Mining Bug Repositories for Multi-Fault Programs

Dylan Callaghan
Stellenbosch University
Stellenbosch, South Africa
21831599@sun.ac.za

Bernd Fischer
Stellenbosch University
Stellenbosch, South Africa
bfischer@sun.ac.za

ABSTRACT

Datasets such as Defects4J and BugsInPy that contain bugs from real-world software projects are necessary for a realistic evaluation of automated debugging tools. However these datasets largely identify only a single bug in each entry, while real-world software projects (including those used in Defects4J and BugsInPy) typically contain multiple bugs at the same time. We lift this limitation and describe an extension to these datasets in which multiple bugs are identified in individual entries. We use test case transplantation and fault location translation, in order to expose and locate the bugs, respectively. We thus provide datasets of true multi-fault versions within real-world software projects, which maintain the properties and usability of the original datasets.

1 INTRODUCTION

Fault localization and program repair tools are typically evaluated over bug repositories such as Defects4J [30] or BugsInPy [46]. These repositories contain faulty program versions and their corresponding fixes and regression test suites, which have been mined from the full version history of multiple open-source Java and Python projects, respectively. However, both Defects4J and BugsInPy overwhelmingly only identify a *single fault* in each faulty program version: the textual difference between faulty and fixed versions is small and focused (typically only on a single line), and the fixed versions pass all tests in the regression test suites.

This single-fault nature limits the usefulness of these bug repositories as evaluation and training data sets. Real-world projects (including, in fact, even those used in Defects4J and BugsInPy) often contain multiple faults that can interact with and mask each other and thus make fault localization and repair harder; the use of single-fault evaluation datasets thus introduces a substantial threat to the validity of the evaluation itself. Similarly, using these bug repositories as training data can introduce bias into learning-based tools such as GRACE [35].

In this paper, we describe the construction of true *multi-fault* variants of Defects4J and BugsInPy. More specifically, we describe how we identify additional, *already existing* faults in the program versions, through a mining process based on *test case transplantation* and *fault location translation*.

Test case transplantation copies tests from the regression test suite of a given bug repository entry to an earlier entry, and checks whether they fail there; if so, this is taken as evidence that the fault fixed in the later program version is already present in its earlier version. Test case transplantation was introduced by An et al. [7] for the Java-based Defects4J bug repository. We demonstrate here that it can also be applied to the Python-based BugsInPy; however, the "Pythonic" programming style used in the underlying projects (e.g., the lack of explicit export interfaces and the corresponding structure of the import clauses) requires a substantially more complex test case extraction step to allow a successful transplantation.

Test case transplantation only indicates that multiple faults may be present but gives no indication where exactly they are located in the different program versions. Since this information is required for the evaluation of tasks such as fault localization, we complement the test case transplantation step by a fault location translation step. This traces the identified fault locations through the versions in the underlying *project* respository back to the version in the bug repository identfied through the test case transplantation.

We applied our technique to Defects4J v1.0.1, and to the current version of BugsInPy. On average, we identified 9.2 faults in each of the 311 versions of the 5 projects in Defects4J also used by An et. al. [7], and 18.6 faults in 501 versions of the 17 projects in BugsInPy. The identification of these faults requires one to two test cases on average to be transplanted per fault.

2 BACKGROUND

2.1 Original datasets

Our datasets are based on the original Defects4J [30] and BugsInPy [46] datasets, which contain collections of versions extracted or reconstructed from the original repositories of different open-source Java and Python projects, respectively. Figure 1 shows the common structure of all of these datasets.

Each underlying project version $v_i = (p_i, T_i)$ consists of the source code p_i and test suite T_i ①. Between any two consecutive versions v_{i-1} and v_i in the project history, there exists a set of changes or *diff* Δ_i ② for both the source code and the test suite such that applying the diff to the older version v_{i-1} will produce exactly the newer version v_i, i.e., $\Delta_i(v_{i-1}) = v_i$.

Each bug repository entry e ③ references two consecutive project versions (v_b, v_f). The "buggy" version v_b contains a single fault exposed by at least one failing test $t \in T_f$ from the "fixed" version v_f; this fault is repaired in v_f and all tests in $T_f = \Delta_f(T_b)$ pass.

The original datasets guarantee three properties that are important for their use as fault localization and program repair benchmarks. First, each fault is *exposed* by a failing test in the buggy version's test suite. Second, each fault is *repaired* in the fixed version, and all tests in the corresponding test suite pass. Third, each diff is *minimal*, i.e., any smaller change is not a repair.

Exposure through failing tests is the only indication of program failure; it is necessary for spectrum-based fault localization tools, which cannot predict faulty source code locations without failing tests. The fixed versions' test suites serve as specifications for program repair tools, and the locations affected by the minimal repairs are taken as fault locations ④, and used to determine the performance of any debugging tool in either locating or fixing the faults. However, the diff only approximates the fault location; this may be

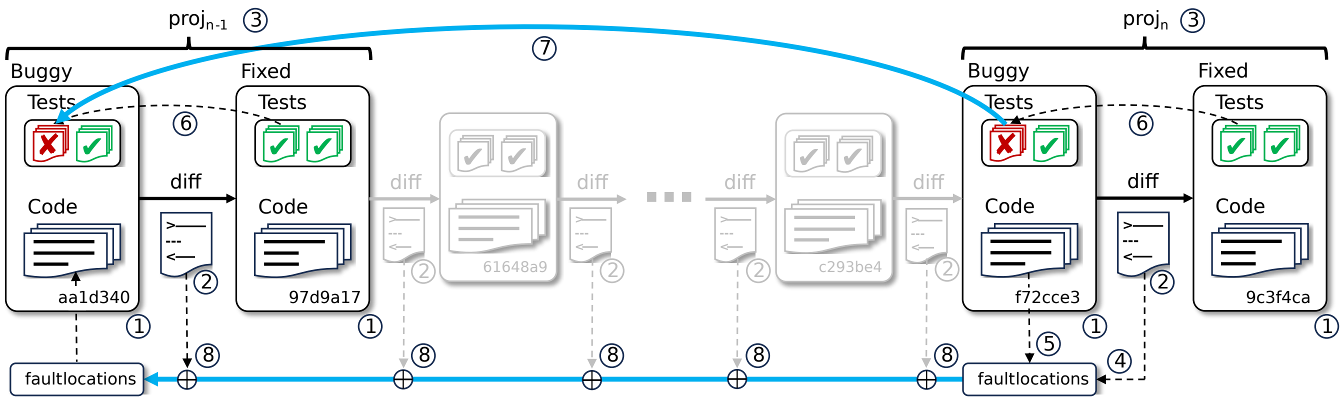

Figure 1: Project layout in original Defects4J [30] and BugsInPy [46] datasets, and construction of multi-fault variants.

