Tests4Py: A Benchmark for System Testing

Marius Smytzek
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
marius.smytzek@cispa.de

Martin Eberlein
Humboldt-Universität zu Berlin
Berlin, Germany
martin.eberlein@hu-berlin.de

Batuhan Serçe
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
batuhan.serce@cispa.de

Lars Grunske
Humboldt-Universität zu Berlin
Berlin, Germany
grunske@hu-berlin.de

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.de

ABSTRACT

Benchmarks are among the main drivers of progress in software engineering research, especially in software testing and debugging. However, current benchmarks in this field could be better suited for specific research tasks, as they rely on weak system oracles like crash detection, come with few unit tests only, need more elaborative research, or cannot verify the outcome of system tests.

Our *Tests4Py* benchmark addresses these issues. It is derived from the popular *BugsInPy* benchmark, including 61 bugs from 7 real-world Python applications and, in addition, 6 bugs from 4 example programs. Each subject in Tests4Py comes with an *oracle* to verify the functional correctness of system inputs. Besides, it enables the generation of system tests *and* unit tests, allowing for qualitative studies by investigating essential aspects of test sets and extensive evaluations. These opportunities make Tests4Py a next-generation benchmark for research in test generation, debugging, and automatic program repair.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Software libraries and repositories**.

KEYWORDS

Benchmark, Python, Test generation

1 INTRODUCTION

For several years, benchmarks of program faults have been the backbone for evaluating methodologies and driving qualitative studies in the domain of software engineering research [8, 11, 20]. These benchmarks aim to allow engineers to investigate faults within a real-world application context. Consequently, these benchmarks predominantly consist of programs with identifiable faults, paving the way for thorough investigations of bugs.

Google's FuzzBench [11], for example, is a service to evaluate fuzz testing tools on various real-world subjects. Similarly, Codeflaws [18] is a benchmark developed for automatic program repair, wherein bugs are sorted into categories to yield insights into the types of bugs that can be repaired. Beyond benchmarks with targeted goals, numerous benchmarks are tailored for specific programming languages. Notable examples include Defects4J [8], BugsJS [5], and BugsInPy [20] for Java, JavaScript, and Python, respectively.

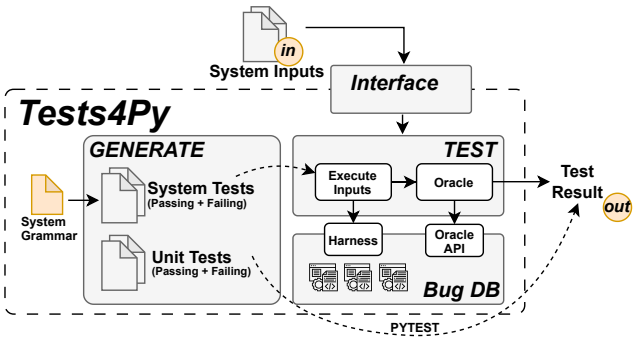

Figure 1: Tests4Py Overview. Tests4Py incorporates components for generating system and unit tests, running them, and assessing their results using generic oracles.

Despite these benchmarks' pivotal role in software engineering research, the complexity of automated debugging and test generation approaches often requires more comprehensive evaluations. For instance, test generation benchmarks currently rely on *generic* oracles, such as crash detection, which, while being practical for gauging program security, could benefit from evaluations that determine their efficacy in uncovering *functional* bugs. Detecting functional bugs requires test generators capable of producing *oracles*, which remains a research challenge.

Another significant limitation is that benchmarks typically offer a set of unit tests with *fixed inputs*, lacking interfaces to incorporate *generated inputs*. This restricts the potential for exploring combinations of test generators and automated repair tools.

Our novel benchmark, Tests4Py, addresses and overcomes these limitations for Python programming. Tests4Py (Figure 1) is an extendable suite that consists of various faults, each meticulously sourced from five real-world Python programs. These bugs are adopted from the BugsInPy database and are thoughtfully augmented with an *oracle*, a *system interface*, and the capability to *incorporate both system and unit test generators*.

