

Systematic Review

# A Systematic Mapping of the Proposition of Benchmarks in the Software Testing and Debugging Domain

Deuslirio da Silva-Junior <sup>1</sup>, Valdemar V. Graciano-Neto <sup>1,\*</sup>, Diogo M. de-Freitas <sup>1</sup>, Plinio de Sá Leitão-Junior <sup>1</sup> and Mohamad Kassab <sup>2</sup>

<sup>1</sup> Instituto de Informática, Universidade Federal de Goiás, Goiânia 74690-900, Goiás, Brazil; deuslirio.junior@gmail.com (D.d.S.-J.); diogom42@gmail.com (D.M.d.-F.); plinio.sa.leitao.junior@ufg.br (P.d.S.L.-J.)

<sup>2</sup> Engineering Division, The Pennsylvania State University, Malvern, PA 16801, USA; muk36@psu.edu

\* Correspondence: valdemarneto@ufg.br

**Abstract:** Software testing and debugging are standard practices of software quality assurance since they enable the identification and correction of failures. Benchmarks have been used in that context as a group of programs to support the comparison of different techniques according to pre-established parameters. However, the reasons that inspire researchers to propose novel benchmarks are not fully understood. This article reports the investigation, identification, classification, and externalization of the state of the art about the proposition of benchmarks on software testing and debugging domains. The study was carried out using systematic mapping procedures according to the guidelines widely followed by software engineering literature. The search identified 1674 studies, from which, 25 were selected for analysis. A list of benchmarks is provided and descriptively mapped according to their characteristics, motivations, and scope of use for their creation. The lack of data to support the comparison between available and novel software testing and debugging techniques is the main motivation for the proposition of benchmarks. Advancements in the standardization and prescription of benchmark structure and composition are still required. Establishing such a standard could foster benchmark reuse, thereby saving time and effort in the engineering of benchmarks for software testing and debugging.

**Keywords:** testing; debugging; benchmark; software engineering



**Citation:** Silva-Junior, D.d.; Graciano-Neto, V.V.; de-Freitas, D.M.; Leitão-Junior, P.d.S.; Kassab, M. A Systematic Mapping of the Proposition of Benchmarks in the Software Testing and Debugging Domain. *Software* **2023**, *2*, 447–475. <https://doi.org/10.3390/software2040021>

Academic Editor: Daniela Soares Cruzes

Received: 26 July 2023

Revised: 18 August 2023

Accepted: 10 October 2023

Published: 12 October 2023



**Copyright:** © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

The adoption of software has been progressively observed in the control of various systems, including—but not limited to—automotive software [1], urban traffic management [2], and disaster monitoring [3]. However, despite advancements, the occurrence of coding errors leading to software bugs remains. Such software bugs can yield substantial economic losses, pose threats, and even compromise the well-being of users' integrity [4,5]. Consequently, both researchers and practitioners have invested efforts in the establishment of mechanisms aimed at diminishing the frequency of bugs and enhancing the quality of the delivered software product. In that context, the pivotal role of software testing and debugging (STD) cannot be overstated. Software testing is the systematic process of evaluating a software application to identify defects and ensure it meets specified requirements; whilst debugging is the process of locating and fixing defects or errors in a software program to restore its intended functionality [6]. These activities constitute fundamental steps of software quality assurance, addressing the identification and rectification of software faults [7–11]. Nonetheless, these activities are frequently labor-intensive, prone to errors, and time-demanding, with their complexity escalating when undertaken manually and as the scale of software projects amplifies [12]. In this vein, STD techniques have been automated with the generation of test cases, pinpointing faults, and facilitating program repair [10,13–15].

Benchmarks recurrently support the evaluation of STD techniques [16]. In the context of STD techniques, benchmarks comprise a set of programs, faulty versions of these programs, a suite of test cases, and bug reports or logs of test case executions. They can be used to evaluate the effectiveness of those techniques to achieve their aims, such as finding a fault [17]. A remarkable and popular instance of a benchmark for that context is Defects4J, which is made up of six Java programs with 438 real bugs and test cases that cover the faulty code. Benchmarks are well-accepted means in the state of the practice to support the evaluation of STD techniques [18,19]. Benchmarks can increase the reliability of the results obtained from the evaluation of STD techniques because they (i) work as a reference to compare the results delivered by different techniques, (ii) allow replication of the evaluation so that other researchers can reproduce the experiment applied in the evaluation and confirm (or refute) its results, and (iii) reduce threats to the validity of results by bringing a more systematic evaluation and producing data for future evaluation. Indeed, some recent literature reviews have included sections exclusively devoted to benchmarks and have presented examples of studies that use benchmarks in their projects [18,19]. However, studies that discuss aspects that motivate or guide the proposition of benchmarks are scarce.

The main contribution of this article is providing an overview of the reasons that have led researchers and practitioners to propose new benchmarks over the years. We analyze the state of the art by using the systematic mapping (SM) approach to collect evidence on this topic [20]. A total of 25 out of the initially retrieved 1674 studies were included and analyzed. The results reveal that (i) benchmarks have predominantly been proposed for software testing, bug diagnosis, and program repair, with fewer allocations for fault localization. (ii) Over a span of ten years, nine benchmarks were introduced, while the subsequent five years witnessed a notable increase to sixteen, emphasizing their pivotal role in technique evaluation. (iii) Approximately 50% of studies featured benchmarks exclusive to C or Java, while the remainder spanned language diversity. (iv) A substantial 92% of proposed benchmarks integrated real bugs sourced from controlled or production environments. (v) Benchmarks primarily arose due to factors including data absence, the imperative for authentic data, the scarcity of specialized data, incomplete bug understanding, and spontaneous result data provision.

The remainder of the article is organized as follows: Section 2 establishes a common vocabulary for our research by presenting useful definitions and a background; Section 3 presents the SM protocol and reporting. Section 4 presents a summary of contributions and research opportunities extracted from our SM; Sections 5 and 6 conclude this study by discussing threats to validity and final remarks, respectively.

## 2. Background

Software development is a costly and complex activity. The process involves humans and it is subject to their interpretation, which can lead to *mistakes* (i.e., a misinterpretation of the requirements). In turn, mistakes can lead to *defects (faults)* in the code, i.e., an implementation that does not conform to the requirements; and defects in the code can lead to *failures*, i.e., program executions that do not match the expected behavior and that generate a perception of one or more defects [21].

Software testing plays an essential role in software quality [9]. Unlike static testing, which focuses on reviewing software artifacts such as requirements documents, test plans, and code, dynamic testing is mainly concerned with revealing failures by executing the program. Benchmarks are used frequently in dynamic testing studies. Then, henceforth, we use the term *software testing* to refer to dynamic testing. Software testing involves the elaboration and execution of test cases [21]. Testing software, therefore, involves verifying the behavior delivered during code execution in response to a finite set of test cases. The set of test cases is made up of all possible inputs of a program (input domain) and its expected outputs [22]. Software is tested in practice through two techniques: functional tests and structural tests [11]. Functional tests deal with software code as a “black box” (that is,

without the tester's awareness of the internal logic of the software), where possible inputs are provided and evaluated to detect whether the code is being developed in accordance with the stakeholders' aims. Structural tests, however, handle software as a "white box", highlighting the internal structure and operation from a developer's perspective. Structural tests are complementary to functional testing techniques and are used to establish and contribute to software quality, both in internal aspects and to meet requirements [21,23]. When the test activities expose failures, the debugging process starts.

The debugging process may be structured into three steps, (i) defect localization, (ii) defect understanding (bug diagnosis), and (iii) defect/program repair[24–26]. According to Hailpern et al. [10], software debugging involves analyzing and modifying a program that does not match its specifications. Thus, the primary goal is to establish a new version of the program that is close enough to the original one but satisfies the previously violated requirements. The first activity (*defect localization*), also called fault localization (FL), consists of the precise determination of the location of the defect in the program; the second one (*defect understanding*) is related to obtaining knowledge about the fault and its behavior; the last step (*program repair*) consists of repairing the defect. Debugging is a time-consuming activity, which motivates the adoption of automated methods to proceed with its inherent steps.

Automated fault localization (FL) techniques are those used in software development to identify the locations or lines of code that are responsible for causing defects or errors in a software program. These techniques aim to narrow down the search for the root cause of a bug, making it easier for developers to locate and fix the issue. By automatically pinpointing faulty code, FL techniques expedite the debugging process and improve the efficiency of software maintenance. Some of the most popular automated fault localization techniques include (i) **spectrum-based techniques**, which analyze the program's execution traces, such as test coverage information or execution frequencies, to identify code segments associated with failing test cases. Examples include Tarantula, Ochiai, and DStar; (ii) **statistical debugging**, which analyzes historical debugging data to identify patterns or correlations between code and defects. Examples include delta debugging and probabilistic models; (iii) **mutation-based techniques**, which involve creating small changes (mutations) in the code to simulate defects and assess the effectiveness of test cases; (iv) **data flow analysis**, which tracks how data flows through the program to identify potential error propagation paths and isolate faulty code; (v) **constraint-solving techniques**, which formulate the debugging problem as a constraint satisfaction problem and use automated solvers to narrow down the possible fault locations; (vi) **search-based techniques**, which map fault localization to a search problem and explore the code space systematically to find the most likely fault locations; (vii) **machine learning-based techniques**, which adopt machine learning algorithms to predict potential fault locations in new code; and (viii) **program slicing**, which involves extracting a subset of codes that directly influence a specific program behavior, aiding in identifying the root cause of defects.

As the first part of the process of debugging, the FL activity aims to indicate the code portions with a high probability of containing the defect. The most popular techniques of *automated FL* use the information from the test case coverage provided by test case executions. Heuristics are then applied to the coverage data to indicate suspicious code. In addition to reducing costs and increasing the reliability of software, automated FL techniques include a low degree of human intervention in the FL process. Subsequently, once faults are located, the activity to diagnose the defect and understand its behavior starts. The experience of previously known bugs may help the debugging expert to understand the fault already found. Also, the reports generated by developers are an indication of the bug type in some cases; such analysis may support fault correction.

The third step of debugging involves program repair. Program repair consists of the replacement of faulty code by a corrected version of it. This activity is usually manual and repetitive, which leads to the adoption of automated methods, the so-called *automated program repair* (APR). APR proposes and adopts automated software bug fixes. APR uses a set of tests to guide the repair process, thereby ensuring better code quality and lower

maintenance costs, producing a variant of the program that meets the project specifications. Traditional APR consists of the generation of corrections and validation. FL techniques are performed to identify suspicious code fragments [27]. Once the fragment was identified, APR techniques can generate corrections. Then, the created code is submitted again to the same set of tests to ensure it is still conforming to them. Fail test cases (reproducing failures), and success test cases (characterizing expected behaviors) are commonly used to validate the correction of the candidate fix [27]. This procedure is repeatedly performed until a valid variant of the buggy code is found.

Novel fault localization and APR techniques are often proposed. To be reliable, they should be assessed and compared with the existing techniques in the state of the art. For this purpose, a common code foundation and a set of test cases are required to yield results that can serve as a comparison baseline among various techniques. The set of programs used for this purpose is often referred to as benchmarks. The IBM Dictionary of Computing [17] defines a benchmark as a reference point in which measures can be applied to evaluate software or hardware. In STD, benchmarks are usually a group of programs with some representation of real-world environments along with all the necessary instruments or characteristics of the techniques under evaluation. For instance, these instruments may be an available test case set, available source code, programs in a specific programming language, or a specific number of lines of code (LoC).

The Siemens Suite (SS) [28] is an example of an artifact that was not built to be specifically used as a benchmark but has become popular and frequently used for FL activities [29]. It consists of a reduced set of small-sized programs in C formed by seven programs and the largest one has less than 500 LoC. The bugs were artificially inserted in the SS programs through code mutation techniques. These artificial bugs are useful for simulating some real defects [30], although there are real bugs that are not reachable by inserting single, small faults (first-order mutations), such as a replacement of logical or arithmetic operators [31]. As the software FL improved, the programs used to demonstrate FL methods also had to be changed, to show the benefits in industry-like environments [18]. When a novel FL technique is proposed, besides comparing its results to real buggy programs, it is common to use SS as well as a comparison parameter.

