
# DyPyBench: A Benchmark of Executable Python Software

ISLEM BOUZENIA, University of Stuttgart, Germany
BAJAJ PIYUSH KRISHAN, University of Stuttgart, Germany
MICHAEL PRADEL, University of Stuttgart, Germany

Python has emerged as one of the most popular programming languages, extensively utilized in domains such as machine learning, data analysis, and web applications. Python's dynamic nature and extensive usage make it an attractive candidate for dynamic program analysis. However, unlike for other popular languages, there currently is no comprehensive benchmark suite of executable Python projects, which hinders the development of dynamic analyses. This work addresses this gap by presenting DyPyBench, the first benchmark of Python projects that is large-scale, diverse, ready-to-run (i.e., with fully configured and prepared test suites), and ready-to-analyze (by integrating with the DynaPyt dynamic analysis framework). The benchmark encompasses 50 popular open-source projects from various application domains, with a total of 681k lines of Python code, and 30k test cases. DyPyBench enables various applications in testing and dynamic analysis, of which we explore three in this work: (i) Gathering dynamic call graphs and empirically comparing them to statically computed call graphs, which exposes and quantifies limitations of existing call graph construction techniques for Python. (ii) Using DyPyBench to build a training data set for LExecutor, a neural model that learns to predict values that otherwise would be missing at runtime. (iii) Using dynamically gathered execution traces to mine API usage specifications, which establishes a baseline for future work on specification mining for Python. We envision DyPyBench to provide a basis for other dynamic analyses and for studying the runtime behavior of Python code.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**.

Additional Key Words and Phrases: Python benchmark, dynamic analysis, executable, collection of software repositories

**ACM Reference Format:**
Islem Bouzenia, Bajaj Piyush Krishan, and Michael Pradel. 2024. DyPyBench: A Benchmark of Executable Python Software. *Proc. ACM Softw. Eng.* 1, FSE, Article 16 (July 2024), 21 pages. https://doi.org/10.1145/3643742

## 1 INTRODUCTION

Python has grown rapidly in recent years to become one of the most frequently used programming languages in a variety of fields, including machine learning, data analysis, and web applications. For example, a 2022 report on programming languages used in projects hosted on GitHub lists Python as the second-most popular of all languages.[1] The success of Python can be attributed to its simplicity, flexibility, and extensive collection of libraries and frameworks. Python's dynamic nature, which allows for dynamic type checking and modification of object structures during runtime, further enhances its appeal to developers.

[1] https://octoverse.github.com/2022/top-programming-languages

Authors' addresses: Islem Bouzenia, University of Stuttgart, Suttgart, Germany, fi_bouzenia@esi.dz; Bajaj Piyush Krishan, University of Stuttgart, Stuttgart, Germany, st173644@stud.uni-stuttgart.de; Michael Pradel, University of Stuttgart, Suttgart, Germany, michael@binaervarianz.de.


ACM 2994-970X/2024/7-ART16
https://doi.org/10.1145/3643742

The dynamic nature of Python also presents challenges when it comes to program analysis, error detection, security assessment, and performance optimization. Various dynamic program analysis techniques have been developed to address these challenges, e.g., to detect bugs [66], to enforce differential privacy [1], to slice programs [15], or to understand the performance of a program via profiling [7]. These techniques play a crucial role in identifying programming errors, security vulnerabilities, and performance bottlenecks. To help develop, evaluate, and compare dynamic analysis techniques, other popular programming languages have benchmark suites of executable code. For example, DaCapo for Java [10], SPEC CPU for C/C++ [31], and Da Capo con Scala [58] have contributed significantly to work by both researchers and practitioners on their respective programming languages.

Despite the popularity and importance of Python, there currently is no equivalent benchmark suite for Python, which hinders the progress and evaluation of dynamic analyses. Instead, researchers, who want to work on a novel dynamic program analysis for Python, would like to study the runtime behavior of Python code, or want to gather data from executions of Python software, must spend significant efforts on creating a suitable set of executable programs. As a result, many evaluations that involve dynamic analysis of Python are performed on a relatively small set of benchmark programs, e.g., nine projects [21], five projects [59], six projects [16], and eleven projects [66]. Moreover, since every team working on Python creates their own benchmark, it is difficult to compare different techniques and results with each other.

In response to this gap, this paper presents DyPyBench, a novel benchmark of executable Python projects. We envision our benchmark to facilitate future work on dynamically analyzing Python. To this end, the benchmark is the first to offer four important properties – large-scale, diverse, ready-to-run, and ready-to-analyze – as described in the following:

- *Large-scale.* To offer users an extensive collection of executable code, the benchmark includes a substantial number of popular and non-trivial projects. The definition of "large-scale" may vary depending on the specific use case. In this work, we have curated a dataset comprising 50 projects, collectively amounting to 681k lines of code. This size is an order of magnitude larger than the benchmarks considered in previous evaluations, as listed above.
- *Diverse.* Given Python's numerous applications, it is critical that a benchmark covers a wide number of application domains, which each have their own code style and libraries. DyPyBench attempts to represent the breadth and diversity of Python usage by including projects systematically sampled from different application domains, enabling extensive assessments and studies of code behavior across different contexts.
- *Ready-to-run.* Executing the code of a real-world, complex project is non-trivial, as it requires a suitable project configuration, installing third-party dependencies, and inputs that exercise the code. The effort spent on setting up projects can be significant, especially for users who are not familiar with the project, and usually, this effort needs to be spent again and again for each new project. DyPyBench comes with all projects already set up and uses the test suites of these projects to provide inputs. To facilitate running the code irrespective of variations in project dependencies, configurations, and execution environments, DyPyBench provides a unified interface that allows users to run the benchmark with a single command. Moreover, we provide a Docker image that encapsulates the benchmark and all its dependencies, ensuring reproducibility and longevity.
- *Ready-to-analyze.* To facilitate the dynamic analysis of the projects in DyPyBench, the benchmark integrates with DynaPyt [21], a general-purpose dynamic analysis framework for Python. The integration allows for instrumenting and analyzing projects with a single

command, making it straightforward to apply a new dynamic analysis to a wide range of projects.

To evaluate the usefulness and practicality of DyPyBench, we perform a series of experiments covering a wide range of research areas. In particular, we apply our benchmark in three usage scenarios:

(1) We use DyPyBench to empirically compare statically and dynamically created call graphs. To this end, we use the dynamic analysis infrastructure built into DyPyBench to gather dynamic call graphs and compare them to the results of a state of the art static call graph generator [54]. Our results help understand the strengths and weaknesses of both approaches, and provide some guidelines for future work on call graph construction.
(2) We use DyPyBench to create training data for LExecutor [59], a recent neural network-based technique for learning-guided execution. In this scenario, we apply the existing source code instrumentation tool provided by LExecutor to the code in DyPyBench and then execute our benchmark. By using the resulting 436,355 data examples as training data for the LExecutor model, we find that more data helps improve the accuracy of the model.
(3) Finally, we use DyPyBench to mine specifications from execution traces. For this application, we again build upon the dynamic analysis support built into our benchmark, this time to extract traces of function calls. DyPyBench provided thousands of sequences of calls that allowed to extract some meaningful patterns, which sets a baseline for future work on dynamic specification mining for Python.

For all three of these applications, one would usually have to spend a significant effort on setting up suitable projects and finding inputs, e.g., in the form of test suites, for exercising their code. Instead, DyPyBench provides this setup, which hugely facilitates these and future dynamic analyses for Python.

