PreciseBugCollector: Extensible, Executable and Precise Bug-fix Collection

Solution for Challenge 8: Automating Precise Data Collection for Code Snippets with Bugs, Fixes, Locations, and Types

Ye He
Carnegie Mellon University
Pittsburgh, US
hey@cs.cmu.edu

Zimin Chen
KTH Royal Institute of Technology
Stockholm, Sweden
zimin@kth.se

Claire Le Goues
Carnegie Mellon University
Pittsburgh, US
clegoues@cs.cmu.edu

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00163

***Abstract*—Bug datasets are vital for enabling deep learning techniques to address software maintenance tasks related to bugs. However, existing bug datasets suffer from precise and scale limitations: they are either small-scale but precise with manual validation or large-scale but imprecise with simple commit message processing. In this paper, we introduce PreciseBugCollector, a precise, multi-language bug collection approach that overcomes these two limitations. PreciseBugCollector is based on two novel components: a) A bug tracker to map the codebase repositories with external bug repositories to trace bug type information, and b) A bug injector to generate project-specific bugs by injecting noise into the correct codebases and then executing them against their test suites to obtain test failure messages.**

We implement PreciseBugCollector against three sources: 1) A bug tracker that links to the national vulnerability data set (NVD) to collect general-wise vulnerabilities, 2) A bug tracker that links to OSS-Fuzz to collect general-wise bugs, and 3) A bug injector based on 16 injection rules to generate project-wise bugs. To date, PreciseBugCollector comprises 1 057 818 bugs extracted from 2 968 open-source projects. Of these, 12 602 bugs are sourced from bug repositories (NVD and OSS-Fuzz), while the remaining 1 045 216 project-specific bugs are generated by the bug injector. Considering the challenge objectives, we argue that a bug injection approach is highly valuable for the industrial setting, since project-specific bugs align with domain knowledge, share the same codebase, and adhere to the coding style employed in industrial projects.

***Index Terms*—Bug datasets, Program repair, Software testing and debugging**

I. Introduction

A precise bug dataset plays a crucial role in various software tasks, including bug detection [20, 21], fault localization [22, 23, 24, 25], pattern mining [26, 27, 28, 29], automated program repair for patch generation [30, 31, 32, 33, 34], and patch assessment [35, 36, 37, 38]. Current bug datasets can be categorized into two main types. The first type involves human curation, resulting in carefully processed but small-scale datasets, such as ManyBugs and IntroClass [2], Defects4J [1] and QuixBugs [39]. All of these datasets contain fewer than 1 000 bugs each. On the other hand, the second type comprises large-scale datasets mined from code repositories, such as ManySStubBs4J [13], which processes commit messages or issues with text processing. However, this approach tends to be imprecise in identifying bugs or classifying their types, relying on simple keyword comparison. Despite this loss of precision, crawling historical bug-fix commits from open-source projects remains a widely used approach to collect training bug-fix data in both academic [16, 40] and industry [41, 42], due to the ease with which it can construct large-scale datasets.

There are several key challenges to collecting a truly useful, precise bug dataset:

- **Problem 1 - Data quality and scalability:** Manual collection ensures the quality of the dataset but does not scale. Automatic collection with keyword matching on commits is scalable, but imprecise [43].
- **Problem 2 - Language diversity:** Most datasets consist only of a single language, limiting their usability and posing a threat to the external validity of software tasks.
- **Problem 3 - Lack of metadata:** The most common metadata provided by the dataset per bug is a commit ID or commit message. Important information such as the type of bug, the severity of the bug, and the date of discovery is missing.
- **Problem 4 - Lack of tests:** Only a handful of bug datasets provide test cases for the exposed bug. This challenges both bug reproduction, and validating bug fixes beyond the one provided by the developer.

In this paper, we introduce the PreciseBugCollector, a novel approach to tackle the imprecision problem in bug dataset creation. Our approach offers a curated collection of software defects, providing not only accurate bug type classification but also meta information to describe the bug, including original buggy code, fix code, precise location, error type, and available executable/reproducible test cases. We denote this information as $< bug, fix, loc, type, (test) >$. [1]

The PreciseBugCollector is based on two components: *bug tracker* for general bug collection, and *bug injection* for project-specific bug generation. The bug tracker focuses on extracting bug-fix commits, leveraging available code repositories (e.g., GitHub, Bitbucket, Gitlab, SVN) and external bug repositories (e.g., OSS-Fuzz, Jira, Apache Bug Report). The code repositories provide the exact source code

[1] Not all the bug-fix data contains executable test cases.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00163

TABLE I: Bug Dataset in the Literature.

Bug Dataset	Publish Venue	Publish Year	Languages	Type	Tests	Source	# Bugs
Defects4J [1]	ISSTA	2014	Java	✓	✓	commits+text processing	835
ManyBugs [2]	TSE	2015	C	✓	✓	commits+text processing	185
IntroClass [2]	TSE	2015	C	✓	✓	student assignments	998
CodeFlaw [3]	ICSE-Companion	2017	C	✓	✓	contest	3902
QuixBugs [4]	SPLASH-Companion	2017	Java/Python	✓	✓	contest	40
CodRep [5]	Arxiv	2018	Java	✗	✗	commits+text processing	58 069
PatchPaser [6]	ICSME	2018	Java	✓	✗	commits+text processing	16 450
Bears [7]	SANER	2019	Java	✓	✓	commits+CI	251
BugsJS [8]	ICST	2019	JavaScript	✓	✓	commits+text processing	453
Bugswarm [9]	ICSE	2019	Java/Python	✓	✓	commits+CI	3 091
Defexts [10]	ICSE-Companion	2019	Kotlin/Groovy	✓	✓	commits+text processing	526
Refactory [11]	ASE	2019	Python	✓	✓	student assignments	1783
BugsInPy [12]	FSE	2020	Python	✓	✓	commits+text processing	493
ManySStuBs4J [13]	MSR	2020	Java	✓	✗	commits+text processing	153 652
CODIT's dataset [14]	TSE	2020	Java	✗	✗	commits+CI	32 473
CodeBERT's dataset [15]	EMNLP	2020	Java/Python/Ruby/JavaScript/Go	✗	✗	commits+text processing	2 millions
CoCoNuT's dataset [16]	ISSTA	2020	Java/Python/C/JavaScript	✗	✗	commits+text processing	23 millions
Megadiff [17]	Arxiv	2021	Java	✗	✗	commits+text processing	663 029
Vul4J [18]	MSR	2022	Java	✓	✓	commits+text processing	79
FixJS [19]	MSR	2022	JavaScript	✗	✗	commits+text processing	300 000

changes made to address bugs, while the bug repositories offer detailed metadata about each bug. However, simply considering each repository in isolation is incomplete. Code repositories contain source code changes but lack clear bug metadata; bug repositories do not always include information about the exact source code changes. Therefore, we devise a method to merge these two sources, combining the advantages of both.