improved by manually constructing the fault location oracle from inspection of the source code and bug fixing diff ⑤ [42].

2.2 Original dataset construction

BugsInPy identifies the project versions (v_b, v_f) referenced in an entry e by first inspecting the commit message related to the diff Δ_f for bugfix-related terms such as “fix”. It then checks for tests $t_j \in T_f$ that pass in the fixed version v_f but fail if they are added to the buggy version v_b (by applying the diff Δ_f to the v_b’s test suite), to ensure exposure of the bug. The addition of these test cases ⑥ to the buggy version changes its test suite T_b, but the tests are already part of the project history, and the code p_b is identical to the repository version. We align the version numbering in our multi-fault variant with the commit dates, and re-label if necessary.

Defects4J also inspects the commit messages of Δ_f for bugfix-related terms to identify the versions (v_b, v_f). However, while BugsInPy only considers bug fixes that are already minimal, Defects4J also selects bug fixes that contain feature additions. It separates the minimal bug fix Δ'_f from Δ_f to ensure minimality, and applies the inverse $\Delta'_f{}^{-1}$ to the fixed version v_f to reconstruct the “clean” buggy version v_b. The test suite T_f is then added to the buggy version v_b using Δ'_f, similar to BugsInPy. Hence, the buggy version v_b contained in Defects4J can differ from the referenced project version contained in the project history, however these differences are only in the feature additions contained within Δ_f.

2.3 Related datasets

Most fault identification datasets such as Defects4J and BugsInPy contain program versions with only a single fault each. The Software Infrastructure Repository (SIR) [21] contains a variety of faulty programs written in multiple programming languages; of these, space [45], an interpreter for an array definition language which contains 33 real-world single fault versions, and the Siemens set of small programs written in C which have been seeded with single faults, are widely used for evaluation. More recent work includes the HasBugs [10] dataset of 25 single-fault Haskell program versions. Note that these datasets are are sometimes (incorrectly) considered to be multi-fault datasets, due to the existence of multi-hunk faults. Our datasets are, in contrast, proper multi-fault datasets.

True multi-fault datasets are limited, and usually either contain synthetic or transplanted faults. Högerle et. al. [25] construct a dataset of 75000 Test Coverage Matrices (TCMs) from 15 open source Java projects. Each project version initially contained a passing test suite, and between 1 and 32 synthetic faults were automatically injected, causing at least one test case to fail. An et al. partially construct a multi-fault dataset with 311 versions from the Defects4J dataset, where the faults are exposed through the transplantation of a failing test case, but are not all identified (i.e. indication of source code in the version responsible for the fault). We build upon the work of An et al. in this paper to construct a full multi-fault dataset from Defects4J. Zheng et al. also constructed multi-fault datasets with 46 versions from the Defects4J dataset and 217 versions from the programs contained in the SIR [47], by manually transplanting faults from *older* versions to *newer* versions in the dataset. Their technique therefore alters the source code of underlying versions in the project history.

3 DATASET DESCRIPTION AND STATISTICS

This paper describes two separate datasets, Defects4J-mf [4] and BugsInPy-mf [3], which we created using the same techniques and for the same purposes. Both are multi-fault extensions to the original, underlying datasets Defects4J and BugsInPy, respectively.

Similar to the original datasets (see Section 2.1), the dataset extensions created in this paper consist of pre-existing, unaltered versions from underlying open-source repositories maintained using version control software. In addition to this, we too identify existing bugs in the versions by means of test case failures. However we differ from the original datasets by identifying *multiple faults* in each version. We do so by *exposing* additional faults in each version by transplanting test cases committed in future versions (see Section 5.1), and by *identifying* the faulty code locations by translating the fault locations identified by the original datasets for the previous versions (see Section 5.2). In order to ensure correctness, we only consider a bug as existing in a version if the test case transplantation and fault location translation processes both succeed; That is, if the bug is both *exposed* in the version by a failing test case, and *identifiable* by at least one line of code.

