit tests are also added (often in the same bug-fixing commit) to cover the buggy feature, allowing for better regression testing. This allows extracting the bug-fixing changes, for example, by *diffing* the buggy and fixed revisions.

Additionally, the benchmark should include the bug report information, including critical times (e.g., when the bug was opened, closed, or reopened), the discussions about each bug, and link to the commits where the bug was fixed.

To fill this gap, in Section 3, we overview BUGSJS [21], a benchmark of real JS bugs, its design and implementation.

3. BUGSJS—THE PROPOSED BENCHMARK

To construct a benchmark of real JS bugs, we identify existing bugs from the programs' version control histories and collect the real fixes provided by developers. Developers often manually label the revisions of the programs in which reported bugs are fixed (*bug-fixing commits* or patches). As such, we refer to the revision preceding the bug-fixing commit as the *buggy commit*. This allowed us to extract detailed bug reports and descriptions, along with the buggy and bug-fixing commits they refer to. Particularly, each bug and fix should adhere to the following properties:

- **Reproducibility.** One or more *test cases* are available in a *buggy commit* to demonstrate the bug. The bug must be reproducible under reasonable constraints. We excluded non-deterministic features and flaky tests from our study, since replicating them in a controlled environment would be excessively challenging.
- **Isolation.** The bug-fixing commit applies to JS source code files only; changes to other artifacts such as documentation or configuration files are not considered. The source code of each commit must be *cleaned* from irrelevant changes (e.g., feature implementations, refactorings,

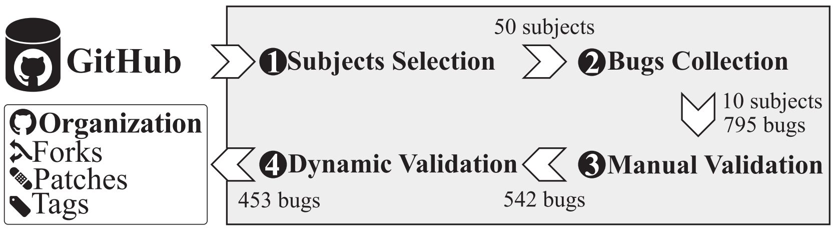


Figure 1. Overview of the bug selection and inclusion process.

and changes to non-JS files). The *isolation* property is particularly important in research areas where the presence of noise in the data has detrimental impacts on the techniques (e.g., automated program repair, or fault localization approaches).

Figure 1 depicts the main steps of the process we performed to construct our benchmark. First, we adopted a systematic procedure to select the JS subjects to extract the bug information from ❶. Then, we collected bug candidates from the selected projects ❷, and manually validated each bug for inclusion by means of multiple criteria ❸. Next, we performed a dynamic sanity check to make sure that the tests introduced in a bug-fixing commit can detect the bug in the absence of its fix ❹. Finally, the retained bugs were cleaned from irrelevant patches (e.g.,whitespaces).

3.1. Subject systems selection

To select relevant programs to include in BUGSJS, we focused on popular and trending JS projects on GitHub. Such projects often engage large communities of developers and therefore are more likely to follow software development best practices, including bug reporting and tracking. Moreover, GitHub's *issue IDs* allow conveniently connecting bug reports to bug-fixing commits.

Popularity was measured using the projects' *Stargazers count* (i.e., the number of stars owned by the subject's GitHub repository). We selected server-side Node.js applications which are popular (Stargazers count $\geq$100) and mature (number of commits $>$200) and have been actively maintained (year of the latest commit $\geq$2017). We currently focus on Node.js because it is emerging as one of the most pervasive technologies to enable using JS in the server side, leading to the so-called full-stack web applications [1]. Limiting the subject systems to server-side applications and specific testing frameworks is due to technological constraints, as running tests for browser-based programs would require managing many complex and time-consuming configurations. We discuss the potential implications of this constraint in Section 6.

We examined the GitHub repository of each retrieved subject system to ensure that bugs were properly tracked and labeled. Particularly, we only selected projects in which bug reports had a dedicated *issue label* on GitHub's *Issues* page, which allows filtering irrelevant issues (pertaining to, for example, feature requests, build problems, or documentation), so that only *actual bugs* are included. Our initial list of subjects included 50 Node.js programs, from which we filtered out projects based on the number of candidate bugs found and the adopted testing frameworks.

3.2. Bugs collection

Collecting bugs and bug-fixing commits.

For each subject system, we first queried GitHub for *closed issues* assigned with a specific bug label using the official GitHub's API.[†††] For each closed bug, we exploit the *links* existing between issues and commits to identify the corresponding bug-fixing commit. GitHub automatically detects these links when there is a specific keyword (belonging to a predefined list[‡‡‡]), followed by an issue ID (e.g., `Fixes #14`).

Each issue can be linked to zero, one, or more source code commits. A closed bug without a bug-fixing commit could mean that the bug was rejected (e.g., it cannot be replicated) or that

[†††]https://developer.github.com/v3/
[‡‡‡]https://help.github.com/articles/closing-issues-using-keywords/

Table II. Subjects included in BUGSJS

	Stats (#)				Tests (#)				Coverage (%)			
	kLOC (JS)	**Stars**	**Commits**	**Forks**	**All**	**Passing**	**Pending**	**Failing**	**Statements**	**Branches**	**Functions**	**Lines**
BOWER	16	15,290	2,706	1,995	455	103	19	36	81.11	66.91	80.62	81.11
ESLINT	240	12,434	6,615	2,141	18,528	18,474	0	54	99.21	98.19	99.72	99.21
EXPRESS	11	40,407	5,500	7,055	855	855	0	0	98.71	94.32	100	99.95
HESSIAN.JS	6	104	217	23	225	223	2	0	96.42	91.27	98.99	96.42
HEXO	17	23,748	2,545	3,277	875	868	7	0	96.20	90.51	98.54	97.27
KARMA	12	10,210	2,485	1,531	331	331	0	0	54.61	34.03	43.98	54.76
MONGOOSE	65	17,036	9,770	2,457	2,107	2,071	36	0	90.97	85.95	89.65	91.04
NODE-REDIS	11	10,349	1,242	1,245	966	965	0	1	99.06	98.19	97.99	99.06
PENCILBLUE	46	1,596	3,675	276	807	802	0	5	35.21	19.09	22.91	35.22
SHIELDS	20	6,319	2,036	1,432	482	469	13	0	75.98	65.60	83.26	75.97

developers did not associate that issue with any commit. We discarded such bugs from our benchmark, as we require each bug to be identifiable by its bug-fixing commit. At last, similarly to existing benchmarks [15], we discarded bugs linked to more than one bug-fixing commit, as this might imply that they were fixed in multiple steps or that the first attempt for fixing them was unsuccessful.

Including corresponding tests.

We require each fixed bug to have *unit tests* that demonstrate the absence of the bug. To meet this requirement, we examined the bug-fixing patches to ensure they also contain changes or additions in the test files. For this filtering, we manually examined each patch to determine whether test files were involved. The result of this step is the list of *bug candidates* for the benchmark. From the initial list of 50 subject systems, we considered the projects having at least 10 bug candidates.

Testing frameworks.

There are several testing frameworks available for JS applications. We collected statistics about the testing frameworks used by the 50 considered JS projects. Our results show that there is no single predominant testing framework for JS (as compared to, for instance, JUnit which is used by most Java developers). We found that the majority of tests in our pool were developed using Mocha[§§§] (52%), Jasmine[¶¶¶] (10%), and QUnit[‖‖‖] (8%). Consequently, the initial version of BUGSJS only includes projects that use Mocha, whose prevalence as JS testing framework is also supported by a recent large-scale empirical study [31].

Final selection.

Table II reports the names and descriptive statistics of the 10 applications we ultimately retained. Notice that all these applications have at least 1,000 LOC (frameworks excluded), thus being representative of modern web applications (Ocariza et al. [9] report an average of 1,689 LOC for AngularJS web applications on GitHub with at least 50 stars).

The subjects represent a wide range of domains. *Bower* is a front-end package management tool that exposes the package dependency model through an API. *Express* is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. *Hessian.js* is a JS binary web service protocol that makes web services usable without requiring a large framework and without learning a new set of protocols. *Hexo* is a blog framework powered by Node.js. *Karma* is a popular framework agnostic test runner tool for JS. *Mongoose* is a MongoDB object modeling tool for Node.js. *Node-redis* is a Node.js client for Redis database. *Pencilblue* is a content management system (CMS) and blogging platform, powered by Node.js. *Shields* is a web service for badges in SVG and raster format.

[§§§]https://mochajs.org/
[¶¶¶]https://jasmine.github.io/
[‖‖‖]https://qunitjs.com/

3.3. Manual patch validation

We manually investigated each bug and the corresponding bug-fixing commit to ensure that only bugs meeting certain criteria are included, as described below.

Methodology.

Two authors of this paper manually investigated each bug and its corresponding bug-fixing commit and labeled them according to a well-defined set of *inclusion criteria* (Table III). The bugs that met all criteria were initially marked as 'Candidate Bug' to be considered for inclusion.

In detail, for each bug, the authors investigated simultaneously the code of the commit to ensure relatedness to the bug being fixed. During the investigation, however, several bug-fixing commits were too complex to comprehend by the investigators, either because domain knowledge was required or because the number of files or lines of code being modified was large. We labeled such complex bug-fixing commits as 'Too complex', and discarded them from the current version of BUGSJS. The rationale is to keep the size of the patches within reasonable thresholds, so as to select a high quality corpus of bugs which can be easily analyzable and processable by both manual inspection and automated techniques. Particularly, we deemed a commit being too complex if the production code changes involved more than three (3) files or more than 50 LOC, or if the fix required more than 5 min to understand. In all such cases, a discussion was triggered among the authors, and the case was ignored if the authors unanimously decided that the fix was too complex.

Another case for exclusion is due to refactoring operations in the analyzed code. First, our intention was to keep the original code's behavior as written by developers. As such, we only restored modifications that did not affect the program's behavior (e.g., whitespaces). Indeed, in many cases, in-depth domain knowledge is very much required to decouple refactoring and bug fixation. JS is a dynamic language, and code can be refactored in many ways. Thus, it is more challenging to observe and account for side effects only by looking at the code than, for instance, in Java. In addition, refactoring may affect multiple parts of a project, it affects metrics such as code coverage, and it makes restoring the original code changes more challenging.

Results.

Overall, we manually validated 795 commits (i.e., bug candidates), of which 542 (68.18%) fulfilled the criteria. Table IV (Manual) illustrates the result of this step for each application and across all applications.

The most common reason for excluding a bug is that the fix was deemed as too complex (136). Other frequent scenarios include cases where a bug-fixing commit addressed more than one bug (32), or where the fix did not involve production code (29), or contained refactoring operations (39). Also, we found four cases in which the patch did not involve the actual test's source code, but rather comments or configuration files.

3.4. Sanity checking through dynamic validation

To ensure that the test cases introduced in a bug-fixing commit were actually intended to test the buggy feature, we adopted a systematic and automatic approach described next.

Methodology.

Let V_{bug} be the version of the source code that contains a bug b, and let V_{fix} be the version in which b is fixed. The existing test cases in V_{bug} do not fail due to b. However, at least one test of V_{fix} should

Table III. Bug-fixing commit inclusion criteria

Rule name	Description
Isolation	The bug-fixing changes must fix only one (1) bug (i.e., must close exactly one (1) issue).
Complexity	The bug-fixing changes should involve a limited number of files ($\leq$3), lines of code ($\leq$50) and be understandable within a reasonable amount of time (max 5 min).
Dependency	If a fix involves introducing a new dependency (e.g., a library), there must also exist production code changes and new test cases added in the same commit.
Relevant changes	The bug-fixing changes must involve only changes in the production code that aim at fixing the bug (whitespace and comments are allowed).
Refactoring	The bug-fixing changes must not involve refactoring of the production code.

Table IV. Manual and dynamic validation statistics per application for all considered commits

		BOWER	ESLINT	EXPRESS	HESSIAN.JS	HEXO	KARMA	MONGOOSE	NODE-REDIS	PENCILBLUE	SHIELDS	Total
Manual	*Initial number of bugs*	10	559	39	17	24	37	56	25	18	10	795
	✗ Fixes multiple issues	0	18	1	0	1	5	2	5	0	0	32
	✗ Too complex	0	94	0	4	8	4	8	7	9	2	136
	✗ Only dependency	1	9	0	0	1	0	2	0	0	0	13
	✗ No production code	0	20	4	0	1	1	2	0	0	1	29
	✗ No tests changed	1	0	1	0	0	0	0	1	1	0	4
	✗ Refactoring	0	36	0	0	0	1	1	1	0	0	39
Dynamic	*After manual validation*	8	382	33	13	13	26	41	11	8	7	542
	✗ Test does not fail at V_{bug}	1	11	6	4	1	2	8	3	1	3	40
	✗ Dependency missing	3	17	0	0	0	1	1	0	0	0	22
	✗ Error in tests	1	7	0	0	0	0	3	1	0	0	12
	✗ Not Mocha	0	14	0	0	0	1	0	0	0	0	15
	✓ Final number of bugs	3	333	27	9	12	22	29	7	7	4	453

fail when executed on V_{bug}. This allows us to identify the test in V_{fix} used to demonstrate b (*isolation*) and to discard cases in which tests immaterial to the considered buggy feature were introduced.

To run the tests, we obtained the dependencies and set up the environment for each specific revision of the source code. Over time, however, developers made major changes to some of the projects' structure and environment, making tests replication infeasible. These cases occurred, for instance, when older versions of required dependencies were no longer available, or when developers migrated to a different testing framework (e.g., from QUnit to Mocha).

For the projects that used scripts (e.g., `grunt`, `bash`, and `Makefile`) to run their tests, we extracted them, so as to isolate each test's execution and avoiding possible undesirable side effects caused by running the complete test suite.

Results.

After the dynamic analysis, 453 bug candidates were ultimately retained for inclusion in BUGSJS (84% of the 542 bug candidates from the previous step).

Table IV (Dynamic) reports the results for the dynamic validation phase. In 22 cases, we were unable to run the tests because dependencies were removed from the repositories. In 15 cases, the project at revision V_{bug} did not use Mocha for testing b. In 12 cases, tests were failing during the execution, whereas in 40 cases no tests failed when executed on V_{bug}. We excluded all such bug candidates from the benchmark.

3.5. *Patch creation*

We performed *manual cleaning* on the bug-fixing patches, to make sure they only include changes related to bug fixes. In particular, we removed *irrelevant files* (e.g., `*.md`, `.gitignore`, `LICENSE`), and *irrelevant changes* (i.e., source code comments, when only comments changed, and comments unrelated to bug-fixing code changes, as well as changes solely pertaining to whitespaces, tabs, or newlines). Furthermore, for easier analysis, we separated the patches into two separate files, the first one including the modifications to the tests, and the second one pertaining to the production code fixes.

3.6. *Final benchmark infrastructure and implementation*

Infrastructure.

Figure 2 illustrates the overall architecture of BUGSJS, which supports common activities related to the benchmark, such as running the tests at each revision or checking out specific commits. The framework's command-line interface includes the following commands:

- `info`: Prints out information about a given bug.