A key characteristic of Tests4Py is its emphasis on *test diversity* to foster a more extensive and rigorous evaluation. Therefore, each faulty program included in our benchmark is coupled with a comprehensive set of carefully constructed system and unit tests that can be used to guarantee test diversity if needed. Half of these tests are designed to pass successfully, while the other half are crafted

to fail, simulating various scenarios. This unique design balances capturing potential faults and affirming the program's functionality, providing a comprehensive and practical benchmarking tool in the Python programming environment.

To illustrate Tests4Py, let us consider bug #2 from the FastAPI [14] project,[1] which occurs when FastAPI establishes a WebSocket while overriding its dependencies. Each bug in our benchmark comes with at least one failing unit test. The unit test for FastAPI bug #2 (Figure 2) is included in the FastAPI project and was adopted by BugsInPy. This particular unit test includes an oracle specific to this test case.

There is nothing wrong with this unit test. However, it cannot be used in conjunction with *generated tests* (as it has only one input and related oracle), and it cannot be used with *system inputs* (given or generated). This is where Tests4Py steps in:

- First, Tests4Py offers an *interface for system tests* (Figure 3) that runs the project through a custom *harness* that is included in the benchmark.
- Second, Tests4Py provides an *oracle* that is suitable for generated inputs (Figure 4). The generic oracle for FastAPI bug #2 examines the run and checks for signals that indicate whether the defect has been triggered.
- On top, Tests4Py provides hand-crafted *grammars*, specifying the input of the program to *generate* and *validate* further inputs (with outcomes checked through its oracles).

Assembling these oracles, tests, and test generation interfaces for each subject and bug required substantial effort and attention to detail. We began with a deep dive into each fault, aided by unit tests and fixes from the BugsInPy database. Based on our understanding, we designed twenty unique unit tests per fault. If a subject lacked an interface capable of triggering the fault, we implemented a harness to fill this gap, furthering the authenticity of our testing environment. While time-intensive, this exhaustive process was integral to creating a more comprehensive and realistic benchmark. By addressing the previously identified limitations, we aimed to enhance the value of our benchmark, Tests4Py, for the software engineering community.

With Tests4Py, we make the following contributions:

An easy-to-use benchmark. We provide a simple *command line interface*. We also incorporate the entire BugsInPy database, which is easily extendable with new bugs in Tests4Py.

Oracles. We provide an *oracle* for every subject, facilitating the verification of (given and generated) system tests, including functional testing.

Interfaces for test generation. We include *system and unit test generators* for each subject, enabling the creation of large and diverse test sets.

Input specifications. We provide *input grammars* to specify the format of the system tests for each included bug.

Tests4Py is available as open source; see Section 6 for details.

2 TESTS4PY AND ITS BENCHMARK

Tests4Py builds on the bugs in the existing BugsInPy [20] benchmark. This decision significantly expedited our process, alleviating the need to identify bugs from the ground up. However, every subject we selected underwent rigorous verification against the initially provided unit tests. Any subject that failed to reproduce the bug using unit tests was duly discarded, a fate that befell several BugsInPy subjects. Given the extensive collection of real-world bugs already at our disposal, the unique feature of the Tests4Py benchmark is its flexibility in testing the included subjects. Each subject must be capable of *accepting inputs to enable system testing*. For this purpose, we implemented *harnesses* for virtually all subjects where direct system input was not feasible, maintaining close fidelity to the original defect. So far, our Tests4Py benchmark includes 61 bugs sourced from seven different real-world projects: *Cookiecutter* [4] with 3 bugs, *FastAPI* [14] with 16 bugs, *HTTPie* [15]

[1] Tests4Py uses the same bug identifiers as BugsInPy.

FastAPI bug #2 failing unit test

```
def test_router_ws_depends_with_override():
    client = TestClient(app)
    app.dependency_overrides[ws_dependency]
        = lambda: "Override"
    with client.websocket_connect("/router-ws-depends/")
        as websocket:
            assert websocket.receive_text() == "Override"
```

Figure 2: The original unit test for the FastAPI bug #2 as included in the FastAPI project.

Tests4Py: *FastAPI* bug #2 interface

```
def run(system_test: List[str]):
    subprocess.run(
        ["python", HARNESS_FILE] + system_test,
        env=execution_environment, stdout=subprocess.PIPE
    ).stdout
```

Figure 3: The Tests4Py interface (simplified) provides a *harness and API* to execute system tests for the FastAPI bug #2. The result of this function gets directly provided to the oracle.