Bug injection automatically generates artificial unseen bugs, each of which is specified by an existing test suite with at least one failing test that exposes the bug. To initiate the bug generation process, we begin with code extracted from various projects, ensuring that all existing tests pass successfully. The code noising tool is then employed to deliberately introduce changes into this code, simulating bug injection. It is essential to acknowledge that not all the injected code is genuinely buggy as some changes might be benign; we use the original passing tests to identify meaningful injected faults. The test failure diagnosis obtained via this validation process precisely identifies the type of bug introduced.

Considering the challenge objective, we consider bugs from both the bug tracker and bug injection components to be beneficial for industrial settings. The bug tracker ensures that we have access to a diverse range of bug fixes from real-world projects, making the dataset more representative. Bug injection allows us to have project-specific bugs that are difficult to learn from a general dataset, and tailored for industrial settings.

The two bug tracking sources that we use for the bug tracker are: the National Vulnerability Dataset (NVD) [2] and OSS-Fuzz [3]. NVD is a repository of vulnerability management data represented using the Security Content Automation Protocol (SCAP); it includes databases of security checklist references, software flaws, misconfigurations, product names, and impact metrics. OSS-Fuzz, provided by Google, is an open source continuous fuzzing service. It uses random inputs (fuzzed data) to discover potential vulnerabilities, bugs, or crashes. OSS-Fuzz is specifically designed to identify security vulnerabilities and defects in open-source projects. We implement the bug injector with 16 single-statement injection rules from previous work [44].

In the end, to date, the PreciseBugCollector collected a total of 1 057 818 bugs from 2 968 open-source projects. We name this dataset `PreciseBugs`. Out of these, 12 602 bugs are contributed by the bug tracker (NVD and OSS-Fuzz), while 1 045 216 bugs are generated by the bug injector. To our knowledge, this is the largest executable bug dataset with precise bug information compared with related work.

To sum up, we make the following contributions:

- We introduce PreciseBugCollector for collecting a precise bug dataset to collect general-wise real-world bug-fix data and project-specific bug-fix data.
- We present a comprehensive dataset named `PreciseBugs`, which includes 1 057 818 bugs based on 2 968 open-source projects across more than six programming languages.

[2]https://nvd.nist.gov/vuln
[3]https://google.github.io/oss-fuzz

- We make our dataset readily accessible and openly available to the community through the link: https://github.com/SophieHYe/PreciseBugs.

II. Background

In this section, we give background on bug-fix datasets in the program repair literature. Table I presents a summary of 20 extensively utilized bug-fix datasets. These bug-fix dataset collection approaches (found in the seventh column of Table I) consists primarily of four commonly used methods. We explain them according to the collection approaches in the following.

Commits+Text Processing: Most datasets use commit mining and filtering against specific keywords (e.g., BugsJS [8]) to approximate bug-fix commits. Commonly used keywords include “fix” and “bug".

Commits+Continuous Integration (CI): Another approach uses continuous integration (CI) infrastructure for bug collection, as observed in projects such as Bears [7] and Bugswarm [9]. This method revolves around checking the CI status of two consecutive commits. When the first commit fails but the second one passes, it is considered a bug-fix pair. One of the advantages of this approach is gaining additional insights into the failing tests that triggered the CI build failure, and thus the bug.

Programming Contests: This category of research involves collecting bug-fix commits from contests or programming competitions, as demonstrated by projects such as CodeFlaw [3] and QuixBugs [4]. In programming contests, developers are presented with a problem description and test case specifications, which facilitates bug-fix collection encompassing both bug types and corresponding failing tests. The bug-fix commits are obtained by analyzing users' submission histories and identifying two submissions where the first submission fails and the second one is accepted. However, this approach is limited by the scarcity of available contests and the relatively low participation of developers.

Student Assignment Submissions: This group of approaches identifies and collects bug-fix commits from student submissions in introductory programming courses, exemplified by projects like IntroClass [2] and Refactor [11]. The process of collecting bug-fix commits in this context is similar to that for programming contests, analyzing student submission history. However, it is important to note that this approach is limited to collecting bug-fix commits from relatively small programs, typical of introductory programming assignments.

By considering different data collection approaches, we make the following implications.

Problem 1 - Data quality and scalability: It becomes evident that manual validation for bug-fix collection is not scalable. While manual validation ensures the quality of the collected bugs by including precise bug types and regression test cases, it requires extensive and laborious efforts in committing searches, data cleaning, and execution. As a result, manual bug collection is impractical for collecting large-scale datasets. For instance, even widely-used datasets like Defects4J [1] and ManyBugs [2] are limited to containing fewer than a thousand bugs each. On the other hand, most of the datasets in Table I use commits+text processing to crawl bugs. Although this approach is scalable and can collect millions of examples. Antoniol et al. have shown that text classification is not enough to classify the intent of a commit [43].

Problem 2 - Language diversity: There is a lack of language diversity in the bug dataset. Of the 20 widely used bug-fix datasets, the majority (15 out of 20) are focused solely on a single programming language, leaving only 5 datasets that cater to multiple languages. This language bias in bug datasets can limit the generalizability of research findings and may not adequately represent the diverse landscape of software development. Different programming languages have unique syntax, semantics, and coding practices, leading to varying types of bugs and bug-fix patterns. Therefore, incorporating multiple programming languages in bug-fix datasets is crucial to enable a more comprehensive understanding of bugs and their repairs across different language ecosystems.

Problem 3 - Lack of metadata: A significant issue with many bug datasets is the lack of sufficient metadata on the bugs. Crucial information, such as the date of bug discovery, bug type, commit author, and bug severity, is often missing. This absence of metadata poses challenges in analyzing the bug dataset beyond its initial intended use case. Proper metadata is essential for conducting in-depth research and understanding the characteristics of bugs, their patterns, and the context in which they occur.

Problem 4 - Lack of tests: Only a few bug datasets come with accompanying tests. Test cases are an indispensable means of specifying program correctness and validating bug-fixes. Having test cases allows researchers to verify the correctness of bug-fixes independently from the developer bug-fix present in the dataset. Moreover, test cases enable various dynamic analyses on the source code, such as fault localization-based on test coverage. Including test cases in bug datasets enhances the overall usability and utility of the dataset, enabling researchers to conduct more extensive and accurate evaluations of program repair techniques.