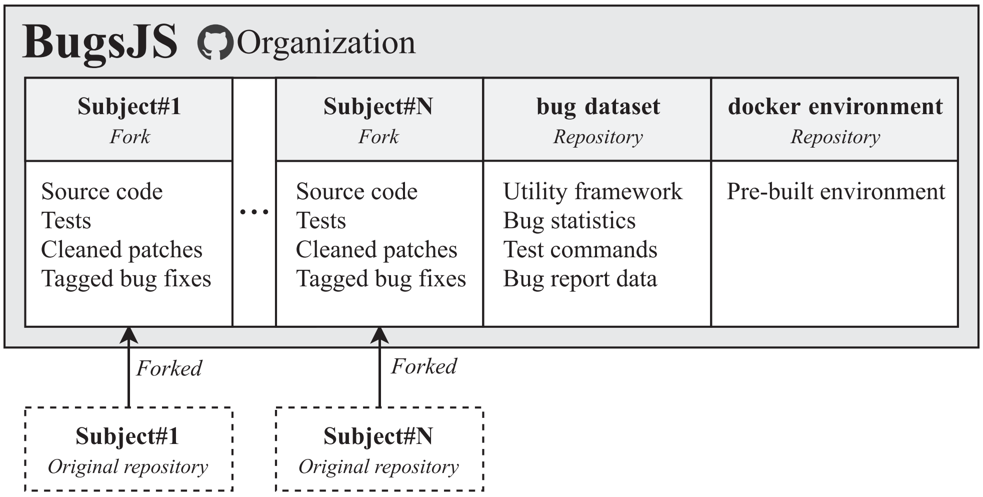


Figure 2. Overview of BUGSJS architecture.

- `checkout`: Checks out the source code for a given bug.
- `test`: Runs all tests for a given bug and measures the test coverage.
- `per-test`: Runs each test individually and measures the per-test coverage for a given bug.

For the `checkout`, `test`, and `per-test` commands, the user can specify the desired code revision: *buggy*, *buggy with the test modifications applied*, or the *fixed version*. BUGSJS is equipped with a prebuilt environment that includes the necessary configurations for each project to execute correctly. This environment is available as a Docker image along with a detailed step-by-step tutorial. The interested reader can find more information on BUGSJS and access the benchmark on our website (https://bugsjs.github.io/).

Source code commits and tests.

We used GitHub's *fork* functionality to make a full copy of the *git* history of the subject systems. The unique identifier of each commit (i.e., the commit `SHA1` hashes) remains intact when forking. In this way, we were able to synchronize the copied fork with the original repository and keep it up-to-date. Importantly, our benchmark will not be lost if the original repositories get deleted.

The fork is a separate *git* repository; therefore, we can push commits to it. Taking advantage of this possibility, we have extended the repositories with additional commits, to separate the bug-fixing commits and their corresponding tests. To make such commits easily identifiable, we tagged them using the following notation (`X` denotes a sequential bug identifier):

- `Bug-X`: The parent commit of the revision in which the bug was fixed (i.e., the buggy revision);
- `Bug-X-original`: A revision with the original bug-fixing changes (including the production code and the newly added tests);
- `Bug-X-test`: A revision containing only the tests introduced in the bug-fixing commit, applied to the buggy revision;
- `Bug-X-fix`: A revision containing only the production code changes introduced to fix the bug, applied to the buggy revision;
- `Bug-X-full`: A revision containing both the cleaned fix and the newly added tests, applied to the buggy revision.

Test runner commands.

For each project, we have included the necessary test runner commands in a CSV file. Each row of the file corresponds to a bug in the benchmark and specifies

1. A sequential bug identifier;
2. The test runner command required to run the tests;
3. The test runner command required to produce the test coverage results;
4. The Node.js version required for the project at the specific revision where the bug was fixed, so that the tests can execute properly;
5. The preparatory test runner commands (e.g., to initialize the environment to run the tests, which we call `pre-commands`);
6. The cleaning test runner commands (e.g., the tear down commands, which we call `post-commands`) to restore the application's state.

Bug report data.

Forking repositories does not maintain the issue data associated with the original repository. Thus, the links appearing in the commit messages of the forked repository still refer to the original issues. In order to preserve the bug reports, we obtained them via the GitHub's API and stored them in the Google's Protocol Buffers[****] format. Particularly, for each bug report, we store the original issue identifier paired with our sequential bug identifier, the text of the bug description, the issue open and close dates, and the `SHA1` of the original bug-fixing commit along with the commit date and commit author identifiers. Lastly, we save the comments from the issues' discussions.

[****]https://developers.google.com/protocol-buffers/

In the following, we list several artifacts that we added to BUGSJS in form of *precomputed data*.[††††] These can be reproduced by performing suitable static and dynamic analyses on the benchmark; however, we supply this information to better facilitate further bug-related research, including bug prediction, fault localization, and automatic repair.

Test coverage data.

As part of the technical extensions to the initial version of the framework [21], we included in BUGSJS precomputed information. We used the tool `Istanbul`[‡‡‡‡] to compute per-test coverage data for `Bug-X` and `Bug-X-test` versions of each bug, and the results are available in the JSON format. Particularly, for each project, we included information about the tests of the `Bug-X` versions in a separate CSV file. Each row in such file contains the following information:

1. A sequential bug identifier;
2. Total LOC in the source code, as well as LOC covered by the tests;
3. The number of functions in the source code, as well as the number of functions covered by the tests;
4. The number of branches in the source code, as well as the number of branches covered by the tests;
5. The total number of tests in the test suite, along with the number of passing, failing, and pending tests (i.e., the tests which were skipped due to execution problems).

Static source code metrics.

Furthermore, to support studies based on source code metrics, we run static analysis on `Bug-X-full` and `Bug-X` versions of each bug. For the static analysis, we used the tool `SourceMeter`[§§§§] which calculates 41 static source code metrics for JS. The results are available in a zip file named `metrics`.

3.7. BugsJS dissection

Sobreira et al. [44] implemented a web-based interface for the bugs in the Defects4J bug benchmark [15]. It presents data to help researchers and practitioners to better understand this bug dataset.[¶¶¶¶] We also utilized this dashboard and ported dissection to BUGSJS (Figure 3), which is available on the BUGSJS website (https://bugsjs.github.io/dissection).

BUGSJS dissection presents the information in the dataset to the user in an accessible and browsable format, which is useful for inspecting the various information related to the bugs, their fixes, their descriptions, and other artifacts such as the precomputed metrics.

More precisely, information provided in BUGSJS Dissection include the following:

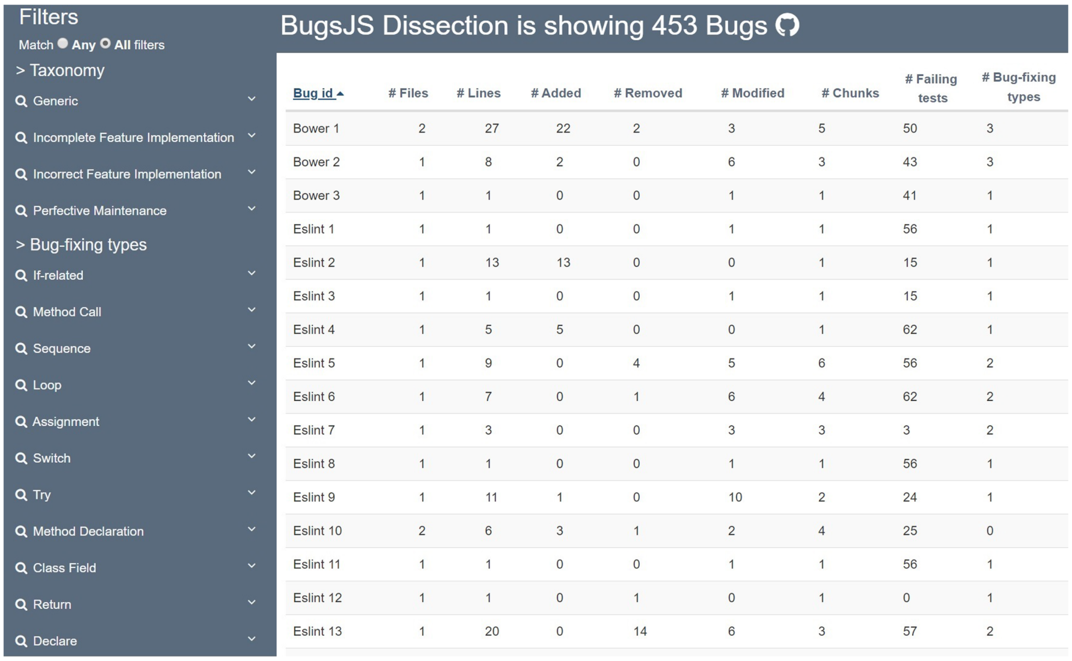
1. # Files: number of changed files.
2. # Lines: number of changed lines.
3. # Added: number of added lines.
4. # Removed: number of removed lines.
5. # Modified: number of modified lines.
6. # Chunks: number of sections containing sequential line changes.
7. # Failing tests: number of failed test cases.
8. # Bug-fixing type: number of bug-fixing types based on the taxonomy by Pan et al. [45].

Inspection of the bugs is supported through different filtering mechanisms that are based on the bug taxonomy and bug-fix types. By clicking on a bug, additional details appear (Figure 4), including bug-fix types, the patch by the developers, taxonomy category, and failed tests.

3.8. Extending BugsJS

BUGSJS was designed and implemented in a way that is easy to extend with new JS projects; however, there are some restrictions. The current version of the framework only supports projects that

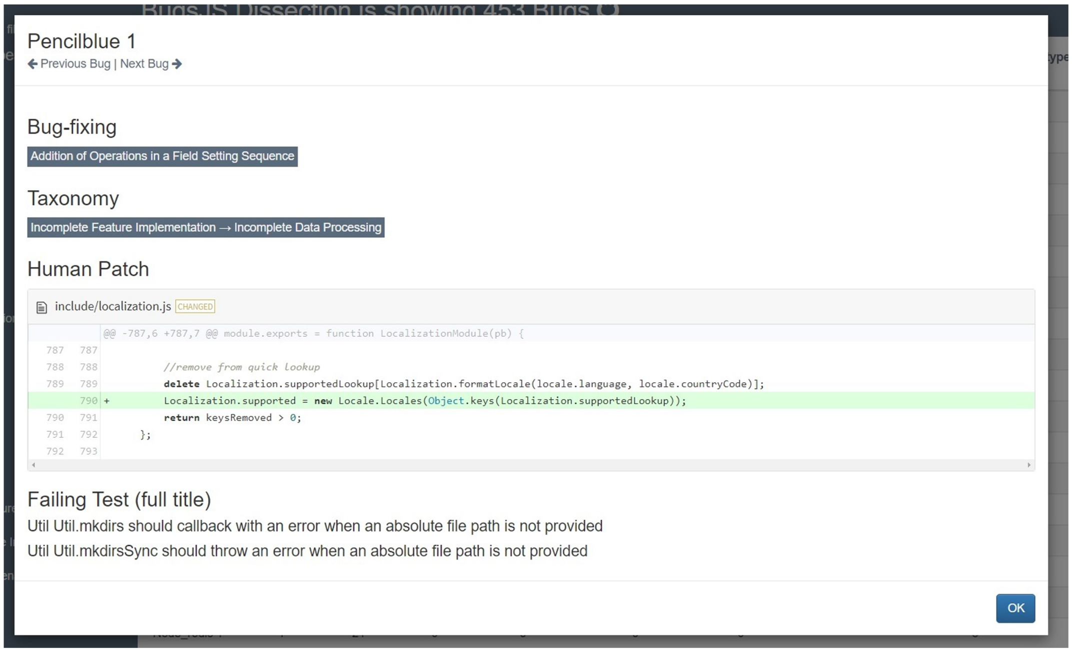
[††††]https://github.com/BugsJS/bug-dataset
[‡‡‡‡]https://istanbul.js.org/
[§§§§]https://www.sourcemeter.com/
[¶¶¶¶]https://github.com/program-repair/defects4j-dissection



BugsJS Dissection is showing 453 Bugs

Bug id	# Files	# Lines	# Added	# Removed	# Modified	# Chunks	# Failing tests	# Bug-fixing types
Bower 1	2	27	22	2	3	5	50	3
Bower 2	1	8	2	0	6	3	43	3
Bower 3	1	1	0	0	1	1	41	1
Eslint 1	1	1	0	0	1	1	56	1
Eslint 2	1	13	13	0	0	1	15	1
Eslint 3	1	1	0	0	1	1	15	1
Eslint 4	1	5	5	0	0	1	62	1
Eslint 5	1	9	0	4	5	6	56	2
Eslint 6	1	7	0	1	6	4	62	2
Eslint 7	1	3	0	0	3	3	3	2
Eslint 8	1	1	0	0	1	1	56	1
Eslint 9	1	11	1	0	10	2	24	1
Eslint 10	2	6	3	1	2	4	25	0
Eslint 11	1	1	0	0	1	1	56	1
Eslint 12	1	1	0	1	0	1	0	1
Eslint 13	1	20	0	14	6	3	57	2

Figure 3. BugsJS dissection overview page.



Pencilblue 1

Bug-fixing

Addition of Operations in a Field Setting Sequence

Taxonomy

Incomplete Feature Implementation → Incomplete Data Processing

Human Patch

include/localization.js CHANGED

```
@@ -787,6 +787,7 @@ module.exports = function LocalizationModule(pb) {

            //remove from quick lookup
            delete Localization.supportedLookup[Localization.formatLocale(locale.language, locale.countryCode)];
+           Localization.supported = new Locale.Locales(Object.keys(Localization.supportedLookup));
            return keysRemoved > 0;
        };

```

Failing Test (full title)

Util Util.mkdirs should callback with an error when an absolute file path is not provided

Util Util.mkdirsSync should throw an error when an absolute file path is not provided

Figure 4. BugsJS dissection page for one bug.

are in a *git* repository and use the Mocha testing framework. If the project is hosted on GitHub, it can be also forked under the BUGSJS's GitHub Organization to preserve the state of the repository.

Mining an appropriate bug to add to BUGSJS takes four steps as described at the beginning of Section 3. It is mainly manual work, but some of it could be done programmatically. In our case, we used GitHub as the source of bug reports, which has a public API; thus, we could automate the bug collection step. Validating the bug-fixing patches requires manual work, but it can be partially supported by automation, for example, for filtering patches that modify both the production code and the tests. However, in our experience during the development of BUGSJS, the location of test files varies across different projects and sometimes across versions as well. Thus, it is still challenging to automatically determine it for an arbitrary set of JS projects. Dynamic validation also requires some manual effort. Despite we limited the support only to the most common testing framework (Mocha), the command that runs the test suite is, in some cases, assembled at run-time (e.g., with `grunt` or `Makefile`) and can change over time. Extracting it programmatically is only possible for standard cases, for example, when it is located in the default `package.json` file. Due to the great variety of JS projects, this process can hardly be automated, as compared to other languages like Java, where project build systems are more homogeneous.

