The Journal of Systems & Software 192 (2022) 111423

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A systematic literature review on benchmarks for evaluating debugging approaches☆

Thomas Hirsch, Birgit Hofer*

Institute of Software Technology, Graz University of Technology, Austria

ARTICLE INFO

Article history:
Received 16 December 2021
Received in revised form 22 June 2022
Accepted 23 June 2022
Available online 30 June 2022

Keywords:
Debugging
Benchmark
Fault localization
Automatic repair

ABSTRACT

Bug benchmarks are used in development and evaluation of debugging approaches, e.g. fault localization and automated repair. Quantitative performance comparison of different debugging approaches is only possible when they have been evaluated on the same dataset or benchmark. However, benchmarks are often specialized towards usage for certain debugging approaches in their contained data, metrics, and artifacts. Such benchmarks cannot be easily used on debugging approaches outside their scope as such approach may rely on specific data such as bug reports or code metrics that are not included in the dataset. Furthermore, benchmarks vary in their size w.r.t. the number of subject programs and the size of the individual subject programs. For these reasons, we have performed a systematic literature review where we have identified 73 benchmarks that can be used to evaluate debugging approaches. We compare the different benchmarks w.r.t. their size and the provided information such as bug reports, contained test cases, and other code metrics. This comparison is intended to help researchers to quickly identify all suitable benchmarks for evaluating their specific debugging approaches. Furthermore, we discuss reoccurring issues and challenges in selection, acquisition, and usage of such bug benchmarks, i.e., data availability, data quality, duplicated content, data formats, reproducibility, and extensibility.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

Fixing bugs is time-consuming and therefore a cost driver of software projects. It is estimated that programmers spend 30 to 90% of their time on locating and fixing bugs (Ang et al., 2017). In a recent questionnaire (Hirsch and Hofer, 2021), programmers indicated to require on average 3 h to reproduce a failure, 4.4 h to locate the faulty code and 2.1 h to fix it, making in total 9.5 h for resolving a bug. These numbers represent typical bugs, not the most difficult bugs, which exceed these debugging times many times over. To support programmers in the debugging process, researchers have developed numerous fault localization (Wong et al., 2016; Zakari et al., 2020) and repair (Gazzola et al., 2019) approaches.

The efficiency and effectiveness of automatic debugging approaches is measured using bug benchmarks. Further, such bug benchmarks are vital to the development process of debugging tools as test subjects. Therefore, bug benchmarks are important resources for practitioners and researchers alike.

A bug benchmark is a curated collection of bugs and their corresponding artifacts and data, created with the intent to evaluate debugging tools upon and/or aid development of such tools. Such benchmarks can take many forms regarding their structure, size, and contained artifacts. The most common components are artifacts exhibiting the incorrect behavior in the form of buildable source code, accompanied by fixed versions, patches (i.e. the diffs to be applied in order to fix the bug), or fault location information, and test cases or inputs that trigger the fault. In addition, a bug instance can be accompanied by its textual bug report, precalculated code metrics, and other information. Bug benchmarks are not necessarily self-contained, e.g., a faulty version may be identified by a commit hash on a public repository, or bug reports represented by links to their corresponding entries on public bug trackers. The subject programs in benchmarks can be real world software projects (e.g., open-source projects), created specially for the benchmark (*fabricated programs*), or other artifacts (e.g., student submissions and coding competitions). The contained bugs are artificially created (i.e., seeded) or real world bugs.

☆ Editor: Matthias Galster.

* Correspondence to: Institute of Software Technology, Graz University of Technology, Inffeldgasse 16b/EG, 8010 Graz, Austria.
E-mail addresses: thirsch@ist.tugraz.at (T. Hirsch), bhofer@ist.tugraz.at (B. Hofer).

https://doi.org/10.1016/j.jss.2022.111423

Numerous benchmarks are available for evaluating debugging approaches. These benchmarks differ in several aspects, e.g., the size (number of different programs and bugs) and scope (language, bug types, availability of test cases and meta data such as bug reports). While certain benchmarks are well-suited for specific types of debugging approaches they often cannot be used as is on other types, as such debugging approaches may rely on other types of information not contained in a benchmark. For example, model-based debugging (Reiter, 1987) requires only one failing test case, but can only be used on small programs, while spectrum-based fault localization (SBFL) (Abreu et al., 2009) requires a test suite with both passing and failing test cases, but can also handle large programs. The plethora of available benchmarks and their wide variety makes it challenging to select the best suited benchmark(s) for a specific task.

Despite the long history of benchmarking debugging approaches, and today's abundance of such benchmarks, there are only little resources supporting researchers in the benchmark selection process. Majd et al. (2019) compared 20 benchmarks w.r.t. fault nature, fault type, fault number, fault categorization, fault isolation, program type, number of projects, program size, source type, and language. In this paper, we provide a comprehensive overview of 73 benchmarks to help researchers to select the best suited benchmark(s) for evaluating a debugging approach. Our work differs from Majd et al. (2019) in the number of investigated benchmarks and the level of detail: We provide quantitative measures over qualitative labels wherever possible (e.g. the number of contained faults). We introduce a number of new fields such as the intended purpose of the benchmark, the availability of patches, virtual environments, and list the metrics contained in the respective datasets. Further, we evaluate the benchmarks regarding the FAIR Guiding principles.

To identify existing debugging benchmarks, we performed a systematic literature review (SLR). We examined each benchmark regarding a pre-defined set of quality criteria. We reviewed a total of 206 papers to create a comprehensive list of debugging benchmarks. We rejected 134 papers that did not fulfill our selection criteria, for example, 14 papers of which the benchmark data has become unavailable. We identified 73 different benchmarks that can be used for evaluating debugging approaches. The selected benchmarks cover 44 different programming languages and the size of the programs in these benchmarks ranges from tiny (containing less than 100 LOC) to large (with more than 1000 KLOC). The largest program we encountered in a benchmark contains 14 million LOC. We provide a json file in our online Appendix (see Section 8) containing the meta-data of the studied benchmarks.

The remainder of this paper is structured as follows: Section 2 discusses the design of this SLR and introduces the research questions. Section 3 describes the selected benchmarks. Section 4 answers the research questions and Section 5 discusses the problems that we identified in some of the benchmarks. Section 6 describes the threats to validity that this paper faces and how we address them. Section 7 concludes the paper.

We provide the study protocol, the extracted data and the used evaluation script as open science material on Zenodo (Hirsch and Hofer, 2022):
https://doi.org/10.5281/zenodo.6670198

2. Study design

2.1. Rationale

This paper aims to provide an overview of benchmarks that can be used to evaluate different debugging approaches, including, but not limited to, fault localization and automatic repair approaches. The resulting catalog of benchmarks should help researchers to select the best suited benchmark.

2.2. Scope

The survey focuses on benchmarks suitable for evaluating debugging techniques that operate on, or require the source code, e.g., fault localization and repair techniques. We exclude benchmarks that exclusively consist of calculated code metrics where the corresponding source code is not provided with the benchmark, or obtainable through information contained in the benchmark. An extensive collection of such datasets, aiming at the creation of predictive software models, is already available in the Promise Software Engineering Repository (Sayyad Shirabad and Menzies, 2005).

2.3. Research questions

We address the following research questions:

- RQ1: What is the content and size of the individual benchmarks?
- RQ2: What is the intended purpose of the benchmarks?
- RQ3: What FAIR Guiding Principles are realized?

2.4. Paper selection process

Research method. We perform a SLR (Kitchenham and Charters, 2007) to identify relevant benchmarks. We select the benchmarks based on a set of predefined inclusion criteria. After the selection of the benchmarks, we extract the relevant data from each benchmark.

Search protocol. Fig. 1 shows the paper selection process. We use Scopus[1] to search for relevant publications. Scopus is an abstract and citation database that hosts over 1.8 billion records from over 5000 publishers, including Elsevier, Springer, and IEEE. We search for papers published in 2000 or later that contain at least one of the terms *benchmark*, *dataset*, or *bug database* and one of the terms *debugging*, *bug*, *fault localization*, or *repair* in their title, abstract, or keywords. We decided to use the terms *debugging*, *bug*, *fault localization* and *repair* instead of the more generic term *fault* in our search string to avoid a high number of false positives. While the term *bug* is strongly associated with software, the term *fault* is in widespread use in many engineering disciplines. The term *fault localization* on the other hand is again strongly associated with software bugs. We limit the subject area to *Computer Science* and the document type to *Conference paper*, *Article*, and *Book Chapter*. Further, we exclude papers from computer vision and human–computer-interaction outlets.

Researcher 2 performs a first filtering step on the papers obtained from the Scopus search in which irrelevant papers are removed. We consider the following types of papers as irrelevant: papers dealing with faults in systems other than software, papers that do not belong to the field of computer science, and papers that deal with bugs in software but do not contain bug benchmarks (Coarse Selection). The filtering decision was based on the title and the abstract of the paper. Whenever a paper mentions in the abstract that a benchmark was used to evaluate an approach, the body of the paper is scanned to identify the benchmark. The paper describing the benchmark is added to the list of papers to be inspected (Snowballing I). The paper that only uses the benchmark is removed from the list of potential benchmark papers. Whenever a potential benchmark paper is identified, the introduction and the related work sections are scanned to find other relevant benchmarks (Snowballing II). This step is recursively applied to all newly identified papers. Whenever several papers describe the same benchmark, a major paper is selected.

[1] https://www.scopus.com/.

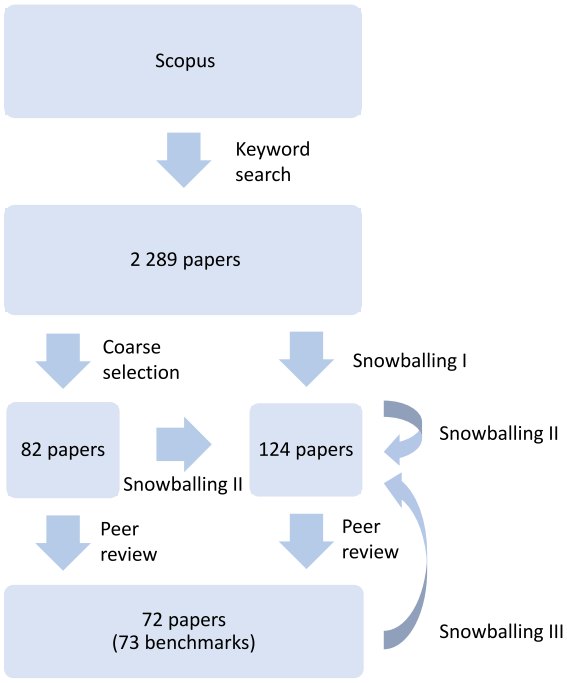

Fig. 1. Paper selection process.

Table 1
Inclusion criteria.

	Criteria
I1	The provided material has to include bug benchmark data and must to be publicly available or upon request.
I2	The provided material has to be complete, i.e., it has to contain data of all fault instances.
I3	The benchmark must not be deprecated.
I4	The benchmark must either contain source code or link to the issue tickets or source code, e.g., in the form of bug fixing commits.
I5	The benchmark must not be an exact copy of an existing benchmark.
I6	The location of the bug must be indicated, either by stating which source code lines are faulty, by providing a patch, or by naming the buggy file (when the code base consists of more than one file). If the patch can be generated in an automated fashion based on a link to a commit or resolved issue ticket that again links to a commit, this is also acceptable.
I7	The benchmark must contain at least 20 bugs.
I8	It must not contain issues other than bugs (e.g., feature requests) if they cannot be distinguished using the provided information.

Usually, journal papers serve as major papers. In case of several conference or workshop papers, the newest version is selected as major paper. Whenever the website for the benchmark does not exist or the download link does not work, we inform the authors via email about the problem. We exclude all papers from the review set where we do not obtain the benchmark within two months after the request.

In the next step, both researchers read the papers and inspect the available online resources. Based on the available information, we decide together on the inclusion of papers based on the criteria presented in Table 1 (Peer review). In a final step, researcher 2 performs forward snowballing on the accepted papers (Snowballing III). We use Google scholar to find the papers that are citing the accepted papers. Papers found in this process are again peer-reviewed as well as forward and backward snowballed.