		Program size (loc)			Existing tests			Added	Drop	$\varnothing_{BPV}$
Project	N	Min	Mean	Max	Min	Mean	Max	tests	rate (%)	
Chart [22]	20	203303	208700.9	232364	1584	1752.9	2183	11.9	0.0	4.3
Lang [8]	61	48029	53116.4	61093	1605	1872.7	2670	10.1	0.0	7.6
Math [9]	104	30521	121701.8	185273	880	2856.6	5187	7.7	0.0	5.7
Time [18]	23	70198	77992.2	99183	3787	3913.8	4002	24.0	0.0	9.2
Closure [17]	103	99385	208393.9	269152	1629	6699.1	7588	28.9	1.4	14.9
Total	311	30521	133981.0	269152	880	3419.0	7588	16.5	0.3	8.3
PySnooper [36]	3	335	560.3	673	5	17.0	29	0.0	0.0	1.0
ansible [20]	18	101706	1124664.3	1590076	3101	7984.1	11020	5.0	0.0	5.5
black [33]	23	5241	66510.7	96049	18	81.0	129	5.1	0.7	5.9
cookiecutter [23]	4	1258	1828.8	2049	156	251.5	298	1.8	0.3	2.0
fastapi [37]	16	2839	4172.4	4954	179	572.0	793	3.8	9.8	3.2
httpie [38]	5	775	3106.2	3911	17	146.4	232	1.8	27.3	2.2
keras [16]	45	36600	39474.9	42438	158	24817.2	45484	5.6	4.3	5.2
luigi [11]	33	14185	20071.3	28751	549	973.9	1581	5.0	26.9	4.4
matplotlib [28]	30	118312	120706.2	123290	7542	7814.3	8191	5.6	8.2	6.1
pandas [43]	169	159369	161675.2	164785	50989	63559.8	88768	59.0	6.0	45.3
sanic [40]	5	5506	7121.2	7604	638	641.3	644	1.0	0.0	2.0
scrapy [41]	40	15636	20352.8	22631	923	1377.0	2050	16.0	52.6	8.3
spacy [26]	10	94575	97907.0	104284	1647	2398.4	2617	1.1	0.0	2.1
thefuck [29]	32	1636	3679.5	6248	283	1087.5	1716	7.1	64.6	4.1
tornado [44]	16	21167	22957.9	24422	16	19.6	23	1.8	5.0	2.5
tqdm [19]	9	655	2348.0	3229	14	61	91	0.9	21.4	1.6
youtube-dl [12]	43	20515	82597.7	137957	324	1530.2	2365	6.3	16.2	6.0
Total	501	335	104690.3	1590076	5	6666.6	88768	7.5	14.3	6.3

Table 1: Dataset statistics. *N* is the number of versions in the project, program and test suite sizes are averaged over all project versions, $\varnothing_{BPV}$ is the average number of bugs available in the multi-fault versions of the project.

We successfully identify 9.2 respectively 18.6 faults in each version from the Defecst4J respectively BugsInPy datasets. Table 1 gives the overall dataset statistics, while Figure 2 gives a more detailed look at the bug distributions. We see from this figure that the Defects4J versions have on average substantially fewer bugs (normalized by program size) identified in our datasets than the BugsInPy versions, and that particular projects within BugsInPy have substantially higher bug densities in their versions than the rest. For each of these versions, we transplant on average 16.5 and 6.3 test cases for Defects4J and BugsInPy respectively, which are necessary to expose the additional bugs in these versions. Figure 3 shows the number of tests transplanted per bug in each version. We also report in Table 1 the number of times a fault was excluded from a version (drop rate), with the test case transplantation process succeeding, but the fault location translation process failing. This indicates the number of times a fault is exposed, but cannot be automatically identified in the version. On average, this occurs 0.3% and 14.3% for Defects4J and BugsInPy respectively, with the anomaly occurring more frequently on certain projects.

The datasets created in this paper also enable detailed perspectives on the underlying software projects and versions themselves. Figure 4 gives one such insight, showing the average number of versions in which a particular bug is available from each of the projects. Combining this information with the information from each projects' git history, we are able to estimate the amount of time a particular bug is active for, which is given in Figure 5. These figures give an estimate of the average lifespan of a bug in a particular program. We note, however, that this is a lower estimate on the lifespan of the bugs, as these bugs could be available in more versions that are not identified by our techniques. As we can see from Figure 4, bugs from the Java-based Defects4J projects last on average 6.9 Defects4J versions, whereas bugs from the Python-based BugsInPy projects last only 4.1 BugsInPy versions on average (excluding the project pandas). Despite this, there are particular bugs and whole projects (such as pandas) where the average version lifespan is much greater. For example, most projects have at least one bug that has a lifespan of on average 35 versions. Figure 5 indicates that the average lifespan of a bug is usually quite small (around one to two weeks), however we also see here that bug lifetimes vary widely, and that most projects again have at least one bug that lasts more than 100 to 200 days). These statistics indicate that although it is uncommon for bugs to last more than a week or two, there are usually individual bugs whose lifespan spans a larger portion of the project history. These findings are corroborated in current literature on the topic of bug lifetimes [15, 32, 39], which indicates both the veracity of the data, and the accuracy of our datasets in identifying faults within versions.

4 DATASET USAGE

As described in Section 2, Defects4J and BugsInPy consist of versions from popular open source projects written in Java and Python,