After a suitable bug is found, some preparatory steps are required before a bug can be added to BUGSJS. If necessary, irrelevant whitespace and comment modifications can be removed from the bug-fix patch, which is re-added to the repository as a new commit. Next, the production code modifications should be separated from the test modifications by committing the changes separately on top of the buggy version. Then, the commits should be tagged according to the notation described in Section 3.6. Finally, the new bug can be submitted to BUGSJS using a GitHub *pull request*. The pull request has to contain the modified or added CSV files that contain the repository URL, the test runner command, and any additional commands (pre and post) if any. Adding precomputed data is not mandatory, but beneficial.

4. TAXONOMY OF BUGS IN BUGSJS

In this section, we present a detailed overview of the root causes behind the bugs in our benchmark. We adopted a systematic process to classify the nature of each bug, which we describe next.

4.1. Manual labeling of bugs

Each bug and associated information (i.e., bug report and issue description) was manually analyzed by four authors (referred to as 'taggers' hereafter) following an open coding procedure [46]. Four taggers specified a descriptive label to each bug assigned to them. The labeling task was performed independently, and the disagreements were discussed and resolved through dedicated meetings. Unclear cases were also discussed and resolved during such meetings.

First, we performed a pilot study, in which all taggers reviewed and labeled a sample of 10 bugs. Bugs for the pilot were selected randomly from all projects in BUGSJS. The consensus on the procedure and the final labels was high; therefore, for the subsequent rounds the four taggers were split into two pairs, which were shuffled after each round of tagging.

The labels were collected in separate spreadsheets; the agreement on the final labels was found by discussion. During the tagging, the taggers could reuse existing labels previously created, should an existing label apply to the bug under analysis. This choice was meant to limit introducing nearly similar labels for the same bug and help taggers to use consistent naming conventions.

When inspecting the bugs, we looked at several sources of information, namely (i) the bug-fixing commit on the GitHub's web interface containing the commit title, the description as well as at the code changes, and (ii) the entire issue and pull request discussions.

In order to achieve internal validation in the labeling task, we performed cross-validation. Specifically, we created an initial version of the taxonomy labeling around 80% of the bugs (353). Then, to validate the initial taxonomy, the remaining 20% (100) were simply assigned to the closest category in the initial taxonomy, or a new category was created, when appropriate. Bugs for the initial taxonomy were selected at random, but they were uniformly selected among all subjects, to avoid over-fitting the taxonomy toward a specific project. Analogously, the validation set was retained so as to make sure all projects were represented. Internal validation of the initial taxonomy is achieved if few or no more categories (i.e., labels) were needed for categorizing the validation bugs. The labeling process involved four rounds: first round (the pilot study) involved labeling 10 bugs, second round 43 bugs, and 150 bugs were analyzed in both third and fourth rounds.

4.2. *Taxonomy construction*

After enumerating all causes of bugs in BUGSJS, we began the process of creating a taxonomy, following a systematic process. During a physical meeting, for each bug instance, all taggers reviewed the bugs and identified candidate equivalence classes to which descriptive labels were assigned. By following a bottom-up approach, we first clustered tags that correspond to similar notions into categories. Then, we created parent categories, in which that categories and their sub-categories follow specialization relationship.

4.3. *Taxonomy internal validation*

We performed the validation phase in a physical meeting. Each of the four tagger classified independently one fourth of the validation set (25 bugs), assigning each of them to the most appropriate category. After this task, all taggers reviewed and discussed the unclear cases to reach full consensus. All 100 validation bugs were assigned to existing categories, and no further categories were needed.

4.4. *The final taxonomy*

Figure 5 presents a graphical view of our taxonomy of bugs in the JS benchmark. Nodes represent classes and subclasses of bugs, and edges represent specialization relationships. Specializations are not complete, disjoint relationships. Even though during labeling we tried assigning the most specific category, we found out during taxonomy creation that we had to group together many app-specific corner cases. Thus, some bugs pertaining to inner nodes were not further specialized to avoid creating an excessive number of leaf nodes with only a few corner cases.

At the highest level, we identified four distinct categories of causes of bugs, as follows:

1. Causes related to an *incomplete feature implementation*. These bugs are related either to an incomplete understanding of the main functionalities of the considered application or a refinement in the requirements. In these cases, the functionalities have already been implemented by developers according to their best knowledge, but over time, users or other developers found out that they do not consider all aspects of the corresponding requirements. More precisely, given a requirement r, the developer implemented a program feature f' which corresponds to only a subset $r' \subset r$ of the intended functionality. Thus, the developer has to adapt the existing functionality $f' \subset f$ to f, in order to satisfy the requirement in r. Typical instances of this bug category are related to one or more specific corner-cases that were unpredictable at the time in which that feature was initially created, or when the requirements for the main functionalities are changed or extended to some extent.
2. Causes related to an *incorrect feature implementation*. These bugs are also related to the mainstream functionalities of the application. Differently from the previous category, the bugs in this category are related to wrong implementation by the developers, for instance, due to an incorrect interpretation of the requirements. More precisely, suppose that given a requirement r, the developer implemented a program feature f', to the best of her knowledge. Over time, other developers found out by the usage of the program that the behavior of f' does not reflect the intended behavior described in r and opened a dedicated issue in the GitHub repository (and, eventually, a pull request with a first fix attempt).
3. Causes related to *generic* programming errors. Bugs belonging to this category are typically not related to an incomplete/incorrect understanding of the requirements by developers, but rather to common coding errors, which are also important from the point of view of a taxonomy of bugs.
4. Causes related to *perfective maintenance*. Perfective maintenance involves making functional enhancements to the program in addition to the activities to increase its performance even when the changes have not been suggested by bugs. These can include completely new requirements to the functionalities or improvements to other internal or external quality

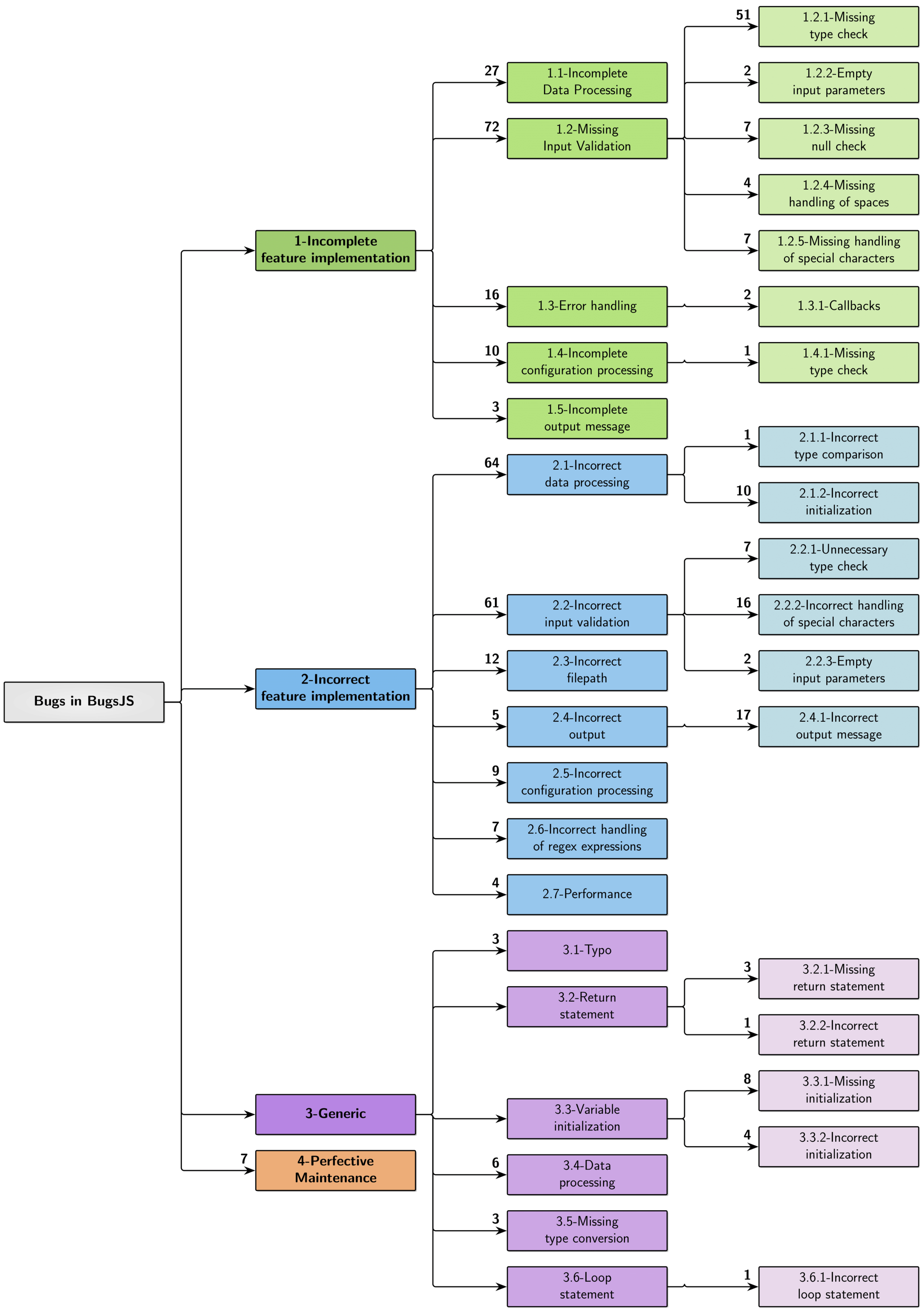

Bugs in BugsJS
1-Incomplete feature implementation
27 1.1-Incomplete Data Processing
72 1.2-Missing Input Validation
51 1.2.1-Missing type check
2 1.2.2-Empty input parameters
7 1.2.3-Missing null check
4 1.2.4-Missing handling of spaces
7 1.2.5-Missing handling of special characters
16 1.3-Error handling
2 1.3.1-Callbacks
10 1.4-Incomplete configuration processing
1 1.4.1-Missing type check
3 1.5-Incomplete output message
2-Incorrect feature implementation
64 2.1-Incorrect data processing
1 2.1.1-Incorrect type comparison
10 2.1.2-Incorrect initialization
61 2.2-Incorrect input validation
7 2.2.1-Unnecessary type check
16 2.2.2-Incorrect handling of special characters
2 2.2.3-Empty input parameters
12 2.3-Incorrect filepath
5 2.4-Incorrect output
17 2.4.1-Incorrect output message
9 2.5-Incorrect configuration processing
7 2.6-Incorrect handling of regex expressions
4 2.7-Performance
3-Generic
3 3.1-Typo
3.2-Return statement
3 3.2.1-Missing return statement
1 3.2.2-Incorrect return statement
3.3-Variable initialization
8 3.3.1-Missing initialization
4 3.3.2-Incorrect initialization
6 3.4-Data processing
3 3.5-Missing type conversion
3.6-Loop statement
1 3.6.1-Incorrect loop statement
7 4-Perfective Maintenance

Figure 5. Taxonomy of bugs in the benchmark of JavaScript programs of BUGSJS.

attributes not affecting existing functionalities. When composing BUGSJS, we aimed at excluding such cases from the candidate bugs (see Section 3); however, the bugs that we classified in the taxonomy with this category were labeled as bugs by the original developers, so we decided to retain them in the benchmark.

We now discuss each of these categories in turn, in each case considering the sub-categories beneath them.

1. *Incomplete feature implementation*

This category contains 45% of the bugs overall and has five sub-categories, which we describe in the following subsection.

1.1. *Incomplete data processing*

The bugs in this category are related to an incomplete implementation of a feature's logic, that is, the way in which the input is consumed and transformed into output.

Overall, 27 bugs were found to be of this type. An example is `Bug#7` of Hexo,[‖‖‖] in which an HTML anchor was undefined, unless correct escaping of markdown characters is used.

```
- var text = $(this).html();
+ var text = _.escape($(this).text());
```

1.2. *Missing input validation*

The bugs in this category are related to an incomplete input validation, that is, the way in which the program checks whether a given input is valid, and can be processed further.

Overall, 16% of the bugs were found to be of this type and a further 16% in more specialized instances. This prevalence was mostly due to the nature of some of our programs. For instance, ESLint provides linting utilities for JS code, and it is the most represented project in BUGSJS (73%). Therefore, being its main scope to actually validate code, we found many cases related to invalid inputs being unmanaged by the library, even though we found instances of these bugs also in other projects. For instance, in `Bug#4` of Karma,[*****] a file parsing operation should not be triggered on URLs having no line number. As such, in the bug-fixing commit, the proposed fix adds one more condition.

```
- if (file && file.sourceMap) {
+ if (file && file.sourceMap && line) {
```

Another prevalent category is due to missing type check on inputs (11%), whereas less frequent categories were missing check of null inputs, empty parameters, and missing handling of spaces or other special characters (e.g., in URLs).

1.3. *Error handling*

The bugs in this category are related to an incomplete handling of errors,that is, the way in which the program manages erroneous cases, that is, exception handling.

Overall, 3% of the bugs were found to be of this type. For instance, in `Bug#14` of Karma,[†††††] the program does not throw an error when using a plugin for a browser that is not installed, which is a corner-case missed in the initial implementation. Additionally, we found two cases specific to callbacks.

1.4. *Incomplete configuration processing*

The bugs in this category are related to an incomplete configuration, that is, the values of parameters accepted by the program.

Overall, 2% of the bugs were found to be of this type. For instance, in `Bug#10` of ESLint,[‡‡‡‡‡] an invalid configuration is used when applying extensions to the default configuration object. The bug fix updates the default configuration object's constructor to use the correct context and to make sure the config cache exists when the default configuration is evaluated.

[‖‖‖]https://github.com/BugsJS/hexo/releases/tag/Bug-7-original
[*****]https://github.com/BugsJS/karma/releases/tag/Bug-4-original
[†††††]https://github.com/BugsJS/karma/releases/tag/Bug-14-original
[‡‡‡‡‡]https://github.com/BugsJS/eslint/releases/tag/Bug-10-original

1.5. Incomplete output message

The last sub-category pertains to bugs related to incomplete output messages by the program.

Only three bugs were found to be of this type. For instance, in `Bug#8` of Hessian.js,[§§§§§] the program casts the values exceeding `Number.MAX_SAFE_INTEGER` as string, to allow safe readings of large floating point values.

2. Incorrect feature implementation

This category contains 48% of the bugs overall and has seven sub-categories, which we describe in the following subsections.

2.1. Incorrect data processing

The bugs in this category are related to a wrong implementation of a feature's logic, that is, the way in which the input is consumed and transformed into output.

Overall, 75 bugs were found to be of this type, with two sub-categories due to a wrong type comparison (1 bug), or an incorrect initialization (10 bugs). An example of this latter category is `Bug#238` of ESLint,[¶¶¶¶¶] in which developers remove the default parser from CLIEngine options to fix a parsing error.

```
- parser: DEFAULT_PARSER
+ parser: ""
```

2.2. Incorrect input validation

The bugs in this category are related to a wrong input validation, that is, the way in which the program checks whether a given input is valid and can be processed further.