2.5. Paper analysis

Data extraction. Researcher 2 performs the initial data collection followed by researcher 1 verifying the collected data. For each benchmark, we read the paper and investigate the available online material to extract the following data:

- Name of the benchmark
- Major publication (and Google citation count as of 2021-11-12)
- Data outlets, including URL, Host (e.g., GitHub, Zenodo), DOI (if available), Accessibility (public or upon request), and Versioning (e.g., version numbers, git)
- Inspected version
- License (name and URL, if available)
- Documentation quality (well-documented, little documentation, no documentation)
- Intended purpose (e.g., repair)
- Domain (e.g., mobile applications)
- Source type (e.g., student code)
- Fault type (artificial or real)
- Details on the bugs, including the programming language, number and size of subject programs, number of faults
- File formats of meta-data
- Version history (full history, old versions available, no history)

Detailed approach to our RQs. To answer RQ1, we investigate the content type provided by each benchmark (e.g., programming languages, availability of test cases and/or patches, issue tickets, calculated code metrics) and the size of the benchmarks (i.e., number of programs and faults, LOC). Furthermore, we provide an overview of the benchmark subjects w.r.t the source type (e.g., fabricated for the purpose of benchmarking, student programs, open source projects) and the fault type (i.e., artificial or real). This information helps researchers to locate benchmarks containing the data required by their approach, e.g., a test suite for SBFL. Information about program size helps researchers to choose a benchmark of appropriate size, but also to determine the size boundaries of their approaches: Certain debugging approaches can be used on programs of any size while others can only be applied on smaller programs.

We answer RQ2 by investigating the intended purposes of the benchmarks as indicated in the papers or in the documentations. The intended purposes help researchers to identify benchmarks that have been used by other researchers for evaluating similar approaches.

We answer RQ3 by applying a catalog of evaluation criteria to score how well FAIR Guiding Principles (Wilkinson et al., 2016) are materialized for each benchmark. Researchers benefit from benchmarks published in accordance to the FAIR Guiding Principles (i.e., Findable, Accessible, Interoperable, and Reusable). Since benchmark contents are subject to change, we introduce reproducibility as an additional principle. Reproduction experiments and comparison of novel approaches to previously published approaches requires the following: The exact state of the benchmark on which the original experiment was performed must be identifiable, and the dataset in this exact state must be obtainable. This reproducibility principle is covered by version control and version history scores. Version control, i.e. in the form of explicit version numbers or commit hashes, allow researchers to indicate the exact version used in their experiments. The version history allows other researchers to obtain the exact version. Perfect reproducibility is given if the exact state of the dataset can be

Table 2
FAIR evaluation criteria.

Criteria	Value	Rating
Findable		
Storage location	Zenodo, figshare	★★★★★
	BitBucket, GitHub, Google Code	★★★★☆
	NIST, GitLab	★★★☆☆
	Google drive, Google sites	★★☆☆☆
	sourceforge	★★☆☆☆
	dedicated or personal website	★★☆☆☆
	University, company, foundation	★★☆☆☆
	Web archive, email	★☆☆☆☆
Actual data stored	true	★★★★★
	false	★☆☆☆☆
Accessible		
Access mode	public	★★★★★
	upon identification	★★★★★
	registration and reply	★☆☆☆☆
	upon email request	★☆☆☆☆
Interoperable		
File format	json, xml, yaml	★★★★★
	csv	★★★★☆
	txt, md, html, arff, SQLite	★★★☆☆
	xlsx, sql, web-based, pickle	★★☆☆☆
	pdf, doc, png, virtual box image	★☆☆☆☆
	code comments	★☆☆☆☆
Reusable		
Documentation	Well-documented	★★★★★
	Little documentation	★★☆☆☆
	No documentation	★☆☆☆☆
License	Permissive, copyleft	★★★★★
	Restr. Copyleft	★★★★☆
	Restricted	★★☆☆☆
	None	★☆☆☆☆
Reproducible		
Version control	Unique version identifiers	★★★★★
	Commit id	★★★★☆
	Change date	★★★☆☆
	None	★☆☆☆☆
Version history	Full history	★★★★★
	Old versions available	★★★★☆
	No history	★☆☆☆☆

uniquely identified and obtained in order to exactly reproduce an experiment. We provide a one to five star rating for each benchmark for each criteria. Table 2 summarizes the evaluation criteria. Details of the evaluation criteria for the principles are discussed below.

Findable. A benchmark is findable if is has a globally unique, persistent identifier and the data is stored in a long-lasting repository (Wilkinson et al., 2016). Zenodo and Figshare are long-lasting repositories and all data stored on them gets assigned a DOI. Therefore, we rate all benchmarks that are hosted on Zenodo or Figshare with five stars. BitBucket, GitHub, and Google Code have a high reputation and are provided by respected companies. Although, we cannot rule out that these companies change their policies for hosting repositories, we are confident that the benchmarks storage is secured in the medium term. Therefore, we rate benchmarks hosted on BitBucket, GitHub (including GitHub IO), or Google Code with four stars. We rate benchmarks hosted by the NIST or on GitLab with three stars. While NIST is a well-reputed institution, there is no guarantee that the benchmarks will be stored there permanently. Although GitLab provides high quality of service, this service is only free of charge for MIT licensed content, and any non-MIT licensed content may be lost if the owners discontinue their subscriptions. Benchmarks hosted on Google drive or Google sites are rated with two stars as there are no guarantees on how long the data will be available there. In particular for Google drive users might delete benchmarks when they need the storage space for other data. Benchmarks hosted on sourceforge are rated with two stars since there is no guarantee on how long the data will be available there. Benchmarks hosted on dedicated websites or researchers' personal websites are rated with two stars since it is uncertain how long the maintainers will fund the service. Benchmarks that are hosted on company, university, or foundation servers are rated with two stars because project websites and employees' websites are often deleted when projects are finished or the employees leave the company/university. We rate benchmarks that can only be obtained via email or with the wayback machine with one star.

Another important factor is the type of data that is actually stored: Some benchmarks provide the complete source code while others only link to bug reports and commits of open source repositories. The latter is problematic since the linked open source repositories may be deleted, moved, migrated, or its history modified. Therefore, self-contained benchmarks are rated with five stars, and their counterparts rated with one star. Benchmarks with mixed content, i.e., some bugs are self-contained while others are only linked (e.g., EDD Caballero et al., 2018), are also rated with one star.

Accessible. A benchmark is accessible if the data can be retrieved using a standardized communications protocol—either open or upon authentication. We rate public repositories and repositories allowing instant access upon registration with five stars. Repositories that require a registration process involving a maintainer manually granting permissions, as well as repositories where access or data is provided only upon an email request are rated with one star. These access modes are often problematic due to invalid email addresses, infrequently checked inboxes, or researchers having left the respective research institution.

Interoperable. A benchmark is interoperable if open or common formats are used (Wilkinson et al., 2016). The used file format should be human readable, machine readable, durable, non-proprietary and part of default installations. We list all discovered file formats that were used to provide the bug information in the benchmarks. Whenever several file formats are used, we differentiate if both file formats are required to access all the provided information. For example, `["html", ["csv", "txt"]]` means that the complete information is provided in an html file or alternatively by using the csv file and the txt file together.

We rate json, xml, and yaml files with five stars because these file formats are human readable and can be easily automatically processed. They are durable, non-proprietary, and can be accessed using any text editor or browser. csv files have nearly the same properties as the above mentioned file formats, except that they have certain limitations w.r.t portability (e.g., delimiters, number formatting, escaping of special characters). We refer to all file types with such a structure as csv regardless of the utilized delimiters (e.g., tab separated, semicolon separated), as long as spreadsheet processors and parsing libraries can directly process them, and rate them with four stars. We rate txt, md, and html files with three stars because these file formats provide the data in an unstructured way requiring some additional effort to parse them. SQLite files are rated with three stars because they are not human readable without the use of additional tools that are not commonly found on most default installations. arff file are also rated with three stars because either the Weka framework or other libraries have to be installed to process them. xlsx files are rated with two stars; while acceptable for data intended for manual inspection and manipulation using spreadsheets, we consider xlsx inferior to csv, xml, and json for automated processing. Even though xlsx files can be converted to other formats, additional steps are required and portability cannot be guaranteed. We rate sql databases with two stars because they require significant overhead to access the data. We consider this overhead unjustified in most cases. We rate pickle files with two stars

because they are not human readable. We rate pdfs and doc files with one star since they require significant effort to automate processing. We rate pngs with one star because they cannot be automatically processed. We rate code comments with one star because of their unstructured nature and the information may be spread over multiple files and locations. While virtual machine images constitute a legitimate way to enable and streamline reproducability, their use as storage and transport medium for a dataset is problematic. This is due to their size and additional layers wrapping the data, which in turn hamper exploration, inspection, or manipulation of the data. We therefore rate them with one star.

Reusable. A benchmark is considered reusable when the data is well documented and licensed. The documentation can be provided as files within the dataset or web-based. We rate benchmarks that are well documented with five stars, benchmarks that have little documentation with two stars, and benchmarks without documentation with one star.

We rate benchmarks that come with a permissive or copyleft license with five stars. We rate licenses that allow the publication of modified versions only upon the acceptance of a peer-reviewed paper with four stars and licenses that do not allow one to freely publish a modified version of the benchmark with two stars. Benchmarks without any license are rated with one star.

Reproducible. A benchmark is reproducible when it provides some form of version control and the full version history is available. Regarding version control, we rate benchmarks with unique version identifiers with five stars and benchmarks using a version control system such as Git (without explicit version numbers) with four stars as commit ids can serve as version identifiers. Further, we rate benchmarks indicating the change date with three stars. Benchmarks without any form of version control are rated with one star. To allow reproduction, previous versions have to be available. Regarding version history, we rate benchmarks that provide their full history with five stars, benchmarks that provide old versions with four stars and benchmarks without any version history with one star.

3. Selected papers and benchmarks

The Scopus search resulted in 2289 papers. After the coarse selection of the initial search results, 82 primary and 124 snowballing papers were inspected by both authors to decide on the inclusion in this survey. During this inspection, we have encountered several problems. Four papers did not provide a link to the benchmark. For 14 papers, the website indicated in the paper did not exist or the download link was broken. Six of these benchmarks had a university URL, one had a company URL, two had dedicated websites, and two were originally hosted on GitHub; one benchmark offered the data to be downloaded via SVN, but the SVN repository did not exist anymore and another benchmark provided the data in a virtual box image that was removed from Google drive. One benchmark required accepting the license agreement via email, however, the email address did not exist anymore. We emailed the authors of the corresponding papers to inform them about the problem and asked them for the benchmarks. 12 research teams answered to our emails and provided a new link or fixed the existing website. From the remainder, we have not received any reply. Therefore, these benchmarks were excluded from this survey. From here on, we present the data of those benchmarks that are included in the survey. In total, we selected 72 papers and 73 benchmarks, as one paper contains two benchmarks.

Appendix A lists all investigated benchmarks and the links to the repositories as well as the inspected version/commit or the date of inspection. Here, we briefly describe the benchmarks sorted after programming language.

Multi-lingual. *NIST SAMATE Juliet Test Suite* (Black, 2011) is a collection of artificial bugs written in Java, C/C++, and C#. *Mandelbugs* (Cotroneo et al., 2013) is a collection of 963 bugs found in 4 open source projects written in C, C++, and Java. *IntroClass* (Le Goues et al., 2015) contains bugs made by students in a C programming course. *IntroClass Java* (Durieux and Monperrus, 2016) is the Java version of this benchmark. The *Software-artifact Infrastructure Repository (SIR)* (Do et al., 2005) is a collection of programs written in C, C++, Java, and C#. The *10 Years Bug-Fix Dataset* (Vieira et al., 2019) contains bug fixing data of 55 open source projects written in Java, Ruby, Scala, Python, C++, and Perl.

NFBugs (Radu and Nadi, 2019) is a collection of non-functional bugs in Java and Python programs. *QuixBugs* (Lin et al., 2017) is a collection of bugs in Java and Python programs fabricated to be fixed by programmers in a competition. *Defexts* (Benton et al., 2019) contains bugs of programs written in Kotlin, Groovy, and Scala.

BugSwarm (Tomassi et al., 2019) is a toolset that detects pass-fail build pairs of programs written in both Java and Python by searching the projects' CI (continuous integration) log records and creates a container with the buggy code version, failing test cases, and fixes for each bug. There exists a benchmark with more than 3000 bug containers, but only 112 of them are suited for evaluating fault localization and automatic repair approaches (Durieux and Abreu, 2019).

SECBench (Reis and Abreu, 2017) is a collection of 676 bugs found in 114 programs from various programming languages. *DataRaceBench* (Liao et al., 2017) is a collection of fabricated OpenMP programs to evaluate data race detection tools. *Code4Bench* (Majd et al., 2019) and *Codeflaws* (Tan et al., 2017) are collections of bugs from the submissions of programming contests. *Denchmark* (Kim et al., 2021) contains 4577 bug reports from 193 deep learning software projects.

TrainTicket (Zhou et al., 2021) is a fabricated microservice system that is written in 4 programming languages (Java, Node.js, Python, and Go) and contains 41 microservices. The Replication Package for fault analysis and debugging contains 22 faulty versions of this system.