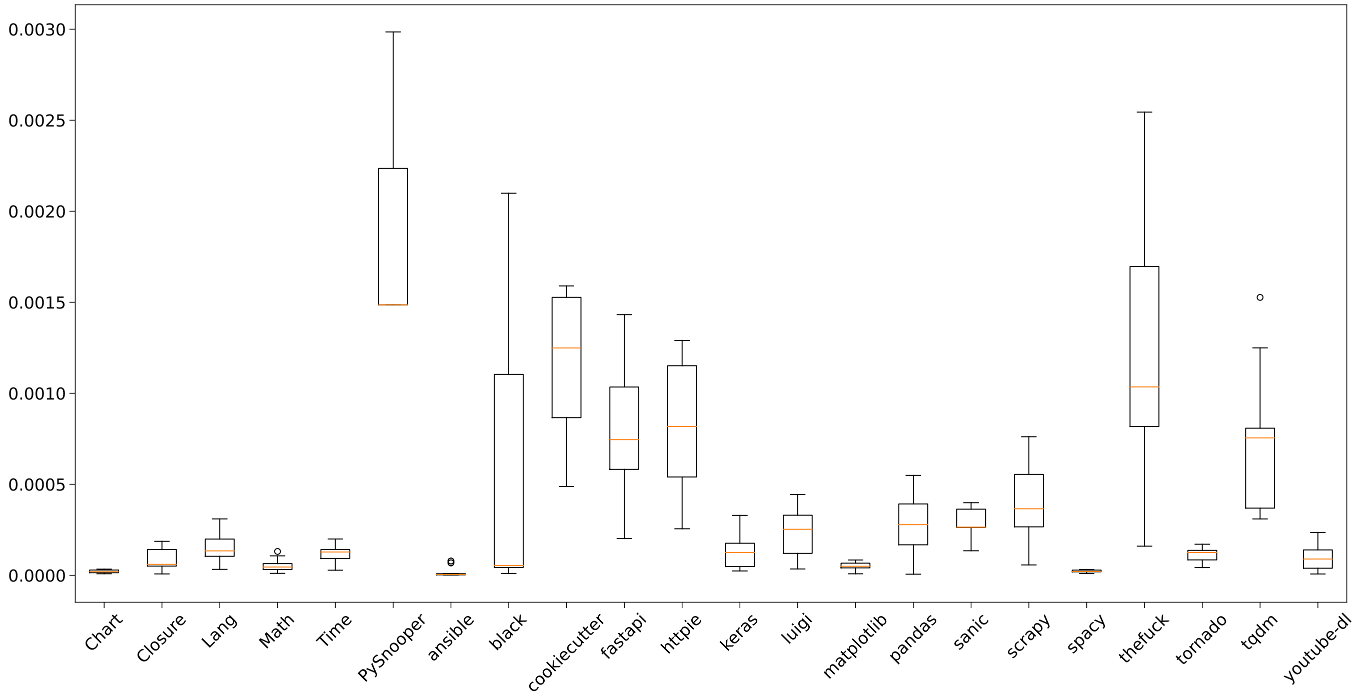

Figure 2: Average number of bugs per version, normalized by the program size of the version.

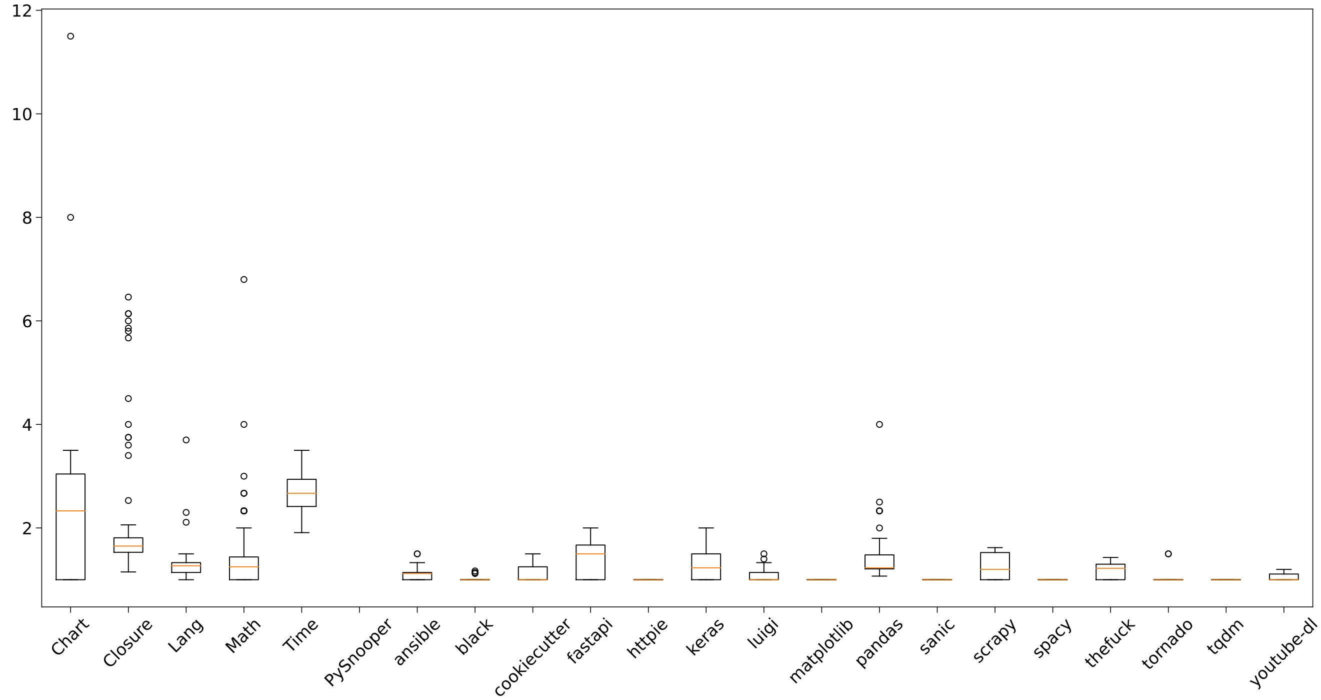

Figure 3: Number of tests transplanted per bug, averaged by version.

however the datasets themselves do not store each version for the projects, but rather provide the facilities to easily clone the versions tracked by the dataset from the original project repositories. We maintain the functionality and setup of the original datasets in our