The APR community maintains a website ([32]) where they suggest that the benchmarks *Defects4J* [33], *Codeflaws* [34], and *IntroClass and ManyBugs* [35] be used in research studies. These benchmarks contain real bugs and are often used to apply FL techniques. Although several benchmarks exist, novel benchmarks are still being proposed, which raises the following question: *why are existing benchmarks not enough?* The following section presents the protocol developed to guide an investigation on this topic.

For the scope of this article, two terms are important: *motivation* and *scope of use*. The *motivation* for creating a benchmark is understood as what lack has motivated the creation of new benchmarks, such as lack of data or benchmarks composed of code excerpts/programs to enable the testing of a specific platform, technology, or programming language; whilst the *scope of use* of the creation of a novel benchmark should be understood as the final target of application for the created benchmark, for instance, for an entire community or a particular research group. An example of motivation for creating a new benchmark is the multi-threaded Java programs [36]. Initially, there were no real programs to be used as benchmarks. Then, artificial defects were seeded in multi-threaded codes to create the preliminary benchmarks until codes with real bugs were provided to be used. Examples of benchmarks for different scopes of use include Defects4J [33] and the benchmark by Jooyong Yi et al. benchmark [34]. The former benchmark was created and delivered for the entire community, whilst the latter was created for the specific scope of research.

### 3. Systematic Mapping Study

This SM was structured according to the guidelines of Kitchenham and Charters, and Petersen et al. [20,37]. The main steps involved **planning**, **conduction**, and **reporting**, as follows.

### 3.1. Planning

In the **planning** step, the research questions were established, and the research protocol was defined.

The **goal** of this SM was to present the state of art on software testing and debugging benchmarks along with a comprehensive analysis of the motivations behind their creation. Hence, the studies of interest include those that introduce benchmarks in the context of STD techniques.

From the established goal, the following **research questions (RQ)** were defined.

**RQ1:** *What are the proposed benchmarks for STD and their target topics?*

**Rationale:** By answering this RQ, we aim to provide a list of benchmarks to support researchers in selecting the benchmarks that they could use to exercise their novel STD techniques empirically. Moreover, answering RQ1 also provides a classification of the reported benchmarks according to their target topics, i.e., the context for which it was proposed. The target topics include software testing, bug diagnosis, program repair, and fault localization.

**RQ2:** *What are the languages used to write the programs that compose the proposed benchmarks?*

**Rationale:** The program repair community website [32] points out existing benchmarks that are highly restricted to Java and C code. The answer to this RQ may provide a broader panorama about the benchmarks that are available and that are composed of programs written in other diverse programming languages.

**RQ3:** *Are the bugs that compose the proposed benchmarks real or artificial?*

**Rationale:** The program debugging community often discusses whether or not bugs artificially introduced in a real program could represent real bugs [31]. Hence, providing such information is an essential contribution to better support researchers when choosing benchmarks for their studies. The proposed benchmarks can be characterized by the nature (real or artificial) of the bugs.

**RQ4:** *What were the identified motivations for proposing the benchmarks?*

**Rationale:** Benchmarks are built to match a set of intentions. This RQ aims to reveal the main motivations and needs that lead the community to create the benchmarks reported in the included studies.

**RQ5:** *What was the identified scope of use for the proposed benchmarks?*

**Rationale:** The aim of answering this RQ is to map the scope of use that led researchers to create benchmarks, as discussed at the beginning of this section.

#### 3.1.1. Search Strategy

A control group was selected, a search string was elaborated, and online databases were chosen to proceed with an automatic search.

**Search databases.** We conducted searches in the following databases by applying filters on the titles, abstracts, and keywords. The databases were selected between the most common publication databases used to conduct systematic literature studies in software engineering [38,39]. The chosen databases comply with the recommendations made by Kitchenham and Charters [37] and Petersen et al. [40].

- IEEEExplore (<http://ieeexplore.ieee.org>) (accessed on 9 October 2023);
- ACM Digital Library (<http://dl.acm.org>) (accessed on 9 October 2023);
- Scopus (<http://www.scopus.com>) (accessed on 9 October 2023);
- Engineering Village (<http://www.engineeringvillage.com>) (accessed on 9 October 2023).

**Control studies.** The program repair community website provides a set of benchmarks considered relevant for the area [32]. Apart from benchmarks, the website also provides the

corresponding study that reports each benchmark proposition. We used that set of studies as a control group, i.e., a set of studies that should be retrieved by the elaborated search string. The control group includes the following benchmarks.

- Defects4J [33];
- Manybugs and IntroClass [35];
- Codeflaws [34];
- DBGBENCH [26];
- QuixBugs [41].

**Search string and calibration.** For the search string elaboration, we adopted the key terms used in the research questions, synonyms, and variations. Initially, the scope of research focused on *benchmarks for software debugging*, once the use of the word “testing” in the search string could retrieve an intractable number of studies. Then, this word was avoided at the first moment. The search string in the first try was:

“benchmark” AND “software” AND (“fault localization” OR “repair”)

The result was 192 studies from Engineering Village, 248 from Scopus, 128 from IEEE Xplore, and 176 from ACM DL; 744 studies is a reasonable number of studies, but unfortunately, some of the control group elements (e.g., Defects4J [33]) were originally proposed for software testing and were not retrieved using this string. So we had to expand the scope with the term “testing”. Also, while the control group was retrieved, this also resulted in more than 4000 results in each base. To narrow it down to only software testing that looks for bugs, we added the term “buggy”, resulting in the final search string presented previously. Hence, the words “benchmark”, “software”, and either “testing” or “debugging” are expected to appear in the relevant primary studies. Since FL and software repair studies might not explicitly use the word “debugging”, the terms “fault localization” and “repair” were also included. Furthermore, a term to represent bugs, such as “buggy”, is expected; for instance, while referring to the number of buggy programs in one benchmark. Thus, the following search string was built:

“benchmark” AND “software” AND (“fault localization” OR “repair” OR “testing” OR “debugging”) AND “buggy”

After identifying relevant synonyms to each term, the string evolved into:

(“benchmark” OR “benchmarking” OR “dataset” OR “dataset” OR “database” OR “datasets” OR “datasets” OR “benchmarks”) AND (“software” OR “program”) AND ((“fault localization” OR “error localization” OR “defect localization” OR “bug localization” OR “error localisation” OR “defect localisation” OR “bug localisation” OR “fault localisation”) OR (“software repair” OR “software fixing” OR “program repair” OR “program fixing” OR “bug fixing” OR “bug-fixing” OR “automatic repair”) OR (“software testing” OR “software test”) OR (“software debugging”)) AND (“bug” OR “defect” OR “buggy” OR “faulty” OR “failing” OR “failed” OR “bugs” OR “defects”)

### 3.1.2. Selection Criteria

We consider that a study proposes a benchmark when it exposes the obtainment of a new dataset or the grouping of information from different benchmarks. The study should also provide a URL to a repository with this novel dataset. If a study only describes other (existing) benchmarks and offers a link to each of them, we do not consider it as a study that proposes a benchmark, thereby justifying their exclusion.

These criteria are aimed at supporting a proper selection of the relevant studies for this SM, i.e., studies that answer the presented research questions. The following inclusion criteria are defined:

IC: The study proposes a benchmark and makes it available as a single project in a URL link.

Conversely, for eliminating non-relevant studies, the following exclusion criteria are defined:

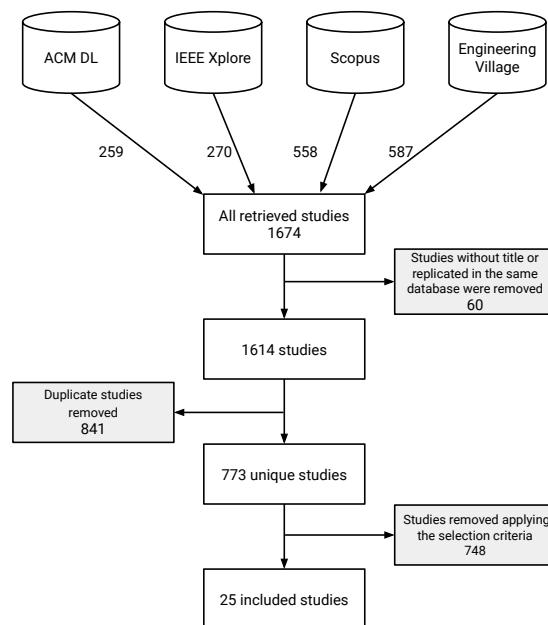
- EC1: The study is not related to software testing or debugging.
- EC2: The study does not propose a new benchmark specific to software testing or debugging techniques.
- EC3: The study is not written in English.
- EC4: The study is not a full article or is not available for access.
- EC5: The study does not provide the proposed benchmark for access as a single project in a unique URL.

### 3.1.3. Data Extraction and Synthesis Method

We defined a form to guide the data extraction process. This data extraction form consists of a set of questions aimed at gathering sufficient data to classify the benchmarks, measure the quality of the studies, and answer all research questions. The form is presented in Appendix A.

### 3.2. Conduction and Data Extraction

This search process was initially conducted in September 2018 and updated in January 2019, so studies published after January 2019 were not included. During the search, we did not limit the initial year. Figure 1 shows the number of studies obtained by applying the search string in the selected databases. Initially, the search string retrieved 1674 studies. Some of the retrieved studies did not present even a title, and others were replicated in the same database. After removing them, 1614 studies remained. The study involved five researchers. Two of them were master's students at that time, and the others were professors holding PhDs, with substantial experience in the software testing area. The professors supervised the students during the activity and contributed to data extraction and synthesis, in addition to resolving conflicts about the selection of studies.



**Figure 1.** Number of studies in each phase of the selection.

In the third step, duplicated studies were eliminated, i.e., the studies that were retrieved in more than one database search. After that, 773 studies remained to be analyzed. We included 47 studies to be entirely analyzed. After applying the inclusion and exclusion criteria in this set, 25 studies were included (22 only mentioned benchmarks, but did not propose them).

### 3.3. Reporting

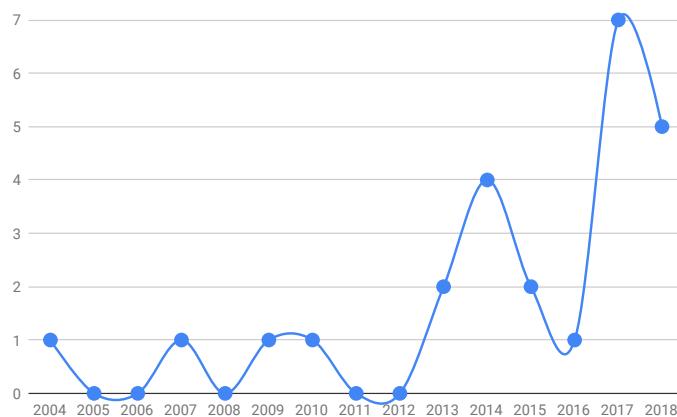
In this section, we report the findings of our SM. We report the quality of the included studies and we answer the research questions and provide data plots and tables to summarize the collected data.

All included studies are listed in Table 1.