Overall, 19% of the bugs were found to be of this type, with three sub-categories due to unnecessary type checks (7 bugs), incorrect handling of special characters (16 bugs), or empty input parameters given to the program (2 bugs). As an example of this latter category, in `Bug#171` of ESLint,[‖‖‖‖‖] the `arrow-spacing` rule did not check for all spaces between the arrow character (=>) within a given code. Therefore, it is updated as follows:

```
- while (t.type !== "Punctuator" || t.value !== "=>") {
+ while (arrow.value !== "=>") {
```

2.3. Incorrect filepath

The bugs in this category are related to wrong paths to external resources necessary to the program, such as files. For instance, in `Bug#6` of ESLint,[******] developers failed to check for configuration files within sub-directories. Therefore, the code was updated as follows:

```
- if (!directory)
+ if (directory) directory = path.resolve(this.cwd, directory);
```

2.4. Incorrect output

The bugs in this category are related to incorrect output by the program. For instance, in `Bug#7` of Karma,[††††††] the exit code is wrongly replaced by null characters (`0x00`), which results in squares (□□□□□□) being displayed in the standard output.

```
- return exitCode
+ return {exitCode: exitCode, buffer: buffer.slice(0, tailPos)}
```

[§§§§§]https://github.com/BugsJS/hessian.js/releases/tag/Bug-8-original
[¶¶¶¶¶]https://github.com/BugsJS/eslint/releases/tag/Bug-238-original
[‖‖‖‖‖]https://github.com/BugsJS/eslint/releases/tag/Bug-171-original
[******]https://github.com/BugsJS/eslint/releases/tag/Bug-6-original
[††††††]https://github.com/BugsJS/karma/releases/tag/Bug-7-original

2.5. *Incorrect configuration processing*

The bugs in this category are related to an incorrect configuration of the program, that is, the values of parameters accepted.

Nine bugs were found to be of this type. For instance, in `Bug#145` of ESLint,[‡‡‡‡‡‡] a regression was accidentally introduced where parsers would get passed additional unwanted default options even when the user did not specify them. The fix updates the default parser options to prevent any unexpected options from getting passed to parsers.

```
- let parserOptions = Object.assign({}, defaultConfig.parserOptions);
+ let parserOptions = {};
```

2.6. *Incorrect handling of regex expressions*

The bugs in this category are related to an incorrect use of regular expressions.

Seven bugs were found to be of this type. For instance, in `Bug#244` of ESLint,[§§§§§§] a regular expression is wrongly used to check that the function name starts with `setTimeout`.

2.7. *Performance*

The bugs in this category caused the program to use an excessive amount of resources (e.g., memory). Only four bugs were found to be of this type. For instance, in `Bug#85` of ESLint,[¶¶¶¶¶¶] a regular expression susceptible to catastrophic backtracking was used. The match takes quadratic time in the length of the last line of the file, causing Node.js to hang when the last line of the file contains more than 30,000 characters. Another representative example is `Bug#1` of Node-Redis,[‖‖‖‖‖‖] in which parsing big JSON files takes substantial time due to an inefficient caching mechanism which makes the parsing time grow exponentially with the size of file.

3. *Generic*

This category contains 6% of the bugs overall and has six sub-categories, which we describe next.

3.1. *Typo.*

This category refers to typographical errors by the developers.

We found three such bugs in our benchmark. For instance, in `Bug#321` of ESLint,[*******] a rule is intended to compare the *start* line of a statement with the end line of the previous token. Due to a typo, it was comparing the *end* line of the statement instead, which caused false positives for multiline statements.

3.2. *Return statement.*

The bugs in this category are related to either missing return statements (3 bugs) or incorrect usage of return statements (1 bug). For instance, in `Bug#8` of Mongoose,[†††††††] the fix involves adding an explicit return statement.

```
- this.constructor.update.apply(this.constructor, args);
+ return this.constructor.update.apply(this.constructor, args);
```

3.3. *Variable inizialization.*

The bugs in this category are related to either missing initialization of variables statements (8 bugs) or to an incorrect initialization of variables (4 bugs). For instance, in `Bug#9` of Express,[‡‡‡‡‡‡‡] the fix involves correcting a wrongly initialized variable.

[‡‡‡‡‡‡]https://github.com/BugsJS/eslint/releases/tag/Bug-145-original
[§§§§§§]https://github.com/BugsJS/eslint/releases/tag/Bug-244-original
[¶¶¶¶¶¶]https://github.com/BugsJS/eslint/releases/tag/Bug-85-original
[‖‖‖‖‖‖]https://github.com/BugsJS/node_redis/releases/tag/Bug-1-original
[*******]https://github.com/BugsJS/eslint/releases/tag/Bug-321-original
[†††††††]https://github.com/BugsJS/mongoose/releases/tag/Bug-8-original

```
-  mount_app.mountpath = path;
+  mount_app.mountpath = mount_path;
```

3.4. *Data processing.*

The bugs in this category are related to incorrect processing of information.

Six bugs were found to be of this type. For instance, in `Bug#184` of ESLint,[§§§§§§§] developers fixed the possibility of passing negative values to the string.slice function.

```
- currentText.slice(node.range[0] - (beforeCount || 0)
+ currentText.slice(Math.max(node.range[0] - (beforeCount || 0), 0)
```

3.5. *Missing type conversion.*

The bugs in this category are related to missing type conversions.

Three bugs were found to be of this type. For instance, in `Bug#4` of Shields,[¶¶¶¶¶¶¶] developers forgot to convert labels to string prior to apply the uppercase transformation.

```
- data.text[0] = data.text[0].toUpperCase();
+ data.text[0] = ('' + data.text[0]).toUpperCase();
```

3.6. *Loop statement.*

We found only one bug of this type—`Bug#304` of Shields,[‖‖‖‖‖‖‖] —related to an incorrect usage of loop statements.

```
- while ((currentAncestor = currentAncestor.parent))
-  if (isConditionalTestExpression(currentAncestor)) {
-    return currentAncestor.parent;
-  }
- }

+ do {
+  if (isConditionalTestExpression(currentAncestor)) {
+    return currentAncestor.parent;
+  }
+ } while ((currentAncestor = currentAncestor.parent));
```

4. *Perfective maintenance*

This category contains only 1% of the bugs. For instance, in `Bug#209` of ESLint,[********] developers fix JUnit parsing errors which treat no test cases having empty output message as a failure.

5. ANALYSIS OF BUG-FIXES

To gain a better understanding about the characteristics of bug-fixes of bugs included in BUGSJS, we have performed two analyses to quantitatively and qualitatively assess the representativeness of our benchmark. This serves as an addition to the taxonomy presented in Section 4.4 which, by connecting the bug types to the bug-fix types, can support applications such as automated fault localization and automated bug repair.

5.1. *Code churn*

Code *churn* is a measure that approximates the rate at which code evolves. It is defined as the sum of the number of lines added and removed in a source code change. The churn is an important measure with several uses in software engineering studies, for example, as a direct or indirect predictor in bug prediction models [47,48].

[‡‡‡‡‡‡‡]https://github.com/BugsJS/express/releases/tag/Bug-9-original
[§§§§§§§]https://github.com/BugsJS/eslint/releases/tag/Bug-184-original
[¶¶¶¶¶¶¶]https://github.com/BugsJS/shields/releases/tag/Bug-4-original
[‖‖‖‖‖‖‖]https://github.com/BugsJS/eslint/releases/tag/Bug-304-original
[********]https://github.com/BugsJS/eslint/releases/tag/Bug-209-original

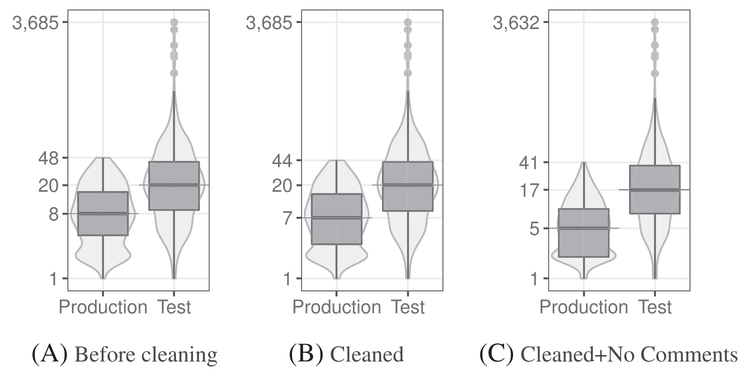


Figure 6. (a–c) Distribution of the churn in bug-fixing commits.

We utilize the distribution of code churn to describe the overall data distribution in the benchmark and understand to what extent it can be used to support software testing techniques (e.g., fault localization and program repair) that are directly affected by the size of the source code changes.

Figure 6 illustrates the distribution of code churn in each bug-fixing commit in BUGSJS, *before* cleaning the patches (Figure 6a), *after* removing changes unrelated to the fix, while retaining related comments added or removed during the bug fix (Figure 6b), and *after cleaning* the fixes and also removing all the comments and whitespaces (Figure 6c). The box and whisker plots show the quartiles of the data, enhanced by the underlying violin plots to better depict the data distribution.

The median value for the code churn *required to fix a bug* is 5 and 17 for production and test code, respectively (excluding irrelevant changes, comments, and whitespaces). This essentially means that for each line of bug-fixing change in the production code, more than three lines of test code were changed on average.

We can also observe that nearly half of the changes done to the production code involve more than five lines of code, suggesting that existing fault localization and repair techniques will fall short in being applicable on real JS bugs, as they currently deal with one-liner changes only. Previous work showed that the same conclusion holds for Java projects [49].

In addition, comparing the median values in Figure 6 suggests that developers, on average, add/remove one line of unrelated code changes (i.e., $8-7$) and two lines of comments/whitespace (i.e., $7-5$) when fixing a bug. This essentially shows the importance of the manual isolation and cleaning performed on the bugs included in BUGSJS.

The largest value for churn in test code is 3,632 lines (Figure 6c), occurring in a bug-fixing commit in the ESLint project. This commit corresponds to generated test data committed along with the production code bug-fixing changes. A closer look at the data revealed that there are five other such commits in this project, all changing more than 1,000 lines of test code. Such commits in which the number of changes is exceptionally high (i.e., outliers) should be carefully handled or discarded when conducting empirical studies.

5.2. *Patterns in bug fixes*

We further analyzed the bugs in BUGSJS to observe occurrence of low-level *bug fixes* for recurring patterns. Previous work [45,49,50] have studied patterns in bug-fixing changes within Java programs. They suggest that the existence of patterns in fixes reveals that specific kinds of code constructs (e.g., if conditionals) could signal weak points in the source code where developers are consistently more prone to introduce bugs [45].

Methodology.

Four authors of this paper manually investigated all 453 bug-fixing commits in BUGSJS and attempted to assign the bug-fixing changes to one of the predefined categories suggested in previous studies. In particular, we used the categories proposed by Pan et al. [45]. These categories, however, are related to Java bug fixes. Our aim is to assess whether they generalize to JS, or whether, in contrast, JS-specific bug-fix patterns would emerge.

Following the original category definitions [45], we assigned each *individual* bug fix to exactly one category. Disagreements concerning classification or potential new categories were resolved by further discussion between the authors. To identify the occurrences of such patterns, we opted for a manual analysis to ensure covering potential new patterns, and to add an extra layer of validation against potential misclassifications (e.g., false positives).

Table V shows the number of bug fix occurrences followed the categories by Pan et al. [45] (for fixes spanning multiple lines, we possibly assigned more than one category to a single bug-fixing commit; hence, the overall number of occurrences is greater than the number of bugs).

Note that, since the categories proposed by Pan et al. have been derived from Java programs, we had to make sure to match them correctly on JS code. In particular, until ECMAScript 2015, JS did not include syntactical support for classes. Classes were *emulated* using functions as constructors, and methods/fields are added to their prototype [51-53]. In addition, *object literals* could represent imaginary classes: comma-separated list of name–value pairs enclosed in curly braces, where the name–value pairs declare the class fields/methods. We have taken all these aspects into account during the assignment task, to avoid misclassifications.

Our analysis revealed that, in 88% of bugs in BUGSJS, the fix includes changes falling into one of the proposed categories. The most prevalent bug fix patterns involve changing an `if` statement (i.e., modifying the `if` condition or adding a precondition), changing assignment statements, and modifying function call arguments (Table V). The same three categories have been also found to be most recurring in Java code, but with a different ordering: Pan et al. [45] report that the most prevalent fix patterns are changes done on method calls, `if` conditions, and assignment expressions. In addition, we found that changes to class fields are also prevalent. This can be explained by the fact that in JS, object literals are frequently created without the need for defining a class or function constructor, and, as far as fixing bugs is concerned, updating their attributes (i.e., fields) is a common practice.

5.2.1. *JavaScript-related bug-fixing patterns.*

We found three *new* recurring patterns in our benchmark, which we describe next.

Changes to the `return` statement's expression. We found a recurring bug-fixing pattern involving changing the return statement's expression of a function, that is

```
- return node.type !== "A";
+ return !(node.type === "A" && lastI.type === "R");
```

Variable declaration. In JS, it is possible to use a variable without declaring it. However, this has implications which might lead to subtle *silent bugs*. For example, when a variable is used inside a function without being declared, it is 'hoisted' to the top of the global scope. As a consequence, it is visible to all functions, *outside its original lexical scope*, which can lead to name clashes. This fix pattern essentially includes declaring a variable which has already been in use.

Table V. Bug-fixing change types by Pan et al. [45] and new JavaScript-related patterns found by our study.

	Category	Example	#
Existing	`if`-related	Changing `if` conditions	291
	Assignments	Modifying the RHS of an assignment	166
	Function calls	Adding or modifying an argument	151
	Class fields	Adding/removing class fields	151
	Function declarations	Modifying a function's signature	94
	Sequences	Adding a function call to a sequence of calls, all with the same receiver	42
	Loops	Changing a loop's predicate	7
	`switch` blocks	Adding/removing a switch branch	6
	`try` blocks	Introducing a new `try-catch` block	1
New	`return` statements	Changing a `return`statement	40
	Variable declaration	Declaring an existing variable	2
	Initialization	Initializing a variable with empty object literal/array	3

Initialization of empty variables. This bug-fixing pattern category corresponds to Hanam et al.'s [25] first bug pattern, that is, dereferenced non-values. To avoid this type of bug, developers can add additional `if` statements, comparing values against 'falsey' values ('`undefined`' type or '`null`'). This bug fix pattern provides a shortcut to using an `if` statement, by using a logical 'or' operator, for example, `a = a || {}`, which means that the value of `a` will remain intact if it already has a 'non-falsey' value, or it will be initialized with an empty object otherwise.

5.3. Bug taxonomy and bug-fixing types

We compared the taxonomy with the bug-fixing patterns used to fix the bugs. Figures 7–9 present Sankey diagrams showing the relationship of the assigned bug categories with the bug-fixing patterns of Pan et al. [45] and the JS-related bug-fixing patterns described in Section 5.3. We focused our analysis on the first three main bug categories of our taxonomy. For presentational clarity, each figure shows a diagram for one of such main categories. The left side of the diagrams shows one of the main bug categories, unfolded into its sub-categories. The right side depicts the associated bug-fixing patterns. The *None* node indicates that no patterns were applicable. In the middle, each bug category is connected to each bug-fixing patterns that are used to fix the bugs belonging to the bug category. The thickness of the curved lines between the nodes indicates the cardinality of the association.