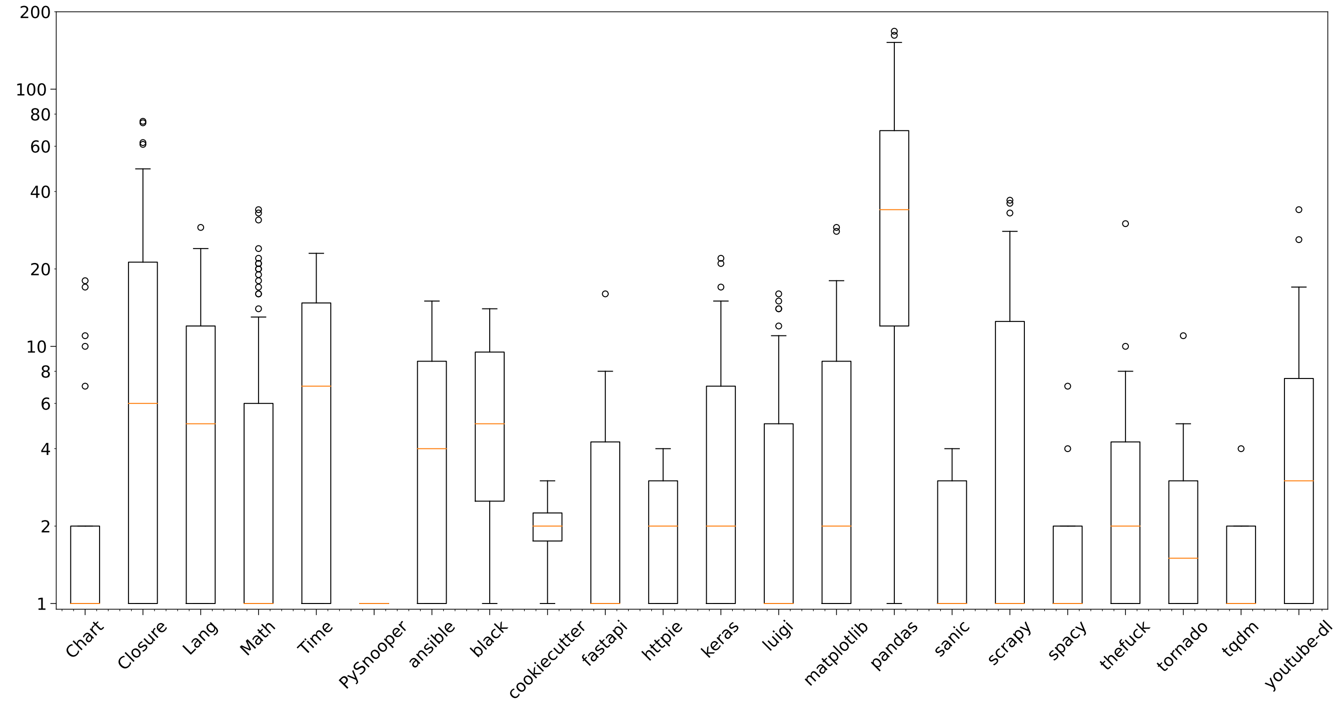

Figure 4: Average number of versions a particular bug is available in (y-axis in log scale).

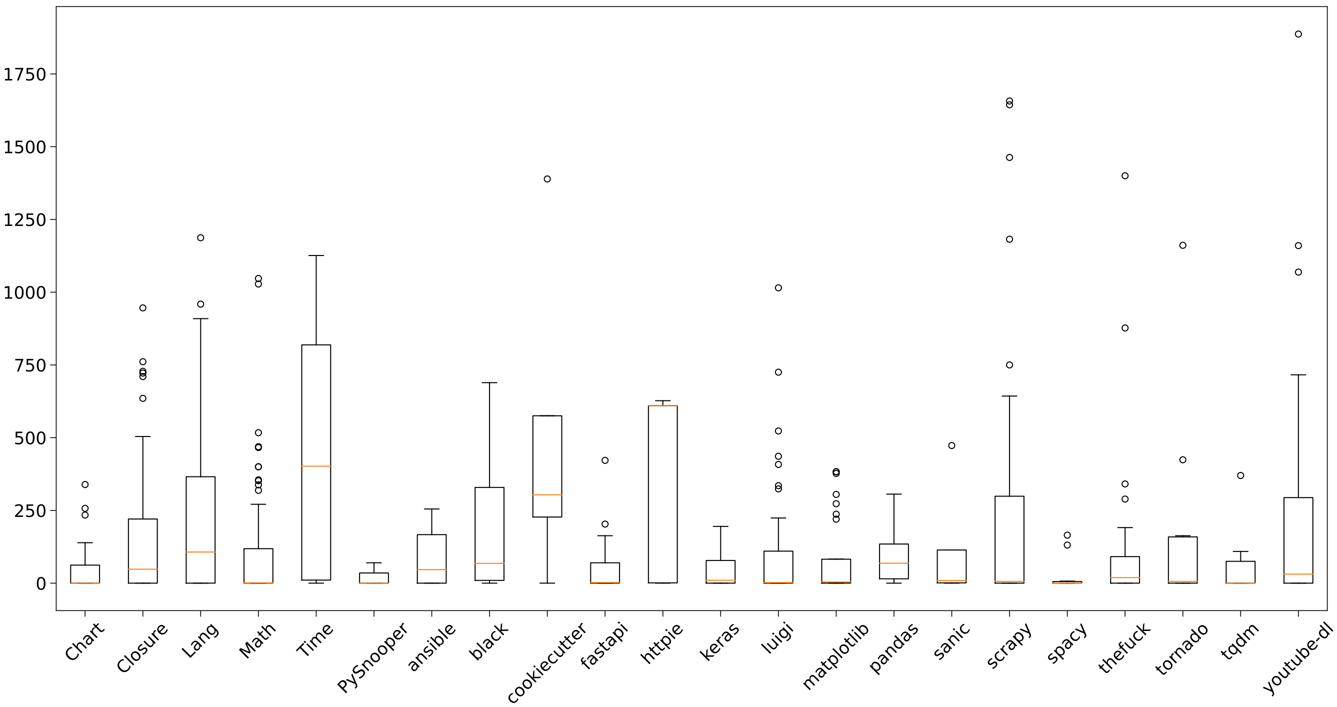

Figure 5: Average number of days between the oldest version a bug is available in and the version in which the bug is fixed (bug lifetime).

Command	Description
`info`	Get the information of a specific project or bug
`checkout`	Checkout a buggy or a fixed project version (use **`multi-checkout`** for BugsInPy multi-fault)
`compile`	Compile sources and developer-written tests of a buggy or a fixed project version
`test`	Run a single test method or a test suite on a buggy or a fixed project version
`coverage`	Run code coverage analysis on a buggy or a fixed project version
`to-tcm`	Output coverage in TCM format (BugsInPy only)
`mutation`	Run mutation analysis on a buggy or a fixed project version
`fuzz`	Run a test input generation from specific bug (BugsInPy only)
`identify`	Add fault location information to elements

Table 2: Defects4J and BugsInPy commands; multi-fault modifications and extensions in bold.

extension, and only add functionality to allow each version to be identified as containing multiple faults.