**Table 1.** Primary studies included.

Study	Ref	Title	Year
S1	[36]	Compiling a benchmark of documented multi-threaded bugs	2004
S2	[42]	Extraction of Bug Localization Benchmarks from History	2007
S3	[43]	Clash of the Titans: Tools and Techniques for Hunting Bugs in Concurrent Programs	2009
S4	[44]	(Un-)Covering Equivalent Mutants	2010
S5	[45]	Empirical Evaluation of Bug Linking	2013
S6	[46]	The Eclipse and Mozilla Defect Tracking Dataset: A Genuine Dataset for Mining Bug Information	2013
S7	[47]	42 Variability Bugs in the Linux Kernel: A Qualitative Analysis	2014
S8	[33]	Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs	2014
S9	[48]	On the Effectiveness of Information Retrieval Based Bug Localization for C Programs	2014
S10	[49]	Automated Bug Finding in Video Games: A Case Study for Runtime Monitoring	2014
S11	[50]	A Dataset of High Impact Bugs: Manually-Classified Issue Reports	2015
S12	[35]	The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs	2015
S13	[51]	TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Data Center Distributed Systems	2016
S14	[52]	A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments	2017
S15	[53]	Automatic detection and demonstrator generation for information flow leaks in object-oriented programs	2017
S16	[34]	Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools	2017
S17	[54]	ELIXIR: Effective Object-Oriented Program Repair	2017
S18	[26]	How Developers Debug Software The DBGBENCH Dataset	2017
S19	[41]	QuixBugs: a multi-lingual program repair benchmark set based on the Quixey challenge	2017
S20	[55]	Secbench: A Database of Real Security Vulnerabilities	2017
S21	[56]	Crashing Simulated Planes is Cheap: Can Simulation Detect Robotics Bugs Early?	2018
S22	[57]	Large-Scale Analysis of Framework-Specific Exceptions in Android Apps	2018
S23	[58]	Mining repair model for exception-related bug	2018
S24	[59]	Pairika-A Failure Diagnosis Benchmark for C++ Programs	2018
S25	[60]	Repairing Crashes in Android Apps	2018

**Distribution over the years.** Figure 2 shows the distribution of studies that reported the proposition of a benchmark over the years. The first identified benchmark was published in 2004 [36]. None of the studies included any proposed benchmarks in 2005 and 2006. In 2007, a study on iBugs was published. iBugs comprised the first benchmark with semiautomatic methods to search faulty programs using the GitHub repository [42]. No relevant studies published in 2008 were found. In 2009, the first identified benchmark for two different languages (Java and C#) was published [43]. A benchmark created from a mutated code was proposed in 2010 [44]. During 2011 and 2012, benchmarks were not proposed from the included studies.



**Figure 2.** Studies per year.

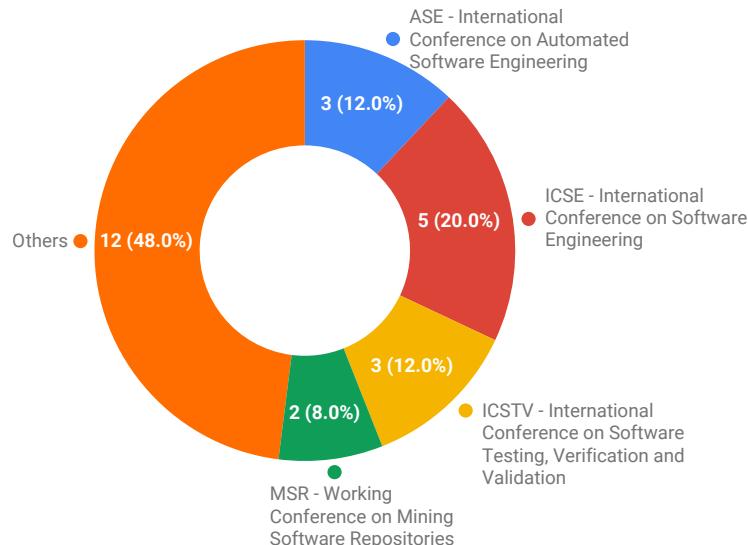
From 2013 onward, new benchmarks began to be proposed and were published at least once a year. We observed that the number of proposed benchmarks oscillated throughout the years; from the information plotted in Figure 2, the data present a waveform with increasing maxima over the years. Moreover, 25 benchmarks were proposed and published during the investigated 15 years, resulting in a publication rate of around 1.6 benchmarks per year. However, the wave amplitude and wavelength subsequently increased over the years, from one benchmark per biennium (one every two years from 2004 to 2011), the number increased over this decade (two benchmarks in 2012 and 2013 (S5 and S6); six benchmarks in 2014 (S7, S8, S9, and S10) and 2015 (S11 and S12); eight from 2016 (S13) to 2017 (S14, S15, S16, S17, S18, S19, and S20); and five in 2018 (S21, S22, S23, S24, and S25)). Hence, the frequency of publication increased, as well as the number of benchmarks proposed. The first decade of analysis produced 9 different benchmarks, while the last five years alone accounted for 16 different benchmarks.

The term ‘Benchmark’ is quite a recent term, which might explain why our search did not find any benchmark before 2004, despite using the term dataset in the search string to alleviate the impact of a single term to denote our research focus. We only considered—as a benchmark proposition—the studies that provided a URL to the dataset, which is also a recent practice. Reference [49] was published in 2014 at a conference and was expanded in a journal article in 2017. In this case, we only considered the earlier version.

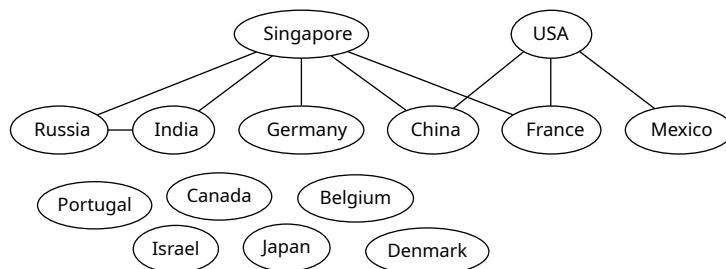
**Publication venues and affiliation data.** Studies that propose benchmarks for software testing and debugging are mainly published in well-known conferences. Five were published in each of the ICSE, ASE, and ICSTV conferences, with these hosting three studies of benchmark proposals each, as shown in Figure 3. The four conferences (ICSE, ASE, ICSTV, MSR) represent 48% (12 studies) of the venues used to publish the included studies. However, some of them were not published in the main track. From the 25 included studies, only 4 (S10, S11, S15, S23) were published in journals. From these data, we interpret and conjecture that most of the studies that only propose benchmarks have been published at conferences. Conversely, all the studies published in journals not only propose benchmarks but also use them to support the evaluation of novel techniques for software testing and debugging.

We analyzed the authors’ affiliations to discover the more representative countries related to benchmark proposals in software testing and debugging. Moreover, 8 out of 25 studies included authors from the USA; 6 studies included authors from Singapore; 5 studies involved authors from Germany; 3 studies were authored by researchers from China; and authors from Belgium, Canada, Denmark, France, India, Israel, Japan, Mexico, Portugal, and Russia were included in (up to) two studies. Some of the selected studies are the results of international collaborations. Figure 4 represents the collaboration network. Notably, Singapore and the USA boast the highest numbers of international collaborators, yet they do not collaborate with each other. Both China and France have developed research

with cooperation from Singapore and the USA. Belgium, Canada, Denmark, Israel, Japan, and Portugal do not exhibit international collaboration in our analysis.



**Figure 3.** Publication venues of studies.



**Figure 4.** Collaboration between authors from different countries.

**Studies Quality.** A set of quality questions (QQs) based on previous mapping studies [61,62] was incorporated into the form to allow measurement of the studies' quality. They can be answered with "Yes", "To some extent", or "No", and enable the assessment of the included studies according to the following parameters. Table 2 summarizes the answers to the QQs. Most of the studies received "Yes" to the quality questions, which indicates an overall good quality of the included studies. For each QQ, except for QQ6 and QQ7, at least 76% of all articles scored "Yes". Overall, studies that propose benchmarks do not discuss threats to validity and limitations (QQ6) or future work (QQ7).

QQ1: There is a rationale for the study to be undertaken.

QQ2: The authors present an overview of the related works and background of the area in which the study is developed.

QQ3: There is an adequate description of the context (industry, laboratory setting, products used, etc.) in which the work was carried out.

QQ4: The study provides a clear justification of the methods used during the study.

QQ5: There is a clear statement of contributions and sufficient data have been presented to support them.

QQ6: The authors explicitly discuss the credibility and limitations of their findings.

QQ7: The authors discuss perspectives of future works based on the contributions of the study.

Studies S2, S3, S8, S14, S15, S16, S18, S19, S24 received a "no" as the answer to QQ6 (credibility analysis and limitations). For QQ7 (discussion on future work), studies S6, S13, S18 received "no" as the answer, and studies S2, S7, S11, S15, S16, S19, S24 were assessed as "To some extent".

**Table 2.** Quality question results.

Quality Question	Yes	To Some Extent	No
QQ1	25	0	0
QQ2	20	4	1
QQ3	21	4	0
QQ4	22	3	0
QQ5	19	5	1
QQ6	13	3	9
QQ7	15	7	3

### 3.3.1. RQ1: What Are the Benchmarks Proposed in the Software Testing and Debugging Context and Their Target Topics?

As mentioned in Section 2, benchmarks consist of a group of programs with some representations of real-world environments, along with all the necessary instruments or characteristics for the techniques under evaluation. For instance, these instruments may be available test case sets, available source codes, programs in a specific programming language, or a specific number of lines of code (LoC).

Table 3 presents 25 benchmarks reported in the selected studies. Each benchmark is addressed by only one study once this MS considers studies on benchmark propositions, regardless of their subsequent use. All selected studies provide an external URL link to access the reported benchmark. However, in some cases, the presented link is not accessible. The included studies were grouped into four categories according to the target domain for which the benchmark was proposed. The benchmarks were proposed for applying and evaluating techniques in the following categories: software testing, fault localization, bug diagnosis, and program repair. The categorization of the studies in the four aforementioned categories was performed using a set of items extracted from the studies, including the keywords and classification of the aims of application of each benchmark mentioned by the authors in the study. In studies where the benchmark is related to more than one target topic, we considered only the most mentioned term in the article. Studies S1, S3, S4, S8, S13, S15, S20, S22 report the proposition of benchmarks for software testing. Studies S2, S9, and S24 report the proposition of benchmarks for fault localization. Studies S5, S6, S7, S10, S11, S21, S18 report the proposition of benchmarks for bug diagnosis. Studies S12, S14, S16, S17, S19, S23, S25 report the proposition of benchmarks for program repair. It is also possible to observe that there is a slight dominance from benchmarks proposed for software testing (eight benchmarks, 32% of the total number of proposed benchmarks found) over the other categories. Bug diagnosis and program repair have the same number of benchmarks (seven benchmarks, 28% of the total number of proposed benchmarks found for each topic), and fault localization only has three benchmarks (12% of the total number of proposed benchmarks found).

Figure 5 presents a cumulative view of the benchmarks proposed over the years in regard to their target topics (software testing, fault localization, bug diagnosis, and program repair). For exemplification purposes, considering software testing, one benchmark was proposed in 2004, and there was a break of new propositions until 2009 when the cumulative number of benchmarks proposed for software testing increased to two. The same pattern is followed in the other categories, showing the cumulative number of existing benchmarks over the years and none for the other categories. During the first five years (2004 to 2008), only one benchmark for software testing was recovered. From 2009 to 2010, two new benchmarks were proposed, remaining the same until 2013. From 2014 to 2018, the number of different benchmarks proposed for software testing over the years reached eight benchmarks. The same rationale was applied to the other categories. One important finding from this plot is that benchmarks for software testing, fault localization, and bug diagnosis have experienced progressive growth in number over the years. In addition, the program repair category experienced rapid growth, increasing the number of benchmarks from one to seven in just three years.