For example, in Figure 7, the node *incomplete feature implementation* is connected to the sub-category *missing input validation* with a thick line, due to the majority of the bugs in this main category belonging to that sub-category. Then, the node of that sub-category is connected to the node of the *IF-CC*bug-fixing pattern with a relatively thick line, because a considerable amount of bugs fixes are classified into that category. The nodes of the bug categories can be wider than the total width of the lines connected from the right side, because bug fixes can be assigned with multiple bug patterns, whereas each bug is assigned with exactly one bug category. Also, there are bugs that are only assigned to a main category, for example, *Missing input validation*, but not to any sub-category, which resulted in the direct lines between pattern and main category nodes.

Table VI gives a description about the bug-fixing pattern abbreviations. We now discuss each of the bug categories in detail.

5.3.1. Incomplete feature implementation.

Figure 7 illustrates the connection between the bugs under the Incomplete feature implementation category and the related bug-fixing patterns.

Missing input validation. Table VII shows that the majority of the bugs assigned with this category are fixed by `if`-related (53), assignment-related (29), and method declaration-related (22) changes. The most common are changing a condition in an `if` statement (*IF-CC*), changing an assignment expression (*AS-CE*), or by adding a method declaration (*MD-ADD*). The most numerous sub-category is the missing type check, and the related bugs are mainly fixed by `if`-related changes (39), the top two are *IF-CC* and *IF-APCJ*, and by changing an assignment expression (18 *AS-CE*). The fixes of bugs in the missing handling of special characters sub-category often contain changes in an assignment expression (6 *AS-CE*) and in an `if` statement (5), mainly adding post-condition checks (*IF-APTC*). Bugs belonging to the missing null check and the empty input parameters sub-categories are mainly fixed by changing an `if` condition (5). The missing handling of spaces sub-category is connected to various bug-fixing patterns and none of them is dominant.

Incomplete configuration processing. The majority of the bug fixes of this category contain `if`-related patterns (9), mainly *IF-APC*, adding a precondition check. The missing type check sub-category of Incomplete configuration processing is too small to draw meaningful conclusions.

Error handling. Bugs of this category are typically fixed with `if`-related fixes (12), namely *IF-APCJ*, *IF-APC* and *IF-CC*. The second most common fix patterns are *MC-DNP*, changing the number of parameters (3) and *MD-ADD*, adding a method declaration (4). The callbacks sub-category of error handling is related to a variety of bug-fixing patterns (4 `if`-related, 1 assignment related, 1 method call related, and 1 sequence related).

Incomplete output message. Bugs assigned with this category are usually fixed by changing the parameters of function calls (2 *MC-DAP*, 1 *MC-DNP*).

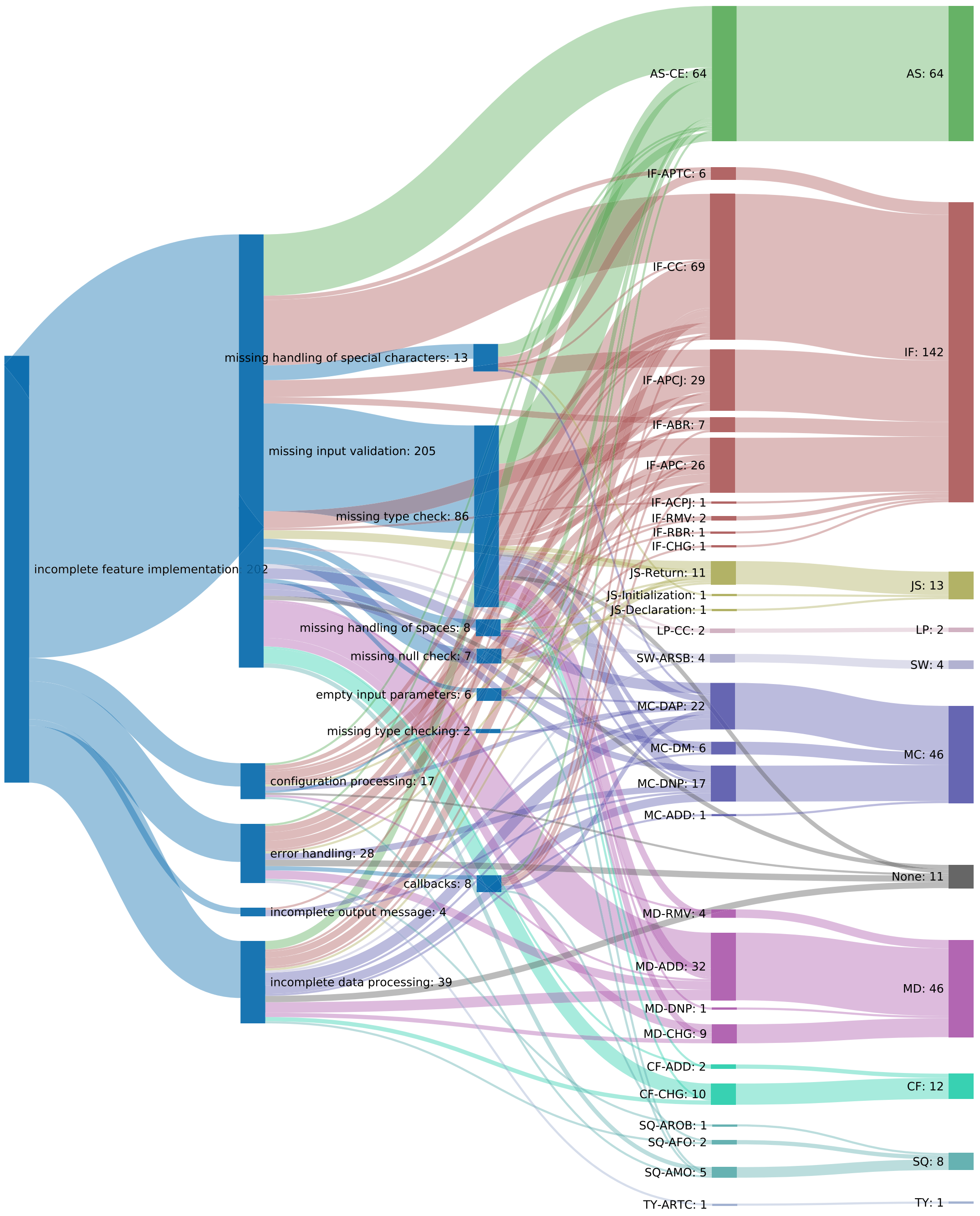


Figure 7. Bug-fixing patterns used in the Incomplete feature implementation category.

Incomplete data processing. Bugs belonging to this category were fixed in a variety of ways. The most common pattern are method call related (11), but there is no dominant bug-fixing pattern.

5.3.2. Incorrect feature implementation. Figure 8 illustrates the connection between the bugs under the Incorrect feature implementation category and the related bug-fixing patterns.

Incorrect input validation. The most dominant bug-fixing patterns of this category are `if`-related (56), method call related (27) and assignment related (25). The most numerous `if`-related pattern is *IF-CC*, and the most numerous method call-related pattern is *MC-DAP*. Furthermore, `return` statement related (12 *JS-Return*) and method declaration related (10 *MD-ADD* and 5

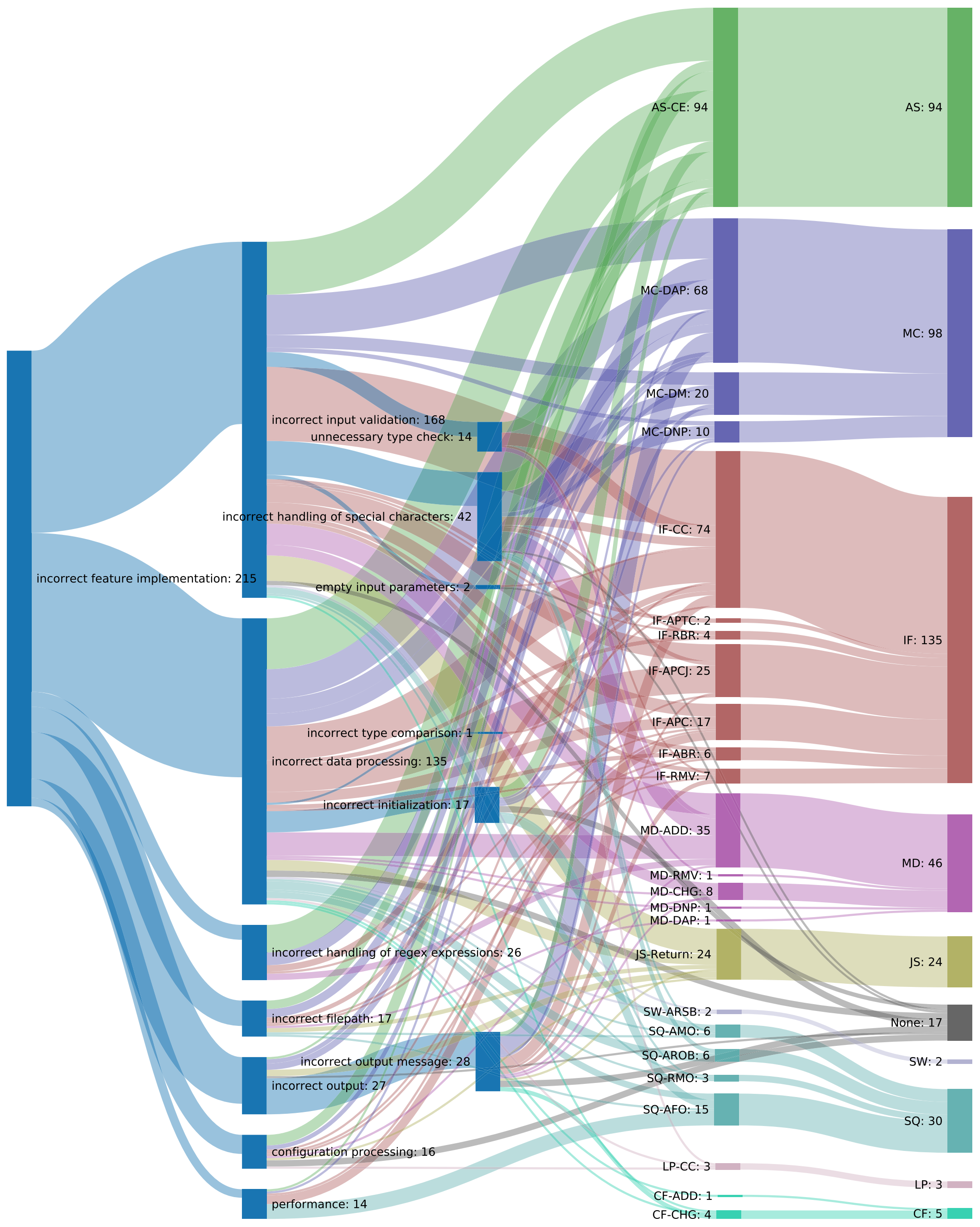


Figure 8. Bug-fixing patterns used in the Incorrect feature implementation category.

MD-CHG) patterns are also quite common. In the unnecessary type check sub-category, the most common pattern is `if`-related (6 *IF-CC* and 1 *IF-RMV*), and the second most common pattern is the assignment-related pattern (5 *AS-CE*). Bugs belonging to the biggest input validation-related sub-category, the incorrect handling of special characters, are usually fixed with method call-related changes (10 *MC-DAP* and 2 *MC-DM*), with assignment-related changes (9 *AS-CE*) and with method declaration-related changes (9 *MD-ADD* and 1 *MD-RMV*). The empty input parameters sub-category of incorrect input validation is too small to draw meaningful conclusions.

Incorrect data processing. Similarly to the incomplete data processing category, bugs belonging to this category were fixed in a variety of ways. There is no dominant bug-fixing pattern. The most

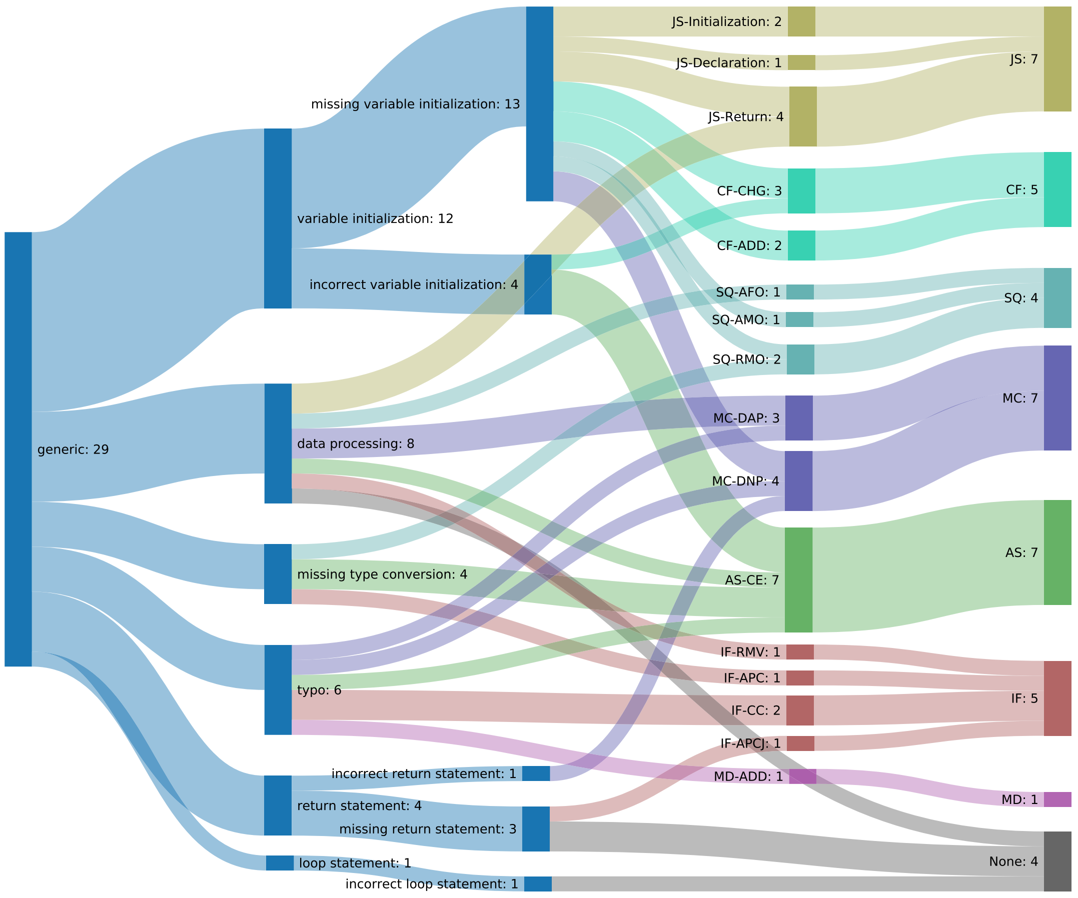


Figure 9. Bug-fixing patterns used in the generic category.

numerous patterns are `if`-related (39), method call related (27) and assignment related (24). The incorrect initialization sub-category is interestingly not connected to `if`-related patterns at all. The bug fixes of this category are connected to only assignment-related (5), sequence-related (5), and method call-related (4) patterns. The incorrect type comparison sub-category of incorrect data processing is too small to draw meaningful conclusions.