In summary our work contributes the following:

- We address the lack of a comprehensive benchmark suite of executable Python projects by creating DyPyBench.
- We integrate our benchmark with the general-purpose dynamic analysis framework DynaPyt [21], which allows for performing arbitrary dynamic analyses on the entire benchmark with minimal effort.
- We illustrate the usefulness of DyPyBench in three application scenarios.

## 2 METHODOLOGY

This section describes our methodology for creating a benchmark of executable Python software. We begin by presenting the criteria for selecting projects to include in the benchmark, and then describe our methodology for making these projects ready-to-execute and ready-to-analyze.

### 2.1 Selecting Projects

To ensure the diversity and representativeness of the benchmark, we select projects from the Awesome Python repository,[2] which is a curated list of popular, open-source Python projects. The Awesome Python repository has 173k stars on GitHub and more than 400 contributors, which are indicators of its popularity and adoption by the community. The curated list contains 679 projects, including libraries, frameworks, and applications, which are classified into 90 categories.

We select a subset of the projects from the curated list, following three criteria designed to ensure the diversity, quality, and representativeness of our benchmark. First, to increase the quality and

[2] https://github.com/vinta/awesome-python

relevance of the benchmark, we consider only projects that have at least 500 stars on their respective GitHub repositories. GitHub stars serve as an indication of a project's popularity, reflecting its usefulness and community support. Second, to make DyPyBench diverse, we sample projects from different categories in the Awesome Python list. Each category covers a different application domain, such as web crawling, machine learning, and robotics. We randomly sample from all projects in the Awesome Python list, with the constraint to pick at most one project from each category. Finally, as our goal is to create an executable benchmark, we focus on projects with test suites. Specifically, we focus on test suites that can be executed with pytest, i.e., one of the most popular testing frameworks for Python. The selected tests include tests written based on Python's built-in `unittest` framework, as well as tests written with pytest itself.

Alternative to sampling projects from the Awesome Python list, we could have selected the most downloaded projects from the Python Package Index (PyPI). However, we observe that the most downloaded projects do not fully represent the diversity of the Python ecosystem, but are biased toward kinds of projects that many others depend on, such as tools to build and set up projects. Instead, the Awesome Python list provides an independently curated list of projects grouped into a diverse set of application domains. A downside of using the Awesome Python list is that it focuses on open-source projects only, and hence, may not be fully representative of the entire Python ecosystem. We decided to focus on open-source projects, as they are more likely to be used in research and are more accessible to the community.

### 2.2 Enabling Execution

As executability is an important property of our benchmark, we invest significant efforts into automating the process of setting up the projects and their test suites. Our benchmark can be set up in two ways. First, we provide a Docker image that encapsulates the benchmark and all its dependencies. This image can be used to run the benchmark on any machine that supports Docker, ensuring reproducibility and longevity. Second, we provide a command-line interface that allows users to automatically install and set up the benchmark and its projects. This process involves cloning the projects, installing their dependencies, and configuring their test suites. To the extent possible, DyPyBench invokes standard Python tools for installing, building, and testing projects. However, some projects require various non-standard steps, such as installing unspecified third-party dependencies or slightly adapting the developer-provided test suites. To address these cases, we manually inspect the projects and make the necessary adjustments to ensure that they are ready-to-execute. From the perspective of a user of our benchmark, this process is transparent, as the benchmark provides a single command for setting up and executing all or individual projects.

The following describes the steps of our automated setup process in detail. At first, we clone the project repositories from GitHub. To ensure reproducibility, we clone the repositories at a specific commit, which is the latest commit on or before January 18, 2023. Then, we create a Python virtual environment for each project, which ensures that the project's dependencies are installed in an isolated manner. Next, we install the project's dependencies within the virtual environment. If available, this step uses the project's `requirements.txt` file, which we augment with additional dependencies if necessary. Finally, we install the project itself, ensuring that it is properly set up within its dedicated virtual environment.

To ensure that the test suites of the projects are ready-to-execute, we configure them to run with pytest. This step involves specifying the locations of the test cases within the project. For some projects, additional dependencies may be required to execute the test suite. These dependencies are often not mentioned in the requirements file but can be found in the project's README instructions. To handle this scenario, we manually collect the necessary dependencies for each project. Furthermore, we overwrite specific test files of some projects, e.g., for tests that run into

Table 1. Projects in the benchmark.

| Project | Domain | Project | Domain | Project | Domain |
|---|---|---|---|---|---|
| akshare | Downloader | grab | Web crawling | python-decouple | Configuration |
| arrow | Date & time | graphene | GraphQL | python-diskcache | Caching |
| black | Code formatter | gunicorn | WSGI servers | python-future | Compatibility |
| blinker | Misc. | html2text | Web content | python-patterns | Algorithms |
| supervisor | DevOps | lektor | Static sites | PythonRobotics | Robotics |
| celery | Task queues | marshmallow | Serialization | pyvips | Parallel img. proc. |
| cerberus | Data validation | mezzanine | CMS | requests | HTTP clients |
| click | CLI app dev. | moviepy | Video | schedule | Job schedul. |
| code2flow | Code analysis | pdoc | Documentation | seaborn | Data visual. |
| delegator.py | Processes | pickledb | Database | streamparse | Distr. comput. |
| dh-virtualenv | Distribution | pillow | Image proc. | structlog | Logging |
| elasticsearch-dsl | Search | pudb | Debugging | thefuck | Cmd-line tools |
| errbot | ChatOps tools | pydub | Audio | uvicorn | ASGI servers |
| flask-api | RESTful API | pyfilesystem2 | Files | webassets | Web asset mgmt. |
| funcy | Functl. progr. | pyjwt | Authentic. | wtforms | Forms |
| furl | URL manip. | pypdf | Format proc. | zerorpc-python | RPC servers |
| geopy | Geolocation | pyquery | HTML manip. | | |

an infinite loop or that consume an unacceptably high amount of memory. This affects a total of 51 test files across 18 projects. Finally, we validate that the test suites are ready-to-execute by running all the above steps on a fresh machine and by checking that the tests execute successfully.

### 2.3 Enabling Dynamic Analysis

One of the main goals of this work is to facilitate dynamic analysis of Python projects. To this end, the benchmark integrates with DynaPyt [21], a general-purpose dynamic analysis framework for Python. DynaPyt supports a range of analyses by providing an API of callbacks that are triggered whenever a specific kind of runtime event occurs, such as a function call, the creation of a new object, or a binary operation. We show two DynaPyt-based applications of DyPyBench in Sections 4.1 and 4.3.

Performing a dynamic analysis with DynaPyt requires instrumenting the source code of the target project. DyPyBench facilitates this step by providing a single command for instrumenting all or selected projects in the benchmark. The command applies DynaPyt's instrumentation tool to the source code of the selected projects. DyPyBench offers two variants of this command: One variant instruments all Python code in a project, which is useful to analyze the executions of both the tests and the project's main code. The other variant instruments only the application code, i.e., excluding the test suite. We use the latter variant, e.g., for performing a dynamic call graph analysis (Section 4.1). Finally, DyPyBench provides a command for running the instrumented projects, which automatically invokes the callbacks into a user-provided dynamic analysis whenever a runtime event occurs that is of interest to the analysis.

Beyond dynamic analyses built with DynaPyt, the benchmark can, of course, also be used with other dynamic analysis tools. For example, one of our applications (Section 4.2) is based on a custom, instrumentation-based dynamic analysis. Furthermore, the benchmark could also be easily run with dynamic analysis tools that do not require source-level instrumentation, but that instead build upon Python's built-in tracing library `sys.settrace`.