4.1 Usage description of the original datasets

For completeness, we describe the usage of the original Defects4J and BugsInPy datasets. Both allow interaction with the underlying project versions through the use of a list of specialized CLI commands. Table 2 lists the commands supported by both Defects4J and BugsInPy. They are run as `defects4j <command>` and `bugsinpy-<command>`, respectively. Note that any of the provided tools can be run on the multi-fault datasets described in this paper.

4.2 Usage description of the multi-fault datasets

The main addition with our multi-fault datasets is the ability to identify versions in the underlying datasets that contain multiple faults. As such, the main difference in the commands is the addition of a multi-fault `checkout` command. These can be run as `defects4j_multi checkout` and `bugsinpy-multi-checkout`, respectively. These commands use the underlying Defects4j and BugsInPy datasets' `checkout` commands to clone the version from the project repository; however, they also add for *each* of the faults identified in the version the fault-exposing tests and the fault locations. In both multi-fault datasets, the transplanted test cases are added to the existing test suite by a process of test case source code alteration ("splicing"). After the multi-fault checkout process is complete, these test cases are accessible in the test suite, and for any test suite related commands in the underlying dataset. The fault locations are available in `bug.locations.<bugId>` files for each bug, in both datasets.

In addition to the `checkout` command, the our datasets also provide useful commands for evaluation purposes. In particular, the coverage commands provided by the original Defects4J and BugsInPy datasets do not collect code coverage per test case which is needed for techniques such as spectrum-based fault localization [24]. We thus additionally provide commands for this purpose. For Defects4J, we alter the original `coverage` command to collect code coverage using Gzoltar [14] instead of Jacoco [2]. For BugsInPy, we change the settings of the `coverage` command (which uses Python's `coverage.py` coverage library) to extract coverage per test, and provide the `to-tcm` command, which converts the collected coverage into TCM [1] format. We then also provide commands in both datasets for identifying each of the faults, based on the fault identification information, within the collected coverage format, using the `identify` command. For both datasets, this command adds the fault as a part of the element (i.e., source code line) name, in the respective format.

5 DATASET CREATION

Like the original datasets, our multi-fault versions guarantee fault exposure and fault location identification. The former is achieved by test case transplantation, the latter by fault location translation. We describe each step in turn.

5.1 Test case transplantation

Test case transplantation copies fault-revealing test cases from the test suite of one bug repository entry to that of an earlier entry. This process does not alter the source code of the project's versions, and all test case logic is *extracted* from an existing projcet version, and not *created.* The top of Figure 1 shows the test case transplantation process ⑦: test cases which expose the fault in the buggy version of an entry are extracted, and then copied to a previous entry. The test cases are then compiled and run, and their output is compared with their output from their original version. If the outputs are similar enough according to the Hunt-Szymanski algorithm [27] for longest common subsequence (LCS), the fault is considered exposed also in the target entry. Each set of fault-exposing test cases is transplanted as far as possible, i.e., until the fault is no longer exposed.

For the Defects4J, we reused the the test case transplantation by An et. al. [7]. They provide tools for extracting and copying the test cases from one version to another, and identify many Defects4J versions in which test cases can be transplanted to expose multiple faults. Their tools and results have been included in the defects4j-mf dataset created in this paper. We note that An et. al. were only able to identify 311 multi-fault versions out of the 396 available bugs in Defects4j v1.0.1. We thus only include these 311 versions in our defects4j-mf dataset.

For BugsInPy, we carried out a similar process in order to achieve the same results. In particular, we provide the tools for extracting and copying test cases from one version to another, and identify the BugsInPy versions in which test cases can be transplanted for the exposure of multiple faults. This process was considerably more complex for the Python project versions in BugsInPy, due to certain Python coding conventions, which encourage test fragment reuse, and specialized imports. We developed a *source code dependency aware* test extraction and copying tool, which allows both the test cases and their respective source code dependencies (e.g. test fixtures, imports, etc.) to be extracted and copied between versions.

5.2 Fault location translation

In the original Defects4J and BugsInPy datasets, the fault identification used for the location oracle for each buggy version b either uses the lines changed in the diff Δ_f as an approximation ④, or relies on a more precise manual identification ⑤. We use either of these methods as a starting point for fault identification in the multi-fault versions. However, these identified locations cannot be used directly to identify the fault locations in prior versions as the changes during the development may have caused the source code locations to shift. We therefore backtrack [5] the starting locations through all versions in the complete project repository, until we reach the buggy target version b' of each test case transplantation step; for each version i, we consider the operations in the diff Δ_i and update the fault locations as follows ⑧. (1) If a source file containing a tracked location is renamed, the tracking respects this renaming. (2) If source code in the same file as a tracked location is altered above this location, then the tracked location is adjusted to reflect the changes. (3) If a tracked location is modified or added in a particular diff, then tracking for this location is stopped; this ensures that the tracked source lines remain unmodified, and are thus identical to the location in the version in which the bug was originally identified. We consider a particular fault identified in a target version if at least one identified fault location is tracked successfully back into that version.

5.3 Limitations and threats to validity

The identification of buggy versions from the underlying real-world, open-source projects used in the Defects4J and BugsInPy datasets was done manually. This manual identification of buggy versions is occasionally incorrect and can lead to incorrect results in the multi-fault versions of the datasets.

Due to the extensive manual effort we did not manually identify the location of each fault in each multi-fault version. Where available, we used existing manual fault identification [42] for each bug in the version where it was discovered by Defects4J or BugsInPy; otherwise, we used the diff Δ_f between the buggy and fixed versions as an approximation. We then extended this to the multi-fault versions using the automated fault location translation process. The approximate fault identification using source code diffs and the automated fault location translation are both susceptible to errors; in particular the location translation may be unable to trace any faulty lines to an earlier version and thus fail to identify all faults. We automatically verified that lines were properly translated, and manually corrected lines incorrectly translated.