Incorrect handling of regex expressions. Bugs assigned to this category are mostly fixed with assignment-related (13 *AS-CE*) and method call-related (6 *MC-DAP*) changes.

Incorrect filepath. Here, the most dominant bug-fixing patterns are assignment related (4), method call related (4), and `if`-related (4). The variety of connected patterns is large.

Incorrect output. Bugs of this category are usually fixed by changing method calls (3*MC-DM* and 2 *MC-DAP*) and by changing `return` statements (3 *JS-Return*). The incorrect output message sub-category contains bugs with fixes that mostly involve changes to the parameters of method calls (9 *MC-DAP* and 1 *MC-DNP*) and changes to condition expressions in `if` statements (5 *IF-CC*, 2 *IF-RMV*, and 1 *IF-APCJ*).

Incorrect configuration processing. The most dominant bug-fixing pattern for this category is assignment related (5 *AS-CE*), but a variety of other patterns occur as well (3 `if`-related, 2 method call related, 1 JS related, 1 loop related, and 1 method declaration related).

Performance. The majority of the bug-fixing patterns used for fixing bugs of this category is *SQ-AFO*, adding operations in an operation sequence of field settings (7). The second most common patterns are `if`-related (5 *IF-APC* and 1 *IF-APCJ*).

5.3.3. Generic. Figure 9 illustrates the connection between the bugs under the generic category and the related bug-fixing patterns.

Table VI. Bug-fixing patterns

Category	Pattern name
	Pan [45]
Assignment (AS)	Change of assignment expression (AS-CE)
Class field (CF)	Addition of a class field (CF-ADD)
	Change of class field declaration (CF-CHG)
	Removal of a class field (CF-RMV)
If-related (IF)	Addition of an else branch (IF-ABR)
	Addition of precondition check (IF-APC)
	Addition of precondition check with jump (IF-APCJ)
	Addition of post-condition check (IF-APTC)
	Change of if condition expression (IF-CC)
	Removal of an else branch (IF-RBR)
	Removal of an if predicate (IF-RMV)
Loop (LP)	Change of loop condition (LP-CC)
	Change of the expression that modifies the loop variable (LP-CE)
Method call (MC)	Method call with different actual parameter values (MC-DAP)
	Different method call to a class instance (MC-DM)
	Method call with different number or types of parameters (MC-DNP)
Method declaration (MD)	Change of method declaration (MD-CHG)
	Addition of a method declaration (MD-ADD)
	Removal of a method declaration (MD-RMV)
Sequence (SQ)	Addition of operations in an operation sequence of field settings (SQ-AFO)
	Addition of operations in an operation sequence of Method calls to an object (SQ-AMO)
	Addition or removal method call operations in a short construct body (SQ-AROB)
	Removal of operations from an operation sequence of field settings (SQ-RFO)
	Removal of operations from an operation sequence of method calls to an object (SQ-RMO)
Switch (SW)	Addition/removal of switch branch (SW-ARSB)
Try (TY)	Addition/removal of a catch block (TY-ARCB)
	Addition/removal of a try statement (TY-ARTC)
	BugsJS
JavaScript (JS)	Changing a return statement (JS-return)
	Initializing a variable with empty object literal/array (JS-initialization)
	Declaring an existing variable (JS-declaration)

Variable initialization. The bug fixes of the missing variable initialization sub-category contain five types of bug-fixing patterns. The most dominants are the JS-related patterns (2 *JS-Return*, 2 *JS-Initialization*, and 1 *JS-Declaration*) and the class field-related patterns (2 *CF-CHG* and 2 *CF-ADD*). In the other sub-category, incorrect variable initialization, changing an assignment expression (3 *AS-CE*) is the most dominant.

Data processing. Similarly to the other two cases of data processing bugs, the associated bug fixes contain a variety of patterns (2 JS related, 2 method call related, 1 `if`-related, 1 assignment related, and 1 sequence related), and there is no dominant bug-fixing pattern.

Missing type conversion. The bug fixes of this category mostly involve changes in an assignment expression (2 *AS-CE*), but adding a precondition check to an `if` statement (1 *IF-APC*) and removing an operation from an operation sequence of method calls (1 *SQ-RMO*) also appears.

Typo. For this category, there is no dominant bug-fixing pattern (2 `if`-related, 2 method call related, 1 assignment related, and 1 method declaration related) which essentially means typos can occur at any place in the code.

Return statement. This category contains too few bug-fixing patterns to draw meaningful conclusions. Surprisingly, the bug fixes do not fall into the *JS-Return* category.

Loop statement. There is only one bug in this category, and its fix does not contain any bug-fixing patterns; therefore, we cannot draw meaningful conclusions here.

5.3.4. Highlights. Table VII provides statistics about the occurrences of each bug-fix type corresponding to the bug types. The table can serve to analyze the emergence of correlations between

Table VII. Taxonomy and bug fixing types

Incomplete feature implementation	AS	CF	IF	JS	LP	MC	MD	None	SQ	SW	TY
Configuration processing	1	0	9	1	0	2	1	1	1	0	0
Missing type check	1	0	0	0	0	1	0	0	0	0	0
Error handling	1	0	12	1	0	3	4	3	1	0	1
Callbacks	1	0	4	0	0	2	0	0	1	0	0
Incomplete data processing	4	2	9	1	0	11	7	3	1	1	0
Incomplete output message	0	0	1	0	0	3	0	0	0	0	0
Missing input validation	29	8	53	4	1	11	22	2	2	2	0
Empty input parameters	1	0	3	0	0	1	0	0	1	0	0
Missing handling of spaces	2	0	2	1	0	1	2	0	0	0	0
Missing handling of special characters	6	0	5	1	0	1	0	0	0	0	0
Missing null check	0	0	5	2	0	0	0	0	0	0	0
Missing type check	18	2	39	2	1	10	10	2	1	1	0
Incorrect feature implementation											
Configuration processing	5	0	3	1	1	2	1	3	0	0	0
Incorrect data processing	24	2	39	5	1	27	13	3	9	1	0
Incorrect initialization	5	0	0	0	0	4	0	3	5	0	0
Incorrect type comparison	0	0	1	0	0	0	0	0	0	0	0
Incorrect filepath	4	0	4	2	0	4	1	0	2	0	0
Incorrect handling of regex expressions	13	0	4	0	0	6	3	0	0	0	0
Incorrect input validation	25	1	56	12	0	27	15	2	4	1	0
Empty input parameters	0	0	1	0	0	0	0	1	0	0	0
Incorrect handling of special characters	9	0	7	0	1	12	9	1	3	0	0
Unnecessary type check	5	0	7	0	0	0	2	0	0	0	0
Incorrect output	1	0	0	3	0	5	0	1	0	0	0
Incorrect output message	2	2	8	1	0	10	2	3	0	0	0
Performance	1	0	5	0	0	1	0	0	7	0	0
Generic											
Data processing	1	0	1	2	0	2	0	1	1	0	0
Loop statement											
Incorrect loop statement	0	0	0	0	0	0	0	1	0	0	0
Missing type conversion	2	0	1	0	0	0	0	0	1	0	0
Return statement											
Incorrect return statement	0	0	0	0	0	1	0	0	0	0	0
Missing return statement	0	0	1	0	0	0	0	2	0	0	0
Typo	1	0	2	0	0	2	1	0	0	0	0
Variable initialization											
Incorrect variable initialization	3	1	0	0	0	0	0	0	0	0	0
Missing variable initialization	0	4	0	5	0	2	0	0	2	0	0
Perfective maintenance	1	0	9	1	0	1	1	0	0	0	0

bug-fix types and bug types. Overall, the most common bug-fixes types are `if`-related (291), the second most common are assignment-related (166), and the third most common are method call-related (152) bug-fixes. These bug-fixes types are mostly related to the most prominent bug categories, namely missing input validation, incorrect input validation, and incorrect data processing. Another correlation is that assignment-related fixes are also the preferred way to fix regexes. These are perhaps the only correlations between bug-fix types and bug types that are observable in our benchmark.

6. DISCUSSION

Our results might drive devising novel software analysis and repair techniques for JS, with BUGSJS being a suitable real-world bug benchmark for their evaluation, as well as inform developers of the most error-prone constructs.

In the rest of this section, we discuss some of the potential uses of our taxonomy, together with possible use cases of our benchmark in supporting empirical studies in software analysis and testing, as well as its limitations and threats to validity of our study.

6.1. Directing research efforts

Our taxonomy and associated data can be useful in several contexts related to JS analysis and testing contexts. Our study reveals that the majority of bugs are related to mistakes by the developers. This finding is in line with those of the previous study by Ocariza et al. [9] on client-side JS programs. Overall the bug fixes included in BUGSJS cover a diverse range of categories, some of which being specific only to JS (Section 5.3). For instance, incorrect/missing input validation and incorrect/incomplete data processing caused the majority of bugs we observed, 50%[††††††††] and 23%,[‡‡‡‡‡‡‡‡] respectively. It has to be said that this prevalence is due to the ESLint project, which is a linting tool, essentially a code validator. While ESLint may be not representative of the majority of JS web services, it is a very popular JS linting tool, being recommended by multiple comparative studies above other tools like JSLint, JSHint, and JSCS.[§§§§§§§§][¶¶¶¶¶¶¶¶] Moreover, ESLint is supported by JetBrains in the WebStorm IDE and by Vue.js to validate their templates and by Facebook's React to help enforce their coding rules. This adoption suggests that major companies recognize input validation/data processing tasks as vital throughout the software development.

Another relevant source of bugs is due to missing type checks, a construct which is particularly problematic due to the dynamically typed nature of JS, which makes it easier for developers to introduce bugs if they are, for instance, more familiar or used to strongly typed languages [9,25]. Researchers have proposed approaches for addressing this class of errors and finding ways to prevent them [1,4,54-57], which suggests that this class of errors deserves considerable attention.

6.1.1. Benchmark for testing techniques. Various fields of testing research can benefit from BUGSJS. First, our benchmark includes more than 25k JS test cases, which makes it a rather large dataset for different regression testing studies (e.g., test prioritization, test minimization, or test selection). Second, BUGSJS can play a role to support research in software oracles (e.g., automated generation of semantically meaningful assertions), as it contains all test suites' evolution as well as examples of real fixes made by developers. Additionally, these can be used to drive the design of automated test repair techniques [58,59]. Finally, test generation or mutation techniques for JS can be evaluated on BUGSJS at a low cost, since precomputed coverage information are available for use.

6.1.2. Bug prediction using static source code analysis. To construct reliable bug prediction models, training feature sets are extracted from the source code, comprising instances of buggy and healthy code, and static metrics. BUGSJS can support these studies since it streamlines the hardest part of constructing the training and testing datasets, that is, determining whether a given code element is affected by a bug. As such, the cleaned fixes included in BUGSJS make this task much easier. Also, the availability of both uncleaned and cleaned bug-fixing patches in the dataset can allow assessing the sensitivity of the proposed models to the noise. In addition, some of the most important static code metrics are readily available as precomputed data.

6.1.3. Bug localization. BUGSJS can support devising novel bug localization techniques for JS. Approaches that use natural language processing can take advantage of our benchmark since bugs are readily available to be processed. Indeed, text retrieval techniques are used to formulate a natural language query that describes the observed bug. To this aim, BUGSJS contains pointers to the natural language bug description and discussions for several hundreds of real-world bugs. Similarly, BUGSJS will be of great benefit for other popular bug localization approaches, for example, the

[††††††††](72+51+2+7+4+7+61+7+16+2)/453%
[‡‡‡‡‡‡‡‡](27+64+1+10)/453%
[§§§§§§§§]https://www.sitepoint.com/comparison-javascript-linting-tools/
[¶¶¶¶¶¶¶¶]https://codekitapp.com/help/jslint/

spectrum-based techniques [60-63] because all the necessary data—test case outcomes, code coverage, and bug information—are readily available.

6.1.4. Automated program repair. Automated program repair techniques aim at automatically fixing bugs in programs, by generating a large pool of candidate fixes, to be later validated. The manually cleaned patches available in BUGSJS can be used as learning examples for patch generation in novel automated program repair for JS. Also, BUGSJS provides an out-of-the-box solution for automatic dynamic patch validation. Detailed classification of the bugs according to our bug-taxonomy and the bug-fix types provides to this kind of research additional useful information.

6.2. Directing developer efforts

6.2.1. Improving manual repair processes. In the absence of automated program repair techniques, knowledge about the causes of bugs could help developers manually fix programs, both by increasing their awareness to bugs causes and helping them prioritize the inspection of possible causes based on the relative importance of such causes in our taxonomy.

6.2.2. Avoiding bugs. Bugs avoidance pertain to cases in which programmers are provided with information to assist them in avoiding bugs, and it is up to them to choose what information to utilize and how. One way to promote bugs avoidance involves educating developers and maintainers of JS applications as to the causes and probabilities of bugs. Such education could be supported by information present in our taxonomy and the data that underlie it. First, consider code changes activities. We observed that many of the bugs involving missing type checks resulted from simple code changes, that is, missing an IF-condition in a statement. Second, consider code creation tasks. When creating new code, programmers can enforce the practice of adding input validation as a must-do of their daily activities. Finally, consider test case creation. Testers can use our taxonomy to focus their test case creation to avoid the most bug-prone categories.

6.2.3. Preventing bugs. Bugs prevention, in contrast to bugs avoidance, involves the use of automated approaches for ensuring that bugs do not occur, even though humans might be included in the feedback loop. For example, programmers may take advantage of tools like checkers and linting tools that enable static and dynamic analysis automatically [64], and our taxonomy could help improve on certain constructs that are particularly challenging for developers.

6.2.4. IDE enhancements. Another class of bugs prevention approaches involves improvements in web programming and testing IDEs. Analysis techniques that operate concurrently with program development and maintenance may be quite effective, and such techniques could also be guided by our taxonomy. Modern IDEs for code development typically employ such approaches: As programmers edit, they point out problems or provide useful information based on the programmers' actions. JS application development IDEs employ such approaches also, but to our knowledge, no such IDEs also build in assistance related to testing efforts. Such IDEs could aid in bugs avoidance by alerting developers to possible effects or bad practices and letting them choose whether or not to act on them. They could aid in prevention by prohibiting certain actions or by recommending the creation of constructs. Let us take as example the return statement-related issues. Our taxonomy highlighted that developers often fix bugs by changing the return statement. Thus, IDEs can be improved with new data flow testing techniques that check that JS objects' states are preserved during the execution before they are returned or that inform a change-impact analysis technique to show how the change to an object affects the final output.