Tests4Py: Oracle for *FastAPI* bug #2

```
def oracle(output) -> TestResult:
    if (mode == "websocket" and url in websockets
        and websockets[url] in overrides
        and overrides[websockets[url]] not in output):
        return TestResult.FAILING
    else:
        return TestResult.PASSING
```

Figure 4: The Tests4Py oracle (excerpt and abstracted) for FastAPI bug #2, used to validate system tests, checks for *generic* issues. The input itself describes the mode, websockets, and overrides.

with 5 bugs, *PySnooper* [13] with 2 bugs, *Sanic* [12] with 5 bugs, *The Fuck* [7] with 20 bugs, and *youtube-dl* [1] with 10 bugs. Moreover, we implemented four example programs with six bugs to enable users to quickly and initially evaluate their setup before evaluating the real-world subject. These programs are labeled as *calculator*, *expression*, *markup*, and *middle*.

2.1 Tests4Py Components

Figure 1 shows the components of Tests4Py. Like its predecessor, BugsInPy, Tests4Py adheres to a *three-tier architecture*. This structure consists of (1) a *bugs database*, serving as a repository that houses the metadata for each subject; (2) a *database abstraction layer*, rendering the information from the bugs database into formats accessible by both humans and machines; and (3) an *execution framework* to test each bug using either the provided or generated tests, bringing the entire process full circle and enabling a comprehensive evaluation of each fault.

The Tests4Py *components* include the *generation* of new test cases and the subsequent *testing* of these cases. The generation process can produce an arbitrary number of tests, with a specified proportion of failing tests. Every generated system test aligns with the provided input specification, i.e., the grammar. The oracle subsequently assesses these system tests. The oracle gets information about the executed subject, scrutinizes the observation, and classifies it as passing, failing, or unknown in cases where the oracle cannot precisely determine it. This meticulous process ensures a comprehensive and reliable testing framework within Tests4Py.

While extending Tests4Py by a new bug is straightforward, fully integrating all of its capabilities requires manual effort. This effort depends, among others, on the design of the harness or the complexity of the required inputs.

2.2 Oracles

The harnesses we have implemented are pivotal in conveying the necessary information to discern a failure. In alignment with this objective, we have incorporated a unique testing *oracle* for each bug to determine if a system input incites the defect. The oracles are diverse, reflecting the varied nature of the projects and bugs. Each subject necessitates an individual oracle, resulting in many implementations. These oracles range from capturing the standard error stream and detecting specific exceptions to asserting the existence of files or directories and verifying the return values of particular functions. The choice of implementation is dictated by what best aids in identifying the inherent defect. Accessing these oracles is a straightforward process. One can create a file with the test input and then incorporate Tests4Py's system test module. This design facilitates easy utilization and encourages the active use of the benchmark in varied testing scenarios.

2.3 Grammars

When introducing system inputs, an essential element is the requirement to adhere to a specific format to ensure they match the interface of the included bugs and are not dismissed during the input parsing and processing phase. Therefore, each subject in Tests4Py includes *grammars* that serve as input specifications for the system tests. Each grammar is valuable for validating the system tests created syntactically. Furthermore, grammars are used within the oracles to gather additional information, helping distinguish between successful and failing test runs. Tests4Py thus guarantees the integrity of the system inputs, increasing benchmarking validity.

2.4 System Tests

By utilizing the harnesses, oracles, and grammars, Tests4Py can accurately distinguish passing and failing tests at the system level. Furthermore, these components lay the groundwork for generating new test cases. Not only can we validate these tests, but our thorough fault analysis also enabled the implementation of a targeted test generation for each subject. As a result, we obtained a precise understanding of how to either trigger or avoid the defect, enhancing the precision and utility of the generated tests. Test generation depends entirely on the specific project and the bug it targets.

The test generation is mostly hand-crafted to suit these dependencies while relying on known testing techniques like fuzzing to generate specific input components. For the example of bug #2 from the FastAPI project from above, this generated input describes what different applications, routers, requests, and responses exist for the test server set up by the harness.