In our PreciseBug dataset, we strive to address these four challenges, with the ultimate goal of creating a comprehensive, precise, and large-scale bug dataset. The methodology used to construct this dataset is detailed in section III.

III. PreciseBugCollector

Figure 1 gives an overview of the PreciseBugCollector collection architecture. PreciseBugCollector consists of two novel components: a bug tracker for general bug collection and bug injection for project-specific bug collection.

The bug tracker creates real-world bug datasets by establishing connections between code repositories and external bug repositories. We use GitHub as the code repository

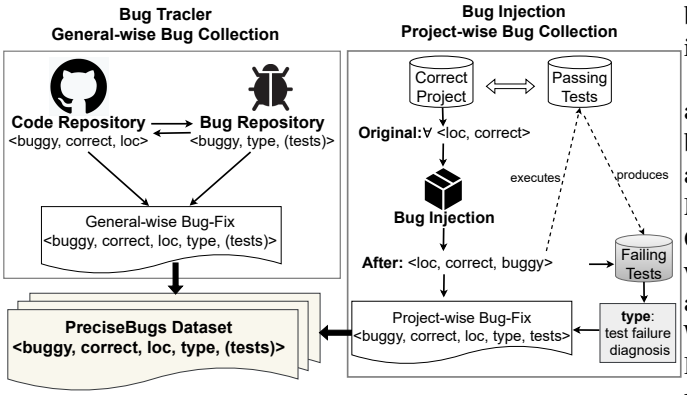


Fig. 1: An Overview of PreciseBugCollector.

instance, given its vast size, hosting over 28 million public repositories [45]. For bug repositories, many external bug repositories can be used, so long as they provide precise bug types and providing unique bug identifications that can be linked to code repositories (like an issue ID). Widely-used repositories that meet these criteria include the Common Vulnerabilities and Exposures (CVE) repository by the NVD, OSS-Fuzz, and Apache bug repositories.

The bug injection component relies on bug injection tools to generate a project specific bug-fix dataset. Bug injection generates new bugs by introducing noise into code that passes all tests in its accompanying test suite, and using those tests to evaluate the bugginess of the modified noisy code. The test suite not only plays a role in determining the bugginess of the noised code but also provides essential test failure diagnosis information, describing the bug.

A. Bug tracker: mapping code and bug repositories

PreciseBugCollector leverages external bug repositories to acquire accurate bug type information. Given a commit from a code repository, one can easily obtain the buggy code and fix code and their diffs to extract the code change locations, denoted as $< buggy, fix, loc >$. However, as discussed in section II, it is challenging to summarize bug types from fix commit messages. Our work introduces a novel approach to establishing mappings between code and bug repositories, revealing the actual bug type, and producing $< buggy, fix, loc, type >$ information. Moreover, there exist bug repositories that also contain failing tests that expose the bugs; this is crucial for buggy code reproduction and execution [46]. Consequently, the bug tracker may obtain the bug-fix dataset in the format of $< buggy, fix, loc, type, (tests) >$, where the test information is optional depending on the external bug repositories.

We use two external bug repositories: NVD and OSS-Fuzz, both of which assign unique identifiers to vulnerabilities and bugs (a CVE in the NVD, and unique identifiers in OSS-Fuzz). Both NVD and OSS-Fuzz are large, with 220,748 CVE records in the NVD, and 28,000 bugs in OSS-Fuzz. Notably, both repositories support different programming languages, including C/C++, Rust, Go, Python, and Java/JVM.

CVE collection-based on NVD. NVD serves as an invaluable resource for mining verified vulnerabilities reported by humans. Many important and well-known vulnerabilities are reported there, including HeartBleed (CVE-2014-0160[4]), Meltdown, (CVE-2017-5754[5]), Spectr (CVE-2017-5753 and CVE-2017-5715 [6]), and Log4Shell (CVE-2021-44228[7]). Each vulnerability reported on NVD includes essential the vulnerability description, its type identified by the CWE (Common Weakness Enumeration) ID, a severity level, and references. References often contain patches to the vulnerabilities, providing source code before and after the vulnerability fix. Our systematic procedure for collecting vulnerabilities from NVD is as follows:

1) We use the NVD API [8] to download the complete vulnerability metadata, a total of 217 403 vulnerabilities.
2) We filter vulnerabilities by identifying external links that lead to GitHub commits and with the *Patch* tag. This narrows the vulnerabilities to 9,759.
3) We extract the fixed source code from the corresponding GitHub commits. Extraction may fail due to name changes or repositories having been removed.

After the last step, we are left with a dataset of 8487 vulnerabilities, each accompanied by its metadata and the corresponding vulnerability fix. The decision to extract vulnerability fixes exclusively from GitHub commits is based on the fact that patches reported to NVD come in various formats, making them challenging to parse. These formats may include links to patches on patched product download sites, blog websites, and bug discussions on various platforms, among others. By focusing on GitHub commits specifically tagged as *Patch*, we can mitigate noise and ensure a more consistent and reliable dataset. Additionally, using GitHub allows us to easily retrieve the project name from the repository name, further enhancing accuracy.

Bugs collection-based on OSS-Fuzz. Fuzz testing [47, 48, 49] is a widely acknowledged technique for detecting programming errors, especially critical issues like buffer overflows. In contrast to the vulnerabilities collected in NVD, each bug found by OSS-Fuzz provides explicit failing tests that expose the bugs. Our systematic procedure of collecting bugs from OSS-Fuzz is as follows:

1) We conduct an exploration of the OSS-Fuzz repository, and retrieve open-source projects that have registered to use the OSS-Fuzz infrastructure, resulting in a total of 28348 bugs from 557 open source projects.
2) To obtain the bug-fixes for each project, we first filter GitHub commit messages and search for OSS-Fuzz identifiers. Then, we use these identifiers to query the

[4]https://nvd.nist.gov/vuln/detail/cve-2014-0160
[5]https://nvd.nist.gov/vuln/detail/CVE-2017-5754
[6]https://nvd.nist.gov/vuln/detail/cve-2017-5753 and https://nvd.nist.gov/vuln/detail/cve-2017-5715
[7]https://nvd.nist.gov/vuln/detail/cve-2021-44228
[8]https://nvd.nist.gov/developers/vulnerabilities