Table 2. Properties of DyPyBench.

| Metric | Value |
|---|---|
| Projects | 50 |
| Lines of code | 681k |
| Test cases: | |
| Total | 29,511 |
| Passing | 27,569 |
| Failing | 270 |
| Skipped | 1,672 |
| Lines of executed code: | |
| Total lines | 558k |
| Coverage | 82% |
| Execution time: | |
| Avg./project | 71 seconds |
| Min. | 1 seconds |
| Max. | 1,362 seconds |
| Total | 3,568 seconds |

## 3 DYPYBENCH

By applying the methodology described in the previous section, we create DyPyBench, a large-scale, diverse, ready-to-run, and ready-to-analyze benchmark of Python projects. This section presents the composition of the benchmark (Section 3.1), its runtime properties (Section 3.2), and details of its implementation (Section 3.3).

### 3.1 Composition of the Benchmark

DyPyBench consists of 50 projects, listed in Table 1. The code in these projects sums up to 681k lines of Python code, which can be considered large-scale in comparison to current community standards. For comparison, recent papers on dynamic analyses for Python are usually evaluated with an order of magnitude fewer projects, e.g., nine projects [21], five projects [59], six projects [16], and eleven projects [66]. Each project in DyPyBench covers a different application domain, where the domains are determined by the Awesome Python repository (Section 2.1). Table 1 lists the 50 domains along with the projects included in the benchmark. Within the selected projects, 94.5% of files are Python files, whereas some files are written in other languages (2.0% JavaScript, 1.2% HTML, 0.8% C).

### 3.2 Runtime Properties of the Benchmark

Because a key property of DyPyBench is to be executable, the following takes a deeper look into the runtime properties of the benchmark. Table 2 summarizes the key figures. The projects in DyPyBench come with a comprehensive set of test cases, totaling 29,511. The number of test cases per project ranges from 1 to 3,947, with an average of 590 test cases per project.

Out of all test cases in the benchmark, 27,569 test cases pass, while 270 test cases fail and the rest gets skipped. DyPyBench has 31 projects with no failing test cases, while 13 projects have between 1 and 10 failing tests. Figure 1 shows the number of successful, failing and skipped test cases per project. The pass rate, i.e., the percentage of passing test cases among all test cases, across all projects in DyPyBench is 93%. To better understand the distribution of test case outcomes across projects, Figure 2 shows the cumulative distribution of test case pass rate. The plot shows that 41

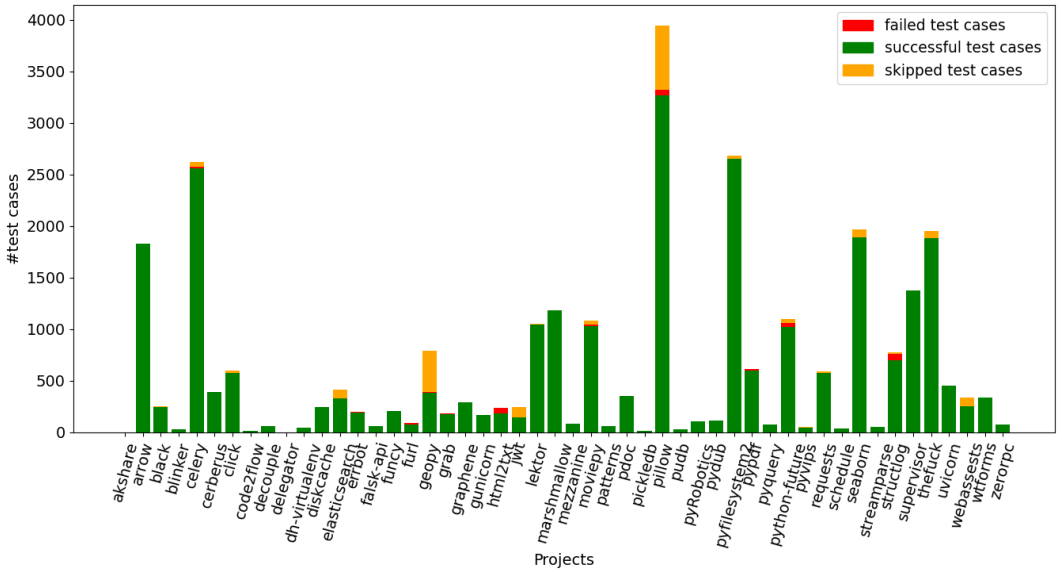


Fig. 1. Number of successful, failed, and skipped test cases for each project of the benchmark.

projects have a pass rate exceeding 90%, and 46 projects have 80% or more test cases passing. This high pass rate not only allows users of our benchmark to execute a lot of code, but also to observe mostly valid executions.

There are two main reasons why some projects have failing test cases. First, some tests fail due to incompatible hardware. For example, some test cases are designed to run on a specific hardware, e.g., a GPU or a specific CPU architecture. Our Docker-based setup provides software abstraction, but not hardware abstraction, which can lead to test failures depending on the used machine. Second, some tests fail due to incompatible or unavailable software. Examples include test cases designed to run on a particular operating system and test cases that use online services that require authentication. Because our Docker-based setup provides a single consistent software environment, including a specific operating system, tests that require another software environment are doomed to fail.

Figure 3 shows the distribution of covered and uncovered statements as a result of test execution for each project. The figure illustrates that the number of missed statements is noticeably low for the vast majority of the projects. Furthermore, 27 projects have coverage levels that exceed 85%. This high coverage not only demonstrates comprehensive testing within our benchmark, but it also plays an important role in providing diverse and meaningful data for a variety of applications, as discussed in more detail in Section 4.

Finally, we report the wall-clock time required to execute the test suites of the projects in DyPyBench. The following numbers are obtained on a standard computer with an Intel Xeon CPU running at 2.10GHz and 32GB of memory. We launch all the test suites on a single CPU core. The time to execute the test suite of single project ranges from 1 second to 1,362 seconds. Figure 4 illustrates the runtime for each project (rounded to full seconds), as well as the corresponding number of test cases. The total runtime for all projects combined amounts to 3,568 seconds, i.e., about one hour. The average time required to execute a complete test suite is 71 seconds. Notably, 39 projects can complete their test suite within 50 seconds or less, while there are four projects with

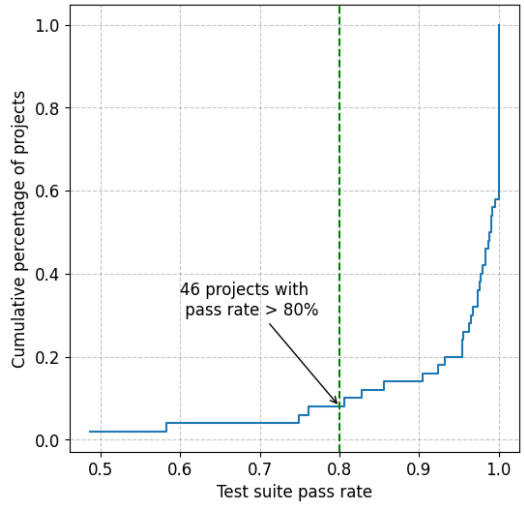


Fig. 2. Cumulative distribution of test suite pass rates for all projects.