Java. *Defects4J* (Just et al., 2014) consists of 835 bugs from 17 open source projects. The dataset of *6 open source Java projects* (Ye et al., 2014) contains more than 22,000 bugs that were found by a keyword search in commit messages. The *Technical Debt Dataset* (Lenarduzzi et al., 2019) contains issue tickets, commits, and metrics of 33 Apache projects. *ManySStuBs4J* (Karampatsis and Sutton, 2020) is an extensive collection of "simple stupid bugs" (SStuBs), i.e., bugs with small and trivial fixes. *High impact bugs* (Ohira et al., 2015) contains 4000 high impact bugs from four Apache projects.

Bugs.jar (Saha et al., 2018) contains 1158 bugs collected from 8 open source Java projects, *Bears* (Madeiral et al., 2019) contains 251 bugs from 72 open source Java projects. Both put an emphasis on automatability in terms of tests and build environments of the selected projects, and have been used to evaluate automatic repair approaches. The *History Driven Program Repair* (Le et al., 2016) dataset contains more than 3000 single line fixes. The *PAR dataset* (Kim et al., 2013) contains 119 bugs from six open-source Java projects. *Bench4BL* (Lee et al., 2018) contains bug reports and fixes from 51 open-source projects and is used to evaluate fault localization approaches. *BugLocator* (Zhou et al., 2012) is an information retrieval based bug localization approach. Parts of the utilized evaluation dataset, i.e., 118 bugs from two open source java applications, are made publicly available. *Cloud Bugs* (Gunawi et al., 2014) is a collection of more than 3000 issues of six open-source cloud systems. *BigFix* (Li et al., 2020) is a collection of the commits and bug reports of eight open-source projects.

JaConTeBe (Lin et al., 2015) is part of the Software-artifact Infrastructure Respository (SIR) (Do et al., 2005) and contains 47 concurrency bugs. *IBM ConTest* (Eytani et al., 2008) is a concurrency benchmark and consists mainly of small programs written by students. *JBench* (Gao et al., 2018) contains nearly 1000 data races from 42 academic example programs and six open-source programs. *JCrashPack* (Soltani et al., 2020) is a collection 200 crashing bugs that were found in 7 open-source projects. *Exception-related bugs* (Zhong and Mei, 2018) contains bugs from six open source projects. *Leak study* (Ghanavati et al., 2020) is a collection of 491 leaks found in 15 Java open-source projects. *CryptoAPI-Bench* (Afrose et al., 2019) contains 171 crypto API misuses that can be used to evaluate security tools. *MUBench* (Amann et al., 2016) is a collection of 89 API misuses found in 33 projects.[2] *APIRepBench* (Kechagia et al., 2021) contains 101 API misuses found in three benchmarks, nameley MUBench, Bugs.jar, and Bears.

IFSpec (Hamann et al., 2018) focuses on information-flow vulnerabilities in Java and Android applications. *Securibench mirco* (Livshits, 2014) is a set of small programs with security vulnerabilities. The *OWASP* benchmark (OWASP Foundation, 2022) is a collection of vulnerabilities in Java Servlets. *VDBENCHWS-PD* (Antunes and Vieira, 2015) is a benchmark to evaluate vulnerability detection tools.

iBugs (Dallmeier and Zimmermann, 2007) is a technique that allows one to build a bug benchmark by searching for keywords such as “bug” or “fixed” followed by an id in the commit messages of software repositories. Three datasets that were created using this tool are made available: Bugs from AspectJ, Rhino, and JodaTime. The *Variability Fault Localization* benchmark (Ngo et al., 2021) is a collection of 570 seeded bugs in 6 small programs.

C/C++. *CoREBench* (Böhme and Roychoudhury, 2014) consists of 70 regression bugs from four open source C projects. The *IR dataset* (Saha et al., 2014) contains more than 7500 bug reports from five open-source C projects. *Lava-1* and *Lava-M* (Dolan-Gavitt et al., 2016) contain bugs from open-source C projects. *ManyBugs* (Le Goues et al., 2015) contains 185 bugs found in 9 open source C programs. *TempList* (Vorobyov et al., 2017) contains a set of C files with temporal memory errors. *DbgBench* (Böhme et al., 2017) contains 27 regression bugs and corresponding patches created in a user study where twelve professional software developers were asked to locate and fix those bugs. DbgBench also contains information on the correctness of the included patches.

APIMU4C (Gu et al., 2019) is a collection of API misuses in C programs. It contains small examples as well as three open source projects. *Bug-Injector* (Kashyap et al., 2019) seeds bugs into real-world programs based on bug templates. The provided dataset contains artificial faults injected into two C/C++ programs. *ClabureDB* (Slaby et al., 2013) contains three bug collections with C/C++ bugs. *Pairika* (Rahman et al., 2018) contains 40 reproducible bugs from an open source C++ project. *Toyota ITC Benchmark* (Shiraishi et al., 2016) is a collection of fabricated C/C++ programs. Each bug is encapsulated into a single function with approximately five LOC. *Magma* (Hazimeh et al., 2020) is a benchmark for evaluating fuzzers, containing 118 real world bugs from 7 open source C/C++ projects. *PerfBugReplication* (Han et al., 2019) is a VirtualBoxImage that contains 17 reproduced performance bugs from two open-source projects. The *Variability Bugs Database* (Abal et al., 2018) consists of 98 variability bugs found in four open-source C/C++ projects.

The *BroSCH dataset* (Kiss and Hodovan, 2019) contains over 13 000 security-related commits from two open-source web browsers. The Systematic concurrency testing benchmark (*SCT-Bench*) (Thomson et al., 2016) is a collection bugs from different benchmarks.

Python. *BugsInPy* (Widyasari et al., 2020) contains bugs from Python programs and is intended to be used to evaluate testing and debugging techniques. The *Refactory dataset* (Hu et al., 2019) is a collection of buggy student programs written in Python.

Mobile applications. *AppLeak* (Riganelli et al., 2019a) contains 40 bugs found in 15 Android applications that are caused by resource leaks. *DroidLeaks* (Liu et al., 2019) consists of 292 resource leaks found in 32 Android apps. *BenchERoid* (Salehnamadi et al., 2020) is a collection of 34 fabricated androids apps, each containing zero to three event races. *Ghera* (Mitra and Ranganath, 2017) contains 60 vulnerabilities in Android apps. The *DataLoss-Repository* (Riganelli et al., 2019b) contains 110 faults in Android apps that cause the loss of data. *Andror2* (Wendland et al., 2021) contains 459 reproduced bug reports from 121 Android apps.

Others. *HireBuild* (Hassan and Wang, 2018) consists of 175 bugs in build scripts. The benchmark of *20 C# projects* (Garnier and Garcia, 2016) was created to evaluate the performance of information-retrieval approaches on 450 bugs found in 20 open source C# software projects. *BugsJS* (Gyimesi et al., 2021) is a collection of JavaScript bugs. *EDD* (Caballero et al., 2018) is short for Erlang Declarative Debugger. The repository hosts not only the source code of the debugger but also sample programs written in Erlang that were used to evaluate the debugger. *GoBench* (Yuan et al., 2021) is a collection of concurrency bugs in Go programs.

Table 3
Most frequent languages.

Language	No. benchmarks
Java	46
C	15
Python	10
C++	7
C/C++ mixed	7
C#	6
JavaScript	4
Go	4
Ruby	4
Scala	4
PHP	3
Kotlin	2
Perl	2
Groovy	2

4. Results

4.1. RQ1: What is the content and size of the individual benchmarks?

Language. The surveyed benchmarks contain programs written in 44 languages. Code4Bench is with 28 different languages the most diverse one. Table 3 lists the frequency of the languages that are used in more than one benchmark. The majority of the benchmarks contain Java programs (46 benchmarks) and C/C++ (29). There are 10 benchmarks with Python programs and 6 with C# programs. Languages that were used in only one benchmark include Befunge, Cobol, Delphi, F#, Haskel, and Erlang.

Content. The majority of the benchmarks are based on open source projects (52) and contain real faults (55). 18 benchmarks contain fabricated programs, 4 benchmarks contain programs from student submissions, 2 benchmarks contain programs from a coding contest and 3 benchmarks have taken programs from other benchmarks. 22 benchmarks contain programs with artificial faults. However, benchmarks containing fabricated programs or student submissions also have their merits, as such small and

[2] We did not consider the Parametric Cryptographic Misuses benchmark (Wickert et al., 2019) as part of MUBench since the pull request is still pending (last checked 2022-03-21).

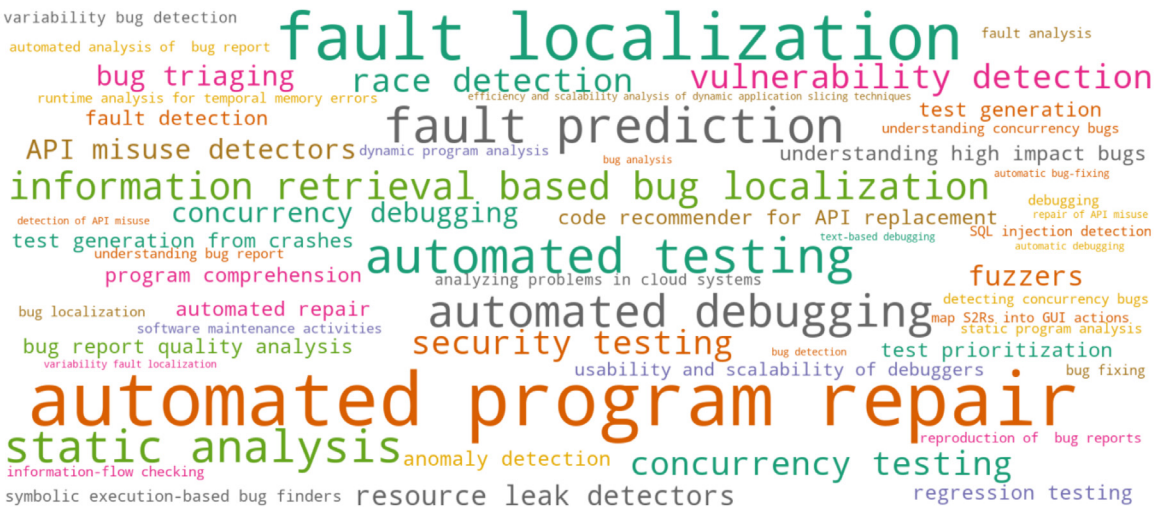

Fig. 2. Mentioned purposes.

well defined bugs are required in tool development and evaluations of proof-of-concept tools. Some of the benchmarks focus on special fault types, e.g., concurrency bugs (7), vulnerabilities (6), leaks (3), variability bugs (2), non-functional bugs (2), and API misuses (3).

15 benchmarks provide a single failing test case for each fault, one benchmark provides a failing test case for some of the programs, 18 benchmarks provide test suites, and 39 benchmarks do not provide any test cases. Some benchmarks provide issue tickets (30), calculated code metrics (6), patches (43) or Docker files (15).

Size. There are on average 128.1 programs (Median: 15.0) per benchmark. NIST SAMATE is the largest benchmark w.r.t. the number of programs with 2 740 different programs. There are on average 54217.6 faults (Median: 450.0) per benchmark. Code4Bench is the largest benchmark w.r.t the number of faults with 3 421 357 faults. Table B.5 in Appendix B lists the number of programs and their sizes as well as the number of bugs per benchmark and the availability of test cases, issue tickets, metrics, patches, and docker files.

Since the information provided in this section is very compressed, we refer the interested reader to our online appendix (see Section 8) where we provide a json file with detailed information for each benchmark including the fault type, information about the fault taxonomy (if provided), and the provided artifacts (including build, execution and download scripts).

4.2. RQ2: What is the intended purpose of the benchmarks?

Fig. 2 illustrates how frequently the different purposes were mentioned in the benchmarks. The purposes are very broad ranging from automated debugging (6) and automated testing (7) over static analysis (6) to fault prediction (7) and bug triaging (2). The majority of the benchmarks were created for evaluating automated program repair (19) and fault localization (14) approaches. Table B.6 in Appendix B lists the purposes of the individual benchmarks.

4.3. RQ3: What FAIR Guiding Principles are realized?

Fig. 3 shows the publication year (of the major publication), the average FAIR rating and the citation count of the individual benchmarks. The average FAIR rating is the mean of the five categories. When a category consists of several subcategories, the category value is the mean of the subcategories. When several benchmarks have the same publication year and the same average FAIR rating, but different citation scores, the benchmarks are listed according to their citation score, i.e., the benchmark with the highest score is listed above the others. Please keep in mind that earlier published benchmarks have a higher citation score compared to recently published benchmarks. Table B.7 in Appendix B shows the rating results of the individual benchmarks for each subcategory while Fig. 4 summarizes the rating results. Below, we discuss the results of the individual categories.

Findable. The majority of the benchmarks are hosted on GitHub (43) and on university servers (11). Seven benchmarks are hosted on Zenodo and two on Figshare. The benchmarks hosted on Zenodo and Figshare are the only benchmarks that have a DOI.