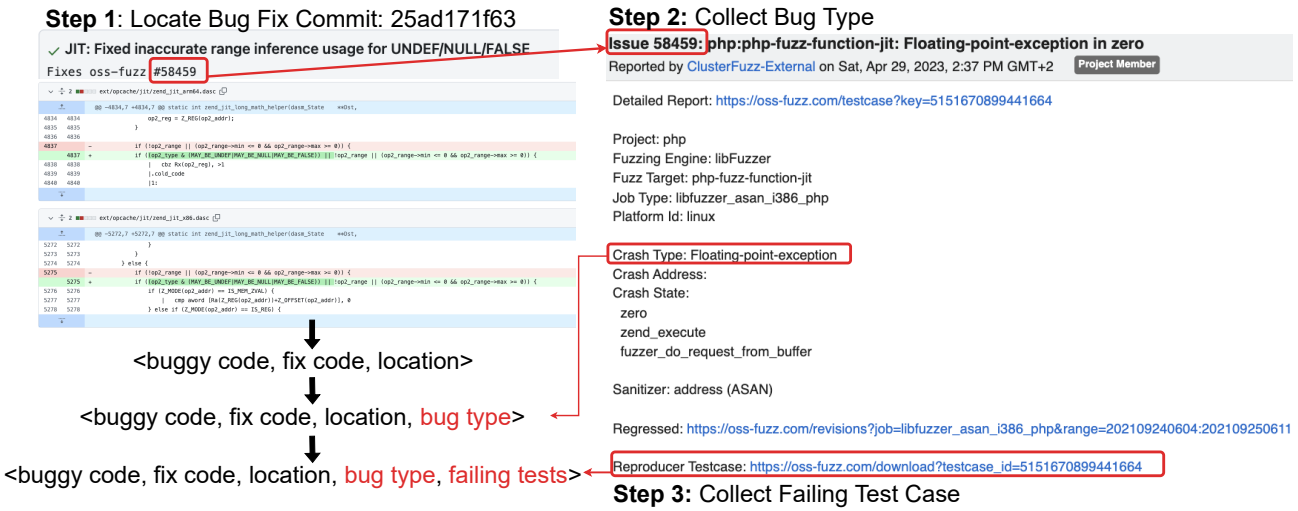


Fig. 2: Running example of the bug tracker component with the external bug repository OSS-Fuzz.

TABLE II: Injection rules for bug creation.

Bug Injection Rules	Description
Rule-1	modify declaring type
Rule-2	modify operator
Rule-3	modify literal
Rule-4	modify constructor
Rule-5	swap argument
Rule-6	modify boolean expression
Rule-7	modify invocation
Rule-8	compound modification
Rule-9	replace similar statement
Rule-10	move statement
Rule-11	insert statement
Rule-12	wrap statement
Rule-13	insert block
Rule-14	delete block
Rule-15	unwrap block
Rule-16	remove block

OSS-Fuzz API, which retrieves crash types per bug. After this step, we are left with a dataset of 4025 bugs collected from six programming languages.

3) (optional) We gather the failing tests that expose the bugs from OSS-Fuzz. However, not all projects support the download of failing test cases due to authorization issues.
4) (optional) We create reproducible bugs by executing collected failing tests against the fixed code (i.e., the patched program) to check if they pass. We execute the same tests against the buggy program to verify that it indeed results in test failures.

Running Example of bug tracker with OSS-Fuzz Figure 2 illustrates the process of collecting bugs from the PHP project using the external bug repository OSS-Fuzz. In the initial step, we browse the PHP GitHub commits to collect the OSS-Fuzz issue identifier. Next, we establish a link between the GitHub code repository and the OSS-Fuzz bug repository to identify the crash type (e.g., floating point exception) and obtain the corresponding failing test cases.

Algorithm 1 Injection-based Bug Creation

```
Input: a correct program CorrectCodebase, Injection rules rules, test suite tests
PreciseBugs ← ∅
for loc, fix in CorrectCodebase do
    type ← apply(rules, loc, fix)
    error, type ← compile(buggy)
    if !error then
        type ← execute(buggy, tests)
    end if
    if type then
        PreciseBugs ← < buggy, fix, loc, type, tests >
    end if
end for
return PreciseBugs
```

Finally, we assess the executability and reproducibility of the collected bugs by separately executing the fix code and buggy code against the test cases.

B. Bug Injection: Creating Project-specific Bugs

Bug injection involves deliberately introducing faults into a stable codebase and is widely used to test software reliability and dependability [50, 51]. Bug injection is heavily used in mutation testing to test the strength of existing test suites [52], as well as to test bug finding tools [53, 54, 55]. Recent work in automatic program repair with machine learning also uses bug injection to create training data to train deep learning program repair models [56, 57].

We adopt this approach to generate project-specific bugs by intentionally corrupting the correct codebase and execute injected bugs against existing test suites to collect precise bug type from compiler and test failure diagnosis. Project-specific bugs hold significant value in the industrial context, particularly for companies working on projects that require domain-specific knowledge and adhere to unique coding styles. These specialized bugs are challenging to learn from general-purpose datasets.

TABLE III: An overview of bug-fix data collected by PreciseBugCollector.

Languages	NVD-based Vulnerabilities		OSS-Fuzz-based Bugs		Injection-based Bugs			Summary	
	#projects	#behavior bugs	#projects	#behavior bugs	#projects	#compilation bugs	#behavior bugs	#projects	#bugs
C	419	2559	12	1604	1	1735	637	424	6535
C++	117	617	53	2439	-	-	-	151	3056
Python	291	468	3	3	3	4372	753	292	1224
Java	199	309	10	48	17	631015	409230	213	1040602
Go	189	323	9	15	-	-	-	195	338
Rust	28	38	5	6	-	-	-	30	44
PHP	597	1846	-	-	-	-	-	597	1846
Others	1065	4173	-	-	-	-	-	1065	4173
All	2905	8487	79	4115	21	637122	410620	2968	1057818

Input and Output. Algorithm 1 illustrates the bug injection process. Bug injection takes a "correct" project codebase and its corresponding tests as input, where "correct" means that the considered project passes all test cases. For every statement present in the project codebase, bug injection introduces noise to that statement with the aim of changing the program's execution behavior (see line 3 in Algorithm 1). The original test suite is then utilized to assess whether the injected code's behavior has changed by yielding at least one failing test.

We first attempt to compile noised code. If any compilation error message is produced, this injected noisy code is deemed to cause a compilation error (line 5). Otherwise, the noisy code is further executed against the test suite to determine whether it causes a behavioral bug (line 6 and line 7). If any tests fail, a test failure diagnosis is performed (line 9 and line 10). Otherwise, it is discarded.