**Table 3.** Benchmarks and their target topics obtained from the included studies.

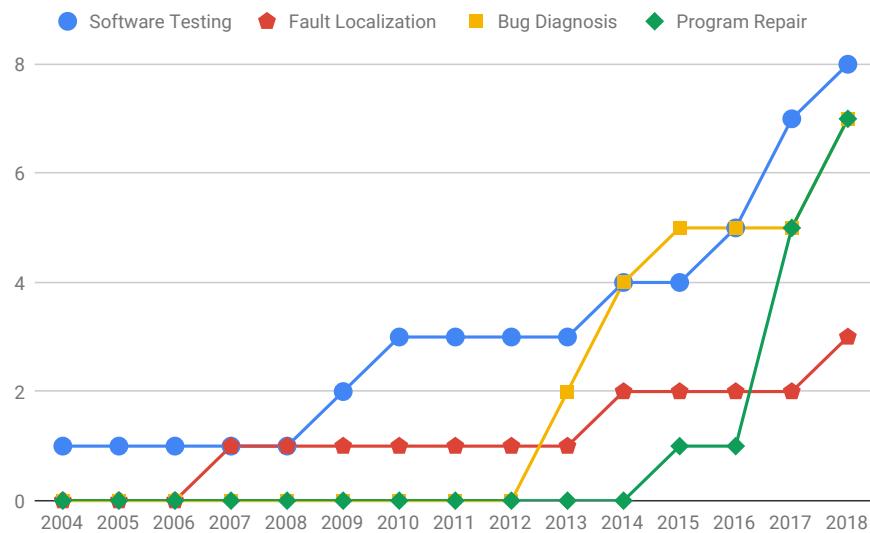
Benchmark Name or Author	Study	Topic
[Eytani et al.]	S1	
[Rungta et al.]	S3	
JAVALANCHE Subject Programs	S4	
Defects4J	S8	
TaxDC	S13	Software Testing
KEG Experiments	S15	
Secbench	S20	
Dataset Crash Analysis	S22	
iBugs	S2	
[Saha et al.]	S9	Fault Localization
Pairika	S24	
[Bissyandé et al.]	S5	
Eclipse and Mozilla Defect Tracking Dataset	S6	
The Variability Bugs Database	S7	
[Varvaressos et al.]	S10	Bug Diagnosis
High Impact Bug Dataset	S11	
DBGENCH	S18	
ArduBugs	S21	
ManyBugs and IntroClass	S12	
[Yi et al.]	S14	
Codeflaws	S16	
Bugs.jar	S17	Program Repair
QuixBugs	S19	
Exception-related bugs	S23	
Droixbench	S25	

A likely explanation for the behavior of the plot displayed in Figure 5 is that the domain of software testing is characterized by its historical precedence, with the progression of associated benchmarks unfolding gradually in parallel with the dissemination of research within this area. Notably, one of the studies included in the mapping reports the first benchmark developed for the software testing domain. For the program repair area, the influence of the study S12 (published in 2015) in the results plotted in Figure 5 is clear, which is a study published by one of the “creators” of the program repair area, which, in addition to proposing the benchmark, already uses it to empirically validate it. Another factor that may justify the rapid rise in the creation of benchmarks for program repair and bug diagnosis is that these areas are novel and on the rise, but such a conclusion cannot be made definitively and demands further investigation into other mappings.

### 3.3.2. RQ2: What Are the Languages Used to Write the Programs That Compose the Proposed Benchmarks?

Benchmarks are often composed of programs written in a specific programming language (or platform). Figure 6 depicts the distribution of the languages over the benchmarks. The most common languages found in the included studies are Java and C. Seven benchmarks were proposed exclusively for Java (S1, S2, S4, S8, S15, S17, S23). In turn, six benchmarks were exclusively proposed for C (S7, S9, S12, S14, S16, S18). It is important to remark that 6 out of the 25 benchmarks are not composed of source code (S5, S6, S11,

S10, S13, S21), e.g., the benchmark proposed by Bissyandé et al. [45] comprises a set of bug reports that can be used in language-independent contexts.



**Figure 5.** Benchmarks available per year, and their target topics.

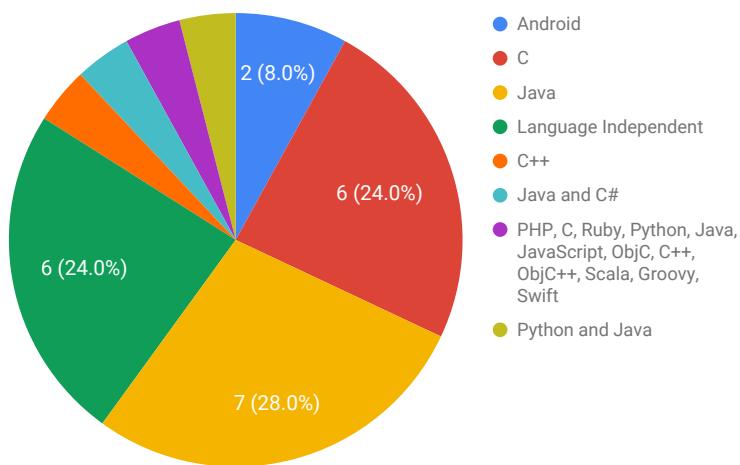
A relevant finding refers to the benchmarks dedicated to the Android platform (S22 and S25). Although these benchmarks involve Java code, different coding libraries and tools are needed to deal with this specific platform, which motivates the creation of those new benchmarks and restricts the use of Java's existing ones.

Then some benchmarks are specific to a programming language, whilst some of them target a specific platform, such as Android. A remarkable benchmark reported by one of the included studies (S20) deals with a specific type of software fault, i.e., this benchmark brings a set of test cases on security vulnerabilities. Furthermore, such a benchmark is not focused on a specific programming language but covers several of them (PHP, C, Ruby, and others). Table 4 complements Figure 6 by explicitly showing the programming languages and their respective included studies. A single benchmark was proposed for C++ (S24). It is also important to remark that 10 out of 25 (40% of the studies) report benchmarks for Java, being exclusive or also applicable to other languages (C# in S3 and Python in S19).

Figure 6 shows a clear predominance of C and Java datasets. Moreover, 52% of the benchmarks (13 of them) are exclusively for Java or C techniques. A likely explanation for C and Java having the highest values is that these are popular languages that ended up becoming mainstream in the areas of testing and debugging, which could be considered an expected/predictable result. This results corroborate prior studies, with the status of data being preserved. In particular, in 2015, a systematic mapping study showed that these languages were already the most present in testing activities, with 50% of the included studies reporting the use of JUnit as the main framework [63]. This information highlights the point that researchers, who wish to explore aspects not provided by certain languages (such as bugs related to functional languages), may not be able to use existing datasets, and need to create new benchmarks to evaluate their research studies. Table 4 shows the programming language related to each study.

### 3.3.3. RQ3: Are the Bugs That Compose the Proposed Benchmarks Real or Artificial?

The origin of bugs was divided between real and artificial bugs. However, during the review, we noticed that the real bugs originate from two different contexts: *a controlled environment* and *a production environment*. The former comprises situations where code and bugs are derived from student exercises or programming competitions. The latter involves most of the benchmarks and contains programs in production with real defects, e.g., Defects4J [33], compiled from large-scale open-source projects.

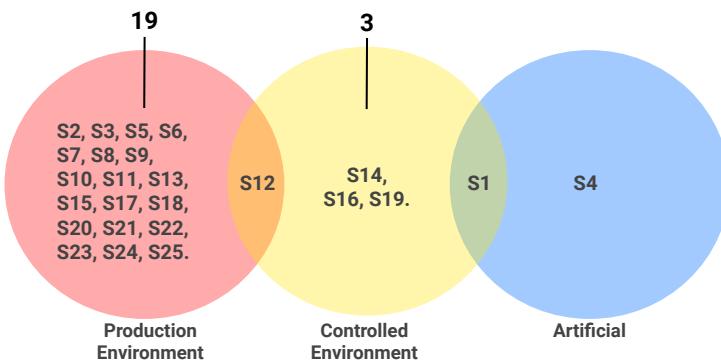


**Figure 6.** Programming language or OS dependency of the benchmarks.

**Table 4.** Benchmarks grouped by programming language or technology dependency.

Programming Language or OS	Studies
Android	S22, S25
C	S7, S9, S12, S14, S16, S18
C++	S24
Language Independent	S5, S6, S10, S11, S13, S21
Java	S1, S2, S4, S8, S15, S17, S23
Java and C#	S3
PHP, C, Ruby, Python, Java, JavaScript, ObjC, C++, ObjC++, Scala, Groovy, Swift.	S20
Python and Java	S19

It is important to highlight that before access to open-source projects was widely provided through a development platform such as GitHub (<https://github.com/>) (accessed on 9 October 2023)), the most used bug benchmarks had artificial bugs or were based on student-made programs. Figure 7 shows that except for S4, which proposes a benchmark for mutant evaluation (purely artificial bugs), and S1, which contains bugs that are intentionally but artificially provided by students in a controlled environment, all the benchmarks present real bugs. Three studies report benchmarks exclusively elaborated with bugs from a controlled environment (S14, S16, and S19). S16 and S19 were obtained from programming competitions. S14, in turn, comprises a dataset with 661 programs obtained from students of an undergraduate course on programming foundations. S12 is the study that combines two different sets of programs for the same benchmark: one set with programs obtained from students, and another set with programs provided from a production environment. The remaining 19 benchmarks correspond to a production environment or are provided as bug reports of existing tools.



**Figure 7.** The origin of bugs in the benchmarks reported from the 25 studies.

### 3.3.4. RQ4: What Were the Identified Motivations for Proposing the Benchmarks?

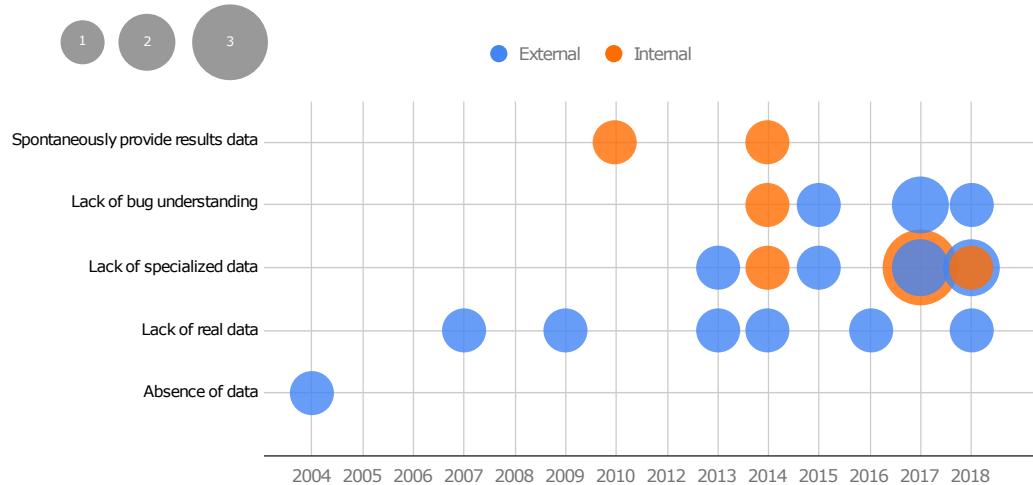
From the data extracted regarding motivations, it was possible to categorize the motivations for the benchmarks into five types, as follows.

1. **Absence of data:** This motivation comprises the situation where a testing or debugging technique exists, and an expert intends to assess the technique. Once the evaluation of those techniques requires a dataset, the absence of data motivates the creation of a new benchmark. In this scenario, even simple datasets with synthetic data are useful. For instance, Eytani et al. show that concurrency defects are difficult to cover and analyze without a standard dataset. To evaluate and compare techniques developed to deal with these types of defects, Eytani et al. proposed the first benchmark of multi-thread programs in 2004 [36]. Only study S1 falls into this class.
2. **Lack of real data:** This motivation targets techniques whose evaluations overcome artificial data or are restricted to real data. With this motivation, iBugs was proposed in 2007 [42]. The authors highlight that until that moment, the benchmarks available for debugging only had artificially seeded defects. Due to the difficulty of validating whether artificial bugs represent reality, it was necessary to create a new benchmark with real defects in large programs. Studies S2, S3, S6, S8, S13, and S23 fit into this class.
3. **Lack of specialized data:** Some methods need specific information to be evaluated. For instance, the evaluation of crashes in a mobile platform inherently requires code and data that are specific to that platform. Hence, highly specialized data are demanded in some categories of software, which motivates the creation of new benchmarks. This motivation is related to benchmarks that aim to fulfill this lack and to make the specialized data available. For instance, in 2013, Bissyandé et al. [45] proposed a benchmark that contains bug reports. Until that moment, techniques of bug localization that use such information could not be evaluated or straightforwardly compared to other techniques due to the lack of specialized data. Android crash automated repair techniques also match this category, since it exposed the need for specialized data available in benchmarks, as reported in S25 (Droixbench benchmark). Benchmarks that fall into this type of motivation often require much attention to avoid biased data, since a technique can be beneficial when the same people create both the benchmark and the technique under evaluation. This is actually a recurrent threat to the validity reported by the included studies themselves as they address the need to develop new benchmarks to evaluate a technique also created by the same authors (e.g., [48,52]). Studies S5, S9, S12, S14, S15, S16, S17, S19, S21, S24, and S26 belong to this class of motivation.
4. **Lack of bug understanding:** This motivation refers to structuring data and providing additional data aiming to support the understanding of classes and origins of bugs. For instance, Reis and Abreu [55] propose a set of security bugs, created specifically to support the analysis of such a defect type. Apart from the previous types of

motivation, this one comprises the lack of more in-depth information about bugs. Studies S7, S11, S18, S20, S23 belong to this class.