46 benchmarks store the actual data, while 27 only link to the actual data. Linking to external data comes with certain risks since the linked content might get deleted, moved, or modified, as we have observed in some benchmarks (see Section 5).

Accessible. The majority of the benchmarks (69) are publicly accessible. One benchmark provides instant access upon registration. Two benchmarks require the users to fill out a registration form and wait for access permissions being granted by a maintainer. One benchmark was obtained upon email request.

Interoperable. The majority of the benchmarks' data (23) is described in txt-files. 12 benchmarks use cvs-files, 12 json, 10 xml, 8 xlsx and 10 html. Less frequently occurring file formats are sql (3), yaml (3), SQLite (2), arff (1), doc (2), and pdf (4). For one benchmark, the data was encapsulated in a virtual box image. One benchmark is provided in the form of a webservice that allows accessing bug instances via API or web-form, however, a download of the full dataset is not intended by the benchmark maintainers.

Reuseable. 50 benchmarks are well-documented, 14 benchmarks have little documentation and 9 benchmarks have no documentation. 13 benchmarks use a BSD license, 6 benchmarks use a GPL or LGPL license, 6 benchmarks use an Apache license and 9 benchmarks use a creative commons license. The other used licenses are the SIR user license[3] (2), CRAPL[4] (1), ODbL.[5] (1) and one custom license[6] 34 benchmarks have no license.

Reproducible. 28 benchmarks used git as versioning system, 21 benchmarks indicated explicit version numbers, one benchmark

[3] https://sir.csc.ncsu.edu/portal/sir-license.php.
[4] https://github.com/stg-tud/MUBench/blob/master/CRAPL-LICENSE.txt.
[5] http://opendatacommons.org/licenses/odbl/1.0/
[6] https://github.com/LLNL/dataracebench/blob/master/LICENSE.txt.

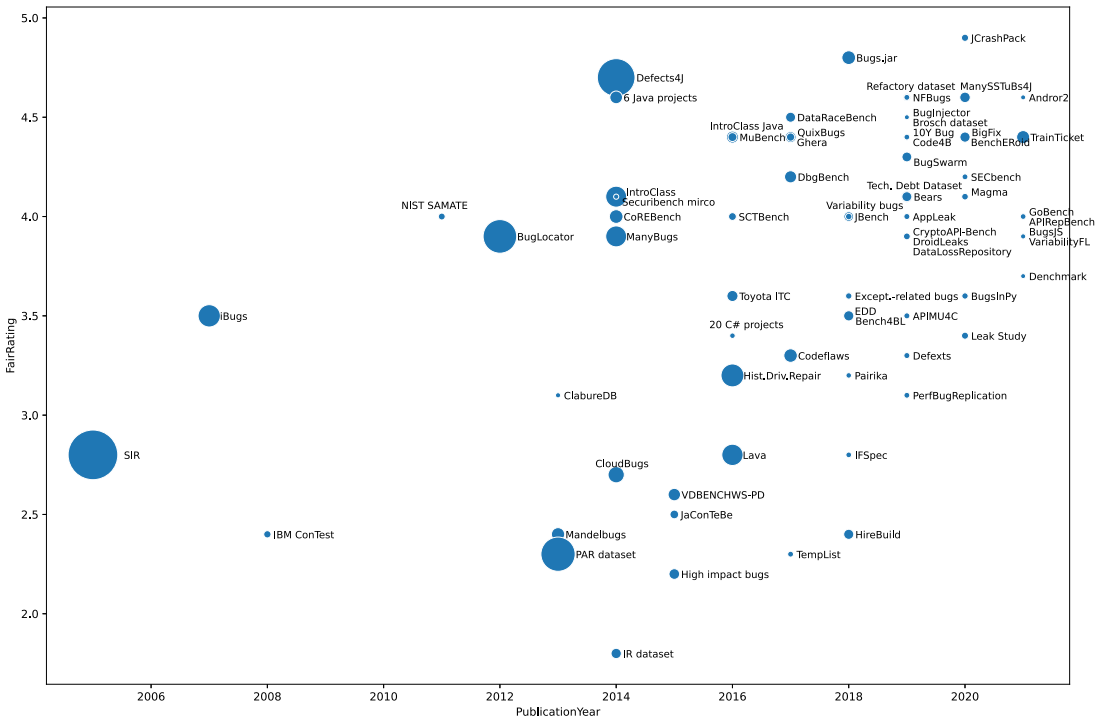

Fig. 3. Publication year, average FAIR rating and citation score (bubble size) of benchmarks.

had version names, four benchmarks indicated the change date, and 19 had no form of versioning.

48 benchmarks provide a full version history (the majority are hosted on GitHub or Zenodo), while 25 benchmarks do not provide any version history.

5. Discussion

We identified several issues while reviewing the benchmarks contained in this study. In the following discussion, we present issues and challenges that should be considered by researchers when creating a benchmark. While we do cite examples to clarify and underpin our points, we want to emphasize that this is to no malign intent as to discredit.

5.1. Data availability

Data availability is the most important factor for bug benchmarks, as an unavailable benchmark voids all further considerations or possible use cases. As already mentioned in Section 3, broken links are a widespread problem. There are many publications of which the original datasets became unavailable to the public over time. The reasons for this span from website gone offline, through data removed due to storage or traffic restrictions, to closed user accounts. For example, some universities have a policy of removing researchers' content some time after they left the institution, and dedicated websites for hosting datasets rely on a long-term financing plan to provide a certain security in data availability.

Beyond this, the practice of chaining multiple online services, or distributing parts of the dataset to different online services introduces more failure points, as the dataset may become unavailable if only one of these services ceases to operate in this regard. Examples of such patterns are:

- Published links point to dedicated domains, that do not contain the actual data, but link to another online service that is the actual data outlet. For example, the PAR dataset (Kim et al., 2013) links to a Google site, but the actual data is stored on Google drive and CloudBugs (Gunawi et al., 2014) refer to a University website while the actual database was hosted on a student's Google drive account.
- Published links point to a Google Forms page that enables its creators to identify the persons who download the data while hiding the actual data outlet behind said form (e.g., AppLeak Riganelli et al., 2019a).

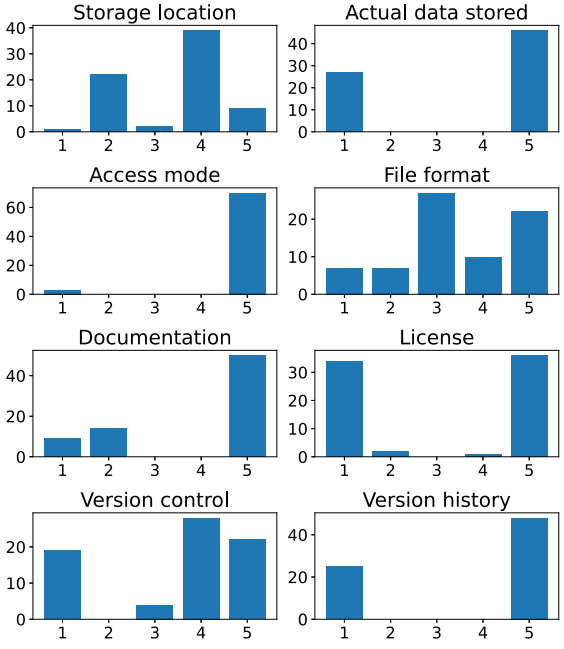

Fig. 4. Histograms of the rating results for the FAIR Guiding Principles Findable (Storage location, Actual data stored), Accessible (Access mode), Interoperable (File format), and Reusable (Documentation, License) and our new principle Reproducible (Version control, version history).

- Published links point to a GitHub repository containing scripts that download the actual data from a different data outlet (e.g., Bench4BL Lee et al., 2018, Defexts Benton et al., 2019, and Codeflaws Tan et al., 2017).

Furthermore, benchmarks that are not self-contained are subject to removal, relocation, and modification of the linked content. All of these problems can be avoided by hosting benchmarks on platforms that not only allow, but to some degree guarantee, long-term storage such as Zenodo and figshare without requiring reoccurring payments or actions from the uploaders; and of course by including *all data* instead of just linking to projects.

5.2. Data quality

Data quality has many facets and dimensions that have to be acknowledged by the developers of a benchmark. Due to this multiplicity of data quality, there are various issues that can lead to a deterioration in overall data quality. Here, we discuss two reoccurring problems that we have discovered and their effects on data quality.

The first problem is insufficient or completely missing metadata and description of the benchmark. Each benchmark should contain a minimum of meta information that allows other researchers to investigate suitability and applicability for a specific purpose. This meta information should at least describe the number and size of the programs, their programming language, the faults (w.r.t number, size and type) as well as the provided artifacts such as test cases, patches, metrics, docker files, and the benchmark's intended purpose in terms of target field of research or methodology.

The second problem concerns comprehensibility, reflected in the amount of effort that is required to understand how to use the benchmark. A benchmark's complexity often grows with the complexity of the methodology that is to be benchmarked or tested, and the amount of data required. However, this issue can be alleviated by providing extensive documentation, automation scripts, and small usage examples, as these are extremely helpful for other researchers.

5.3. Duplicated content

Duplicated content is endemic in uncurated student submission benchmarks. Student assignments are rather small and often simple, leading to the emergence of only a small number of unique solutions and bugs occurring in them, when putting aside differences in names and code styles. This causes an extensive degree of duplication, inflating the reported size of such benchmarks (see e.g., IntroClass Le Goues et al., 2015 and the Refactory dataset Hu et al., 2019). This is problematic since it reduces the expressiveness of results obtained from experiments performed on such benchmarks, by obscuring the sample size and diversity contained in the dataset. While there are scenarios where such student submission benchmarks have advantages over big real world bugs, they require curation to become more transparent in terms of sample size and contained diversity.

5.4. Data formats

Data formats used in the benchmark dictate the degree of portability and automatability of the benchmark. The wrong formatting choice can quickly deteriorate these properties, and induce unnecessary complexity in the dataset, impeding the utility of the dataset.

Given today's availability, portability, and ease of use of data formats for structured data, such standard formats should be used instead of implementing custom data formats as for example done in the Exception-related bugs benchmark (Zhong and Mei, 2018). However, if established data formats are used, the data must adhere to the selected format and must not be malformed (e.g., the test cases of QuixBugs Lin et al., 2017). Such misuse can be source of frustration and additional effort for users if identified, and could cause significant problems if it stays undetected. Further, information stored in machine unreadable formats as for example screenshots of textual content (e.g., Pairika Rahman et al., 2018) significantly reduce the practicality of a dataset.

Benchmark developers should not make assumptions about the context and environment their dataset is going to be used in, and henceforth, any included implementations or scripts should be considered optional and not mandatory for the user to work with the data. Benchmark data should therefore employ portable, machine-readable, well established, and preferably open, data formats.

The usage of fully fledged relational databases as for example MySQL (used e.g., in Code4Bench Majd et al., 2019 and HireBuild Hassan and Wang, 2018) is located on the other end of this data format spectrum. While such databases support concurrent access, offer high availability and performance, these qualities scarcely play a role for bug benchmarks. The data and structures transported in such relational databases, including their lightweight, and transportable, alternatives as e.g., SQLite (used e.g., in the Technical Debt Dataset Lenarduzzi et al., 2019 and ClabureDB Slaby et al., 2013), are most often simple enough to be well captured in plain formats such as xml, json, or even csv. While such databases fulfill aforementioned criteria of being machine readable, being well established, and to some degree portable, the introduced overhead is unwarranted almost every time in the domain of bug benchmarks.

5.5. Data set size

Excessive size combined with improper transfer format can make a dataset unwieldy. Some file archive formats have certain advantages and disadvantages, for example, the sequential nature of *tar.gz* files (used e.g., BigFix Li et al., 2020) lacking a table of contents that discourages its use for big collections of files where random access is desirable. While virtual machine images can contribute significantly towards reproducibility (e.g., PerfBugReplication Han et al., 2019), they do not constitute suitable transport mediums for datasets due to the excessive storage and tooling overhead inherent to virtual machines.

Despite today's high speed internet access, single file downloads in the magnitude of 100 GB can become an issue due to several reasons, including connection stability, speed, and available storage capacity. Aforementioned meta information becomes even more important for big datasets. Big datasets should be split into meaningful and workable subsets. Further, a small sample dataset containing a subset of the original data in the exact same structure should be made available, to aid researchers in selecting and investigating the benchmark's suitability, and enabling preliminary experiments before running their experiment on the full dataset.