2.5 Unit Tests

While system tests are crucial for facilitating test generation, there are requirements for specific methodologies that they cannot fulfill. For example, many automated program repair strategies heavily rely on unit tests. To accommodate this and to ensure that each subject can be thoroughly tested, we incorporated the ability to generate and execute unit tests into our benchmark. Figure 1 also delineates the unit test module of our benchmark. Instead of generating system inputs defined by a grammar, we directly produce Python code as a `unittest.TestCase`. This generated test case is then executed by the PYTEST testing framework as part of the entire test set or as individual tests. This enhancement broadens the scope of testing, making Tests4Py more versatile and comprehensive.

2.6 Usage

To install Tests4Py, run `pip install tests4py` from the command line. After installation, use commands like

(1) `t4p info` to retrieve information of the included projects;
(2) `t4p checkout -p FastAPI -i 2` to download bug #2 from the `FastAPI` project;
(3) `t4p build` from the generated directory to build the virtual environment and install the subject in it;
(4) `t4p systemtest generate -n 10` to generate ten system tests in the newly created folder; and
(5) `t4p systemtest test` to run these tests.

3 TESTS4PY USE CASES

We want to highlight and discuss several use cases for Tests4Py.

3.1 Evaluating Test Generation

Every subject in Tests4Py comes equipped with an oracle and input specification, providing a fertile ground for evaluating test generation techniques such as fuzzing. Unlike previous benchmarks that

relied on crashes or coverage to assess these techniques, Tests4Py showcases the ability of test generators to identify functional bugs. Despite the *oracle problem* representing a challenge in automatically categorizing the generated tests, Tests4Py offers more profound insight into the effectiveness of test generators. This approach could pave the way for a new breed of generators specifically targeting functional defects. The benchmark also presents a variety of real-world faults, allowing for the evaluation of test generation techniques across different bug types and analyzing their effectiveness in uncovering and diagnosing these issues. Given the capability of Tests4Py to generate tests, it provides ample material to study and learn from when setting up a test generator—for example, when employing symbolic execution.

3.2 Mining Input Grammars

The included input grammar allows to validate input specification mining approaches and serves as a ground truth to calculate precision and recall for the derived specification, i.e., how many from the specification derived inputs are correct according to the included input grammar and how many actual inputs are accepted by the mined specification.

3.3 Driving Automatic Program Repair

The Tests4Py benchmark holds great potential for enhancing automatic program repair (APR) methodologies. APR is an exciting area of research focused on devising techniques and tools that autonomously rectify software bugs. Typically, APR strategies hinge on pinpointing the root cause of a defect, generating a fix, and validating it through test leveraging.

With its capacity to generate new tests, Tests4Py offers APR a dynamic platform to probe how different test sets featuring varying attributes impact APR. Users can generate diverse test sets of different sizes and proportions of failing and passing tests and then apply APR to generate patches. The accuracy of these generated solutions can then be evaluated using a concealed test set. This investigation can provide insights into the properties a test set must possess to yield adequate repairs through APR.

Additionally, the subject-specific oracles provided in Tests4Py can be used to evaluate the synergistic combination of APR and test generation. Existing research, such as the work by Yang et al. [21], has already integrated test generation and APR to minimize overfitting, albeit relying on weaker oracles. However, our benchmark enables the combination of APR and functional test generation. Further exploration in this direction could significantly advance the research in this domain.

3.4 Improving Automated Debugging

Tests4Py also holds significant potential for refining automated debugging techniques, most of which depend on the size and quality of a test set. The benchmark's ability to *generate new tests as needed* opens the door for deeper investigation into the requirements of a test set to unveil the fault-causing statements in a program. As Tests4Py includes a patch for each subject and the possibility of retrieving the faulty statements from it, assessing the accuracy of the identified statements is a straightforward task. Some debugging techniques already integrate test generation to refine their hypotheses, for example, ALHAZEN [9] or AVICENNA [3]. These capabilities make Tests4Py a benchmark of choice for evaluating such methodologies in the context of real-world functional bugs. Moreover, Tests4Py provides an embedded statistical fault localization with the integration of SFLKit [17] that enables further research in this direction of automated debugging.