6.3. Limitations

BUGSJS includes only server-side JS applications developed with the Node.js framework. As such, experiments evaluating the client-side (e.g., the DOM) are not currently supported. While our survey revealed a large number of subjects being used for evaluating such techniques, the majority of these programs could not be directly included in our proposed benchmark.

Indeed, in the JS realm, the availability of many implementations, standards, and testing frameworks poses major technical challenges with respect to devising a uniform and cohesive bugs infrastructure. Similar reasoning holds for selecting Mocha as a reference testing framework.

Running tests for browser-based programs may require complex and time-consuming configurations. When dealing with a large and diverse set of applications, achieving isolation would require automating each single configuration for all possible JS development and testing frameworks, which is a cumbersome task. Clearly, this is a potential limitation and bias for experiments that use BUGSJS, and we are considering for a future revision of the benchmark to support other environments as well. We must note, however, that due to the mentioned specialities of the JS ecosystem, we do not expect to be able to fully cover the plethora of the different execution environments. Nevertheless, all the subjects included in BUGSJS have been previously used by at least one work in our literature survey (e.g., BOWER, SHIELDS, KARMA, NODE-REDIS, and MONGOOSE are all used in bug-related studies).

6.4. *Threats to validity*

The main threat to the *internal validity* of this work is the possibility of introducing bias when selecting and classifying the surveyed papers and the bugs included in the benchmark.

Our paper selection was driven by the keywords related to software analysis and testing for JS (Section 2). We may have missed relevant studies that are not captured by our list of terms. We mitigate this threat by performing an issue-by-issue, manual search in the major software engineering conference proceedings and journals, followed by a snowballing process. We, however, cannot claim that our survey captures all relevant literature; yet, we are confident that the included papers cover the major related studies.

Concerning the bugs, we manually classified all candidate bugs into different categories (Section 3.3) and the retained bugs into categories pertaining to existing bug and fix taxonomies (Section 5.2). To minimize classification errors, multiple authors simultaneously analyzed the source code and performed the classifications individually, and disagreements were resolved by further discussions among the authors. Concerning the bug's classification for taxonomy construction, the first four authors classified the bugs manually. This task, however, requires reasoning that cannot be automated, so it is difficult to envision less threat-prone approaches. To reduce the subjectivity involved in the task, the authors followed a systematic and structured procedure, with multiple interactions.

Threats to the *external validity* concern the generalization of our findings. We, by no means, claim that our benchmark represents all relevant web apps. We selected only 10 applications and our bugs may not generalize to different projects. Also, other relevant classes of bugs might be unrepresented or underrepresented within our benchmark, which is to date quite overfitted toward ESLint, that is, the most represented project. Nevertheless, we tried to mitigate this threat by selecting applications with different sizes and pertaining to different domains. We hope that our framework will provide an entry point and a reference for future improvements as other subject systems are necessary to fully confirm the generalizability of our results and corroborate our findings.

Another generalization threat concerns our taxonomy. Taxonomies are conceptual maps derived from empirical observations; as such they typically evolve as additional observations of the world are made. We expect the same to be true of our taxonomy, and thus, it is not necessarily the case that any attempt to apply the taxonomy to subsequent programs will allow every type of bug in those applications to be categorized. In such cases, the taxonomy will require adjustments. This work attempts to reduce this threat by applying a validation phase to our initial taxonomy.

With respect to *reproducibility* of our results, all classifications, subjects, and experimental data are available online, making the analysis reproducible.

7. RELATED WORK

7.1. Benchmarks

7.1.1. C, C++, and C# Benchmarks. The Siemens benchmark suite [65] was one of the first datasets of bugs used in testing research. It consists of seven C programs, containing manually seeded faults. The first widely used benchmark of real bugs and fixes is the SIR [14], which includes the Siemens benchmark and extends it with nine additional large C programs and seven Java programs. SIR also features test suites, bug data, and automation scripts. The benchmark contains both real and seeded faults, the latter being more frequent.

Le Goues et al. [16] proposed two benchmarks for C programs called ManyBugs and IntroClass,[‖‖‖‖‖‖] which include 1,183 bugs in total. The benchmarks are designed to support the comparative evaluation of automatic repair, targeting large-scale production (ManyBugs) as well as smaller (IntroClass) programs. ManyBugs is based on nine open-source programs (5.9MLOC and over 10k test cases) and it contains 185 bugs. IntroClass includes 6 small programs and 998 bugs.

Rahman et al. [66] examined the OpenCV project mining 40 bugs from seven out of 52 C++ modules into the benchmark *Pairika*. The seven modules analyzed contain more than 490kLOC, about 11k test cases and each bug is accompanied by at least one failing test.

Lu et al. [67] propose *BugBench*, a collection of 17 open-source C/C++ programs containing 19 bugs pertaining to memory and concurrency issues.

Codeflaws [68] contains nearly 4k bugs in C programs, for which annotated ASTs with annotated syntactic differences between buggy and patch code are provided.

7.1.2. Java benchmarks. Just et al. [15] presented Defects4J, a bug database and extensible framework containing 357 validated bugs from five real-world Java programs. BugsJS shares with Defects4J the idea of mining bugs from the version control history. However, BugsJS has some additional features: Subject systems are accessible in the form of *git* forks on a central GitHub repository, which maintains the whole project history. Further, all programs are equipped with prebuilt environments in form of Docker containers. Moreover, in this paper, we also provide a more detailed analysis of subjects, tests, and bugs.

Bugs.jar [69] is a large-scale dataset intended for research in automated debugging, patching, and testing of Java programs. Bugs.jar consists of 1,158 bugs and patches, collected from eight large, popular open-source Java projects.

iBugs [70] is another benchmark containing real Java bugs from bug-tracking systems originally proposed for bug localization research. It is composed of 390 bugs and 197kLOC coming from three open source projects.

7.1.3. Multi-language benchmarks. *QuixBugs* [18] is a benchmark suite of 40 confirmed bugs used in program repair experiments targeting Python and Java with passing and failing test cases.

BugSwarm [17] is a recent dataset of real software bugs and bug fixes to support various testing empirical experiments such as test generation, mutation testing, and fault localization.

Code4Bench [71] is another cross-language benchmark comprising C/C++, Java, Python, and Kotlin programs among others. Code4Bench also features a coarse-grained bug classification based on an automatic fault localization process for which faults were classified only in three groups, namely addition, modifications, and deletion. In contrast, BugsJS focuses on JS bugs, for which we provide a fine-grained analysis based on a rigorous manual process.

7.1.4. Benchmarks comparison. We summarize the related benchmarks and compare their main features to BugsJS in Table VIII. The table includes the language(s) in which the programs were written and the kind of bugs the benchmarks contain. Further, the table indicates whether the modified versions have been cleaned from irrelevant changes, for example, achieving the isolation property, whether the benchmark includes quantitative or qualitative analyses of the faults. These information were retrieved in the papers in which the benchmarks were proposed first.

Table VIII. Properties of the benchmarks

Benchmark	Language(s)	Fault type	# Bugs	Isolation	Quantitative analysis	Qualitative analysis
Siemens/SIR [14]	C/Java	Real/seeded	662	✓	✗	✗
ManyBugs [16]	C	Real	185	✓	✓	✓
IntroClass [16]	C	Real	998[†]	✓	✓	✓
Pairika [68]	C++	Real	40	✓	✓	✗
BugBench [69]	C/C++	Real	19	✗	✗	✓
Defects4J [15]	Java	Real	357[‡]	✓	✓	✓
Bugs.jar [71]	Java	Real	1,158	✓	✗	✗
iBugs [72]	Java	Real	390	✗	✓	✗
QuixBugs [18]	Java/Python	Seeded	40	✓	✓	✓
Codeflaws [66]	C	Real	3,902	✓	✗	✓
BugSwarm [17]	Java/Python	Real	3,165	✗	✓	✗
Code4Bench [73]	Multiple	Real	N/A[§]	✗	✓	✗
BUGSJS	JavaScript	Real	453	✓	✓	✓

[†]*The total number of bugs is 1,623, of which 998 are those in common between two test suites.*
[‡]*This is reported in the original publication; the newer versions of the benchmark include additional bugs.*
[§]*Created by independent authors* [44].
[*]*Only the number of faulty program versions is reported.*

The table highlights that BUGSJS is the only benchmark that contains JS programs. This paper also provides both a quantitative analysis of the benchmark and a qualitative analysis of the bugs (from which a taxonomy was derived) and the bug fixes (by comparing them with existing taxonomies). For instance, in the case of Defect4J, the original paper proposed only the benchmark [15], whereas a quantitative analysis was added in a subsequent paper by Sobreira et al. [44]. More qualitative analyses were also made by Sobreira et al. [44] and Motwani et al. [72], who independently propose two orthogonal classification of repairs.

To summarize, BUGSJS is the first benchmark of bugs and related artifacts (e.g., source code and test cases) that targets the JS domain. In addition, BUGSJS differentiates from the previously mentioned benchmarks in the following aspects: (i) The subjects are provided as *git* forks with complete histories, (2) a framework is provided with several features enabling convenient usage of the benchmark, (3) the subjects and the framework itself are available as GitHub repositories, (4) Docker container images are provided for easier usage, (5) the bug descriptions are accompanied by their natural language discussions, as well as (6) a manually derived bug taxonomy and a comparison with an existing bug-fixes taxonomy.

7.2. *Bug taxonomies*

There are several industry standards for categorizing software bugs, such as the IEEE Standard Classification for Software Anomalies [73] or IBM's Orthogonal Defect Classification [74]. However, these are either too generic or more process-related and are not suitable for categorizing bugs in BUGSJS. Also, there are countless categorization schemes proposed by various testing and defect management tool and service vendors, which are also less relevant for our research.

Hanam et al. [25] discuss 13 cross-project bug patterns occurring in JS pertaining to six categories, which are the following: *Dereferenced non-values* (e.g., Uninitialized variables), *Incorrect API config* (e.g., Missing API call configuration values), *Incorrect comparison* (e.g., === and == used interchangeably), *Unhanded exceptions* (e.g., Missing `try-catch` block), *Missing arguments* (e.g., Function call with missing arguments), and *Incorrect* `this` *bounding* (e.g., Accessing a wrong `this`reference).

Since this is probably the closest related work to our taxonomy presented in Section 4.4, we tried to assign all 453 bugs of BUGSJS to one of these categories as well. Our analysis found 42 occurrences of the categories proposed by Hanam et al., most of them (35) belonging to *Dereferenced non-values*. This shows that these patterns do exist in the bugs that we have in BUGSJS, but they only cover a small subset of them. The majority of the rest of the bugs are indeed logical errors made by developers during the implementation which do not necessarily fall into recurring patterns. This shows that the bugs included in BUGSJS are rather diverse in nature, making it ideal for evaluating a wide range of analysis and testing techniques. Our taxonomy seemed more appropriate for the categorization of such logical errors in BUGSJS, with the price that our categories are more high level and independent of the language and the domain of the subject systems.

The most common pattern according to the Hanam et al. scheme, *Dereferenced non-values*, can also be identified in other related work. Previous work showed that this pattern occurs frequently also in client-side JS applications [9]. Developers could avoid these syntax-related bugs by adopting appropriate coding standards. Moreover, IDEs can be enhanced to alert programmers to possible effects or bad practices. They could also aid in prevention by prohibiting certain actions or by recommending the creation of stable constructs.

Catolino et al. [75] analyzed 1,280 bug reports of 119 popular projects with the aim of building a taxonomy of the types of reported bugs. They devised and evaluated the automated classification model which is able to classify the reported bugs according to the defined taxonomy. The authors defined a three-step manual method to build the taxonomy, which was similar to our approach. The final taxonomy defined in this work contains nine main common bug types over the considered systems: configuration, network, database-related, GUI-related, performance, permission/deprecation, security, program anomaly, and test code-related issues. This classification is less suitable to apply to BUGSJS because it is a very high level one and is not related to JS but to web applications in general.

Li et al. [76] used natural language text classification techniques to automatically analyze 29,000 bugs from the Bugzilla databases of Mozilla and Apache HTTP Server. The authors classified the bugs in three dimensions: root cause (RC), impact (I) and software component (SC). According to RC, bugs can be classified into three disjoint groups (and sub-groups): semantic, memory, and concurrency. Some of the root cause sub-categories are similar to the categories in our taxonomy.

Tan et al. [77] proposed a work that is related to the previous study. They examined more than 2,000 randomly sampled real-world bugs in three large projects (Linux kernel, Mozilla, and Apache) and manually analyzed them according to the three dimensions defined by Li et al. [78]. They created a bug type classification model, which used machine learning techniques to automatically classify the bug types.

Zhang et al. [78] investigated the symptoms and root causes of TensorFlow bugs. They identified the bugs from the GitHub issue tracker using commit and pull request messages. The authors collected the common root causes (which were based on structure, model tensor, and API operation) and symptoms (based on error, effectiveness, and efficiency) into categories and classified each bug accordingly.

Thung et al. [79] presented a semi-supervised defect prediction approach (Learning with Diverse and Extreme Examples) to minimize the manual bug labeling. The researchers used a benchmark that contains 500 defects from three projects that have been manually labeled based on IBM's Orthogonal Defect Classification (ODC). In their approach, hand-labeled samples were used to learn and build the model, which uses non-labeled elements to refine the model.

In another study, Thung et al. [80] proposed a classification-based approach used to categorize the bugs into control and data flow, structural or non-functional groups. They performed natural language processing preprocessing and feature extraction operations on the text mined from JIRA. The resulting data was used to build the model based on support vector machine.

Nagwani et al. [81] used the bug-tracking system to collect textual information and several attributes on bugs. They presented a methodology to bug classification, which are based on a generative statistical model (latent Dirichlet allocation) in natural language processing.

8. CONCLUSIONS

The increasing interest of developers and industry around JS has fostered a huge amount of software engineering research around this language. Novel analysis and testing techniques are being proposed every year; however, without a centralized benchmark of subjects and bugs, it is difficult to fairly evaluate, compare, and reproduce research results.

To fill this gap, in this paper, we presented BUGSJS, a benchmark of 453 real, manually validated JS bugs from 10 popular JS programs. Our quantitative and qualitative analyses, including a categorization of bugs in a dedicated taxonomy, show the diversity of the bugs included in BUGSJS that can be used for conducting highly reproducible empirical studies in software analysis and testing research related to, among others, regression testing, bug prediction, and fault localization for JS. Using BUGSJS in future studies is further facilitated by a flexible framework implemented to automate checking out specific revisions of the programs' source code, running each of the test cases demonstrating the bugs, and reporting test coverage.

As part of our ongoing and future work, we plan to include more subjects (and corresponding bugs) to the benchmark. Our long-term goal is to also include client-side JS web applications in BUGSJS. Furthermore, we are planning to develop an abstraction layer to allow easier extensibility of our infrastructure to other JS testing frameworks.

ACKNOWLEDGEMENTS

Gyimesi and Vancsics were supported by project EFOP-3.6.3-VEKOP-16-2017-0002, co-funded by the European Social Fund. Beszédes was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled 'Internet of Living Things'. Ferenc was supported by grant 2018-1.2.1-NKP-2018-00004 'Security Enhancing Technologies for the IoT' funded by the Hungarian National Research, Development and Innovation Office. This research was supported by grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, Hungary. Mesbah, Stocco, and Mazinanian were supported in part by NSERC Discovery and DAS grants.