5.6. Reproducibility

Reproducibility is enabled when researchers document the exact version of a dataset that was used in an experiment. The easiest way to allow for this is by providing specific version numbers for each release of a benchmark. However, a factor often overlooked, is that past versions of the dataset have to be kept available in order to allow reproduction. A popular choice are Git repositories. If handled correctly, they allow to document the exact state of the benchmark in form of a commit hash while

at the same time providing the full history of the benchmark. However, we identified some anti patterns that break these characteristics of Git repositories: Storing separate bug instances in separate branches (e.g., Bears Madeiral et al., 2019), or separate repositories (e.g., BugsJS Gyimesi et al., 2021), leads to an n-fold increase of commit hashes necessary to uniquely identify a specific benchmark state for reproduction experiments. While there are ways to split such a dataset into multiple repositories while maintaining the expressiveness of a single commit hash given a "main" repository (e.g., Bugs.jar Saha et al., 2018), we consider it preferable to use only a single repository for all data for the sake of simplicity.

5.7. Extensibility

Extensibility allows fellow researchers and users of the benchmark to extend or expand the dataset in various directions. A lack of extensibility does not only prevent other researchers from increasing the sample size in order to obtain more statistically relevant results, but also narrows the scope of possible applications of the benchmark (e.g., Zhong and Mei, 2018). Extensibility can be achieved by providing the code, algorithms, and preliminary steps that were used in creating the benchmark in addition to the resulting dataset (e.g., SECbench Reis and Abreu, 2017 and DroidLeaks Liu et al., 2019).

6. Threats to validity

This SLR underlies several threats to validity.

The list of benchmarks might be incomplete for two reasons: First, we used only one search engine for primary paper identification. Second, the used search string might be inadequate. To counteract this threat, we have chosen Scopus as a search engine because it is well suited as a primary search resource (Gusenbauer and Haddaway, 2020) and we tested if the search string is suited to find all benchmarks contained in a manually collected list of benchmarks known to the authors. While we carefully chose a primary paper search engine and a search string, benchmark papers might remain undetected. However, we performed backward and forward snowballing to identify additional papers.

Manual data extraction is an error-prone process and may lead to inconsistencies in the results. In order to reinforce consistency, Researcher 2 solely collected the data from the papers, the websites, and the benchmarks. In order to reduce the amount of errors, Researcher 1 validated the data collected by Researcher 2.

The ratings of the FAIR evaluation criteria are a threat to the internal validity. First, they are based on the personal judgment of the authors of this paper, and second, the ratings might be outdated in a couple of years as best practices and technology continue to evolve.

While this literature review provides a good overview over existing benchmarks, their size, purpose and quality w.r.t the FAIR criteria, it does not assess the quality of the benchmarks for the individual bugs. We did not analyze compatibility of the provided source code versions, if the bugs are reproducible and if the provided patches actually fix the bugs.

The presented data is based on a snapshot of the benchmarks at a discrete moment in time. Benchmarks are subject to change and might become unavailable, while new benchmarks will be introduced over time. To counteract, we host the data extracted from the benchmarks on Zenodo where we will provide new versions of the data whenever we are informed about changes.

7. Conclusion

Many benchmarks have been published in the last two decades. There are multiple possible reasons for this: 1. Existing benchmarks may lack in data and metrics researchers actually need for performing their experiments. 2. Existing benchmarks might be so hard to use, that researchers rather build new datasets that they in turn feel familiar with. 3. Up to now, no comprehensive listing or catalog of existing benchmarks has been available which would support findability.

This systematic literature review provides an overview of benchmarks for evaluating debugging approaches. It supports researchers to choose the best suited benchmark(s) for evaluating their approaches. The provided information can also help reviewers to assess benchmarks used in evaluations and/or to suggest alternatives. Evaluating debugging approaches with the best suited benchmark(s) helps to show a realistic picture of the state of the art in automated debugging and its limitations.

The abundance of benchmarks complicates quantitative comparison of the effectiveness of debugging approaches. Instead of creating more benchmarks, the research community should focus on improving existing benchmarks, e.g., raising the quality of the provided data, and providing sub-benchmarks focusing on certain aspects, e.g., concurrency bugs. Some benchmarks, e.g., Defects4J (Just et al., 2014) explicitly welcome contributions from other researchers. The advantage of a single large benchmark over several smaller benchmarks lies in the reduced overhead of having to deal with only one benchmark structure and format.

8. Supplemental material

A json-file with details on the investigated benchmarks and the evaluation script are publicly available on Zenodo (Hirsch and Hofer, 2022). The benchmarks presented in this literature review are subject to change. Benchmarks might be extended or might become unavailable. We see the supplemental material as a living dataset that we want to maintain over the next years and we encourage researchers to report detected changes to the existing benchmarks but also to inform us about new or missing benchmarks.

CRediT authorship contribution statement

Thomas Hirsch: Conceptualization, Validation, Writing – original draft, Writing – review & editing, Visualization. **Birgit Hofer:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The work described in this paper has been funded by the Austrian Science Fund (FWF): P 32653 (Automated Debugging in Use).

Table A.4
Benchmark overview.

Benchmark	Link	Investigated
10Y Bug-Fix DS (Vieira et al., 2019)	https://figshare.com/articles/Replication_Package_-_PROMISE_19/8852084	Version 4
20 C# projects (Garnier and Garcia, 2016)	https://mgarnier.github.io/sbes16.html	visited 2021-02-04
6 Java projects (Ye et al., 2014)	http://dx.doi.org/10.6084/m9.figshare.951967	visited 2020-11-19
Andror2 (Wendland et al., 2021)	https://zenodo.org/record/4646313	Version 1.0
APIMU4C (Gu et al., 2019)	https://github.com/imchecker/compsac19	Commit a9227be
APIRepBench (Kechagia et al., 2021)	https://github.com/SOLAR-group/APIARTy	Commit 914b9c7
AppLeak (Riganelli et al., 2019a)	https://goo.gl/forms/JZWWaeOK5TMbkacA2	Commit 1a4d70e
Bear (Madeiral et al., 2019)s	https://github.com/bears-bugs/bears-benchmark	Commit 912bb98
Bench4BL (Lee et al., 2018)	https://github.com/exatoa/Bench4BL	Commit 5480cb0
BenchERoid (Salehnamadi et al., 2020)	https://github.com/seal-hub/bencheroid	Commit ef4d046
BigFix (Li et al., 2020)	https://github.com/OOPSLA-2019-BugDetection/OOPSLA-2019-BugDetection	visited 2021-11-09
BroSCH (Kiss and Hodovan, 2019)	https://github.com/renatahodovan/brosch-dataset	Version r2018
BugInjector (Kashyap et al., 2019)	https://doi.org/10.5281/zenodo.3341585	visited 2020-11-20
BugLocator (Zhou et al., 2012)	https://code.google.com/archive/p/bugcenter/downloads	visited 2020-12-10
Bugs.jar (Saha et al., 2018)	https://github.com/bugs-dot-jar/bugs-dot-jar	Commit 8410717
BugsInPy (Widyasari et al., 2020)	https://github.com/soarsmu/BugsInPy	Commit a7ebd31
BugsJS (Gyimesi et al., 2021)	https://bugsjs.github.io/	Version 1.0
BugSwarm (Tomassi et al., 2019)	http://www.bugswarm.org/	Version 1.1.3
ClabureDB (Slaby et al., 2013)	http://decibel.fi.muni.cz:3000/	visited 2021-02-08
CloudBugs (Gunawi et al., 2014)	https://ucare.cs.uchicago.edu/projects/cbs/	visited 2020-12-10
Code4Bench (Majd et al., 2019)	https://doi.org/10.5281/zenodo.2582968	Version 1.0.0
Codeflaws (Tan et al., 2017)	http://codeflaws.github.io	visited 2020-11-13
CoREBench (Böhme and Roychoudhury, 2014)	https://www.comp.nus.edu.sg/~release/corebench/	Commit 7861cd5
CryptoAPIBench (Afrose et al., 2019)	https://github.com/CryptoAPI-Bench/CryptoAPI-Bench	Commit 6100c2e
DataLossRepository (Riganelli et al., 2019b)	https://gitlab.com/learnERC/DataLossRepository	Commit 39298e5
DataRaceBench (Liao et al., 2017)	https://github.com/LLNL/dataracebench	Version 1.3.2
DbgBench (Böhme et al., 2017)	http://www.st.cs.uni-saarland.de/debugging/dbgbench/	2020–11–19
Defects4J (Just et al., 2014)	http://defects4j.org	Version v2.0.0
Defexts (Benton et al., 2019)	http://www.github.com/defexts/defexts	Commit 4d8b018
Denchmark (Kim et al., 2021)	https://github.com/RosePasta/Denchmark_BRs	Commit ae893de
DroidLeaks (Liu et al., 2019)	https://zenodo.org/record/2589909	visited 2020-11-25
EDD (Caballero et al., 2018)	https://github.com/tamarit/edd	Commit 867f287
Except.-related bugs (Zhong and Mei, 2018)	https://github.com/drzhonghao/exception.bugs	Commit 0454039
Ghera (Mitra and Ranganath, 2017)	https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks	Commit ea1dbe2
GoBench (Yuan et al., 2021)	https://github.com/timmyyuan/gobench	3038ba4
High impact bugs (Ohira et al., 2015)	http://floss-lab.org/?p=1009	visited 2020-11-30
HireBuild (Hassan and Wang, 2018)	https://sites.google.com/site/buildfix2017/	visited 2020-12-01
Hist. Driv. Repair (Le et al., 2016)	https://github.com/xuanbachle/data-bugfixes	Commit da3ab4b
IBM ConTest (Eytani et al., 2008)	https://researcher.watson.ibm.com/researcher/view_person_subpage.php?id=5722	visited 2020-11-24
iBugs (Dallmeier and Zimmermann, 2007)	https://www.st.cs.uni-saarland.de/ibugs/	visited 2020-11-18
IFSpec (Hamann et al., 2018)	http://www.spp-rs3.de/IFSpec	visited 2021-02-04
IntroClass (Le Goues et al., 2015)	https://repairbenchmarks.cs.umass.edu/	visited 2020-11-13
IntroClassJava (Durieux and Monperrus, 2016)	https://github.com/Spirals-Team/IntroClassJava	Commit aec9de3
IR dataset (Saha et al., 2014)	upon request via email	2020–11–20
JaConTeBe (Lin et al., 2015)	http://sir.unl.edu/portal/bios/JaConTeBe.php	SIR Version 1.0
JBench (Gao et al., 2018)	https://github.com/buptsseGJ/ComRaDe/tree/master/benchmark-suite	Commit 2ffb088
JCrashPack (Soltani et al., 2020)	https://github.com/STAMP-project/JCrashPack	Version 1.0.1
Lava-1/Lava-M (Dolan-Gavitt et al., 2016)	https://sites.google.com/site/steelix2017/home/lava	visited 2021-02-03
LeakStudy (Ghanavati et al., 2020)	https://github.com/heiqs/leak_study	Commit f532d0b
Magma (Hazimeh et al., 2020)	https://github.com/HexHive/magma	Version V1.1.0
Mandelbugs (Cotroneo et al., 2013)	http://wpage.unina.it/roberto.natella/datasets/mandelbugs_oss/	visited 2020-12-14
ManyBugs (Le Goues et al., 2015)	https://repairbenchmarks.cs.umass.edu/	visited 2020-11-13
ManySStuBs4J (Karampatsis and Sutton, 2020)	https://doi.org/10.5281/zenodo.3653444	Version 2.0
MuBench (Amann et al., 2016)	https://github.com/stg-tud/MUBench	Commit 44558f3f
NFBugs (Radu and Nadi, 2019)	https://github.com/ualberta-smr/NFBugs	Commit 65d9ef6
NIST SAMATE (Black, 2011)	https://samate.nist.gov/SARD/testsuite.php	Version 1.3
OWASP	https://owasp.org/www-project-benchmark/	Version 1.2
Pairika (Rahman et al., 2018)	https://github.com/tum-i22/Pairika	Commit 3b34e68
PAR dataset (Kim et al., 2013)	https://sites.google.com/site/autofixhkust/	visited 2021-02-03
PerfBugReplication (Han et al., 2019)	https://github.com/xha225/PerfBugReplication	Commit ddf3dbe
QuixBugs (Lin et al., 2017)	https://github.com/jkoppel/QuixBugs	Commit 09d59c6
Refactory dataset (Hu et al., 2019)	https://github.com/githubhuyang/refactory	Commit 4df4216
SCTBench (Thomson et al., 2016)	https://github.com/mc-imperial/sctbench	Commit 4a8e554f
SECbench (Reis and Abreu, 2017)	https://github.com/TQRG/secbench	Commit ad661b5
Securibench mirco (Livshits, 2014)	https://github.com/too4words/securibench-micro	Commit 6a5a724
SIR (Do et al., 2005)	https://sir.csc.ncsu.edu/	visited 2020-12-07
Tech. Debt Dataset (Lenarduzzi et al., 2019)	https://github.com/clowee/The-Technical-Debt-Dataset	Version 1.0.1
TempList (Vorobyov et al., 2017)	http://nikolai.kosmatov.free.fr/TempLIST.zip	visited 2020-12-15
Toyota ITC (Shiraishi et al., 2016)	https://github.com/regehr/itc-benchmarks	Commit 7bfb56a
TrainTicket (Zhou et al., 2021)	https://fudanselab.github.io/research/MSFaultEmpiricalStudy/	visited 2021-11-10
Variability bugs (Abal et al., 2018)	https://bitbucket.org/itu-square/vbdb/	Commit 307bd3a
VariabilityFL (Ngo et al., 2021)	https://tuanngokien.github.io/splc2021/	Version 2
VDBENCHWS-PD (Antunes and Vieira, 2015)	http://eden.dei.uc.pt/~nmsa/publications/vdbench-ws.zip	visited 2020-12-16