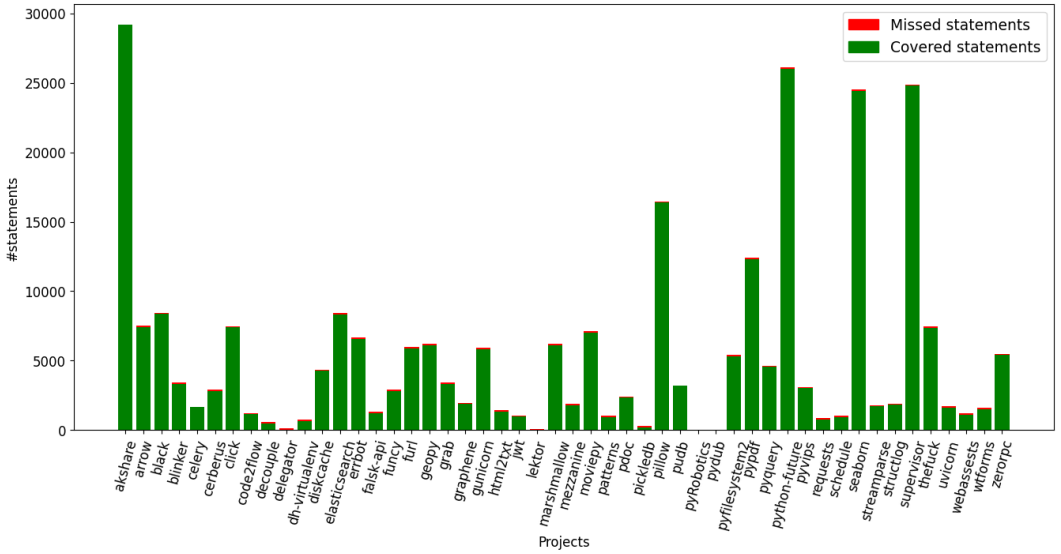


Fig. 3. Distribution of covered statements vs uncovered statements during the execution of test suites for each project.

runtimes exceeding 200 seconds. The diversity in project sizes and runtimes of their corresponding test suites allows users of DyPyBench to sample projects with different characteristics, depending on the specific needs.

### 3.3 Implementation

Inspired by other widely used benchmarks in the community [18, 33], DyPyBench comes with a convenient command-line interface that allows users to interact with the benchmark and its projects.

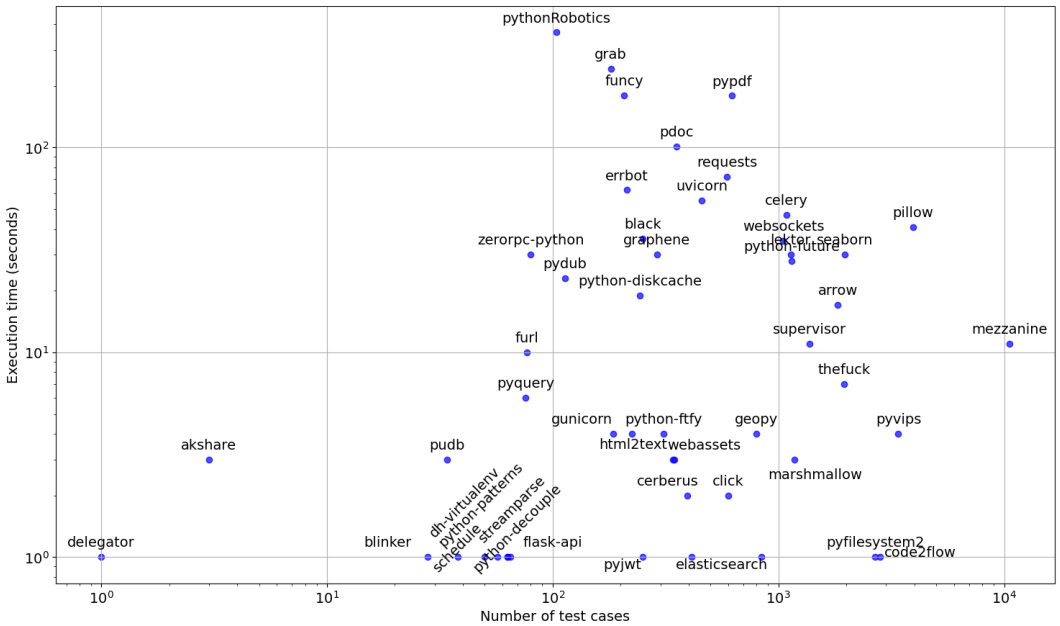


Fig. 4. Number of test cases vs. execution time for each projects.

For example, the interface supports running all or selected projects, instrumenting their source code, or applying a dynamic analysis to them. For reproducibility and to ensure the longevity of the benchmark, we provide an Ubuntu 20.04-based Docker image that encapsulates the benchmark and all its dependencies. We use Python 3.10 as the default Python version across all our Python environments, considering its widespread usage and stability. To support dynamic analysis, the benchmark integrates with DynaPyt version 0.3.1.

Currently, DyPyBench contains 50 projects, but it is designed to be easily extensible. The benchmark can be extended by adding the GitHub URL of new projects to a specific text file. Our installation script will try to install and configure the new projects based on frequently observed setup patterns. In case our script fails to automatically install a project, the user can edit the installation script to add a special case.

All our experiments are conducted on a machine with an Intel Xeon Gold 6230 CPU, 64GB RAM limit and 300GB disk space. The disk space is needed only when performing the applications described below on all projects at once.

## 4 APPLICATIONS

We demonstrate the usability and usefulness of DyPyBench by studying three applications of the benchmark. First, we compare dynamic and static call graphs extracted for the programs in DyPyBench (Section 4.1). Second, we use the benchmark to generate training data for LExecutor [59], a machine learning-based technique for executing incomplete code snippets (Section 4.2). Finally, we dive into the domain of mining specifications from execution traces (Section 4.3).

### 4.1 Studying Static and Dynamic Call Graphs

Call graphs are a fundamental building block for various program analyses. Because of their importance, various algorithms for computing call graphs have been proposed, e.g., for C++ [6],

Java [61], Scala [2], JavaScript [22], and Python [54]. To better understand the properties of such algorithms, several studies compare the call graphs produced by different analyses with each other, e.g., for C [40], Java [51, 60], JavaScript [13], and WebAssembly [35]. However, to the best of our knowledge, there is no study that compares dynamic and static call graphs for Python.

The following application of DyPyBench fills this gap by comparing call graphs generated by PyCG [54], a state-of-the-art static call graph analysis for Python, with dynamic call graphs generated by DynaPyt [21]. Because DyPyBench is ready-to-analyze and already integrated with DynaPyt, obtaining dynamic call graphs of the 50 projects in our benchmark requires relatively little effort.

*4.1.1 Experimental Setup.* A call graph $G = (F, E)$ consists of set $F$ of functions and a set $E$ of edges. An edge $e \in F \times F$ represents the fact that a caller function invokes a callee function. If a function is called multiple times by the same caller, the call graph contains only a single between the caller and the callee. To identify functions, both DynaPyt and PyCG resolve the fully qualified name, including the module name. For a given Python project, we construct a static call graph $G_{static}$ by applying PyCG to the project's source code. As mentioned in the documentation of PyCG, we specify the project path and the set of all Python files in the project. To generate a dynamic call graph $G_{dynamic}$, we use the existing dynamic call graph analysis provided by DynaPyt [21]. To this end, we instrument and then execute the projects in our benchmark, as described in Section 2.3. We set a time limit of six hours and a memory limit of 60GB for the analysis of a single project. Furthermore, we set a timeout of 30s for each individual test case when run with the DynaPyt analysis. Moreover, we ignore the results of an analysis on a project in case the analysis itself crashes.