Injection Rules. We employ abstract syntax tree (AST)-based 16 bug injection rules from prior work [44]. Table II details the rules, which cover different granularities of code transformation, such as type, operation, literal, valuables, expression, statement, and block. Note that these injection rules provide more diverse code transformation rules than existing works that mostly focus on operators and variables [58, 59, 60, 56].

C. Comparison of Collected Bugs

TABLE IV: Comparisons of different sources considered by PreciseBugCollector.

Sources	Compilation Bug	Behavior Bug	Fix	Location	Type	Failing Tests	
						New	Existing
CVE collection-based on NVD	✗	✓	✓	✓	✓	✗	✗
Bugs collection-based on OSS-Fuzz	✗	✓	✓	✓	✓	✓	✗
Bugs based on Injection	✓	✓	✓	✓	✓	✗	✓

Table IV provides a comparison of the three considered bug collection approaches. All three approaches gather bug information in the format of $< bug, fix, location, type >$. Both bug tracker with OSS-Fuzz and bug injection approaches can collect reproducible bugs along with their corresponding failing tests. However, the failing tests from OSS-Fuzz include additional new test cases that go beyond the existing test suite. Failing tests from the bug injection are the existing test cases in the test suite that were originally passed before the bug injection took place.

Notably, Bug injection is the only one that produces compilation bugs, where the error type and error message directly come from the compilers. Therefore, bug injection is a powerful approach to constructing both compilation and behavior bugs.

IV. Evaluation

To evaluate the PreciseBugCollector, we propose the three following research questions:

- **RQ1:** What is the effectiveness of the PreciseBugCollector to construct bug-fix commits?
- **RQ2:** What is the distribution of bug types collected by PreciseBugCollector?
- **RQ3:** What is the distribution of the time period covered by bugs collected by PreciseBugCollector?

A. RQ1: Number of Bugs

Methodology for RQ1. We summarize the number of bugs that PreciseBugCollector has collected. Specifically, we analyze 1) the number of major programming languages, 2) the number of projects, and 3) the number of bugs that PreciseBugCollector is able to cover.

Result for RQ1. Table III gives an overview of the bug-fix data obtained by PreciseBugCollector. PreciseBugCollector collected 8487 CVEs from 2905 projects-based on NVD dataset, 4115 bugs from 79 projects-based on OSS-Fuzz bug repository, and respectively 637122 compilation bugs and 410620 behavior bugs from 21 open source projects.

In total, PreciseBugCollector obtained 1057818 bugs from 2968 projects and more than six programming languages. To our knowledge, this is the largest bug-fix collection with precise bug type and execution information to date. All three are able to collect thousands of bugs, and each of them individually collects more than the

largest prior dataset shown in Table I. Listing 1 gives three examples of collected bug-fix data from each approach. Now, we discuss the implications of these results.

CVEs from NVD cover many projects. We have collected CVEs from 2905 open source projects by tracking to NVD, many more than the 79 and 21 projects covered by the other two approaches. This discrepancy is primarily due to the NVD's establishment as one of the earliest and widely utilized bug repositories, dating back to 2004 (whereas OSS-Fuzz emerged in 2016).

Bug injection generates the most bugs. The bug injection approach, while covering the fewest projects, proves to be highly effective in generating a large number of bugs. This is primarily attributed to the project-specific nature of the bug injection technique, which aims to traverse every statement in the program. However, it is worth noting that the execution cost for the bug injection is relatively high, which is why we restricted our experimentation to only 21 projects from three languages.

Bug injection generates various numbers of bugs for different projects despite the same code corruption rules. This is because the number of bugs generated relies on the number of lines of code (LOC) and test suite size and strength. In our experiment, the considered 17 Java projects (e.g., Closure, JacksonDatabind, etc.) are comprise 25,000 LOC and more than 2000 test cases, meaning that more bugs are generated compared with projects in C and Python.

> Answer to RQ1: PreciseBugCollector gathered a total of 1,057,818 precise bugs from 2,968 open source projects.

B. RQ2: Types of Bugs

Methodology for RQ2. In this RQ, we investigate the type of collected bugs. Specifically, we look at unique bug types that each component brings to PreciseBugCollector.

Result for RQ2. Figure 3 gives the top-10 bug types from each component of PreciseBugCollector. The distribution of bug types varies by component. This is due to the differences among three sources: CVE types are labeled by humans, OSS-Fuzz types are labeled by fuzzing tests, and injection-based bug types are extracted by test failure diagnosis.

CVE-based Types. Collected CVE bug-fix commits contain CWE IDs, a community-developed list of software and hardware weakness types. The most common CVE type in PreciseBugCollector is CWE-79[9]: `Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')`, which accounts for 17.1% (1455/8487) of the total CVE types. The remaining top-2 bug types are out-of-the-bounds read and out-of-the-bounds write.

[9]https://cwe.mitre.org/data/definitions/79.html

```
366 - if(length<0xffffffff && length+index < size())
366 + if(length < size() - index)
CVE ID: CVE-2012-1584
Type: CWE-189 Numeric Errors
Loc: Before: 366, After: 366
```

(a) An NVD-based vulnerability collected from project TagLib.

```
2859 - buf[i + 0] = -(USE_FIXED + 1)*buf[i + 0];
2859 - buf[i + 1] = (USE_FIXED + 1)*buf[i + 1];
2859 + buf[i + 0] = -(int)(USE_FIXED+1U)*buf[i + 0];
2860 + buf[i + 1] = (int)(USE_FIXED + 1U)*buf[i + 1];

OSS-Fuzz Issue ID: 57986
Type: Integer-overflow
Loc: Before: (2859,2860), After: (2859,2860)
```

(b) An OSS-Fuzz-based bug collected from project FFmpeg.

```
4493 - Collection c = r.getAnnotations();
4493 + if (r != null) {
4494 + Collection c = r.getAnnotations();
4495 + }
Failing Test: LogAxisTests:testXYAutoRange1
Type: NullPointerException
Loc: Before: 4493, After: (4493,4495)
```

(c) An injection-based bug collected from project JFreeChart.

Listing 1: Examples of collected bugs from three sources.