Table B.5
Description of the individual benchmarks. |Prog.|: Number of programs, B: other benchmarks, C: commercial software, CON: programming contest submissions, OS: open source projects, RW: real world projects, S: student programs, art.: artificial, fab.: fabricated.

	|Prog.|	|Faults|	Program size	Source Type	Fault Type	Test Cases	Issue Tickets	Metrics	Docker	Patch
10Y Bug-Fix DS	55	70000	mid-large	OS	real	none	True	True	False	False
20 C# projects	20	450	mid-large	OS	real	none	True	True	False	True
6 Java projects	6	22747	large	OS	real	none	True	False	False	True
Andror2	43	90	mid	OS	real	none	True	True	False	True
APIMU4C	3	2272	small-large	fab., OS	both	none	False	False	False	False
APIRepBench	29	101	large	B	real	1	True	False	True	True
AppLeak	15	40	small-large	OS	real	1	True	False	False	True
Bears	72	251	small-large	OS	real	suite	False	False	False	True
Bench4BL	51	10017	mid-large	OS	real	none	True	False	False	False
BenchERoid	34	36	tiny	fab.	art.	1	False	False	False	False
BigFix	8	16928	large	OS	real	none	True	False	False	True
Brosch dataset	2	13365	large	OS	real	none	False	False	False	True
BugInjector	2	3994	mid-large	OS	art.	1	False	False	False	True
BugLocator	2	118	unknown	OS	real	none	True	False	False	False
Bugs.jar	8	1158	large	OS	real	suite	True	False	False	True
BugsInPy	17	493	small-large	OS	real	suite	False	False	False	True
BugsJS	10	453	mid	OS	real	suite	True	False	True	True
BugSwarm	179	3232	unknown	OS	real	1	False	False	True	True
ClabureDB	3	847	large	OS	real	none	True	False	False	False
CloudBugs	6	4003	large	OS	real	none	True	False	False	False
Code4Bench	NaN	3421357	unknown	CON	real	suite	False	False	False	False
Codeflaws	1280	3902	tiny	CON	real	suite	False	False	False	True
CoREBench	4	70	mid	OS	real	1	True	False	True	True
CryptoAPI-Bench	171	135	tiny	fab.	art.	none	False	False	False	False
DataLossRepository	48	110	mid-large	OS	real	1	True	False	False	True
DataRaceBench	334	165	tiny	fab.	art.	none	False	False	False	False
DbgBench	2	27	mid	OS	real	1	True	False	True	True
Defects4J	17	835	mid	OS	real	suite	True	False	True	True
Defexts	416	654	unknown	OS	real	suite	False	False	False	True
Denchmark	193	4577	mid-large	OS	real	none	True	True	False	True
DroidLeaks	32	292	small-large	OS	real	none	True	False	False	True
EDD	37	40	tiny-small	fab., OS	art.	1	False	False	False	True
Except.-related bugs	5	2018	large	OS	real	none	True	False	False	False
Ghera	60	60	small	fab.	art.	1	True	False	False	True
GoBench	9	185	mid-large	fab., OS	real	none	False	False	True	True
High impact bugs	4	4000	large	OS	real	none	True	False	False	False
HireBuild	54	175	unknown	OS	real	none	False	False	False	True
Hist.Driv.Repair	772	3000	mid-large	OS	real	0–1	False	False	False	True
IBM ConTest	40	40	tiny-small	S, C, OS	art.	none	False	False	False	False
iBugs	3	390	mid	OS	real	suite	True	False	False	True
IFSpec	232	80	tiny	B	art.	none	False	False	False	False
IntroClass	6	572	tiny	S	real	suite	False	False	False	False
IntroClass Java	6	297	small	S	real	suite	False	False	False	False
IR dataset	5	7173	large	OS	real	none	False	False	False	False
JaConTeBe	7	47	tiny-small	OS	real	1	False	False	False	False
JBench	48	985	small-large	fab., OS	both	1	False	False	False	False
JCrashPack	7	200	mid-large	OS	real	none	True	False	False	False
Lava	5	2850	mid-large	OS	art.	1	False	False	False	False
Leak Study	15	491	large	OS	real	none	True	True	False	True
Magma	7	118	large	OS	real	1	False	False	True	True
Mandelbugs	4	852	mid-large	OS	real	none	True	False	False	False
ManyBugs	9	185	large	OS	real	suite	True	False	True	True
ManySSTuBs4J	1100	179191	small-large	OS	real	none	False	False	False	True
MuBench	66	89	unknown	OS	real	none	False	False	True	True
NFBugs	65	133	unknown	OS	real	none	False	False	False	True
NIST SAMATE	335	121922	tiny-small	fab.	art.	none	False	False	False	False
OWASP	2740	2740	tiny	fab.	art.	none	False	False	True	False
Pairika	1	40	large	OS	real	suite	True	False	False	True
PAR dataset	6	119	small	OS	real	suite	False	False	False	False
PerfBugReplication	2	17	unknown	OS	real	none	False	False	True	True
QuixBugs	80	80	tiny	fab.	art.	suite	False	False	False	True
Refactory dataset	5	1783	tiny	S	real	suite	False	False	True	True
SCTBench	15	49	small	B	both	none	False	False	True	False
SECbench	114	1352	unknown	OS	real	none	False	False	False	True
Securibench mirco	96	134	tiny	fab.	art.	none	False	False	False	False
SIR	85	471	tiny-large	fab., OS	both	suite	False	False	False	True
Tech. Debt Dataset	33	40890	unknown	OS	real	none	True	True	False	True
TempList	15	23	tiny	fab.	art.	none	False	False	False	False
Toyota ITC	52	638	tiny	fab.	art.	none	False	False	False	False
TrainTicket	1	22	large	fab.	art.	1	False	False	True	True
Variability bugs	4	98	mid-large	OS	real	none	True	False	False	True
VariabilityFL	6	1570	small	fab.	art.	suite	False	False	False	True
VDBENCHWS-PD	3	49	small	fab.	art.	none	False	False	False	False

Table B.6
Purposes of the individual benchmarks as mentioned in the papers.

	Purpose
10Y Bug-Fix DS	fault prediction
20 C# projects	information retrieval based bug localization
6 Java projects	fault localization
APIMU4C	API misuse detectors
APIRepBench	detection of API misuse, repair of API misuse
Andror2	automated analysis of bug report, understanding bug report, reproduction of bug reports, bug localization, fault localization, bug fixing, software maintenance activities, map S2Rs into GUI actions, static program analysis, dynamic program analysis, efficiency and scalability analysis of dynamic application slicing techniques
AppLeak	resource leak detectors, fault localization, automated program repair
Bears	automated program repair
Bench4BL	information retrieval based bug localization
BenchERoid	race detection
BigFix	bug detection
Brosch dataset	fault prediction
BugInjector	static analysis
BugLocator	fault localization
BugSwarm	test generation, test prioritization, fault localization, automated program repair, anomaly detection
Bugs.jar	automated debugging, automated program repair, automated testing, information retrieval based bug localization, bug triaging, bug report quality analysis, program comprehension
BugsInPy	automated testing, automated debugging
BugsJS	automated testing, fault prediction, automated repair, fault localization
ClabureDB	static analysis
CloudBugs	analyzing problems in cloud systems
CoREBench	regression testing, automated debugging, automated program repair
Code4Bench	automated program repair, automated testing, fault localization, fault prediction
Codeflaws	automated program repair
CryptoAPI-Bench	security testing
DataLossRepository	fault detection, automated debugging
DataRaceBench	race detection
DbgBench	automated debugging
Defects4J	automated testing
Defexts	static analysis, fault localization, automated program repair, fault prediction

(*continued on next page*)

Table B.6 (*continued*).

	Purpose
Denchmark	automatic debugging, bug analysis, fault localization, automatic bug-fixing, text-based debugging
DroidLeaks	resource leak detectors
EDD	usability and scalability of debuggers, concurrency debugging
Except.-related bugs	automated program repair
Ghera	vulnerability detection
GoBench	understanding concurrency bugs, detecting concurrency bugs
High impact bugs	understanding high impact bugs, fault prediction, fault localization, bug triaging
HireBuild	automated program repair
Hist.Driv.Repair	automated program repair
IBM ConTest	concurrency testing, concurrency debugging
IFSpec	information-flow checking
IR dataset	information retrieval based bug localization
IntroClass	automated program repair
IntroClass Java	automated program repair, fault localization
JBench	race detection
JCrashPack	test generation from crashes
JaConTeBe	concurrency testing
Lava	fuzzers, symbolic execution-based bug finders
Leak Study	fault localization, automated program repair
Magma	fuzzers
Mandelbugs	fault prediction
ManyBugs	automated program repair
ManySSTuBs4J	automated program repair
MuBench	API misuse detectors
NFBugs	code recommender for API replacement
NIST SAMATE	static analysis
OWASP	vulnerability detection
PAR dataset	automated program repair
Pairika	fault localization
PerfBugReplication	automated testing, automated debugging
QuixBugs	automated program repair
Refactory dataset	automated program repair
SCTBench	concurrency testing
SECbench	security testing
SIR	automated testing
Securibench mirco	security testing
Tech. Debt Dataset	static analysis
TempList	runtime analysis for temporal memory errors
Toyota ITC	static analysis
TrainTicket	fault analysis, debugging
VDBENCHWS-PD	SQL injection detection, vulnerability detection
Variability bugs	variability bug detection
VariabilityFL	variability fault localization
iBugs	fault localization

Table B.7
FAIR rating for the individual benchmarks. (1a) Storage location, (1b) Actual data stored, (2) Access mode, (3) File format, (4a) Documentation, (4b) License, (5a) Version control, (5b) Version history, (O) Overall rating.

	(1a)	(1b)	(2)	(3)	(4a)	(4b)	(5a)	(5b)	(O)
10Y Bug-Fix DS	5	1	5	4	5	5	5	5	4.4
20 C# projects	4	5	5	5	2	1	1	1	3.4
6 Java projects	5	1	5	5	5	5	5	5	4.6
Andror2	5	1	5	5	5	5	5	5	4.6
APIMU4C	4	5	5	2	2	1	4	5	3.5
APIRepBench	4	1	5	5	5	1	4	5	4.0
AppLeak	4	5	5	3	5	1	4	5	4.0
Bears	4	5	5	5	5	5	1	1	4.1
Bench4BL	2	5	5	5	5	1	1	1	3.5
BenchERoid	4	5	5	3	5	5	4	5	4.4
BigFix	5	5	5	2	5	5	5	5	4.4
Brosch dataset	4	1	5	5	5	5	5	5	4.5
BugInjector	5	5	5	3	5	5	5	5	4.6
BugLocator	4	5	5	5	1	5	3	1	3.9
Bugs.jar	4	5	5	5	5	5	4	5	4.8
BugsInPy	4	1	5	3	5	1	4	5	3.6
BugsJS	4	5	5	4	5	5	1	1	3.9
BugSwarm	2	5	5	5	5	1	5	5	4.3
ClabureDB	2	1	5	3	5	5	1	1	3.1
CloudBugs	2	1	5	3	5	1	1	1	2.7
Code4Bench	5	5	5	2	5	5	5	5	4.4
Codeflaws	2	5	5	4	5	1	1	1	3.3
CoREBench	4	1	5	3	5	5	4	5	4.0
CryptoAPI-Bench	4	5	5	2	2	5	4	5	3.9
DataLossRepository	3	5	5	3	5	1	4	5	3.9
DataRaceBench	4	5	5	3	5	5	5	5	4.5
DbgBench	4	1	5	4	5	5	4	5	4.2
Defects4J	4	5	5	4	5	5	5	5	4.7
Defexts	2	5	5	4	5	1	1	1	3.3
Denchmark	4	1	5	5	2	1	4	5	3.7
DroidLeaks	5	1	5	3	2	5	5	5	3.9
EDD	4	1	5	1	5	5	4	5	3.6
Except.-related bugs	4	1	5	3	5	1	4	5	3.6
Ghera	4	5	5	3	5	5	4	5	4.4
GoBench	4	1	5	5	5	1	4	5	4.0
High impact bugs	2	1	5	2	2	1	1	1	2.2
HireBuild	2	1	5	2	2	1	3	1	2.4
Hist.Driv.Repair	4	1	5	3	1	1	4	5	3.2
IBM ConTest	2	5	5	1	2	1	1	1	2.4
iBugs	2	5	5	5	5	1	1	1	3.5
IFSpec	2	5	5	3	2	1	1	1	2.8
IntroClass	2	5	5	5	5	5	3	1	4.1
IntroClass Java	4	5	5	5	5	1	4	5	4.4
IR dataset	1	1	1	5	1	1	1	1	1.8
JaConTeBe	2	5	1	3	2	2	5	1	2.5
JBench	4	5	5	5	1	1	4	5	4.0
JCrashPack	4	5	5	5	5	5	5	5	4.9
Lava	2	5	5	3	2	1	1	1	2.8
Leak Study	4	1	5	4	1	1	4	5	3.4
Magma	4	5	5	3	5	1	5	5	4.1
Mandelbugs	2	1	5	3	2	1	1	1	2.4
ManyBugs	2	5	5	4	5	5	3	1	3.9
ManySSTuBs4J	5	1	5	5	5	5	5	5	4.6
MuBench	4	1	5	5	5	4	5	5	4.4
NFBugs	5	1	5	5	5	5	5	5	4.6
NIST SAMATE	3	5	5	3	5	1	5	5	4.0
OWASP	4	5	5	5	5	5	5	5	4.9
Pairika	4	1	5	1	5	1	4	5	3.2
PAR dataset	2	5	5	1	1	1	1	1	2.3
PerfBugReplication	2	5	5	1	5	5	1	1	3.1
QuixBugs	4	5	5	3	5	5	4	5	4.4
Refactory dataset	4	5	5	4	5	5	4	5	4.6
SCTBench	4	5	5	3	5	1	4	5	4.0
SECbench	4	1	5	4	5	5	4	5	4.2
Securibench mirco	4	5	5	3	2	5	4	5	4.1
SIR	2	5	1	3	5	2	5	1	2.8
Tech. Debt Dataset	4	1	5	3	5	5	5	5	4.1
TempList	2	5	5	1	1	1	1	1	2.3
Toyota ITC	4	5	5	1	1	5	4	5	3.6
TrainTicket	4	5	5	3	5	5	4	5	4.4
Variability bugs	4	5	5	3	1	5	4	5	4.0
VariabilityFL	2	5	5	3	5	1	5	5	3.9
VDBENCHWS-PD	2	5	5	2	2	1	1	1	2.6