4 THREATS TO VALIDITY

For each bug in our benchmark, we investigated its causes to ensure the quality of the created test cases and the test generation. However, we may include tests that pass or fail for reasons other than the underlying defect. To counter this threat, we verified that all tests (included or generated) pass or fail based on the oracle according to their labels.

Even though we tried to stay as close as possible to the underlying defects, we may implement code that does not reveal the original bug. To minimize this risk, we verified the execution and oracles of all created test cases for correctness.

5 RELATED WORK

Tests4Py bears significant parallels to BugsInPy [20], a source of inspiration and foundation for our work. BugsInPy contributed significantly as the pioneer benchmark of real-world faulty Python programs. Another Python program benchmark is Refactory [6], based on student assignments and designed to evaluate automated program repair. In the context of automated program repair, additional benchmarks such as Codeflaws [18] for C and Bears [10] for Java programs exist. The BugSwarm [19] benchmark comprises faulty programs and their fixes for Python and Java programs.

While we focus on Python, benchmarks are available for several other programming languages. A prominent example is Defects4J [8], a benchmark for faulty Java programs. Another benchmark for buggy Java programs is Bugs.jar [16], which comprises a staggering 1,158 subjects.

In test generation, Google's FuzzBench [11] stands out as a popular benchmark for evaluating and comparing fuzz testing based on the achieved coverage. In debugging, we want to showcase the work by Böhme et al. [2]. Their efforts culminated in a benchmark where human experts analyzed defects and provided a diagnosis for each of the included subjects.

6 CONCLUSION AND FUTURE WORK

We introduce Tests4Py, a benchmark of real-world Python bugs. Each bug in this benchmark is accompanied by a test oracle and the capability to generate and execute system and unit tests. Tests4Py establishes an easy-to-use, readily integrable architecture ready for everyday use.

Our future work will focus on the following topics: First, we continue expanding our benchmark by including even more bugs from the BugsInPy database—eventually assimilating all subjects into Tests4Py. We are also designing two studies to explore the impact of various test set properties on automatic program repair and statistical fault localization, as discussed in Sections 3.1 and 3.3.

Tests4Py is available as open source under

https://github.com/smythi93/Tests4Py

REFERENCES

[1] R. Amine. youtube-dl, 2021. https://github.com/ytdl-org/youtube-dl.

[2] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller. Where is the bug and how is it fixed? An experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 117–128, 2017.

[3] M. Eberlein, M. Smytzek, D. Steinhöfel, L. Grunske, and A. Zeller. Semantic debugging. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 438–449, New York, NY, USA, 2023. Association for Computing Machinery.

[4] A. R. Greenfeld. Cookiecutter, 2022. https://www.cookiecutter.io/.

[5] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah. BugsJS: a benchmark of JavaScript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101, 2019.

[6] 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)*, pages 388–398, 2019.

[7] V. Iakovlev. The fuck, 2022. https://github.com/nvbn/thefuck.

[8] 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*, ISSTA 2014, page 437–440, 2014.

[9] A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller. When does my program do this? Learning circumstances of software behavior. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1228–1239, 2020.

[10] F. Madeiral, S. Urli, M. Maia, and M. Monperrus. BEARS: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478, 2019.

[11] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya. FuzzBench: An open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1393–1403, 2021.

[12] S. C. Organization. Sanic, 2024. https://sanic.dev.

[13] R. Rachum, A. Hall, I. Yanokura, et al. Pysnooper: Never use print for debugging again, jun 2019. https://github.com/cool-RR/PySnooper.

[14] S. Ramírez. FastAPI, 2018. https://fastapi.tiangolo.com/.

[15] J. Roztocil. Httpie, 2022. https://httpie.io/.

[16] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad. Bugs.Jar: A large-scale, diverse dataset of real-world Java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 10–13, 2018.

[17] M. Smytzek and A. Zeller. Sflkit: a workbench for statistical fault localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1701–1705, New York, NY, USA, 2022. Association for Computing Machinery.

[18] 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)*, pages 180–182, 2017.

[19] 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*, pages 339–349. IEEE / ACM, 2019.

[20] 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 P. Devanbu, M. B. Cohen, and T. Zimmermann, editors, *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*, pages 1556–1560, 2020.

[21] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 831–841, 2017.