Once we obtain the call graphs $G_{static}$ and $G_{dynamic}$ for the projects in DyPyBench, we compare them with each other. For a quantitative comparison, we consider three sets: (i) the set of call graph edges, (ii) the set of all callers, i.e., nodes with at least one outgoing edge, and (iii) the set of callees, i.e., nodes with at least one incoming edge. For both analyses, we ignore calls made by the underlying unit testing tool and calls made directly inside a test case, because we are interested in the calls made within the analyzed project. For example, if a test file `test_app.py` contains a test case function called `test_1`, then the call to `test_1` appears neither in $G_{static}$ nor $G_{dynamic}$. Likewise, any calls made directly inside `test_1` are also ignored. In contrast, calls made by functions invoked by `test_1` are added to the call graphs. In addition to the quantitative comparison, we manually inspect a sample of 97 differences between the static and dynamic call graphs. The goal of this inspection is to understand the differences and to classify them by their root cause.

*4.1.2 Results.*

*Static call graphs.* PyCG successfully creates a call graph for 39 out of 50 projects. The missing projects are due to six projects that exceed the six-hour timeout, three projects that exceed the memory limit of 60GB, and two projects where PyCG crashes with an exception. Across the 39 successful projects, the static call graphs have a total of 60,565 edges, with an average of 1,552 edges per project. These edges are between 33,795 unique callers and 10,537 unique callees, where uniqueness is computed at the project level, i.e., if the same function appears in two different projects we count it twice.

*Dynamic call graphs.* DynaPyt successfully creates dynamic call graphs for all 50 projects. On average, executing DyPyBench while computing the call graph takes 215 minutes per project. This time is longer than simply executing DyPyBench without computing call graphs because DynaPyt introduces additional overhead by instrumenting the code. The dynamic call graphs have a total of 9,575 edges, giving an average of 191 edges per project. These edges are between 3,078 unique

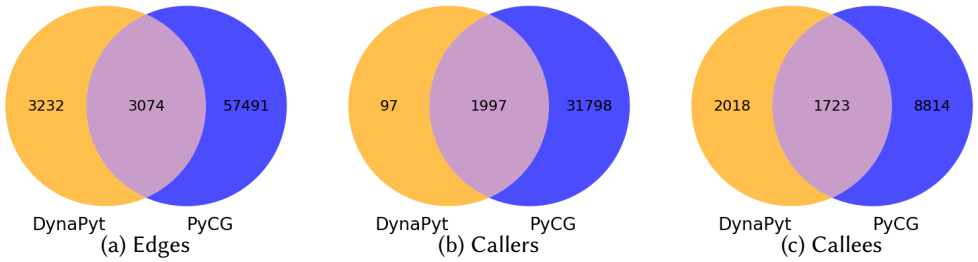


Fig. 5. Venn diagrams of edges, callers, and callees in dynamic call graphs (left) and static call graphs (right).

callers and 5,384 unique callees. Considering only the 39 projects where PyCG runs successfully, DynaPyt generates 6,306 edges between 2,049 callers and 3,741 callees.

*Comparison of edges.* Figure 5a compares the edges found in the static and dynamic call graphs across the 39 projects that could be successfully analyzed by both the static and dynamic analysis. The static call graphs have many more edges than their dynamic counterparts, which is expected because the static analysis considers all code of a project, whereas the dynamic analysis focuses on code exercised by the tests in DyPyBench. Surprisingly, many edges found by the dynamic analysis are not detected by the static analysis. Ideally, a static call graph should be sound, i.e., include all call edges that can occur at runtime. Instead, we find that only 49% of the edges present within DynaPyt's call graphs are within the set of edges found in PyCG's call graphs.

*Comparison of callers.* To better understand the differences between the static and dynamic call graphs, we compare their sets of callers and callees. As illustrated in Figure 5b, we notice a big difference when comparing the sets of callers derived from PyCG and DynaPyt. While almost all the callers collected by DynaPyt (95%) are present in the set of callers identified by PyCG, the static call graph contains many callers missed by the dynamic analysis. We attribute this difference to the limited code coverage in the exercised tests, which does not exercise all possible behavior of the projects.

*Comparison of callees.* When checking the set of callees, an even more important disparity emerges between DynaPyt and PyCG, as shown in Figure 5c. Only 48.5% of DynaPyt's callees are included in the set of PyCG's callees. That is, the static call graph is missing many called functions, even though DyPyBench offers evidence in the form of actual executions that these functions get called.

*Root causes of differences.* To understand the reasons that cause the static and dynamic call graphs to differ, we first automatically analyze why edges are missing, and then manually inspect a sample of missing callers and callees. Our exploration begins with an examination of edges in the dynamic call graph ($G_{dynamic}$) that do not find correspondence in the static call graph ($G_{static}$). Remarkably, 51% of the edges in $G_{dynamic}$ remain unrepresented in $G_{static}$. A detailed analysis of these disparities reveals that for 92% of such edges, the reason is that the callee is missing in $G_{static}$ (while the callers exist). Furthermore, in 8% of cases, both the callers and callees coexist within $G_{static}$, albeit without an edge connecting them. We also analyze $G_{static}$ edges that do not appear in $G_{dynamic}$. This analysis shows that for 79% of such edges both the callers and the callees are missing from the dynamic call graph, while for 20% of these edges only the callees are missing from $G_{dynamic}$.

Next, we manually inspect disparities in callers, more specifically the 5% of callers appearing in $G_{dynamic}$ but not in $G_{static}$. Our manual inspection shows that the mismatch is caused by differences

in the way DynaPyt and PyCG refer to the fully qualified name of some callers. For example, within the Flask project, there is a wrapper around the `Requests` library, which makes PyCG resolve the function to `FlaskAPI.Requests.something`. Instead, DynaPyt refers directly to the original caller, without the wrapper, which results in the caller's name being `Requests.something`.

Finally, we analyze the callees captured by $G_{dynamic}$ but still missing from $G_{static}$. We identify two primary causes for callees missed by the static analysis:

- Our manual inspection shows that PyCG's is missing many calls to methods offered by Python's built-in types. For instance, a call `"abc".strip()` is adequately detected by DynaPyt, recording them it as a call to `str.strip`, but PyCG fails to capture the call. Based on a list of Python's built-in function names, we find that this particular phenomenon accounts for 59% of all callees missing in the static call graph. In addition, calls in the form of `super.method()` are also not included in the call graph of PyCG.
- Another reason is, again, the resolution of modules names, which causes differences in the fully qualified names of functions. The Flask example given above illustrates this problem. Differences in function name resolution account for roughly 12% of the observed mismatches.

*4.1.3 Implications.* To the best of our knowledge, the call graphs generated with DyPyBench are the first large-scale dataset of dynamic call graphs for Python. Such a dataset may serve as a ground truth of calls that a sound static analysis should at least detect. For example, our finding that PyCG is missing method calls made on objects of built-in types, such as strings, calls for considering such calls in future static call graph analyses. Moreover, the dataset provides a basis for a more detailed empirical study of call graph generation algorithms for Python. One challenge to be addressed in such a study is to define a uniform way of resolving functions into a fully qualified name, to avoid spurious differences between different call graphs for the same project. Furthermore, the dynamic call graphs can help evaluate and develop techniques for optimizing and pruning statically generated call graphs [63]. Finally, our observation that dynamic call graphs can be generated quicker than static call graphs for some projects motivates future work on more efficient static call graph algorithms.

### 4.2 Gathering Runtime Data for Training a Neural Model

Neural models of software are becoming increasingly popular for various applications [46], e.g., code completion, bug detection, and type prediction. While the majority of approaches focuses on training models on source code and other static artifacts of software, there is an emerging subfield that trains models on data gathered during the execution of programs [43–45, 59]. A major challenge in this line of work is to gather large-scale datasets of runtime data, which is required to obtain effective neural models. The following demonstrates that DyPyBench can help address this challenge.