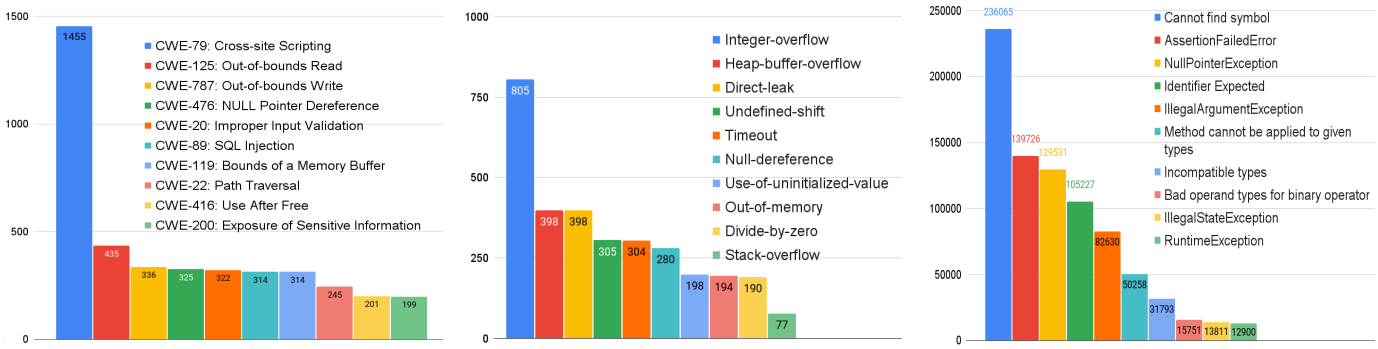
OSS-Fuzz-based Types. For each collected OSS-Fuzz-based bug-fix commit, a unique issue ID leads to a certain crash type produced by fuzzing tests. The dominant bug type from OSS-Fuzz is integer-overflow, which accounts for 19.6 (805/4115) of the total OSS-Fuzz-based collected bugs. The remaining top-2 bug types are heap-buffer-overflow and direct-leak, which are not included in the top 10 of another two sources.

Injection-based Types. Each collected injection-based bug-fix commit is accompanied by a precise error message. Either this message comes from a compiler or a test suite execution result. The most frequent compilation error type from injection-based bugs is "cannot find symbols", while the top-2 behavior bug types are assertion failures with concrete error messages and null pointer exceptions.

> Answer to RQ2: PreciseBugCollector contains a bug-fix dataset with diverse and precise bug types, and each source contributes unique types of bugs.

C. RQ3: Time Period and Data Leakage

Methodology for RQ3. Large language models (LLMs) are evaluated on many existing bug datasets [61, 62]. Yet, these LLMs are also trained on data available prior to 2022 on the internet, which poses a threat of data leakage [63]. We analyze the PreciseBugCollector bug-fix dataset by year and particularly look at the bug-fixes available as of 2022 and 2023.



(a) NVD-based CVE Type (b) OSS-Fuzz-based Bug Type (c) Injection-based Bug Type

Fig. 3: Top-10 bug types in three considered sources by PreciseBugCollector.

TABLE V: Year distribution of collected bug-fix dataset. The rows is valuable to avoid data leakage evaluation for LLM models.

Year	NVD-based	OSS-Fuzz-based	Injected-based	Summary
2023	717	161	1 047 724	1 048 620
2022	1941	435	-	2376
2021	1399	521	-	1920
2020	865	413	-	1278
2019	662	495	-	1157
2018	644	1033	-	1677
2017	723	1054	-	1777
2016	520	3	-	523
≤ 2015	1016	-	-	1016

Result for RQ3. Table V summarizes the bug-fixes mined by PreciseBugCollector over timesince 2015. CVEs have a broader time distribution, as it is one of the earliest and widely used vulnerability datasets. OSS-Fuzz began in 2016, and the majority of its bugs are repaired since 2017. Injection-based bugs by their nature are not available until the bug is injected, therefore, all the bugs are unseen by LLMs. Compared to related work, PreciseBugCollector provides an up-to-date bug-fix dataset. The rows highlighted in Table V are useful to avoid data leakage in state-of-the-art software engineering maintenance tasks (bug detection, fault localization, program repair, etc).

> Answer to RQ3: PreciseBugCollector contains valuable bug-fixes mined from 2022 and 2023, unseen by modern LLMs. These are valuable datasets for both training and testing to avoid data leakage.

V. Related Work

A. Bug-fix Dataset

Now we present all the related work that collected a bug dataset.

Defects4J is a collection of 835 real-world Java programs with known software defects [1]. The bugs are curated from well known open source projects such as Apache Lang and Mockito.

ManyBugs and IntroClass datasets are both bug datasets focusing on C programs [2]. ManyBugs contains 185 defects collected from version control repositories of 9 projects. IntroClass consists of 998 defects from student written assignments for an introductory C programming course. All defects are reproducible with corresponding test cases.

Codeflaws is another bug dataset focusing on C programs [3]. It aims to fix the diversity and size problem of ManyBugs and IntroClass dataset. It crawls Codefroces for rejected submissions and finds another accepted submission by the same user for the same programming problem.

QuixBugs is a multi-lingual bug dataset, with 40 programs translated to Python and Java [4]. All bugs are in one line of code, and each bug has passing and failing test cases.

CodRep is a machine learning on source code competition [64]. The goal of this competition is to find the line number in a file to insert a given code line. All data are extracted from real-world one line commits.

Liu et al. collected 16450 bugs to do a systematic and fine-grained study to gain insights into tuning automatic program repair tools [6]. The bugs are collected from 6 Java projects using keyword matching and bug linking.

Bears is an extensible bug benchmark for program repair tools [7]. It collects bugs in a unique way by looking at the commit building state from continuous integration to find potential bug-fix commits.

BugsJS is a JavaScript bug benchmark that can be used to facilitate the research of fault localization [8]. All bugs are extracted from the issue tracking system of the selected popular repositories from GitHub.

BugSwarm, similar to Bears, is an extensible bug benchmark by mining bugs from continuous integration [9]. The collected bugs are in two languages, Java and Python. All bugs are reproducible within the packaged container.

Defexts is a bug benchmark dataset aiming to collect bugs for the less popular JVM programming languages, namely Kotlin and Groovy [10]. The bugs are collected

through finding a commit message with keywords related to bugs.

Hu et al. collected 1783 bugs from 5 Python programming assignments in a Python introductory course [11]. This dataset is then used to evaluate their tool for generating patches for student programs.

BugsInPy is a Python bug dataset, aiming to create a bug dataset similar to Defects4J, but for Python. 493 bugs are collected manually identify bug-fix commits, reproduce bugs with failing test cases and isolate the bug-fix change from other unrelated changes.

ManySStuBs4J is a collection of 153652 single statement bugs from 1000 popular Java projects [13]. The bugs are curated by classifying bug-fix commits using keywords and filtering our clear refactors. The remaining bugs are then classified into 16 bug patterns.