REFERENCES

1. Alimadadi S, Mesbah A, Pattabiraman K. Understanding asynchronous interactions in full-stack JavaScript. In *Proc. of 38th International Conference on Software Engineering (ICSE)*, 2016. https://doi.org/10.1145/2884781.2884864
2. Wang J, Dou W, Gao C, Gao Y, Wei J. Context-based event trace reduction in client-side JavaScript applications. In *Proc. of International Conference on Software Testing, Verification and Validation (ICST)*, 2018. https://doi.org/10.1109/ICST.2018.00022
3. Wang J, Dou W, Gao Y, Gao C, Qin F, Yin K, Wei J. A comprehensive study on real world concurrency bugs in Node.js. In *Proc. of International Conference on Automated Software Engineering*, 2017. https://doi.org/10.1109/ASE.2017.8115663
4. Alimadadi S, Mesbah A, Pattabiraman K. Hybrid DOM-sensitive change impact analysis for JavaScript. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, 2015.
5. Madsen M, Tip F, Andreasen E, Sen K, Møller A. Feedback-directed instrumentation for deployed JavaScript applications. In *Proc. of 38th International Conference on Software Engineering (ICSE)*, 2016. https://doi.org/10.1145/2884781.2884846
6. Adamsen CQ, Møller A, Karim R, Sridharan M, Tip F, Sen K. Repairing event race errors by controlling nondeterminism. In *Proc. of 39th International Conference on Software Engineering (ICSE)*, 2017. https://doi.org/10.1109/ICSE.2017.34
7. Ermuth M, Pradel M. Monkey see, monkey do: effective generation of GUI tests with inferred macro events. In *Proc. of 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016. https://doi.org/10.1145/2931037.2931053
8. Billes M, Møller A, Pradel M. Systematic black-box analysis of collaborative web applications. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.
9. Ocariza F. S., Bajaj K, Pattabiraman K., Mesbah A.. A study of causes and consequences of client-side JavaScript bugs. *IEEE Transactions on Software Engineering*. 2017; **43**(2): 128–144. https://doi.org/10.1109/TSE.2016.2586066
10. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*. 2011; **37**(5). https://doi.org/10.1109/TSE.2010.62
11. Andrews J. H., Briand L. C., Labiche Y.. Is mutation an appropriate tool for testing experiments. In *Proc. of International Conference on Software Engineering*, 2005.
12. Just R, Jalali D, Inozemtseva L, Ernst M, Holmes R, Fraser G. Are mutants a valid substitute for real faults in software testing. In *Proc. of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.
13. Gopinath R., Jensen C., Groce A. Mutations: how close are they to real faults. In *Proc. of International Symposium on Software Reliability Engineering*, 2014.

14. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with resting techniques: an infrastructure and its potential impact. *Empirical Software Engineering*. 2005; **10**(4): 405–435. https://doi.org/10.1007/s10664-005-3861-2
15. Just R, Jalali D, Ernst MD. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proc. of 2014 International Symposium on Software Testing and Analysis*, 2014. https://doi.org/10.1145/2610384.2628055
16. Le Goues C, Holtschulte N, Smith E, Brun Y, Devanbu P, Forrest S, Weimer W.. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering (TSE)*. 2015; **41**(12): 1236–1256. https://doi.org/10.1109/TSE.2015.2454513
17. Dmeiri N, Tomassi DA, Wang Y, et al. BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes. In *Proc. of 41st International Conference on Software Engineering (ICSE)*, 2019.
18. Lin D, Koppel J, Chen A, Solar-Lezama A. QuixBugs: a multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proc. of International Conference on Systems, Programming, Languages, and Applications: Software for Humanity: Companion*. https://doi.org/10.1145/3135932.3135941
19. Fraser G., Arcuri A.. Sound empirical evidence in software testing. In *Proc. of 34th International Conference on Software Engineering (ICSE)*, 2012. https://doi.org/10.1109/ICSE.2012.6227195
20. Gkortzis A, Mitropoulos D, Spinellis D. VulinOSS: a dataset of security vulnerabilities in open-source systems. In *Proc. of 15th International Conference on Mining Software Repositories*, 2018. https://doi.org/10.1145/3196398.3196454
21. Gyimesi P, Vancsics B, Stocco A, Mazinanian D, Beszédes Á, Ferenc R, Mesbah A. BugJS: a benchmark of JavaScript bugs. In *Proceedings of 12th IEEE International Conference on Software Testing, Verification and Validation*, ICST 2019. IEEE, 2019; 12 pages.
22. Svenonius Elaine. *The Intellectual Foundation of Information Organization*. MIT Press: Cambridge, MA, USA, 2000.
23. Wohlin Claes. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proc. of EASE '14*, 2014; 1–10.
24. Gao Z, Bird C, Barr ET. To type or not to type: quantifying detectable bugs in JavaScript. In *Proc. 39th International Conference on Software Engineering*, 2017.
25. Hanam Q, Brito FS, Mesbah A. Discovering bug patterns in JavaScript. In *Proc. of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.
26. Ocariza Jr FS, Pattabiraman K, Mesbah A. Detecting unknown inconsistencies in web applications. In *Proc. of 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.
27. Ocariza Jr, Frolin S, Pattabiraman K, Mesbah A. Detecting Inconsistencies in JavaScript MVC applications. In *Proc. of 37th International Conference on Software Engineering (ICSE)*, 2015.
28. Ocariza FS, Li G, Pattabiraman K, Mesbah A. Automatic fault localization for client-side JavaScript. *Software Testing Verified Reliability*. 2016; **26**(1). https://doi.org/10.1002/stvr.1576
29. Ocariza Jr, Frolin S., Pattabiraman K, Mesbah A. Vejovis: suggesting fixes for JavaScript faults. In *Proc. of 36th International Conference on Software Engineering*. https://doi.org/10.1145/2568225.2568257
30. Davis J, Thekumparampil A, Lee D. Node.Fz: fuzzing the server-side event-driven architecture. In *Proc. of 12nd European Conference on Computer Systems (EuroSys)*: Belgrade, Serbia, 2017.
31. Fard AM, Mesbah A. JavaScript: the (un)covered parts. In *Proc. of IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017. https://doi.org/10.1109/ICST.2017.28
32. Mirshokraie S, Mesbah A. JSART: JavaScript assertion-based regression testing. In *Web Engineering (ICWE)*, 2012; 238–252.
33. Mirshokraie S., Mesbah A., Pattabiraman K. Efficient JavaScript mutation testing. In *Proc. of 6th International Conference on Software Testing, Verification and Validation (ICST)*, 2013. https://doi.org/10.1109/ICST.2013.23
34. Mirshokraie S, Mesbah A, Pattabiraman K. Guided mutation testing for JavaScript web applications. *IEEE Transactions on Software Engineering*. 2015; **41**(5): 429–444. https://doi.org/10.1109/TSE.2014.2371458
35. Mirshokraie S, Mesbah A, Pattabiraman K. Atrina: inferring unit oracles from GUI test cases. In *Proc. of International Conference on Software Testing, Verification and Validation (ICST)*, 2016. https://doi.org/10.1109/ICST.2016.32
36. Mirshokraie S, Mesbah A, Pattabiraman K. JSEFT: automated JavaScript unit test generation. In *Proc. of 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015. https://doi.org/10.1109/ICST.2015.7102595
37. Quist C, Mezzetti G, Møller A. Analyzing test completeness for dynamic languages. In *Proc. of International Symposium on Software Testing and Analysis*.
38. Fard A. M., Mesbah A, Wohlstadter E. Generating fixtures for JavaScript unit testing. In *Proc. of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.
39. Artzi S., Dolby J., Jensen S. H., Moller A., Tip F.. A framework for automated testing of JavaScript web applications. In *33rd International Conference on Software Engineering (ICSE)*, 2011.
40. Mesbah A., Van Deursen A., Roest D.. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering*. 2012.
41. Andreasen E, Gong L, Møller A, Pradel M, Selakovic M, Sen K, Staicu C. A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys*. 2017; **50**(5): 66:1–66:36.

42. Hong S., Park Y., Kim M.. Detecting concurrency errors in client-side JavaScript web applications. In *Proc. of IEEE 7th International Conference on Software Testing, Verification and Validation*, 2014. https://doi.org/10.1109/ICST.2014.17
43. Dhok M, Ramanathan MK, Sinha N. Type-aware concolic testing of JavaScript programs. In *Proc. of 38th International Conference on Software Engineering*, 2016. https://doi.org/10.1145/2884781.2884859
44. Sobreira V, Durieux T, Madeiral F, Monperrus M, Maia MA. Dissection of a bug dataset: anatomy of 395 patches from defects4J. In *Proceedings of SANER*, 2018.
45. Pan K, Kim S, Whitehead EJ. Toward an understanding of bug fix patterns. *Empirical Software Engineering*. 2009; **14**(3): 286–315. https://doi.org/10.1007/s10664-008-9077-5
46. Seaman CarolynB.. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*. 1999; **25**(4): 557–572. https://doi.org/10.1109/32.799955
47. Nagappan N, Ball T.. Use of relative code churn measures to predict system defect density. In *Proc. of 27th International Conference on Software Engineering*.
48. Giger E, Pinzger M, Gall HC. Comparing fine-grained source code changes and code churn for bug prediction. In *Proc. of 8th Working Conference on Mining Software Repositories (MSR)*, 2011; 83–92. https://doi.org/10.1145/1985441.1985456
49. Campos E. C., Maia MDA. Common bug-fix patterns: a large-scale observational study. In *Proc. of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017. https://doi.org/10.1109/ESEM.2017.55
50. Zhong H, Su Z. An empirical study on real bug fixes. In *Proc. of 37th International Conference on Software Engineering (ICSE)*, 2015; 913–923.
51. Rostami S., Eshkevari L, Mazinanian D, Tsantalis N.. Detecting Function Constructors in JavaScript. In *Proc. of IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.
52. Eshkevari L, Mazinanian D, Rostami S, Tsantalis N. JSDeodorant: Class-awareness for JavaScript Programs. In *Proceedings of the 39th International Conference on Software Engineering Companion*, 2017.
53. Silva LH, Valente MT, Bergel A. Refactoring legacy JavaScript code to use classes: the good, the bad and the ugly. In *Mastering Scale and Complexity in Software Reuse*, 2017.
54. Thiemann Peter. Towards a type system for analyzing JavaScript programs. In *Programming Languages and Systems*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2005; 408–422.
55. Malik RS, Patra J, Pradel M. NL2Type: inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19. IEEE Press: Piscataway, NJ, USA, 2019; 304–315. https://doi.org/10.1109/ICSE.2019.00045
56. Anderson C, Giannini P, Drossopoulou S. Towards type inference for JavaScript. In *ECOOP 2005 - Object-Oriented Programming*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2005; 428–452.
57. Pradel M, Schuh P, Sen K. TypeDevil: dynamic type inconsistency analysis for JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015; 314–324. https://doi.org/10.1109/ICSE.2015.51
58. Hammoudi M, Rothermel G, Stocco A. WATERFALL: an incremental approach for repairing record-replay tests of web applications. In *Proc. of 24th International Symposium on Foundations of Software Engineering (FSE)*, 2016.
59. Stocco A, Yandrapally R, Mesbah A. Visual web test repair. In *Proc. of 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.
60. Perez A, Abreu R, D'Amorim M.. Prevalence of single-fault fixes and its impact on fault localization. In *Proc. of IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.
61. Pearson S, Campos J., Just R., et al. Evaluating and improving fault localization. In *Proc. of 39th International Conference on Software Engineering (ICSE)*, 2017.
62. Perez A, Abreu R, Van Deursen A. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proc. of 39th International Conference on Software Engineering (ICSE)*, 2017.
63. Wong WE, Gao R, Li Y, Abreu R, Wotawa F. A survey on software fault localization. *IEEE Transactions on Software Engineering*. 2016; **42**(8).
64. Krishna Murthy D. R., Pradel M.. Change-aware dynamic program analysis for JavaScript. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018; 127–137. https://doi.org/10.1109/ICSME.2018.00023
65. Hutchins M., Foster H, Goradia T, Ostrand T. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proc. of 16th International Conference on Software Engineering*, 1994.
66. Rahman MdR, Golagha M, Pretschner A. Pairika: a failure diagnosis benchmark for C++ programs. In *Proc. of 40th International Conference on Software Engineering: Companion*: Gothenburg, Sweden, 2018. https://doi.org/10.1145/3183440.3195097
67. Lu S, Li Z, Qin F, Tan L, Zhou P, Zhou Y.. Bugbench: benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, 2005.
68. Tan SH, Yi J, Mechtaev S, Roychoudhury A. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017; 180–182.
69. Saha RK, Lyu Y, Lam W, Yoshida H, Prasad MR. Bugs.Jar: a large-scale, diverse dataset of real-world Java bugs. In *Proc. of 15th International Conference on Mining Software Repositories (MSR)*, 2018. https://doi.org/10.1145/3196398.3196473

70. Dallmeier V, Zimmermann T. Extraction of bug localization benchmarks from history. In *Proc. of International Conference on Automated Software Engineering*.
71. Majd A, Vahidi-Asl M, Khalilian A, Baraani-Dastjerdi A, Zamani B. Code4Bench: a multidimensional benchmark of Codeforces data for different program analysis techniques. *Journal of Computer Language*. 2019; **53**: 38–52. https://doi.org/10.1016/j.cola.2019.03.006
72. Motwani M, Sankaranarayanan S, Just R, Brun Y. Do automated program repair techniques repair hard and important bugs. *Empirical Software Engineering*. 2018; **23**(5): 2901–2947. https://doi.org/10.1007/s10664-017-9550-0
73. IEEE standard classification for software anomalies, IEEE Computer Society, 2009.
74. Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong MY. Orthogonal defect classification—a concept for in-process measurements. *IEEE Transactions on Software Engineering*. 1992; **18**(11): 943–956.
75. Catolino G, Palomba F, Zaidman A, Ferrucci F. Not all bugs are the same: understanding, characterizing, and classifying bug types. *Journal of System Software*. 2019; **152**: 165–181. https://doi.org/10.1016/j.jss.2019.03.002
76. Li Z, Tan L, Wang X, Lu S, Zhou Y, Zhai C. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ACM, 2006; 25–33.
77. Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C. Bug characteristics in open source software. *Empirical Software Engineering*. 2014; **19**(6): 1665–1705.
78. Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, 2018; 129–140.
79. Thung F, Le XBD, Lo D. Active semi-supervised defect categorization. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, IEEE Press, 2015; 60–70.
80. Thung F, Lo D, Jiang L. Automatic defect categorization. In *2012 19th Working Conference on Reverse Engineering*, IEEE, 2012; 205–214.
81. Nagwani NK, Verma S., Mehta K. K.. Generating taxonomic terms for software bug classification by utilizing topic models based on latent Dirichlet allocation. In *2013 Eleventh International Conference on ICT and Knowledge Engineering*, 2013; 1–5.