5. **Spontaneously providing results data:** This motivation refers to a spontaneous contribution by creating a basis for the evaluation of further techniques; i.e., the authors made their results available as benchmark data. Studies S4 and S10 made their study results (respectively, mutated code and run-time log) available for community use to compare other techniques in the same context. Only studies S4 and S10 belong to this class.

Figure 8 shows the accumulated evolution of motivation for the proposition of benchmarks over the years. One interesting finding is related to the accelerated increase in the diversity of motivations for proposing benchmarks over the years. For instance, benchmarks proposed due to the *lack of specialized data* increased in the last three years. Those benchmarks were proposed to support the evaluation of specific characteristics of a software testing or debugging techniques that have not been exploited by existent benchmarks yet according to the authors. One example is S19, which was created to enable researchers to analyze the same defect in programs written in different languages. Then the authors proposed a benchmark with equivalent programs with the same defect but written in two different languages (Java and Python). This was a different motivation when compared to the previous ones. Another example is S24, which reported a benchmark for techniques to be applied in C++ since none of the previous benchmarks could be used for that purpose.



**Figure 8.** The motivation behind the selected primary studies, classified as the absence of data, lack of real data, lack of specialized data, lack of bug understanding, and spontaneously providing results data.

Benchmarks motivated by the *lack of real data* have emerged over the years. They were motivated by the lack of real data and were created to be applied in techniques where artificial data were not enough. One example of a lack of real data was the dataset crash analysis [57], created for Android crash repair techniques. This was the first benchmark of the area that supported the evaluation of techniques with real data for that context.

### 3.3.5. RQ5: What Were the Identified Scopes of Use for the Proposed Benchmarks?

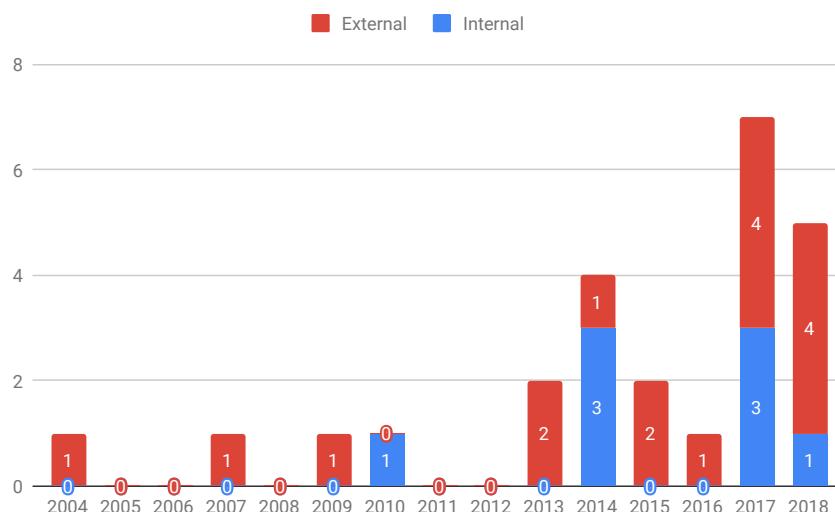
Multiple objectives can guide the designing of benchmarks. However, those purposes can be classified into two general categories, which are:

1. **External:** This is the case where benchmarks are made available to supply the community needs. For instance, Pairika (S24) is a benchmark proposed for the bug diagnosis of C++ programs, i.e., it was not created for a specific technique, but an open-accessed use of an entire programming community. Several research studies can use the datasets provided by benchmarks of this category. We identified 17 studies

that are addressed to this category: S1, S2, S3, S5, S6, S8, S11, S12, S13, S16, S18, S19, S20, S21, S22, S23, and S24.

2. **Internal:** This class of objectives represents the benchmarks built to evaluate the technique of a particular research group. In general, they may be used in other related studies but the main objective was to provide a benchmark because no other existing one could be used in the study. In those cases, the included study actually presents testing or debugging techniques, and the benchmark is jointly proposed to introduce the technique being reported. Eight studies are addressed to this category, S4, S7, S9, S10, S14, S15, S17, and S25.

Figure 9 shows the predominance of the external scope of use among the reported benchmarks. This shows that most of the proposed benchmarks are available for the entire community, not being specific to a particular technique or method.



**Figure 9.** The scope-of-use distribution (external or internal) over the years.

### 3.4. Synthesis

This section provides synthesis results obtained from (i) additional findings that are not directly related to the answers to the research questions (Section 3.4.1), and (ii) information extracted from crossing two or more research questions (Section 3.4.2). Also, Appendix B presents a table that compiles the information obtained by answering each RQ, which allows the reader to confirm the findings reported here and obtain more useful data from them. We provide a repository with a complete list of the studies and the URL link to their reported benchmarks ([https://github.com/I4Soft/Testing\\_and\\_Debugging\\_Benchs](https://github.com/I4Soft/Testing_and_Debugging_Benchs) (accessed on 9 October 2023)).

#### 3.4.1. Additional Findings

The additional findings refer to the information obtained from the included studies as a subjective perception that may be relevant to testing and debugging practitioners, as follows.

**Create benchmarks regardless of software testing and debugging techniques.** New techniques may have no data in the literature that support their assessment in regard to other techniques (for instance, study S9 proposes a dataset with bug reports and codes and uses it to validate a novel bug localization technique), which require the creation of new benchmarks. Then a recurrent threat reported in the included studies is the creation of benchmarks along with the creation and evaluation of techniques: a benchmark was created ‘for’ that technique, and the benchmark is suitable for the technique, not evaluating it at large. To avoid such excessive fit, benchmarks should be unbiased, i.e., when comparing different techniques on the same dataset, the dataset

should not positively or negatively influence the results of the techniques. Thus, from the review of the included studies, a possible perception is that benchmarks should be proposed independently of the techniques they are used to evaluate.

**Benchmarks are often surpassed.** A motivational example comprises the use of benchmarks composed of programs with artificial bugs. The proposition of iBugs [42], for instance, was motivated by the lack of real data. Specifically, they report that the existent benchmarks, such as Siemens Suite [28], were wholly composed of programs with artificial bugs. Then, at some moment, the techniques were well-succeeded to deal with artificial bugs but not validated with real bugs. In turn, the iBugs was not exhaustive about real bugs since it was only one program with multiple versions of real bugs. This characteristic raised the need to create Defects4J (initially composed of five programs) [33], superseding iBugs since novel techniques demanded benchmarks with a larger number of programs with real bugs to deliver a better evaluation of testing techniques. In line with this perspective, in Section 3.3, we observed a tendency for benchmarks to be created in a wave pattern, i.e., in regular periods (about every two years, and increasing). We conjecture that this pattern will be repeated in forthcoming years, as novel techniques can be created in future years due to similar motivations: lack of data, novel techniques, and existing benchmarks being deprecated.

**Out-of-the-box use.** Another interesting finding is the fact that benchmarks can be used in other areas—even if they are not explicitly created for that purpose. An instance is Defects4J, which is recurrently used in program repair [64] and fault localization [65], despite the fact that software testing was its original target topic. Other benchmarks exhibit this same phenomenon, reinforcing the out-of-the-box usage of them. For instance, ‘Codeflaws’, which was proposed for program repair, has also been utilized in fault localization [14].

### 3.4.2. Triangulation

Triangulation is a procedure that combines results obtained from answering the elaborated individual research questions, but that could not be found by analyzing them in isolation.

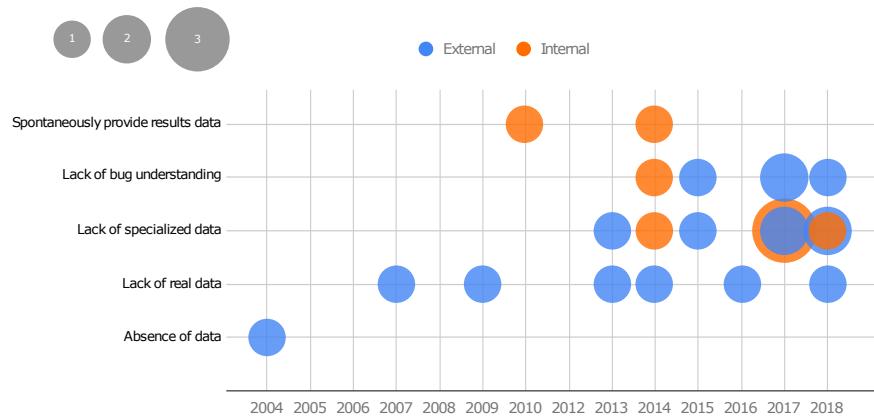
Figure 10 shows a bubble plot that combines results obtained from RQ4 and RQ5 over the years, i.e., the plot shows how the motivation type (absence of data, lack of real data, lack of specialized data, lack of bug understanding, and spontaneously providing results data) impacts the scope of use of the benchmark proposition (internal and external) over time.

From the data, it is possible to observe that, throughout the years, all the benchmarks motivated by a wish to spontaneously provide results data were conceived for an internal scope of use, which means that the benchmarks were created in association with a particular technique and it was used to support the evaluation of the associated technique. However, they were made available to the community. On the other hand, all the proposed benchmarks motivated by the absence of data and lack of real data have an external scope of use, i.e., the main aim of their creation was to make them available to the community. Among the seven benchmark propositions in 2017, three were motivated by the lack of specialized data and with an internal scope of use.

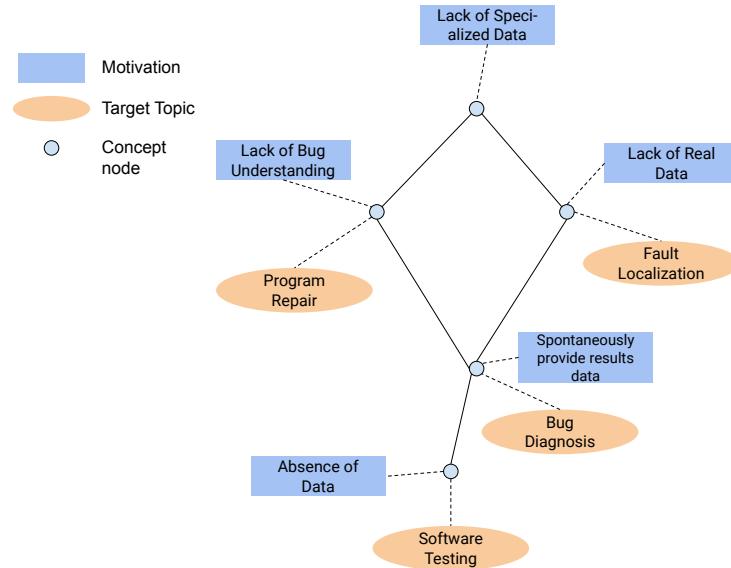
We also exploited the relationship between RQ1 and RQ4, respectively, related to the target topic and motivation. Figure 11 presents a formal concept analysis (FCA) of the literature on benchmark proposals. FCA [66] is an analysis technique that can be applied to data that report objects, attributes, and binary relationships between them. Ellipses represent the objects, and rectangles illustrate the attributes.

A concept is illustrated as a node, and it is not associated with a name. However, it links a set of objects (target topics) to a set of attributes (motivations). An additional recommendation for reading such a plot is as follows: if the plot is read from top to bottom, a motivation was applied (in the included studies) to all the target topics underneath it in the diagram and all of them that could be reached through some edge from it. For instance,

reading the plot, we can infer that the motivation '*lack of specialized data*' was a motivation for all four target topics (program repair, fault localization, bug diagnosis, and software testing). In turn, the motivation '*spontaneously provides results data*' was used in studies that report bug diagnosis and software testing, but not for fault localization and program repair. From this plot, we can also infer that the '*lack of real data*' was used for all target topics, except program repair. Conversely, '*lack of bug understanding*' was used for all target topics, except for fault localization. In turn, the '*absence of data*' was found as a motivation in the included studies only in benchmarks proposed for software testing.