Appendix A. Investigated benchmarks

See Table A.4.

Appendix B. Details on benchmarks

We classify the size of programs into four categories: tiny (<100 LOC), small (<5KLOC), medium (<100 KLOC), and large (>100 KLOC).

See Table B.5, Table B.6, and Table B.7

References

Abal, I., Melo, J., Stanciulescu, S., Brabrand, C., Ribeiro, M., Wąsowski, A., 2018. Variability bugs in highly configurable systems: A qualitative analysis. ACM Trans. Softw. Eng. Methodol. 26 (3), 1–34. http://dx.doi.org/10.1145/3149119.

Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J., 2009. A practical evaluation of spectrum-based fault localization. J. Syst. Softw. 82 (11), 1780–1792. http://dx.doi.org/10.1016/j.jss.2009.06.035.

Afrose, S., Rahaman, S., Yao, D.D., 2019. CryptoAPI-bench: A comprehensive benchmark on Java cryptographic API misuses. In: Secure Development. SecDev, pp. 49–61. http://dx.doi.org/10.1109/SecDev.2019.00017.

Amann, S., Nadi, S., Nguyen, H.A., Nguyen, T.N., Mezini, M., 2016. MUBench: A benchmark for API-misuse detectors. In: 13th Working Conference on Mining Software Repositories. MSR, pp. 464–467. http://dx.doi.org/10.1145/2901739.2903506.

Ang, A., Perez, A., Van Deursen, A., Abreu, R., 2017. Revisiting the practical use of automated software fault localization techniques. In: Int. Symposium on Software Reliability Engineering Workshops. ISSREW, pp. 175–182. http://dx.doi.org/10.1109/ISSREW.2017.68.

Antunes, N., Vieira, M., 2015. Assessing and comparing vulnerability detection tools for web services: benchmarking approach and examples. IEEE Trans. Serv. Comput. 8 (2), 269–283. http://dx.doi.org/10.1109/TSC.2014.2310221.

Benton, S., Ghanbari, A., Zhang, L., 2019. Defexts: A curated dataset of reproducible real-world bugs for modern JVM languages. In: 41st International Conference on Software Engineering: Companion. ICSE-Companion, pp. 47–50. http://dx.doi.org/10.1109/ICSE-Companion.2019.00035.

Black, P.E., 2011. Counting bugs is harder than you think. In: 11th International Working Conference on Source Code Analysis and Manipulation. SCAM, pp. 1–9. http://dx.doi.org/10.1109/SCAM.2011.24.

Böhme, M., Roychoudhury, A., 2014. CoREBench: Studying complexity of regression errors. In: International Symposium on Software Testing and Analysis. ISSTA, pp. 105–115. http://dx.doi.org/10.1145/2610384.2628058.

Böhme, M., Soremekun, E.O., Chattopadhyay, S., Ugherughe, E., Zeller, A., 2017. Where is the bug and how is it fixed? An experiment with practitioners. In: 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2017, Vol. Part F1301, pp. 117–128. http://dx.doi.org/10.1145/3106237.3106255.

Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S., 2018. Declarative debugging of concurrent Erlang programs. J. Log. Algebraic Methods Program. 101, 22–41. http://dx.doi.org/10.1016/j.jlamp.2018.07.005.

Cotroneo, D., Grottke, M., Natella, R., Pietrantuono, R., Trivedi, K.S., 2013. Fault triggers in open-source software: An experience report. In: 24th International Symposium on Software Reliability Engineering. ISSRE, pp. 178–187. http://dx.doi.org/10.1109/ISSRE.2013.6698917.

Dallmeier, V., Zimmermann, T., 2007. Extraction of bug localization benchmarks from history. In: International Conference on Automated Software Engineering. ASE, pp. 433–436. http://dx.doi.org/10.1145/1321631.1321702.

Do, H., Elbaum, S., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir. Softw. Eng. 10 (4), 405–435. http://dx.doi.org/10.1007/s10664-005-3861-2.

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R., 2016. LAVA: Large-scale automated vulnerability addition. In: IEEE Symposium on Security and Privacy. SP 2016, pp. 110–121. http://dx.doi.org/10.1109/SP.2016.15.

Durieux, T., Abreu, R., 2019. Critical review of BugSwarm for fault localization and program repair. CoRR abs/1905.0. arXiv:1905.09375.

Durieux, T., Monperrus, M., 2016. IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. Tech. Rep., Technical Report, Universite Lille, \#hal-01272126. URL https://hal.archives-ouvertes.fr/hal-01272126.

Eytani, Y., Tzoref, R., Ur, S., 2008. Experience with a concurrency bugs benchmark. In: Workshop On a Benchmark For Software Testing (TestBench'08) - ICST Workshop proceedings. ICSTW'08, pp. 379–384. http://dx.doi.org/10.1109/ICSTW.2008.17.

Gao, J., Yang, X., Jiang, Y., Liu, H., Ying, W., Zhang, X., 2018. JBench: A dataset of data races for concurrency testing. In: 5th International Conference on Mining Software Repositories. MSR, pp. 6–9. http://dx.doi.org/10.1145/3196398.3196451.

Garnier, M., Garcia, A., 2016. On the evaluation of structured information retrieval-based bug localization on 20 C\# projects. In: 30th Brazilian Symposium on Software Engineering. SBES, pp. 123–132. http://dx.doi.org/10.1145/2973839.2973853.

Gazzola, L., Micucci, D., Mariani, L., 2019. Automatic software repair: a survey. IEEE Trans. Softw. Eng. 45 (1), 34–67. http://dx.doi.org/10.1109/TSE.2017.2755013.

Ghanavati, M., Costa, D., Seboek, J., Lo, D., Andrzejak, A., 2020. Memory and resource leak defects and their repairs in Java projects. Empir. Softw. Eng. 25 (1), 678–718. http://dx.doi.org/10.1007/s10664-019-09731-8.

Gu, Z., Wu, J., Liu, J., Zhou, M., Gu, M., 2019. An empirical study on API-misuse bugs in open-source C programs. In: 43rd Annual Computer Software and Applications Conference. COMPSAC, pp. 11–20. http://dx.doi.org/10.1109/COMPSAC.2019.00012.

Gunawi, H.S., Gunawi, H.S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K.J., Laksono, A., Lukman, J.F., Martin, V., Satria, A.D., 2014. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In: 5th Symposium on Cloud Computing. SOCC, pp. 1–14. http://dx.doi.org/10.1145/2670979.2670986.

Gusenbauer, M., Haddaway, N.R., 2020. Which academic search systems are suitable for systematic reviews or meta-analyses? Evaluating retrieval qualities of Google Scholar, PubMed, and 26 other resources. Res. Synth. Methods 11 (2), 181–217. http://dx.doi.org/10.1002/JRSM.1378.

Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, A., Ferenc, R., Mesbah, A., 2021. BUGSJS: A benchmark and taxonomy of JavaScript bugs. Softw. Test. Verif. Reliab. 31 (4), e1751. http://dx.doi.org/10.1002/STVR.1751.

Hamann, T., Herda, M., Mantel, H., Mohr, M., Schneider, D., Tasch, M., 2018. A uniform information-flow security benchmark suite for source code and bytecode. In: Lecture Notes in Computer Science. Vol. 11252, pp. 437–453. http://dx.doi.org/10.1007/978-3-030-03638-6_27.

Han, X., Carroll, D., Yu, T., 2019. Reproducing performance bug reports in server applications: The researchers' experiences. J. Syst. Softw. 156, 268–282. http://dx.doi.org/10.1016/j.jss.2019.06.100.

Hassan, F., Wang, X., 2018. HireBuild: AN automatic approach to history-driven repair of build scripts. In: 40th Int. Conference on Software Engineering. ICSE, pp. 1078–1089. http://dx.doi.org/10.1145/3180155.3180181.

Hazimeh, A., Herrera, A., Payer, M., 2020. Magma: a ground-truth fuzzing benchmark. ACM Meas. Anal. Comput. Syst. 4 (3), 1–29. http://dx.doi.org/10.1145/3428334.

Hirsch, T., Hofer, B., 2021. What we can learn from how programmers debug their code. In: 8th International Workshop on Software Engineering Research and Industrial Practice. SER-IP, pp. 37–40. http://dx.doi.org/10.1109/SER-IP52554.2021.00014.

Hirsch, T., Hofer, B., 2022. Supplemental Material for a Systematic Literature Review on Benchmarks for Evaluating Debugging Approaches. Zenodo, http://dx.doi.org/10.5281/zenodo.6670198.

Hu, Y., Ahmed, U.Z., Mechtaev, S., Leong, B., Roychoudhury, A., 2019. Refactoring based program repair applied to programming assignments. In: 34th International Conference on Automated Software Engineering. ASE, pp. 388–398. http://dx.doi.org/10.1109/ASE.2019.00044.

Just, R., Jalali, D., Ernst, M.D., 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In: International Symposium on Software Testing and Analysis. ISSTA, pp. 437–440. http://dx.doi.org/10.1145/2610384.2628055.

Karampatsis, R.M., Sutton, C., 2020. How often do single-statement bugs occur?: the ManySStuBs4J dataset. In: 7th International Conference on Mining Software Repositories. MSR, pp. 573–577. http://dx.doi.org/10.1145/3379597.3387491.

Kashyap, V., Ruchti, J., Kot, L., Turetsky, E., Swords, R., Pan, S.A., Henry, J., Melski, D., Schulte, E., 2019. Automated customized bug-benchmark generation. In: 19th International Working Conference on Source Code Analysis and Manipulation. SCAM, pp. 103–114. http://dx.doi.org/10.1109/SCAM.2019.00020.

Kechagia, M., Mechtaev, S., Sarro, F., Harman, M., 2021. Evaluating automatic program repair capabilities to repair API misuses. IEEE Trans. Softw. Eng. http://dx.doi.org/10.1109/TSE.2021.3067156.

Kim, M., Kim, Y., Lee, E., 2021. Denchmark: A bug benchmark of deep learning-related software. In: 18th International Conference on Mining Software Repositories. MSR, pp. 540–544. http://dx.doi.org/10.1109/MSR52588.2021.00070.

Kim, D., Nam, J., Song, J., Kim, S., 2013. Automatic patch generation learned from human-written patches. In: International Conference on Software Engineering. ICSE, pp. 802–811.

Kiss, A., Hodovan, R., 2019. Security-related commits in open source web browser projects. In: 34th International Conference on Automated Software Engineering Workshops. ASEW, pp. 57–60. http://dx.doi.org/10.1109/ASEW.2019.00029.