Chakraborty et al. gathered 32473 patches to train and evaluate their tree-based neural model for code editing [14]. The dataset is collected from 48 Java projects that used TravisTorrent, have at least 50 commits, and 10 watchers on GitHub.

Lutellier et al. collected a multi-lingual bug dataset to train an ensemble model for program repair. The bugs are collected by keyword matching on commits. In total, they collected 23M samples for Java, Python, C, and JavaScript.

Megadiff is a collection of possible bug-fix commits for Java. Megadiff is curated by keyword matching on commit messages and filter commits that do not change Java or changed more than 40 lines of code.

Vul4J is a dataset of reproducible Java vulnerabilities [18]. All vulnerabilities from filtered from the Project KB knowledge base. The filter criteria include that it is Java, contains Java test suite, and is reproducible and isolated.

FixJS is a dataset of bug-fix commits for the JavaScript language [19]. The dataset is constructed by matching keywords on GitHub commits and filtering commits that did not change JavaScript files.

PreciseBugCollector stands apart from all the previously mentioned dataset construction approaches due to the following distinctions. Firstly, while the aforementioned datasets focus solely on codebases, PreciseBugCollector takes a different approach by connecting codebases to external bug repositories, allowing it to track precise bug information. Secondly, PreciseBugCollector includes project-specific bugs generated by bug injector, which is not considered by any aforementioned datasets.

B. Bug Injection

Bug injection is most commonly employed in mutation testing to assess if an existing test suite captures a small source code change and to evaluate source code analyzing tools. Additionally, it has been utilized to generate training data for machine learning models focused on source code analysis. In this section, we will explore related works that involve creating artificial bugs.

Mutation testing has a lengthy history of altering source code to introduce bugs and verifying if the existing test suite detects them. For further information on mutation testing, we direct readers to two surveys [52, 65] as there are numerous studies on this subject. The main distinction between mutation testing and PreciseBugCollector bug injection lies in their focus. While mutation testing aims to identify bugs that do not cause test failures, PreciseBugCollector bug injection is employed to construct a bug dataset.

The Juliet test suite comprises over 81,000 synthetic C/C++ and Java programs with 181 categories of inserted faults [66]. Developed by the National Security Agency's Center for Assured Software (CAS), this test suite aims to evaluate the effectiveness of software assurance tools. However, the process by which the Juliet test suite creates these synthetic bugs remains unclear.

Shiraishi et al. created 638 C/C++ programs with intentionally injected faults [67]. These injected faults are categorized into 9 defect types. The dataset is used to evaluate static analysis tools. However, the method employed by Shiraishi et al. to create these bugs remains unclear.

LAVA is an automated system capable of injecting faults into large open-source C programs [53]. The bugs are triggered by pre-defined inputs, whereas normal inputs are unlikely to trigger them. This system is used to evaluate bug-finding tools. LAVA differs from PreciseBugCollector in that it uses execution traces to insert bugs and only incorporates out-of-bound read/write bugs. On the other hand, PreciseBugCollector utilizes more rewrite results to create a wider variety of bug types.

EvilCoder is another automated tool that injects vulnerable code [54]. It identifies potentially vulnerable code locations and modifies them to become actually vulnerable by utilizing data flow analysis. This tool facilitates the systematic evaluation of bug-finding tools. EvilCoder varies from PreciseBugCollector in that the bugs it inserts are characterized by improperly secured data flow. In contrast, PreciseBugCollector employs more rewrite results to create a broader range of bug types.

Apocalypse is an automated bug injection tool based on symbolic execution [55]. It aims to inject realistic bugs, distinguishing itself from LAVA and EvilCoder. Apocalypse demonstrated its ability to generate diverse, difficult, and highly realistic bugs according to various metrics. The key difference between PreciseBugCollector and Apocalypse lies in the bug creation process. Apocalypse utilizes symbolic execution, whereas PreciseBugCollector relies on rewrite rules.

SemSeed introduced a novel method for automatically inserting realistic bugs [60]. It employs machine learning techniques to learn patterns and characteristics from collected bug-fixing commits. The learned features are then used to seed new bugs into existing programs. The main distinction between PreciseBugCollector and SemSeed is in their bug creation approach. SemSeed relies on machine learning, while PreciseBugCollector uses rewrite rules.

BugLab is an approach that utilizes self-supervised training for bug detection and repairs [56]. They create rewrite rules to artificially insert bugs into programs, which serve as training data. The rewrite rules used by BugLab include variable swap, argument swap, operator swap, and literal swap. The key difference between PreciseBugCollector and BugLab is the number of rewrite rules used. BugLab does not execute the injected bugs, therefore, no type and error message are obtained in their approach.

DrRepair is a graph-based program repair approach that also uses self-supervised training to generate artificial training data [57]. It randomly deletes, inserts, or replaces operators, punctuation, identifier, and keywords to create bugs and record the compiler message. DrRepair is different from PreciseBugCollector in that it focus on compiler errors instead of bug that are exposed by the test suite.

SelfAPR is another example of using a bug injection model to create training data for machine learning models on program repair [44]. They use 16 rewrite rules to create artificial bugs for training data, ensuring that all bugs are validated against the compiler and test suite. PreciseBugCollector's rewrite rules are essentially derived from SelfAPR, however, we have re-formatted them to construct a bug dataset that includes important bug location and error message information. On the contrary, SelfAPR largely ignores these aspects as its goal is to focus on code change pattern learning.

VI. Conclusion

We introduce a comprehensive and extensive bug-fix collection approach named *PreciseBugCollector*, which encompasses three distinct sources for bug acquisition: CVEs from NVD, bugs from OSS-Fuzz, and injection-based bugs. This endeavor has resulted in a total of 1 057 818 bugs across 2 968 open source projects. Notably, this dataset stands out as the largest bug-fix collection to date, encompassing precise bug types and accompanying message information, making it valuable for future software maintenance tasks, such as bug detection, fault localization, and automated program repair.

PreciseBugCollector offers solutions to create two types of bug-fix datasets: a general-wise dataset composed of real-world bug fixes made by developers, and a project-specific dataset that incorporates domain knowledge and aligns with the code style of the project. We believe that addressing the industry challenge of imprecise bug-fix datasets requires both components to build deep learning models that can learn broadly and in-depth. In industry settings, where private and sensitive projects exist, having a project-specific bug-fix dataset becomes essential to enable training and learning from the same codebase while ensuring data security and privacy. Furthermore, the flexibility of both components is significant as they are extensible, allowing for future expansion and adaptation.