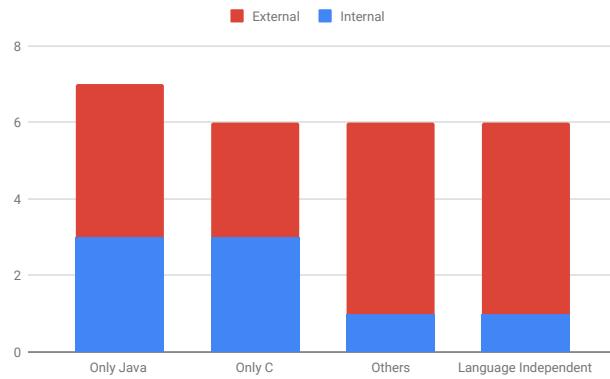
**Figure 10.** The benchmarks classified per motivation and scope of use.



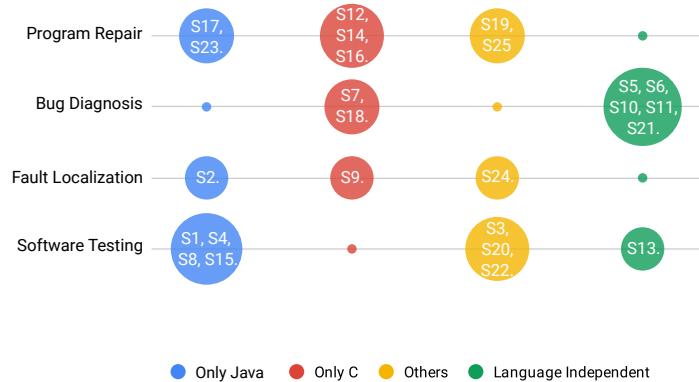
**Figure 11.** Concept analysis about motivation and target topics.

Figure 12 shows the relationship between RQ2 (programming language) and RQ5 (scope of use). The aim of combining those answers was to understand the relationship between the programming language communities and the benchmark's scope of use. Among the seven benchmarks exclusively proposed for Java, three (43% of them) are for the internal scope of use, and four (57% of them) for the external scope of use. Six benchmarks were proposed for C: three (50%) for internal and three (50%) for external scope of use. In turn, among the six benchmarks proposed for programming languages that are not only Java and C, such as C++, Python, and C#, five of them (83% of them) are proposed for external scope of use. The same happens for language-independent (or technology-independent) benchmarks. We noticed that benchmarks that are not exclusively proposed for Java or C languages were mostly created to supply the needs of particular communities that did not have so many possible benchmarks as Java and C.

Figure 13 uses data from RQ1 (target topic) and RQ2 (programming languages). Bubble sizes represent the number of studies for each target topic (rows) displayed according to their respective programming language colors (columns). We can observe that bug diagnosis is the main target topic for benchmarks proposed for language-independent contexts. An example of this is bug reports, which do not require specific code, but only information about the bug. Other important findings are that (i) benchmarks with only Java programs (four studies) are tightly related to software testing, and Benchmarks in the class Others (three studies) are also highly related to software testing, likely attempting to offer an alternative for software testing beyond Java. (ii) C is closely related to programming repair (three studies).



**Figure 12.** Programming languages related to the scope of use.



**Figure 13.** Programming languages related to target topics.

#### 4. Summary of Contributions and Research Opportunities

This section summarizes the main results and contributions obtained from this study and the research opportunities raised from this mapping.

The contributions of this article include:

- **Mapping of the area:** This study provides an overview of the proposition of benchmarks for software testing and debugging, including several dimensions of the area, such as (i) research topics for which benchmarks were proposed, (ii) programming languages contained in the dataset associated with the benchmark, (iii) sources of defects in the programs contained in the benchmark, and (iv) information related to the intention of the proposition of a new benchmark, as motivation and scope of use.
- **A list of proposed benchmarks for software testing and debugging:** A significant contribution from this study is a list of 25 proposed benchmarks for testing and debugging software in the period 2004–2018. This set of benchmarks can be adopted and used by researchers and practitioners who create novel software testing and debugging techniques. Professionals can adopt or adapt an existing benchmark that matches the requirements (programming languages, research topics, or origin of bugs) of their

novel techniques to support the evaluation of their techniques instead of creating new benchmarks. This contribution can increase the level of reusability of benchmarks and reduce the efforts frequently undertaken to elaborate new benchmarks. Moreover, the adoption of the same benchmark can also increase the level of reproducibility of the studies and assessments performed on testing and debugging techniques.

This mapping was focused on studies that reported the *proposition of benchmarks*. However, another important gap to be exploited is related to the *use* of benchmarks. So we envision the following research opportunities related to benchmark proposals and usages, as follows.

- **Guidelines for proposing benchmarks:** During the analysis of the included studies, we do not extract some characteristics due to the lack of uniformity of the provided data about the benchmarks and how they are structured. For instance, QuixBugs [41] has 40 Python and Java programs; the study that presented Defects4J [33] had five programs, but currently, Defects4J is composed of six programs. iBugs initially only had one program, but currently, it has three programs with different versions. S10 only has runtime data from program execution, and S11, in turn, only has bug reports that are manually classified. The material that composes a proposed benchmark is often presented in different sizes (the number of versions and lines of code), but some studies do not even have descriptions of these aspects. From these data, we observe that a lack of standard description and format exists among the benchmarks, potentially hindering a more in-depth analysis of other characteristics not covered herein. By defining guidelines on how to propose benchmarks, a list of requirements could be presented and should be considered for creating and reporting new datasets, which could foster standardization for the area, enabling an even more solid analysis, apart from the selection and use of benchmarks.
- **Suitability of a benchmark according to the evaluation interests:** Benchmarks are often used for the assessment of testing and debugging techniques, revealing the potential of benchmark characteristics to impact the evaluation results. Hence, it is best to be aware of how a benchmark could influence the evaluation results regarding some attributes, such as efficacy, type of faults, and efficiency, among others. In this sense, more development is required to create a framework to conduct a benchmark analysis in the context of testing and debugging target topics. For instance, the results of such a study could help one to evaluate whether a benchmark is suitable to be used in the evaluation procedures of different techniques in specific research fields. Such a framework could also contribute to comparing several benchmarks and evaluating whether one of them is more effective for a research topic.
- **Benchmark usage by research field:** Benchmark usage refers to the search for studies in order to analyze how a benchmark has been applied by addressing target topics, methods (or method categories) under evaluation, evaluation metrics, and research questions. More review studies about benchmarks are also needed, especially from the point of view of different research topics (for example, those used in Section 3.3.1) seeks to expose which benchmarks have been and are being used during the evaluation of new approaches in that particular research area, as well as map the evaluation metrics and methods that are commonly used as baselines. This type of mapping may be one more artifact-used to support researchers in selecting *which* benchmarks to use, but also in providing guidance on *how* to use them.
- **Interchangeability/Customization of benchmarks between different domains:** Some benchmarks have been used in topics other than their initial aim, as seen in Section 3.4.1. After analyzing the selected studies, we noticed that not all benchmarks suggested by the program repair community were proposed for this purpose; however, this did not prevent them from being used for a different context. Hence, a study about the interchangeability of benchmarks between areas such as fault localization and program repair may expand the suite of programs that researchers in both areas may use.

- **Definition of guidelines for benchmark selection:** In this mapping, we reported the potential bias that can emerge from the use of a benchmark jointly proposed with the technique that uses it to be evaluated. Another potentially biased context is the selection of benchmarks, i.e., when a researcher chooses a benchmark to use in his/her study. Studies often present evaluations obtained by using only some parts of a dataset offered by a benchmark. For instance, not all programs contained in Defects4J have been used during the analysis and evaluation of some techniques, which can bring about some imprecision to the obtained results. For reliability purposes, benchmark-based technique evaluations should be conducted according to guidelines. These guidelines could advise researchers on how to select benchmarks to reduce bias, and maximize the empirical value of the results since the most popular benchmark is not necessarily the most suitable for a context.

## 5. Threats to Validity

The results presented by this systematic mapping may have been affected by some factors that we discuss in this section. We identified the main threats regarding the *omission of important primary studies, selection reliability, data extraction, and quality assessment*. We seek to minimize them by employing some mitigation actions.

**The omission of important primary studies:** Important studies can also be missed during the automated search in the selected databases. To alleviate this threat, we (i) adopted the set of bibliographic bases recommended by Dyba et al. [67], which could return a more significant number of relevant studies of the area, (ii) did not delimit a time period in order to obtain the maximum of relevant studies, (iii) considered keywords suggested by software testing and debugging experts, (iv) performed trial searches to calibrate the search string, and (v) used a set of studies as a control group to confirm that our search string retrieved the main studies of the area.

**Selection reliability:** The inclusion and exclusion processes can also be critical since a misunderstanding could cause the exclusion of relevant studies or the inclusion of irrelevant studies. At least two researchers were involved in each step of the study, aiming to mitigate this problem and others related to the interpretation during the selection process. If a conflict emerged, a consensus meeting was used to reduce the possibility of misinterpretation in this step.

**Data extraction:** A recurrent threat to this systematic mapping refers to how the data were extracted from the primary studies. To alleviate this threat, we performed the extraction and synthesis of data in a cooperative way with two reviewers working together to reach an agreement on possible conflicts caused by the extracted data and their classification. Furthermore, when disagreement occurred, consensus meetings were conducted to ensure a full agreement between the reviewers.

**Quality assessment:** We elaborated a set of quality questions to assess the quality of each selected study. A possible threat that emerges is that the results of the quality questions can be influenced by the interpretation of the reviewers. However, quality questions were also performed in a double-check procedure so that both reviewers involved in the extraction answered the same quality questions for the same studies in order to obtain a reliable opinion about the quality of that study, which reduces the threat to the validity of the conclusions obtained about the quality of the included studies.

## 6. Final Remarks and Future Work

This article reported the results of a systematic mapping carried out to offer a broad panorama on the proposition of benchmarks to support the evaluation of software testing and debugging techniques. Twenty-five primary studies were selected and analyzed. From the analysis, we extracted important lessons that can be used to support testing and debugging techniques. We bring the following conclusions:

- Benchmarks are mainly proposed for software testing, bug diagnosis, and program repair, rather than fault localization (only three studies reported the proposition of benchmarks for this domain);
- The first decade of analysis produced nine different benchmarks, whilst the period 2013–2018 was responsible for 16 different benchmarks, showing a significant increase in the number of benchmarks being proposed. This result endorses the importance of benchmarks for supporting the evaluation of software testing and debugging techniques and an increase in the interest over the years;
- Approximately 50% of the retrieved studies report benchmarks proposed for exclusively C or Java. The other half refers to language-independent benchmarks or benchmarks proposed for other languages;
- Most of the proposed benchmarks (92% of the studies, 23 out of 25) are composed of real bugs from a controlled environment (we understand a controlled environment as a non-commercial situation in which software testing activities are carried out, such as in academic environments and competitions) or a production environment (a production environment comprises the software testing environment for commercial software in production);
- The motivation for a proposition of benchmarks could be classified into five different categories: (i) *absence of data*, representing benchmarks proposed because there are no available data (such as a set of buggy programs or execution logs) until that moment to support the proper evaluation of a testing or debugging technique; (ii) *lack of real data*, revealing that several techniques demanded real data and the use of programs with real defects; (iii) *Lack of specialized data*, which indicates that while some benchmarks do exist, they may not be composed of programs in a specific programming language or contain descriptive data about bug fixes, revealing a lack of specialized data that motivates the proposition of novel benchmarks; (iv) *lack of bug understanding*, since some benchmarks exist, but their data are not structured or provided in a way that supports understanding the classes and origins of bugs, thereby motivating the proposition of new ones, and (v) *spontaneously providing results data*, as some benchmarks are obtained through the execution of techniques and are provided to the community as a public dataset.
- The scope of use for the creation of benchmarks can be split into two classes: internal and external. The former refers to an inner evaluation of particular testing or debugging techniques, which demand the creation of a benchmark. The latter refers to benchmarks created to be available for community needs.