Kitchenham, B., Charters, S., 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering. Tech. Rep., EBSE Technical Report, pp. 1–57, URL https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.471.

Le, X.B.D., Lo, D., Le Goues, C., 2016. History driven program repair. In: 23rd International Conference on Software Analysis, Evolution, and Reengineering. pp. 213–224. http://dx.doi.org/10.1109/saner.2016.76.

Le Goues, C., Holtschulte, N., Smith, E.K., Brun, Y., Devanbu, P., Forrest, S., Weimer, W., 2015. The ManyBugs and IntroClass benchmarks for automated repair of c programs. Trans. Softw. Eng. 41 (12), 1236–1256. http://dx.doi.org/10.1109/TSE.2015.2454513.

Lee, J., Kim, D., Bissyandé, T.F., Jung, W., Le Traon, Y., 2018. Bench4BL: Reproducibility study on the performance of IR-based bug localization. In: 27th International Symposium on Software Testing and Analysis. ISSTA, pp. 61–72. http://dx.doi.org/10.1145/3213846.3213856.

Lenarduzzi, V., Saarimäki, N., Taibi, D., 2019. The technical debt dataset. 15th Conference on Predictive Models and Data Analytics in Software Engineering 2–11. http://dx.doi.org/10.1145/3345629.3345630.

Li, Y., Wang, S., Nguyen, T.N., Van Nguyen, S., 2020. Improving bug detection via context-based code representation learning and attention-based neural networks. In: 42nd International Conference on Software Engineering. ICSE, pp. 602–614. http://dx.doi.org/10.1145/3360588.

Liao, C., Lin, P.H., Asplund, J., Schordan, M., Karlin, I., 2017. DataRaceBench: A Benchmark suite for systematic evaluation of data race detection tools. In: International Conference for High Performance Computing, Networking, Storage and Analysis. SC 2017, Vol. 14, pp. 1–14. http://dx.doi.org/10.1145/3126908.3126958.

Lin, D., Chen, A., Koppel, J., Solar-Lezama, A., 2017. QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge. In: ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. SPLASH Companion, pp. 55–56. http://dx.doi.org/10.1145/3135932.3135941.

Lin, Z., Marinov, D., Zhong, H., Chen, Y., Zhao, J., 2015. JaConTeBe: A Benchmark suite of real-world Java concurrency bugs. In: 30th International Conference on Automated Software Engineering. ASE, pp. 178–189. http://dx.doi.org/10.1109/ASE.2015.87.

Liu, Y., Wang, J., Wei, L., Xu, C., Cheung, S.C., Wu, T., Yan, J., Zhang, J., 2019. DroidLeaks: A comprehensive database of resource leaks in Android apps. Empir. Softw. Eng. 24 (6), 3435–3483. http://dx.doi.org/10.1007/s10664-019-09715-8.

Livshits, B., 2014. Securibench micro. URL https://github.com/too4words/securibench-micro.

Madeiral, F., Urli, S., Maia, M., Monperrus, M., 2019. BEARS: An extensible java bug benchmark for automatic program repair studies. In: 26th International Conference on Software Analysis, Evolution, and Reengineering. SANER, pp. 468–478. http://dx.doi.org/10.1109/SANER.2019.8667991.

Majd, A., Vahidi-Asl, M., Khalilian, A., Baraani-Dastjerdi, A., Zamani, B., 2019. Code4Bench: A Multidimensional benchmark of Codeforces data for different program analysis techniques. J. Comput. Lang. 53, 38–52. http://dx.doi.org/10.1016/j.cola.2019.03.006.

Mitra, J., Ranganath, V.P., 2017. Ghera: A repository of android app vulnerability benchmarks. In: 13th International Conference on Predictive Models and Data Analytics in Software Engineering. PROMISE, Vol. 10, pp. 43–52. http://dx.doi.org/10.1145/3127005.3127010.

Ngo, K.-T., Nguyen, T.-T., Nguyen, S., Vo, H.D., 2021. Variability fault localization: a benchmark. In: 25th ACM International Systems and Software Product Line Conference (SPLC)- Volume A. pp. 120–125. http://dx.doi.org/10.1145/3461001.3473058.

Ohira, M., Kashiwa, Y., Yamatani, Y., Yoshiyuki, H., Maeda, Y., Limsettho, N., Fujino, K., Hata, H., Ihara, A., Matsumoto, K., 2015. A dataset of high impact bugs: Manually-classified issue reports. In: International Working Conference on Mining Software Repositories. MSR, pp. 518–521. http://dx.doi.org/10.1109/MSR.2015.78.

OWASP Foundation, 2022. OWASP benchmark. URL https://owasp.org/www-project-benchmark/.

Radu, A., Nadi, S., 2019. A dataset of non-functional bugs. In: International Working Conference on Mining Software Repositories. MSR, pp. 399–403. http://dx.doi.org/10.1109/MSR.2019.00066.

Rahman, M.R., Golagha, M., Pretschner, A., 2018. Poster: Pairika: A failure diagnosis benchmark for C++ programs. In: International Conference on Software Engineering. ICSE, pp. 204–205. http://dx.doi.org/10.1145/3183440.3195097.

Reis, S., Abreu, R., 2017. SECBENCH: A database of real security vulnerabilities. In: International Workshop on Secure Software Engineering in DevOps and Agile Development. SecSE, pp. 70–85.

Reiter, R., 1987. A theory of diagnosis from first principles. Artificial Intelligence 32 (1), 57–95. http://dx.doi.org/10.1016/0004-3702(87)90062-2.

Riganelli, O., Micucci, D., Mariani, L., 2019a. From source code to test cases: A comprehensive benchmark for resource leak detection in Android apps. Softw. - Pract. Exp. 49 (3), 540–548. http://dx.doi.org/10.1002/spe.2672.

Riganelli, O., Mobilio, M., Micucci, D., Mariani, L., 2019b. A benchmark of data loss bugs for android apps. In: International Working Conference on Mining Software Repositories. MSR, pp. 582–586. http://dx.doi.org/10.1109/MSR.2019.00087.

Saha, R.K., Lawall, J., Khurshid, S., Perry, D.E., 2014. On the effectiveness of information retrieval based bug localization for c programs. In: 30th International Conference on Software Maintenance and Evolution. ICSME, pp. 161–170. http://dx.doi.org/10.1109/ICSME.2014.38.

Saha, R.K., Lyu, Y., Lam, W., Yoshida, H., Prasad, M.R., 2018. Bugs.jar: A large-scale, diverse dataset of real-world Java bugs. In: 15th International Conference on Mining Software Repositories. MSR, pp. 10–13. http://dx.doi.org/10.1145/3196398.3196473.

Salehnamadi, N., Alshayban, A., Ahmed, I., Malek, S., 2020. A benchmark for event-race analysis in android apps. In: 18th International Conference on Mobile Systems, Applications, and Services. MobiSys, pp. 466–467. http://dx.doi.org/10.1145/3386901.3396602.

Sayyad Shirabad, J., Menzies, T., 2005. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, URL http://promise.site.uottawa.ca/SERepository.

Shiraishi, S., Mohan, V., Marimuthu, H., 2016. Test suites for benchmarks of static analysis tools. In: International Symposium on Software Reliability Engineering Workshops. ISSREW, pp. 12–15. http://dx.doi.org/10.1109/ISSREW.2015.7392027.

Slaby, J., Strejček, J., Trtík, M., 2013. ClabureDB: Classified bug-reports database tool for developers of program analysis tools. In: Lecture Notes in Computer Science. Vol. 7737 LNCS, pp. 268–274. http://dx.doi.org/10.1007/978-3-642-35873-9_17.

Soltani, M., Derakhshanfar, P., Devroey, X., van Deursen, A., 2020. A benchmark-based evaluation of search-based crash reproduction. Empir. Softw. Eng. 25 (1), 96–138. http://dx.doi.org/10.1007/s10664-019-09762-1.

Tan, S.H., Yi, J., Yulis, Mechtaev, S., Roychoudhury, A., 2017. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In: 39th International Conference on Software Engineering Companion. ICSE-C, pp. 180–182. http://dx.doi.org/10.1109/ICSE-C.2017.76.

Thomson, P., Donaldson, A.F., Betts, A., 2016. Concurrency testing using controlled schedulers: An empirical study. ACM Trans. Parallel Comput. 2 (4), 1–37. http://dx.doi.org/10.1145/2858651.

Tomassi, D.A., Dmeiri, N., Wang, Y., Bhowmick, A., Liu, Y.C., Devanbu, P.T., Vasilescu, B., Rubio-Gonzalez, C., 2019. BugSwarm: Mining and continuously growing a dataset of reproducible failures and fixes. In: International Conference on Software Engineering. ICSE, pp. 339–349. http://dx.doi.org/10.1109/ICSE.2019.00048.

Vieira, R., Da Silva, A., Rocha, L., Gomes, J.P., 2019. From reports to bug-fix commits: A 10 years dataset of bug-fixing activity from 55 Apache's Open source projects. In: 15th International Conference on Predictive Models and Data Analytics in Software Engineering. PROMISE, Vol. 19, pp. 80–89. http://dx.doi.org/10.1145/3345629.3345639.

Vorobyov, K., Kosmatov, N., Signoles, J., Jakobsson, A., 2017. Runtime detection of temporal memory errors. In: Lecture Notes in Computer Science. Vol. 10548, pp. 294–311. http://dx.doi.org/10.1007/978-3-319-67531-2_18.

Wendland, T., Sun, J., Mahmud, J., Hasan Mansur, S.M., Huang, S., Moran, K., Rubin, J., Fazzini, M., 2021. Andror2: A dataset of manually-reproduced bug reports for android apps. MSR, 18th International Conference on Mining Software Repositories MSR, 600–604.http://dx.doi.org/10.1109/MSR52588.2021.00082,

Wickert, A.K., Reif, M., Eichberg, M., Dodhy, A., Mezini, M., 2019. A dataset of parametric cryptographic misuses. In: International Working Conference on Mining Software Repositories. MSR, pp. 96–100. http://dx.doi.org/10.1109/MSR.2019.00023.

Widyasari, R., Sim, S.Q., Lok, C., Qi, H., Phan, J., Tay, Q., Tan, C., Wee, F., Tan, J.E., Yieh, Y., Goh, B., Thung, F., Kang, H.J., Hoang, T., Lo, D., Ouh, E.L., 2020. BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies. In: 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE, pp. 1556–1560. http://dx.doi.org/10.1145/3368089.3417943.

Wilkinson, M.D., Dumontier, M., Aalbersberg, I.J., Appleton, G., Axton, M., 2016. Comment: The FAIR Guiding Principles for scientific data management and stewardship. Sci. Data 3 (1), 1–9. http://dx.doi.org/10.1038/sdata.2016.18.

Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740. http://dx.doi.org/10.1109/TSE.2016.2521368.

Ye, X., Bunescu, R., Liu, C., 2014. Learning to rank relevant files for bug reports using domain knowledge. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 689–699. http://dx.doi.org/10.1145/2635868.2635874.

Yuan, T., Li, G., Lu, J., Liu, C., Li, L., Xue, J., 2021. GoBench: a benchmark suite of real-world go concurrency bugs. In: International Symposium on Code Generation and Optimization. CGO 2021, pp. 187–199. http://dx.doi.org/10.1109/CGO51591.2021.9370317.

Zakari, A., Lee, S.P., Abreu, R., Ahmed, B.H., Rasheed, R.A., 2020. Multiple fault localization of software programs: a systematic literature review. Inf. Softw. Technol. 124, 106312. http://dx.doi.org/10.1016/j.infsof.2020.106312.

Zhong, H., Mei, H., 2018. Mining repair model for exception-related bug. J. Syst. Softw. 141, 16–31. http://dx.doi.org/10.1016/j.jss.2018.03.046.

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., Ding, D., 2021. Fault analysis and debugging of microservice systems: industrial survey, benchmark system, and empirical study. IEEE Trans. Softw. Eng. 47 (2), 243–260. http://dx.doi.org/10.1109/TSE.2018.2887384.

Zhou, J., Zhang, H., Lo, D., 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: International Conference on Software Engineering. ICSE, pp. 14–24. http://dx.doi.org/10.1109/ICSE.2012.6227210.

Thomas Hirsch is a Ph.D. candidate at the Doctoral School of Computer Science at Graz, University of Technology, Austria. He received his B.Sc. and M.Sc. degrees in computer science from Graz University of Technology. His research interests include fault localization, and natural language processing and machine learning applied on artifacts of software development processes.

Dr. Birgit Hofer holds a master's degree in Software Engineering and Economics (2009) and a Ph.D. degree in Computer Science (2013) from the Graz University of Technology, Austria. Her main research interest comprises the automatic localization and correction of faults in imperative and object-oriented software and spreadsheets. In particular, she is interested in combinations of spectrum-based fault localization, model-based debugging techniques and genetic programming.