*4.2.1 Experimental Setup.* As a case study, we use LExecutor [59], a recent technique that queries a neural model to predict otherwise missing values, and hence, allows for executing arbitrary code snippets. The neural model underlying LExecutor is trained on a dataset of value-use events. Each such event consists of a value observed during the execution of a program and the code context in which the value gets used. The original dataset used to train LExecutor consists of 214,365 value-use events, which were gathered from five projects.

To evaluate the usefulness of DyPyBench for training LExecutor, we use the benchmark to generate an additional dataset of value-use events. To this end, we instrument the projects in DyPyBench with the custom Python instrumentation that is part of LExecutor and then execute the instrumented benchmark, which yields a dataset of value-use events. We keep a random 5% of the

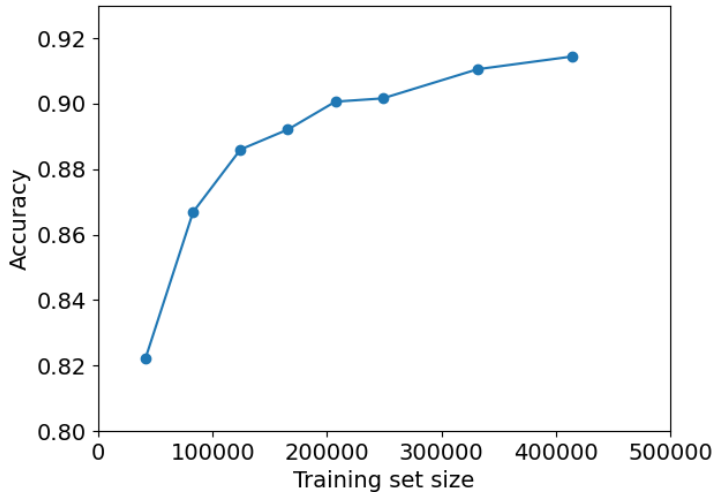


Fig. 6. Accuracy of LExecutor [59] model trained on DyPyBench-based data.

dataset for validation, and use the remaining 95% to train a new LExecutor model. During training, we follow the default setup in the original LExecutor work, i.e. its fine-grained value abstraction and the CodeT5 model, except that we train for five instead of ten epochs due to computational constraints.

*4.2.2 Results.* Executing DyPyBench results in a total of 436,355 value-use events. That is, thanks to DyPyBench, the available training data is more than double the size compared to the original LExecutor work.

To assess the impact of using training datasets of different sizes, we train different LExecutor models with increasingly large subsets of the combined training dataset. Specifically, we train models with subsets of 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100% of the data, respectively. We then measure accuracy on the held-out validation data. Following the original LExecutor work, we report top-1 accuracy, i.e., the percentage of predictions for which the correct value is the most likely prediction. Figure 6 shows the accuracy of the resulting models. The results illustrate that, perhaps unexpectedly, having more training data leads to a more accurate model, i.e., DyPyBench provides an easy way to improve the accuracy of the original model.

*4.2.3 Implications.* The continuously increasing interest in neural models of software calls for large-scale datasets of runtime data. DyPyBench can help address this need by providing a ready-to-analyze benchmark of executable Python projects. We envision future work to build upon our benchmark to generate various other kinds of runtime datasets, e.g., for predicting function names from traces of function executions, for predicting static type annotations from types observed at runtime, or for predicting the next value of a variable from its previous values.

### 4.3 Mining Specifications from Execution Traces

Specification mining [4] is a technique that extracts specifications from existing software. Various techniques for mining specifications have been proposed [52], many of which focus on temporal

constraints between function calls. Specification mining plays a role in various applications, including identifying inconsistencies and anomalies, establishing best practices, and gaining a deeper understanding about the common behavior of a given programs. The following demonstrates that DyPyBench can be used to mine specifications from execution traces of Python code.

*4.3.1 Experimental Setup.* In our experiment, we leverage call traces generated by DynaPyt to uncover patterns in function calls within the projects in DyPyBench. The process unfolds as follows:

(1) We customize DynaPyt's call graph analysis to capture traces of function calls, where each trace entry consists of a caller and a callee function.
(2) We instrument and execute the projects in DyPyBench to capture sequences of function calls for each project. Similar to the call graphs analysis, we only instrument and collect traces of the main source code of a project, without instrumenting the test cases themselves.
(3) To refine the data, we post-process the call sequences to inline callees into each caller functions, and to remove sequences that are incomplete due to exceptions.
(4) We apply an efficient implementation[3] of the PrefixSpan algorithm [29] to mine frequent patterns from the sequences of functions called within a specific function. We give PrefixSpan a list of sequences, each of which corresponds to the functions called within another function, in the order of their execution. We apply the mining algorithm to the aggregated set of sequences of all projects in DyPyBench. To reduce the computational cost of the mining algorithm, we consider only sequences of calls that have a length lower or equal to 100 calls, which corresponds to 94% of all call traces.
(5) Finally, we extract the top-100 patterns with a length greater than two.

Due to time constraints we limit the collection of call traces to 72 hours, which results in traces from 27 projects. While collecting traces using DynaPyt takes a relatively long time, generating the patterns with PrefixSpan is quite fast when invoked with no constraints on the collected patterns. For example, for all the sequences we collect, PrefixSpan is able to generate the top-100 patterns in less than ten seconds. However, this time is highly influenced by the length of sequences, which is the reason why we exclude sequences longer than 100 calls.

*4.3.2 Results.* In total, we extract 16,538 sequences, with an average of 612 sequences per project. The average length is eight, which means that the average function transitively performs eight function invocations. The CDF in Figure 7 shows the distribution of the length of the collected sequences. The plot indicates that most of the sequences (91%) contain less than ten calls. This implies that the most frequent patterns extracted by PrefixSpan are going to be relatively short patterns. In addition, the plot in Figure 8 illustrates the number of call sequences collected from each project, which ranges between tens of sequences and several thousands. The distribution, however, is far from uniform, which based on our initial results, leads to the patterns extracted from one project dominating the top-100 patterns that we mine from the entire set of projects. To prevent this bias, we set a limit on the number of sequences that we consider from each project during the pattern mining. We define the limit as the mean number of sequences (613) over all projects, plus one standard deviation (1,280), which totals to 1,893 sequences. Among the top-100 patterns, we randomly pick five patterns and present them in Table 3 alongside their frequency and an example of their occurrence in a project in DyPyBench. The patterns range from sequences of built-in functions and standard library functions (e.g., in rows 1 and 2) to library-specific patterns (e.g., in rows 3 and 4). While the extracted patterns look simple and straightforward, the code reflecting those patterns can be complicated due to multiple branches (e.g., in row 3 and 4) or

[3] https://github.com/chuanconggao/PrefixSpan-py

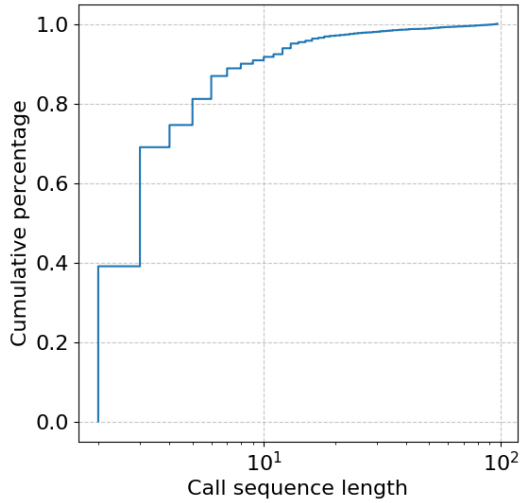


Fig. 7. Distribution of extracted calls sequences by length.