Although new benchmarks have recently been proposed, the field still demands the proposition and consolidation of guidelines to support the introduction of further benchmarks. Our investigation revealed that, in numerous instances, new benchmarks were developed due to the deficiencies of existing ones; however, the newly created benchmarks neither exhibited a high degree of quality nor showed substantial potential for reuse or adaptability. Hence, a more in-depth understanding of customization, quality, and reuse of benchmarks is also needed.

Nevertheless, this study contributed through an analysis of the recent history of benchmark propositions and unveiled several vital characteristics of existing ones. Moreover, a list of numerous benchmarks, along with their characteristics, was made available to the community. This availability enables researchers and practitioners to identify benchmarks that align with the requirements of their testing or debugging techniques, thereby fostering evaluation activities within that domain, and preventing professionals from developing novel benchmarks instead of using existing ones.

From the results obtained in this mapping, we could also raise some important future work in the field. We observed a latent need to define standards and guidelines for the creation of new benchmarks in testing and debugging, reinforcing the need for them to be reusable in different contexts in order to allow the comparison between the results obtained in each case. In parallel, it is also important to ensure the neutrality of the datasets

used so that bias is not inserted, in order to favor metrics (such as accuracy) or better performance for specific methods. In addition, strategies to measure these biases should also be investigated and proposed.

We hope that the results of this research not only present consistent information and provide a panorama of available benchmarks for supporting the evaluation of software testing and debugging techniques, but also support researchers and practitioners to understand the characteristics of those benchmarks better to foster reuse, customization, and productivity by avoiding unnecessary efforts, saving both time and costs for real projects.

**Author Contributions:** Conceptualization, D.d.S.-J., V.V.G.-N., and P.d.S.L.-J.; methodology, D.M.d.-F., M.K.; investigation, all authors; writing—original draft preparation, all authors; writing—review and editing, all authors. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior-Brasil (CAPES)-Finance Code 001.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## Appendix A

RQ1 is addressed by the following form questions: “What is the study title?”, “What is the benchmark link?”, and “What is the classification of the techniques to which the benchmark was intended to be used?”; RQ2 is addressed by the following question: “Does a programming language or platform apply to the benchmark? If so, which one?”; RQ3 is addressed by the following: “As for the origin of defects (or behaviors), are they real, artificial or both?”; RQ4 is addressed the following questions: “What are the motivations described in the article? Why was the benchmark built?”; and RQ5 is addressed by the following question: “What are the objectives to the benchmark described in the article? What was the benchmark built for?“.

The following questions make up the data extraction form used.

1. What is the study title?
2. What is the publication vehicle name? Is it a conference or a journal?
3. What are the authors' affiliation?
4. What is a benchmarks' name?
5. What is a benchmarks' link?
6. Does a programming language or platform apply to a benchmark? If so, which one?
7. As for the origin of defects (or behaviors), are they real, artificial or both?
8. What are the motivations described in the study? Why was the benchmark built?
9. What are the objectives described in the study? What was the benchmark built for?
10. What is the classification of the techniques to which the benchmark was intended to be used? If fault localization, automated program repair, bug analysis, etc.
11. Other important notes.

## Appendix B

**Table A1.** All studies and their classifications—Part 1.

Study	Title	Year	Bench. Name	Topic	PL or Technology	Bugs Origins	Motivation	Purpose	Vehicle
S1	Compiling a benchmark of documented multi-threaded bugs	2004	[Eytani et al.]	Software Testing	Java	Artificial and Controlled Environment	Absence of data	External	Conference
S2	Extraction of Bug Localization Benchmarks from History	2007	iBugs	Fault Localization	Java	Production Environment	Lack of real data	External	Conference
S3	Clash of the Titans: Tools and Techniques for Hunting Bugs in Concurrent Programs	2009	[Rungta et al.]	Software Testing	Java and C#	Production Environment	Lack of real data	External	Conference
S4	(Un-)Covering Equivalent Mutants	2010	JAVALANCHE Subject Programs	Software Testing	Java	Artificial	Spontaneously-provided results data	Internal	Conference
S5	Empirical Evaluation of Bug Linking	2013	[Bissyandé et al.]	Bug Diagnosis	Language Independent	Production Environment	Lack of specialized data	External	Conference
S6	The Eclipse and Mozilla Defect Tracking Dataset: A Genuine Dataset for Mining Bug Information	2013	Eclipse and Mozilla Defect Tracking Dataset	Bug Diagnosis	Language Independent	Production Environment	Lack of real data	External	Conference
S7	42 Variability Bugs in the Linux Kernel: A Qualitative Analysis	2014	The Variability Bugs Database	Bug Diagnosis	C	Production Environment	Lack of bug understanding	Internal	Conference
S8	Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs	2014	Defects4J	Software Testing	Java	Production Environment	Lack of real data	External	Conference
S9	On the Effectiveness of Information Retrieval Based Bug Localization for C Programs	2014	[Saha et al.]	Fault Localization	C	Production Environment	Lack of specialized data	Internal	Conference
S10	Automated Bug Finding in Video Games: A Case Study for Runtime Monitoring	2014	[Varvaressos et al.]	Bug Diagnosis	Language Independent	Production Environment	Spontaneously-provided results data	Internal	Conference
S11	A Dataset of High Impact Bugs: Manually-Classified Issue Reports	2015	High Impact Bug Dataset	Bug Diagnosis	Language Independent	Production Environment	Lack of bug understanding	External	Conference
S12	The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs	2015	ManyBugs and IntroClass	Program Repair	C	Production and Controlled Environment	Lack of specialized data	External	Journal
S13	TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Data Center Distributed Systems	2016	TaxDC	Software Testing	Language Independent	Production Environment	Lack of real data	External	Conference

**Table A2.** All studies and their classifications—Part 2.

Study	Title	Year	Bench. Name	Topic	PL or Technology	Bugs Origins	Motivation	Purpose	Vehicle
S14	A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments	2017	[Yi et al.]	Program Repair	C	Controlled Environment	Lack of specialized data	Internal	Conference
S15	Automatic detection and demonstrator generation for information flow leaks in object-oriented programs	2017	KEG Experiments	Software Testing	Java	Production Environment	Lack of specialized data	Internal	Journal
S16	Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools	2017	Codeflaws	Program Repair	C	Controlled Environment	Lack of specialized data	External	Conference
S17	ELIXIR: Effective Object-Oriented Program Repair	2017	Bugs.jar	Program Repair	Java	Production Environment	Lack of specialized data	Internal	Conference
S18	How Developers Debug Software The DBGBENCH Dataset	2017	DBGBENCH	Bug Diagnosis	C	Production Environment	Lack of bug understanding	External	Conference
S19	QuixBugs: a multi-lingual program repair benchmark set based on the Quixey challenge	2017	QuixBugs	Program Repair	Python and Java	Controlled Environment	Lack of specialized data	External	Conference
S20	SECBENCH: A Database of Real Security Vulnerabilities	2017	Secbench	Software Testing	PHP, C, Ruby, Python, Java, JavaScript, ObjC, C++, ObjC++, Scala, Groovy, Swift	Production Environment	Lack of bug understanding	External	Conference
S21	Crashing Simulated Planes is Cheap: Can Simulation Detect Robotics Bugs Early?	2018	ArduBugs	Bug Diagnosis	Language Independent	Production Environment	Lack of specialized data	External	Conference
S22	Large-Scale Analysis of Framework-Specific Exceptions in Android Apps	2018	Dataset Crash Analysis	Software Testing	Android	Production Environment	Lack of real data	External	Conference
S23	Mining repair model for exception-related bug	2018	Exception-related bugs	Program Repair	Java	Production Environment	Lack of bug understanding	External	Journal
S24	Pairika-A Failure Diagnosis Benchmark for C++ Programs	2018	Pairika	Fault Localization	C++	Production Environment	Lack of specialized data	External	Conference
S25	Repairing Crashes in Android Apps	2018	Droixbench	Program Repair	Android	Production Environment	Lack of specialized data	Internal	Conference

## References

1. Pelliccione, P.; Kobetski, A.; Larsson, T.; Aramrattana, M.; Aderum, T.; Agren, S.M.; Jonsson, G.; Heldal, R.; Bergenhem, C.; Thorsén, A. Architecting cars as constituents of a system of systems. In Proceedings of the International Colloquium on Software-Intensive Systems-of-Systems at 10th European Conference on Software Architecture (SiSoSECSA'16), Copenhagen, Denmark, 28 November–2 December 2016; pp. 5:1–5:7. [[CrossRef](#)]
2. Wang, X.; Ning, Z.; Hu, X.; Ngai, E.C.H.; Wang, L.; Hu, B.; Kwok, R.Y. A city-wide real-time traffic management system: Enabling crowdsensing in social internet of vehicles. *IEEE Commun. Mag.* **2018**, *56*, 19–25. [[CrossRef](#)]
3. Horita, F.E.A.; Rhodes, D.H.; Inocêncio, T.J.; Gonzales, G.R. Building a conceptual architecture and stakeholder map of a system-of-systems for disaster monitoring and early-warning: A case study in brazil. In Proceedings of the XV Brazilian Symposium on Information Systems (SBSI'19), Aracaju, Brazil, 20–24 May 2019; pp. 6:1–6:8. [[CrossRef](#)]
4. Fraser, G.; Rojas, J.M. *Software Testing*; Cha, S., Taylor, R.N., Kang, K., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 123–192. [[CrossRef](#)]
5. Zhivich, M.; Cunningham, R.K. The real cost of software errors. *IEEE Secur. Priv.* **2009**, *7*, 87–90. [[CrossRef](#)]
6. IEEE Std 1059-1993; IEEE Guide for Software Verification and Validation Plans. IEEE: Piscataway, NJ, USA, 1994; pp.1–87. [[CrossRef](#)]
7. Lopes, V.C.; Norberto, M.; RS, D.W.; Kassab, M.; da Silva Soares, A.; Oliveira, R.; Neto, V.V.G. A systematic mapping study on software testing for systems-of-systems. In Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing, Natal, Brazil, 20–21 October 2020; pp. 88–97.
8. Kassab, M.; Laplante, P.; DeFranco, J.; Neto, V.V.G.; Destefanis, G. Exploring the profiles of software testing jobs in the United States. *IEEE Access* **2021**, *9*, 68905–68916. [[CrossRef](#)]
9. Myers, G.J.; Sandler, C. *The Art of Software Testing*; John Wiley Sons, Inc.: Hoboken, NJ, USA, 2004.
10. Hailpern, B.; Santhanam, P. Software debugging, testing, and verification. *IBM Syst. J.* **2002**, *41*, 4–12. [[CrossRef](#)]
11. Kassab, M.; DeFranco, J.F.; Laplante, P.A. Software testing: The state of the practice. *IEEE Softw.* **2017**, *34*, 46–52. [[CrossRef](#)]
12. Vessey, I. Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Trans. Syst. Man. Cybern.* **1986**, *16*, 621–637. [[CrossRef](#)]
13. Godefroid, P.; de Halleux, P.; Nori, A.V.; Rajamani, S.K.; Schulte, W.; Tillmann, N.; Levin, M.Y. Automating software testing using program analysis. *IEEE Softw.* **2008**, *25*, 30–37. [[CrossRef](#)]
14. De-Freitas, D.M.; Leitao-Junior, P.S.; Camilo-Junior, C.G.; Harrison, R. Mutation-based evolutionary fault localisation. In Proceedings of the 2018 IEEE Congress on Evolutionary Computation (CEC), Rio de Janeiro, Brazil, 8–13 July 2018; pp. 1–8.
15. Goues, C.L.; Dewey-Vogt, M.; Forrest, S.; Weimer, W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 3–13.
16. Sim, S.E.; Easterbrook, S.; Holt, R.C. Using benchmarking to advance research: A challenge to software engineering. In Proceedings of the 25th International Conference on Software Engineering (ICSE'03), Portland, OR, USA, 3–10 May 2003; pp. 74–83. Available online: <http://dl.acm.org/citation.cfm?id=776816.776826> (accessed on 9 October 2023).
17. McDaniel, G. *IBM Dictionary of Computing*; McGraw-Hill, Inc.: New York, NY, USA, 1994.
18. Wong, W.E.; Gao, R.; Li, Y.; Abreu, R.; Wotawa, F. A survey on software fault localization. *IEEE Trans. Softw. Eng.* **2016**, *42*, 707–740. [[CrossRef](#)]
19. Gazzola, L.; Micucci, D.; Mariani, L. Automatic software repair: A survey. *IEEE Trans. Softw. Eng.* **2019**, *45*, 34–67. [[CrossRef](#)]
20. Petersen, K.; Feldt, R.; Mujtaba, S.; Mattsson, M. Systematic mapping studies in software engineering. In Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08), Bari, Italy, 26–27 June 2008; pp. 68–77.
21. Ammann, P.; Offutt, J. *Introduction to Software Testing*, 2nd ed.; Cambridge University Press: Cambridge, UK; New York, NY, USA, 2017.
22. Bertolino, A. Knowledge area description of software testing. In *Guide to the Software Engineering Body of Knowledge SWEBOk (v 07)*; IEEE: Piscataway, NJ, USA, 2000.
23. Bernstein, I. *Practical Software Testing: A Process-Oriented Approach*; Springer Professional Computing; Springer: New York, NY, USA, 2003.
24. Parnin, C.; Orso, A. Are automated debugging techniques actually helping programmers? In Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11), Toronto, ON, Canada, 17–21 July 2011; pp. 199–209. [[CrossRef](#)]
25. Böhme, M.; Soremekun, E.O.; Chattopadhyay, S.; Ugherughe, E.; Zeller, A. Where is the bug and how is it fixed? An experiment with practitioners. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, 4–8 September 2017; pp. 117–128. [[CrossRef](#)]
26. Böhme, M.; Soremekun, E.O.; Chattopadhyay, S.; Ugherughe, E.J.; Zeller, A. How Developers Debug Software—The DBGBENCH Dataset. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina, 20–28 May 2017; pp. 244–246.
27. Qi, Y.; Mao, X.; Lei, Y.; Dai, Z.; Wang, C. The strength of random search on automated program repair. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), Hyderabad, India, May 31–June 7 2014; pp. 254–265. [[CrossRef](#)]