The test case transplantation process may also not always work correctly. In order to prevent this from interfering with the quality of the dataset, we tested each transplanted test case, and only accepted transplanted test cases that compiled (i.e., did not produce any compile-time or runtime errors) and produced the same result (i.e., expected output or error) as in the original Defects4J or BugsInPy version, and whose fault location could be fully translated. This ensures a conservative approach; only bugs that are truly exposed in a version are identified, but some bugs that may actually be active in a version may be missed.

As noted in [31], the underlying datasets used in this paper may have the limitation that their test cases are usually only available in the project version which contains the corresponding bug, and thus could be contaminated by the knowledge of the bug-fix. This limitation may also therefore have an impact on the datasets created in this paper. In addition, this limitation may be compounded by the fact that the test case transplantation process used in this paper modifies the test suite of a version by including test cases that were only available in subsequent versions, and thus could contaminate previous versions with the knowledge of fixes from later versions. This limitation may have an adverse effect, mainly on automated program repair techniques, as discussed in [31], but may also have a minor effect on localization techniques. However based on a sample of bugs from each underlying dataset, we notice that the test cases can be reasonably constructed from the corresponding bug report without the knowledge of the bug-fix, indicating a lack of dependence of the test cases on the future knowledge. This suggests that this limitation may be mitigated by further study on the composition of the underlying datasets' test suites. We leave such study as future work.

The fault location translation process used in this paper to identify the faulty locations does not allow the inclusion of the developer-written bug patches in the multi-fault versions. This presents a limitation for techniques such as automated program repair (APR), which often require these patches for evaluation of the techniques.

6 FUTURE DATASET USAGE

6.1 Multi-fault localization

Multi-fault localization [6] and program repair are open problems; multi-fault datasets mined from real-world software projects such as these described here will therefore be useful for training and evaluation of multi-fault debugging tools. More specifically, we already used our multi-fault version of Defects4J for the evaluation of our spectrum-based fault localization tool FLITSR [13], and showed that it can localize multiple faults at the same time.

Automated debugging techniques that rely on machine learning also require large datasets of bugs as training data. Previously, these techniques have used datasets of synthetic (injected) faults, and single fault datasets such as Defects4J [34, 35]. However, this leads to bias in the machine learning model [13]. Our multi-fault datasets can be used as more realistic training data for such machine learning models to improve real-world applicability.

We also see qualitative uses of our datasets. In particular, the identification of software project versions with multiple bugs existing simultaneously may be used in an analysis of the presence of multiple bugs in software systems. In addition, the fact that test cases could be transplanted from newer versions to expose bugs in previous versions may also provide insight into research questions such as "can better test suites expose more bugs?". This has applications in software fuzzing and automated test case generation.

6.2 Future work

Due to automated data construction, any improvements of the underlying Defects4J and BugsInPy datasets will improve the quality of our multi-fault versions as well. This leads to many avenues for further improvement of this dataset by improving: (1) the fault identification for each bug through manual fault identification as in [42] for other projects, (2) the bug isolation and reproducibility by better automation of the set-up for each version, and (3) the size of the datasets by adding more versions exposing more bugs. Each of the above changes will result in improvements in the multi-fault counterparts as well, allowing for better identification, reproducibility, and more bugs in each multi-fault version.

The fault location translation process does not yet fully support complex branching in the git history. We leave it as future work to add this functionality. The ideas used to extend the Defects4J and BugsInPy datasets in this paper can be generalized to other datasets involving many other languages. An example of such a dataset to which these techniques can be applied is the HasBugs dataset [10]. We leave the extension of such datasets using the techniques provided as future work. As mentioned in Section 5.3, bug patches for all faults are not included in the multi-fault versions. We identify the addition of these patches as future work.

7 CONCLUSION

In this paper we present extensions to the Defects4J and BugsInPy datasets of bugs in real-world software projects which expose the existence of multiple bugs in each of the versions. We find on average 9.2 and 18.6 bugs in the Defects4J and BugsInPy versions, respectively. The extension uses test case transplantation and fault location translation to identify these multi-fault versions. In doing so, we do not create or modify any of the existing real-world software project's code, and only use test cases produced by the developers of the corresponding software project.

We have made the creation of the multi-fault extension of the datasets mostly automatic, simplifying reproducibility and future verification. In addition, the process requires only minimal manual efforts when run on new projects or versions added to the underlying Defects4J and BugsInPy datasets.

We maintain the existing frameworks' extensibility and ease of use by allowing all existing functionality to be used in the extensions. We additionally add useful functionality for coverage collection for use in fault localization and program repair.

REFERENCES

[1] 2014. "More Debugging in Parallel" Resource Page. https://www.fernuni-hagen.de/ps/prjs/PD/.

[2] 2017. JaCoCo Java Code Coverage Library. https://www.eclemma.org/jacoco/.

[3] 2023. BugsInPy multi-fault repository. https://github.com/DCallaz/bugsinpy-mf.

[4] 2023. Defects4J multi-fault repository. https://github.com/DCallaz/defects4j-mf.

[5] 2023. Fault location translation (backtracking) tool. https://github.com/DCallaz/bug-backtracker.