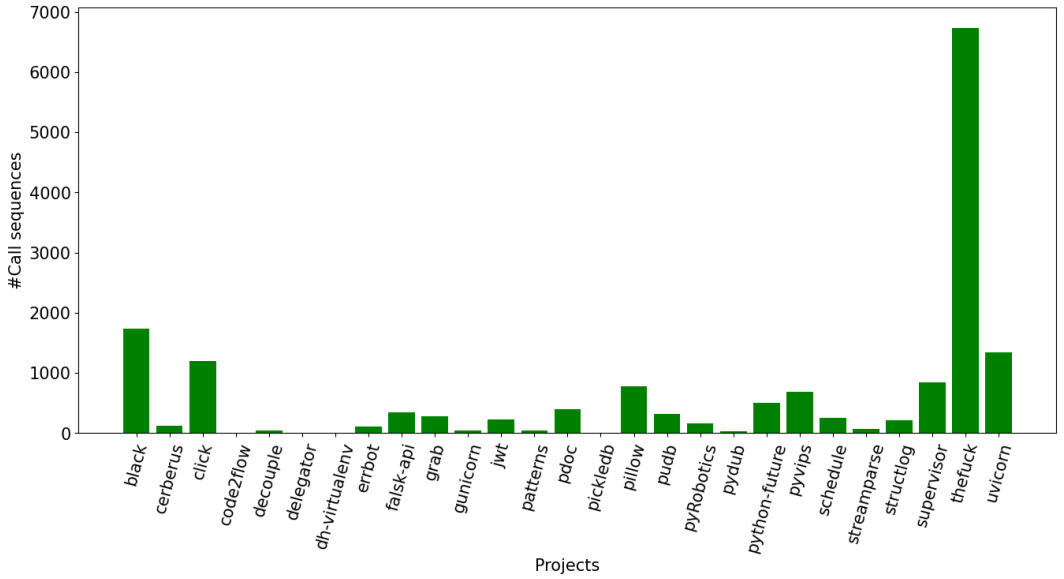


Fig. 8. Number of extracted sequences per project.

because they occur in a complex expressions, such as in row 2. Collecting execution traces and mining frequent patterns condenses such behavior into a temporal specification.

*4.3.3 Implications.* Despite many research results on specification mining in general [52], our work is the first application of specification mining to Python. DyPyBench provides a natural foundation for developing and exploring specification mining in Python projects, as it provides a large number of ready-to-run and ready-to-analyze projects. Beyond specification mining, future work could

Table 3. Examples of patterns among top-100 mined patterns.

| Pattern | Freq. | Code examples from DyPyBench |
|---|---|---|
| `builtins.isinstance`<br>`builtins.isinstance` | 1,701 | `if isinstance(ty, tuple):`<br>`  return Tuple(ty)`<br>`if isinstance(ty, ParamType):`<br>`  return ty` |
| `bytes.join`<br>`builtins.str`<br>`str.encode` | 1,008 | `return b"".join([b"HTTP/1.1",`<br>`  str(status_code).encode(),`<br>`  b"\s", phrase, b"\r\n"])` |
| `Pattern.match`<br>`Match.span`<br>`str.isidentifier` | 730 | `pseudomatch=pseudoprog.match(line,pos)`<br>`if pseudomatch:  # scan for tokens`<br>`  start, end=pseudomatch.span(1)`<br>`  #code in between`<br>`if ...`<br>`elif initial.isidentifier():`<br>`  # ...` |
| `thefuck...encode_utf8`<br>`str.replace`<br>`shlex.split`<br>`str.replace`<br>`thefuck...decode_utf8` | 565 | `encoded = self.encode_utf8(command)`<br>`try:`<br>`  splitted=[s.replace("??", "\\ ")`<br>`  for s in shlex.split(`<br>`  encoded.replace('\\ ','??'))]`<br>`except ValueError:`<br>`  splitted = encoded.split(' ')`<br>`return self.decode_utf8(splitted)` |
| `logging.getLogger`<br>`logger.setLevel` | 205 | `logger = logging.getLogger(`<br>`  "urllib3.connectionpool")`<br>`logger.setLevel(logging.WARNING)` |

explore other applications in software analysis that leverage our dataset of call sequences, e.g., to detect anomalies and inconsistencies.

## 5 RELATED WORK

*Dynamic analysis and testing of Python.* Python has been the target of several dynamic analyses, e.g., to slice programs [15], to detect type-related bugs [66], and to enforce differential privacy [1]. Scalene [7] offers an efficient CPU and memory profiler. DyPyBench can serve as a dataset to test and evaluate these and future dynamic analyses. Pynguin [38] is a unit-level, search-based test generator for Python. LExecutor [59] proposes learning-guided execution, which executes code by injecting otherwise missing runtime values. Both Pynguin and LExecutor share with DyPyBench the goal of executing code. DyPyBench complements these existing efforts by providing a reproducible setup that allows for executing and analyzing a set of real-world projects with little effort. DynaPyt [21] is a general-purpose dynamic analysis framework. Because it allows for implementing arbitrary dynamic analyses, we integrate DynaPyt into this work and show two applications of it (Sections 4.1 and 4.3).

*Other work on Python.* Beyond dynamic analysis, the increasing importance of Python has lead to a stream of work on studying Python software and on supporting Python developers. A study of flaky tests [27] relates to this work because they try to automatically execute the test suites of

thousands of Python projects. DyPyBench differs by focusing on a reusable benchmark, where each project and its setup is carefully checked by a human before being included into the benchmark, whereas the prior work is based on a large-scale, fully automated experiment. Another difference is the size and characteristics of the projects. As highlighted in [27], approximately 75% of their projects have fewer than 10 tests, 60% exhibit a coverage below 10%, and over 90% of the projects consist of less than 10k lines of code. In contrast, our benchmark contains only 4% of projects with less than 10 test cases, a minimum coverage of 48% (average: 82%), and an average code base size of 14k lines per project.

The lack of static type annotations by default has lead to several techniques for inferring and predicting type annotations [3, 47, 67, 68], to work on comparing gradual type systems for Python [50], and to work on studying type annotations in the Python ecosystem [26]. DyPyBench could be used to create a ground truth of types observed at runtime, which can serve as training data for learning-based type predictors or be used during their evaluation. Another study investigates the performance benefits of using Python idioms [71]. Future work on studying the performance of Python code can benefit from DyPyBench as a ready-to-execute collection of real-world code.

*Dynamic analysis of other languages.* Beyond Python, there are various dynamic analyses, e.g., to detect concurrency bugs [23, 42, 55], type-related problems [5, 49], and other common bug patterns [25]. Other analyses find optimization opportunities [62, 65], or infer API usage protocols [48, 69] and input grammars [32]. Security-oriented analyses include taint analysis [17] and dynamic detectors of similar functions [20]. To reduce the effort of building a dynamic analysis, dynamic analysis frameworks have been proposed for most popular languages, e.g., DynamoRIO [12], Pin [37], and Valgrind [41], which all target x86 binaries, Jalangi [57] for JavaScript, Wasabi [34] for WebAssembly, DiSL [39] for Java, and RoadRunner [24], which specifically targets concurrency-related dynamic analyses. As DyPyBench targets Python, it cannot directly support the above analyses, but the breadth and depth of prior work underlines the importance of dynamic analysis, and hence, benchmarks like ours.

*Benchmarks of executable code.* Our work has been partially inspired by other benchmarks of executable code. Similar to DyPyBench, several benchmarks offer a set of real-world programs with inputs to execute their code, e.g., DaCapo [10] for Java, SPEC CPU [31] for C/C++, and Da Capo con Scala [58]. Other executable benchmarks are aimed at specific tools and techniques, e.g., LAVA [19] and Magma [30], which are aimed at evaluating fuzzers, SecBenchJS [8], which is aimed at evaluating techniques for detecting or mitigating vulnerabilities in JavaScript, and Parsec, which offers multi-threaded C/C++ programs [9]. Importantly, none of the above benchmarks is for Python, which is the gap our work tries to fill.