28. Hutchins, M.; Foster, H.; Goradia, T.; Ostrand, T. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, 16–21 May 1994; pp. 191–200.
29. Zakari, A.; Lee, S.P.; Alam, K.A.; Ahmad, R. Software fault localisation: A systematic mapping study. *IET Softw.* **2019**, *13*, 60–74. [[CrossRef](#)]
30. Andrews, J.H.; Briand, L.C.; Labiche, Y. Is mutation an appropriate tool for testing experiments? [software testing]. In Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), St. Louis, MO, USA, 15–21 May 2005; pp. 402–411.
31. Jia, Y.; Harman, M. Higher order mutation testing. *Inf. Softw. Technol.* **2009**, *51*, 379–1393. [[CrossRef](#)]
32. Program-Repair.org-Community-Driven Website on Automated Program Repair (automatic bug fixing). Available online: <http://program-repair.org/> (accessed on 15 January 2019).
33. Just, R.; Jalali, D.; Ernst, M.D. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014), San Jose, CA, USA, 21–25 July 2014; pp. 437–440. [[CrossRef](#)]
34. Tan, S.H.; Yi, J.; Yulis; Mechtaev, S.; Roychoudhury, A. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C'17), Buenos Aires, Argentina, 20–28 May 2017; pp. 180–182. [[CrossRef](#)]
35. Goues, C.L.; Holtschulte, N.; Smith, E.K.; Brun, Y.; Devanbu, P.; Forrest, S.; Weimer, W. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Trans. Softw. Eng.* **2015**, *41*, 1236–1256. [[CrossRef](#)]
36. Eytani, Y.; Ur, S. Compiling a benchmark of documented multi-threaded bugs. In Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS 2004 (Abstracts and CD-ROM), Santa Fe, NM, USA, 26–30 April 2004; Volume 18, pp. 3641–3648. Available online: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-12444295463&&partnerID=40&&md5=f7c2175aeb242de8e8c043cce33de7db> (accessed on 15 January 2019).
37. Kitchenham, B.; Charters, S. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*; Technical Report EBSE 2007-001; Keele University Keele, UK; Durham University: Durham, UK 2007.
38. Guessi, M.; Graciano Neto, V.V.; Bianchi, T.; Felizardo, K.; Oquendo, F.; Nakagawa, E.Y. A systematic literature review on the description of software architectures for systems of systems. In Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC'15). Association for Computing Machinery, New York, NY, USA, 13–17 April 2015; pp. 1433–1440. [[CrossRef](#)]
39. Dyba, T.; Kitchenham, B.; Jorgensen, M. Evidence-based software engineering for practitioners. *IEEE Softw.* **2005**, *22*, 58–65. [[CrossRef](#)]
40. Petersen, K.; Vakkalanka, S.; Kuzniarz, L. Guidelines for conducting systematic mapping studies in software engineering: An update. *Inf. Softw. Technol.* **2015**, *64*, 1–18. [[CrossRef](#)]
41. Lin, D.; Koppel, J.; Chen, A.; Solar-Lezama, A. QuixBugs: A Multi-lingual Program Repair Benchmark Set Based on the Quixey Challenge. In Proceedings of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH Companion 2017, Vancouver, BC, Canada, 22–27 October 2017; pp. 55–56. [[CrossRef](#)]
42. Dallmeier, V.; Zimmermann, T. Extraction of Bug Localization Benchmarks from History. In Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE'07), Atlanta, GA, USA, 5–9 November 2007; pp. 433–436. [[CrossRef](#)]
43. Rungta, N.; Mercer, E.G. Clash of the Titans: Tools and Techniques for Hunting Bugs in Concurrent Programs. In Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD'09), Chicago, IL, USA, 19–20 July 2009; pp. 9:1–9:10. [[CrossRef](#)]
44. Schuler, D.; Zeller, A. (Un-)Covering Equivalent Mutants. In Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST'10), Paris, France, 6–10 April 2010; pp. 45–54. [[CrossRef](#)]
45. Bissyande, T.F.; Thung, F.; Wang, S.; Lo, D.; Jiang, L.; Reveillere, L. Empirical Evaluation of Bug Linking. In Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR'13), Genova, Italy, 5–8 March 2013; pp. 89–98. [[CrossRef](#)]
46. Lamkanfi, A.; Pérez, J.; Demeyer, S. The Eclipse and Mozilla Defect Tracking Dataset: A Genuine Dataset for Mining Bug Information. In Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13), San Francisco, CA, USA, 18–19 May 2013; pp. 203–206. Available online: <http://dl.acm.org/citation.cfm?id=2487085.2487125> (accessed on 9 October 2023).
47. Abal, I.; Brabrand, C.; Wasowski, A. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14), Västerås, Sweden, 15–19 September 2014; pp. 421–432. [[CrossRef](#)]
48. Saha, R.K.; Lawall, J.; Khurshid, S.; Perry, D.E. On the Effectiveness of Information Retrieval Based Bug Localization for C Programs. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014; pp. 161–170.
49. Varvareos, S.; Lavoie, K.; Massé, A.B.; Gaboury, S.; Hallé, S. Automated bug finding in video games: A case study for runtime monitoring. In Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, Cleveland, OH, USA, 31 March–4 April 2014; pp. 143–152.

50. Ohira, M.; Kashiwa, Y.; Yamatani, Y.; Yoshiyuki, H.; Maeda, Y.; Limsettho, N.; Fujino, K.; Hata, H.; Ihara, A.; Matsumoto, K. A Dataset of High Impact Bugs: Manually-Classified Issue Reports. In Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, Italy, 16–17 May 2015; pp. 518–521.
51. Leesatapornwongsa, T.; Lukman, J.F.; Lu, S.; Gunawi, H.S. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems-ASPLOS, Atlanta, GA, USA, 2–6 April 2016; pp. 517–530. [CrossRef]
52. Yi, J.; Ahmed, U.Z.; Karkare, A.; Tan, S.H.; Roychoudhury, A. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017), Paderborn, Germany, 4–8 September 2017; pp. 740–751. [CrossRef]
53. Do, Q.H.; Bubel, R.; Hähnle, R. Automatic detection and demonstrator generation for information flow leaks in object-oriented programs. *Comput. Secur.* **2017**, *67*, 335–349. [CrossRef]
54. Saha, R.K.; Lyu, Y.; Yoshida, H.; Prasad, M.R. ELIXIR: Effective Object Oriented Program Repair. In Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), Urbana, IL, USA, 30 October–3 November 2017; pp. 648–659. Available online: <http://dl.acm.org/citation.cfm?id=3155562.3155643> (accessed on 9 Ocotober 2023).
55. Reis, S.; Abreu, R. Sebbench: A Database of Real Security Vulnerabilities. In *CEUR Workshop Proceedings*; Volume 1977; pp. 69–85. Available online: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85035242201&partnerID=40&md5=2677af851481f16b0d8c44668e1d16b> (accessed on 9 Ocotober 2023).
56. Timperley, C.S.; Afzal, A.; Katz, D.S.; Hernandez, J.M.; Goues, C.L. Crashing Simulated Planes is Cheap: Can Simulation Detect Robotics Bugs Early? In Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), Västerås, Sweden, 9–13 April 2018; pp. 331–342.
57. Fan, L.; Su, T.; Chen, S.; Meng, G.; Liu, Y.; Xu, L.; Pu, G.; Su, Z. Large-scale Analysis of Framework-specific Exceptions in Android Apps. In Proceedings of the 40th International Conference on Software Engineering (ICSE'18), Gothenburg, Sweden, 27 May–3 June 2018; pp. 408–419. [CrossRef]
58. Zhong, H.; Mei, H. Mining repair model for exception-related bug. *J. Syst. Softw.* **2018**, *141*, 16–31. [CrossRef]
59. Rahman, M.R.; Golagha, M.; Pretschner, A. Pairika: A Failure Diagnosis Benchmark for C++ Programs. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeедings (ICSE'18), Gothenburg, Sweden, 27 May 2018–3 June 2018; pp. 204–205. [CrossRef]
60. Tan, S.H.; Dong, Z.; Gao, X.; Roychoudhury, A. Repairing Crashes in Android Apps. In Proceedings of the 40th International Conference on Software Engineering (ICSE'18), Gothenburg, Sweden, 27 May–3 June 2018; pp. 187–198. [CrossRef]
61. Dybå, T.; Dingsøyr, T. Empirical studies of agile software development: A systematic review. *Inf. Softw. Technol.* **2008**, *50*, 833–859. [CrossRef]
62. Ali, M.S.; Ali.Babar, M.; Chen, L.; Stol, K.J. A systematic review of comparative evidence of aspect-oriented programming. *Inf. Softw. Technol.* **2010**, *52*, 871–887. [CrossRef]
63. Yusifoglu, V.G.; Amannejad, Y.; Can, A.B. Software test-code engineering: A systematic mapping. *Inf. Softw. Technol.* **2015**, *58*, 123–147. [CrossRef]
64. Martinez, M.; Durieux, T.; Sommerard, R.; Xuan, J.; Monperrus, M. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empir. Softw. Eng.* **2017**, *22*, 1936–1964. [CrossRef]
65. Pearson, S.; Campos, J.; Just, R.; Fraser, G.; Abreu, R.; Ernst, M.D.; Pang, D.; Keller, B. Evaluating and improving fault localization. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; pp. 609–620.
66. Snelting, G. Concept analysis—A new framework for program understanding. In Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98), Montreal, QC, Canada, 16 June 1998; pp. 1–10. [CrossRef]
67. Dyba, T.; Dingsoyr, T.; Hanssen, G.K. Applying systematic reviews to diverse study types: An experience report. In Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), Madrid, Spain, 20–21 September 2007; pp. 225–234.

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.