VII. Acknowledgements

We thank the anonymous reviewers for the insightful feedback. This work was partially supported by The Wallenberg Foundation and WASP Postdoctoral Scholarship Program - KAW 2022.0368, and partially supported by the TrustFull project financed by the Swedish Foundation for Strategic Research.

References

[1] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[2] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 12, pp. 1236–1256, December 2015.

[3] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 180–182.

[4] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 55–56. [Online]. Available: https://doi.org/10.1145/3135932.3135941

[5] Z. Chen and M. Monperrus, "The codrep machine learning on source code competition," arXiv, Tech. Rep. 1807.03200, 2018.

[6] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon, "A closer look at real-world patches," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 275–286.

[7] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*, 2019.

[8] B. Vancsics, P. Gyimesi, A. Stocco, D. Mazinanian, A. Beszedes, R. Ferenc, and A. Mesbah, "Poster: Supporting javascript experimentation with bugsjs," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 375–378.

[9] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes," in *ICSE*. IEEE / ACM, 2019, pp. 339–349.

[10] S. Benton, A. Ghanbari, and L. Zhang, "Defexts: A curated dataset of reproducible real-world bugs for modern jvm languages," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 47–50. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00035

[11] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury, "Re-factoring based program repair applied to programming assignments," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2019, pp. 388–398.

[12] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1556–1560. [Online]. Available: https://doi.org/10.1145/3368089.3417943

[13] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset." New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3379597.3387491

[14] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, 2020.

[15] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[16] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," ser. ISSTA 2020, 2020.

[17] M. Monperrus, M. Martinez, H. Ye, F. Madeiral, T. Durieux, and Z. Yu, "Megadiff: A Dataset of 600k Java Source Code Changes Categorized by Diff Size," Arxiv, Tech. Rep. 2108.04631, 2021. [Online]. Available: http://arxiv.org/pdf/2108.04631

[18] Q.-C. Bui, R. Scandariato, and N. E. D. Ferreyra, "Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 464–468.

[19] V. Csuvik and L. Vidacs, "Fixjs: A dataset of bug-fixing javascript commits," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 712–716.

[20] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *2009 Ninth International Conference on Quality Software*, 2009, pp. 305–314.

[21] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, vol. 61, pp. 93–106, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584914002602

[22] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 82–101, 2022.

[23] A. Riboira and R. Abreu, "The gzoltar project: A graphical debugger interface," ser. TAIC PART'10. Berlin, Heidelberg: Springer-Verlag, 2010.

[24] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007, pp. 89–98.

[25] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "Hydra: Massively compositional model for cross-project defect prediction," *IEEE Transactions on Software Engineering*, vol. 42, no. 10, pp. 977–998, 2016.

[26] K. Liu, J. Zhang, L. Li, A. Koyuncu, D. Kim, C. Ge, Z. Liu, J. Klein, and T. F. Bissyandé, "Reliable fix patterns inferred from static checkers for automated program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, may 2023. [Online]. Available: https://doi.org/10.1145/3579637

[27] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016.

[28] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, 2021.

[29] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Softw. Engg.*, vol. 25, no. 3, p. 1980–2024, may 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09780-z

[30] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 54–72, 2012.

[31] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "Ifixr: Bug report driven program repair," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 314–325. [Online]. Available: https://doi.org/10.1145/3338906.3338935

[32] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering*, 2021.

[33] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018.

[34] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 31–42.

[35] H. Tian, Y. Li, W. Pian, A. K. Kaboré, K. Liu, A. Habib, J. Klein, and T. F. Bissyandé, "Predicting patch correctness based on the similarity of failing test cases," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, aug 2022. [Online]. Available: https://doi.org/10.1145/3511096

[36] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering*, 2021.

[37] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2020, pp. 981–992.

[38] H. Tian, Y. Li, W. Pian, A. K. Kaboré, K. Liu, J. Klein, and T. F. Bissyande, "Checking patch behaviour against test specification," *ACM Trans. Softw. Eng. Methodol.*, 2022.

[39] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," *Journal of Systems and Software*, vol. 171, p. 110825, 2021.

[40] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[41] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360585

[42] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, "Generating bug-fixes using pretrained transformers," in *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–8. [Online]. Available: https://doi.org/10.1145/3460945.3464951

[43] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? a text-based approach to classify change requests," in *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, ser. CASCON '08. New York, NY, USA: Association for Computing Machinery, 2008. [Online]. Available: https://doi.org/10.1145/1463788.1463819

[44] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *37th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 2022.

[45] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 384–387. [Online]. Available: https://doi.org/10.1145/2597073.2597126

[46] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 474–484.

[47] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang, "Fuzz testing in practice: Obstacles and solutions," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 562–566.

[48] H. Shi, R. Wang, Y. Fu, M. Wang, X. Shi, X. Jiao, H. Song, Y. Jiang, and J. Sun, "Industry practice of coverage-guided enterprise linux kernel fuzzing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 986–995. [Online]. Available: https://doi.org/10.1145/3338906.3340460

[49] J. Gao, Y. Xu, Y. Jiang, Z. Liu, W. Chang, X. Jiao, and J. Sun, "Em-fuzz: Augmented firmware fuzzing via memory checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and*

Systems, vol. 39, no. 11, pp. 3420–3432, 2020.
[50] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.
[51] N. K. Salih, D. Satyanarayana, A. S. Alkalbani, and R. Gopal, "A survey on software/hardware fault injection tools and techniques," in *2022 IEEE Symposium on Industrial Electronics & Applications (ISIEA)*, 2022, pp. 1–7.
[52] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.
[53] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 110–121.
[54] J. Pewny and T. Holz, "Evilcoder: automated bug insertion," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 214–225.
[55] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 224–234.
[56] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," in *NeurIPS*, 2021.
[57] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," in *International Conference on Machine Learning (ICML)*, 2020.
[58] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.
[59] R. Just, B. Kurtz, and P. Ammann, "Inferring mutant utility from program context," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. ACM, 2017, p. 284–294.
[60] J. Patra and M. Pradel, "Semantic bug seeding: a learning-based approach for creating realistic bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 906–918.
[61] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 959–971. [Online]. Available: https://doi.org/10.1145/3540250.3549101
[62] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE 2023. Association for Computing Machinery, 2023.
[63] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, "Is chatgpt the ultimate programming assistant – how far is it?" 2023.
[64] Z. Chen and M. Monperrus, "The codrep machine learning on source code competition," 2018.
[65] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.
[66] T. Boland and P. E. Black, "Juliet 1. 1 c/c++ and java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012.
[67] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," in *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2015, pp. 12–15.