[6] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 88–99. https://doi.org/10.1109/ASE.2009.25

[7] Gabin An, Juyeon Yoon, and Shin Yoo. 2021. Searching for Multi-fault Programs in Defects4J. In *Search-Based Software Engineering - 13th International Symposium, SSBSE 2021, Bari, Italy, October 11-12, 2021, Proceedings (Lecture Notes in Computer Science)*, Vol. 12914. Springer, 153–158. https://doi.org/10.1007/978-3-030-88106-1_11

[8] Apache Software Foundation. 2002. Commons Lang. https://commons.apache.org/proper/commons-lang/

[9] Apache Software Foundation. 2007. Apache Commons Math. https://commons.apache.org/proper/commons-math/

[10] Leonhard Applis and Annibale Panichella. 2023. HasBugs - Handpicked Haskell Bugs. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 223–227. https://doi.org/10.1109/MSR59073.2023.00040

[11] Erik Bernhardsson, Elias Freider, and contributors to Luigi. 2012. Luigi. https://github.com/spotify/luigi

[12] Daniel Bolton and contributors to Youtube-dl. 2011. Youtube-dl. https://github.com/ytdl-org/youtube-dl

[13] Dylan Callaghan and Bernd Fischer. 2023. Improving Spectrum-Based Localization of Multiple Faults by Iterative Test Suite Reduction. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1445–1457. https://doi.org/10.1145/3597926.3598148

[14] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. GZoltar: an eclipse plug-in for testing and debugging. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*. ACM, 378–381. https://doi.org/10.1145/2351676.2351752

[15] Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano Di Penta. 2011. How Long Does a Bug Survive? An Empirical Study. In *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, Martin Pinzger, Denys Poshyvanyk, and Jim Buckley (Eds.). IEEE Computer Society, 191–200. https://doi.org/10.1109/WCRE.2011.31

[16] François Chollet et al. 2015. Keras. https://keras.io.

[17] Closure Compiler Authors. 2009. Closure Compiler. https://developers.google.com/closure/compiler/

[18] Stephen Colebourne and contributors to Joda-Time. 2014. Joda-Time. https://www.joda.org/joda-time/

[19] Casper O da Costa-Luis. 2019. tqdm: A fast, extensible progress meter for python and cli. *Journal of Open Source Software* 4, 37 (2019), 1277.

[20] Michael DeHaan and contributors to Ansible. 2013. Ansible. https://github.com/ansible/ansible

[21] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empir. Softw. Eng.* 10, 4 (2005), 405–435. https://doi.org/10.1007/S10664-005-3861-2

[22] David Gilbert and contributors to JFreeChart. 2000. JFreeChart. https://www.jfree.org/jfreechart

[23] Audrey Roy Greenfeld and contributors to Cookiecutter. 2014. Cookiecutter. https://github.com/cookiecutter/cookiecutter

[24] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, André van Hoorn, Antonio Filieri, and David Lo. 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Softw. Pract. Exp.* 49, 8 (2019), 1197–1224. https://doi.org/10.1002/spe.2703

[25] Wolfgang Hogerle, Friedrich Steimann, and Marcus Frenkel. 2014. More Debugging in Parallel. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*. IEEE Computer Society, 133–143. https://doi.org/10.1109/ISSRE.2014.29

[26] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. spaCy: Industrial-strength Natural Language Processing in Python. (2020). https://doi.org/10.5281/zenodo.1212303

[27] James W. Hunt and Thomas G. Szymanski. 1977. A Fast Algorithm for Computing Longest Common Subsequences. *Commun. ACM* 20, 5 (may 1977), 350–353. https://doi.org/10.1145/359581.359603

[28] John D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science and Engineering* 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55

[29] Vladimir Iakovlev and contributors to The Fuck. 2015. The Fuck. https://github.com/nvbn/thefuck

[30] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21-26, 2014*. ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[31] Vinay Kabadi, Dezhen Kong, Siyu Xie, Lingfeng Bao, Gede Artha Azriadi Prana, Tien-Duy B. Le, Xuan-Bach Dinh Le, and David Lo. 2023. The Future Can't Help Fix The Past: Assessing Program Repair In The Wild. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2023, Bogotá, Colombia, October 1-6, 2023*. IEEE, 50–61. https://doi.org/10.1109/ICSME58846.2023.00017

[32] Sunghun Kim and E. James Whitehead Jr. 2006. How long did it take to fix bugs?. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*, Stephan Diehl, Harald C. Gall, and Ahmed E. Hassan (Eds.). ACM, 173–174. https://doi.org/10.1145/1137983.1138027

[33] Łukasz Langa and contributors to Black. 2018. *Black: The uncompromising Python code formatter*. https://github.com/psf/black

[34] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 169–180. https://doi.org/10.1145/3293882.3330574

[35] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 664–676. https://doi.org/

10.1145/3468264.3468580

[36] Ram Rachum, Alex Hall, Iori Yanokura, et al. 2019. *PySnooper: Never use print for debugging again.* https://doi.org/10.5281/zenodo.10462459

[37] Sebastián Ramírez. 2018. *FastAPI.* https://github.com/tiangolo/fastapi

[38] Jakub Roztocil and contributors to Httpie. 2012. Httpie. https://github.com/jakubroztocil/httpie

[39] Ripon K. Saha, Sarfraz Khurshid, and Dewayne E. Perry. 2014. An empirical study of long lived bugs. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, Serge Demeyer, Dave W. Binkley, and Filippo Ricca (Eds.). IEEE Computer Society, 144–153. https://doi.org/10.1109/CSMR-WCRE.2014.6747164

[40] Sanic Community Organization. 2017. Sanic. https://github.com/sanic-org/sanic

[41] Scrapy Developers. 2012. Scrapy. https://github.com/scrapy/scrapy

[42] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 130–140. https://doi.org/10.1109/SANER.2018.8330203

[43] The pandas development team. 2010. *pandas-dev/pandas: Pandas.* https://doi.org/10.5281/zenodo.3509134

[44] Tornado Developers. 2013. Tornado Web Server. https://github.com/tornadoweb/tornado

[45] Filippos I. Vokolos and Phyllis G. Frankl. 1998. Empirical Evaluation of the Textual Differencing Regression Testing Technique. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*. IEEE Computer Society, 44–53. https://doi.org/10.1109/ICSM.1998.738488

[46] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1556–1560. https://doi.org/10.1145/3368089.3417943

[47] Yan Zheng, Zan Wang, Xiangyu Fan, Xiang Chen, and Zijiang Yang. 2018. Localizing multiple software faults based on evolution algorithm. *J. Syst. Softw.* 139 (2018), 107–123. https://doi.org/10.1016/j.jss.2018.02.001