*Other benchmarks.* Many other benchmark sets have proven valuable, of which we can list only a few here. To evaluate and compare bug-related techniques, there are Defects4J [33], Bugs.jar [53], and BugSwarm [18] for Java, BugsJS for JavaScript [28], and BugsInPy [64] for Python. More specialized benchmarks include a set of JavaScript performance bugs [56] and a collection of concurrency bugs [70]. To evaluate deep learning models of code, benchmarks such as CodeXGLUE [36] and HumanEval [14], are widely used. We envision DyPyBench to help support future work on dynamic analysis for Python in a way similar to the above benchmarks.

## 6 CONCLUSION

This paper addresses a significant gap in the Python programming ecosystem by introducing DyPyBench, a dynamic benchmark suite of executable Python projects. The benchmark offers a unique combination of features by being large-scale, diverse, ready-to-run, and ready-to-analyze.

With 50 projects, 681k lines of code (of which 558k are executed), and 29,511 test cases, DyPyBench offers a rich set of executions to analyze and study. We illustrate the practicality and utility of DyPyBench in three usage scenarios: comparing static and dynamic call graphs; creating training data for learning-guided execution; and mining specifications from execution traces. We envision our work to provide a foundation for many other dynamic analyses and for studying the runtime behavior of Python software.

## DATA AVAILABILITY

Our DyPyBench image is available on Zenodo: https://doi.org/10.5281/zenodo.11097202[11]. Alternatively, the image can also be pulled from DockerHub: https://hub.docker.com/r/islemdockerdev/dypybench. Project page for updates and issues: https://github.com/sola-st/DyPyBench

## ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys, DeMoCo, and QPTest projects.

## REFERENCES

[1] Chike Abuah, Alex Silence, David Darais, and Joseph P. Near. 2021. DDUO: General-Purpose Dynamic Analysis for Differential Privacy. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 1–15. https://doi.org/10.1109/CSF51468.2021.00043

[2] Karim Ali, Marianna Rapoport, Ondřej Lhoták, Julian Dolby, and Frank Tip. 2014. Constructing call graphs of Scala programs. In *European Conference on Object-Oriented Programming*. Springer, 54–79.

[3] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *PLDI*.

[4] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 4–16.

[5] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic inference of static types for Ruby.. In *POPL*. 459–472.

[6] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 324–341.

[7] Emery D Berger, Sam Stern, and Juan Altmayer Pizzorno. 2023. Triangulating Python Performance Issues with {SCALENE}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 51–64.

[8] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. 2023. SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1059–1070. https://doi.org/10.1109/ICSE48619.2023.00096

[9] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[10] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 169–190.

[11] Islem Bouzenia, Bajaj Piyush Krishan, and Michael Pradel. 2024. *DyPyBench Docker Image*. https://doi.org/10.5281/zenodo.11097202

[12] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 265–275.

[13] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. 2022. Automatic root cause quantification for missing edges in javascript call graphs. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[15] Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C. Chu, and Baowen Xu. 2014. Dynamic Slicing of Python Programs. In *IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014*. IEEE Computer Society, 219–228. https://doi.org/10.1109/COMPSAC.2014.30

[16] Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C Chu, and Baowen Xu. 2014. Dynamic slicing of Python programs. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, 219–228.
[17] James A. Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 196–206.
[18] Naji Dmeiri, David A. Tomassi, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. *CoRR* abs/1903.06725 (2019). arXiv:1903.06725 http://arxiv.org/abs/1903.06725
[19] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 110–121.
[20] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 303–317.
[21] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A Dynamic Analysis Framework for Python. In *ESEC/FSE '22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM.
[22] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 752–761.
[23] Cormac Flanagan and Stephen N. Freund. 2004. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 256–267.
[24] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner dynamic analysis framework for concurrent programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 1–8.
[25] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*. 94–105.
[26] Luca Di Grazia and Michael Pradel. 2022. The Evolution of Type Annotations in Python: An Empirical Study.. In *ESEC/FSE*.
[27] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 148–158. https://doi.org/10.1109/ICST49551.2021.00026
[28] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 90–101.
[29] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2001. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*. IEEE, 215–224.
[30] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.
[31] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.
[32] Matthias Höschele and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 720–725.
[33] 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*. 437–440.
[34] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *ASPLOS*.
[35] Daniel Lehmann, Michelle Thalakottur, Frank Tip, and Michael Pradel. 2023. That's a Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023*. 892–903. https://doi.org/10.1145/3597926.3598104
[36] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).
[37] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[38] Stephan Lukasczyk. 2019. Generating Tests to Analyse Dynamically-Typed Programs. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1226–1229. https://doi.org/10.1109/ASE.2019.00146

[39] Lukás Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012*, Robert Hirschfeld, Éric Tanter, Kevin J. Sullivan, and Richard P. Gabriel (Eds.). ACM, 239–250. https://doi.org/10.1145/2162049.2162077

[40] Gail C Murphy, David Notkin, William G Griswold, and Erica S Lan. 1998. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 2 (1998), 158–191.

[41] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 89–100.

[42] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM, 167–178.

[43] Jibesh Patra and Michael Pradel. 2021. Semantic bug seeding: a learning-based approach for creating realistic bugs. 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*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 906–918. https://doi.org/10.1145/3468264.3468623

[44] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: Fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 690–702.

[45] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680* (2020).

[46] Michael Pradel and Satish Chandra. 2022. Neural software analysis. *Commun. ACM* 65, 1 (2022), 86–96. https://doi.org/10.1145/3460348

[47] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-based Validation. 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*. 209–220. https://doi.org/10.1145/3368089.3409715

[48] Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *International Conference on Automated Software Engineering (ASE)*. 371–382.

[49] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *International Conference on Software Engineering (ICSE)*.

[50] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 Types in the Wild: A Tale of Two Type Systems. In *DLS*.

[51] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 251–261. https://doi.org/10.1145/3293882.3330555

[52] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2012. Automated API property inference techniques. *IEEE Transactions on Software Engineering* 39, 5 (2012), 613–637.

[53] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. 2018. Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories*. 10–13.

[54] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146

[55] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems* 15, 4 (1997), 391–411.

[56] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *International Conference on Software Engineering (ICSE)*. 61–72.

[57] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 488–498.

[58] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 657–676.

[59] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. In *FSE*.
[60] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the Recall of Static Call Graph Construction in Practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1049–1060. https://doi.org/10.1145/3377811.3380441
[61] Frank Tip and Jens Palsberg. 2000. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 281–293.
[62] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 607–622.
[63] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. 2022. Striking a Balance: Pruning False-Positives from Static Call Graphs. In *ICSE*.
[64] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. 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*. 1556–1560.
[65] Guoqing (Harry) Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 419–430.
[66] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. 2016. Python predictive analysis for bug detection. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 121–132. https://doi.org/10.1145/2950290.2950357
[67] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 607–618. https://doi.org/10.1145/2950290.2950343
[68] Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. 2023. DLInfer: Deep Learning with Static Slicing for Python Type Inference. In *ICSE*.
[69] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining temporal API rules from imperfect traces. In *International Conference on Software Engineering (ICSE)*. ACM, 282–291.
[70] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. 2021. Gobench: A benchmark suite of real-world go concurrency bugs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 187–199.
[71] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence, In ICSE. *arXiv preprint arXiv:2301.12633*.

Received 2023-09-29; accepted 2024-01-23