EXCEPy: A Python Benchmark for Bugs with Python Built-in Types

Xin Zhang¹,³, Rongjie Yan¹,³ Jiwei Yan²,³,†, Baoquan Cui¹,³, Jun Yan¹,²,³, Jian Zhang¹,³,†
¹ State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
² Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
³ University of Chinese Academy of Sciences
Email: {zhangxin19, yrj, yanjw, cuibq, yanjun, zj}@ios.ac.cn

Abstract—As bugs of Python built-in types can cause code crashes, detecting them is critical to the robustness of the software. Researchers have concluded plenty of patterns for the bug causes and applied these patterns in detection tools. But these tools are only evaluated on handcrafted bugs or bugs obtained from QA pages. Because such bugs cannot reflect the complex code structures and various bug types encountered in real-world projects, the evaluation result is untrustworthy when applied to these projects. As a result, a collection of real-world reproducible bugs is essential for tool evaluation and future bug-related research.

In this paper, we propose EXCEPy, a benchmark for providing bugs of Python built-in types. We collect 180 bugs from the evolution of 15 real-world open-source Python projects on GitHub and then manually build test scripts for bug reproduction. Meanwhile, to improve tool evaluation efficiency, we present a code pruning technique that can minimize buggy code size while retaining bug reproducibility and apply it to EXCEPy to provide simplified buggy code. To demonstrate the benefits of EXCEPy, we use three static analyzers and two fuzzers to detect bugs collected in EXCEPy. We found that simplified code can significantly reduce running time and avoid many tool crashes, and bugs supplied by EXCEPy can reveal limitations of existing tools in reporting real-world bugs.

Index Terms—Benchmark for bugs, built-in bug types of Python, static analyzers, the evolution of Python projects.

I. INTRODUCTION

The need for bug detection tools is growing in tandem with the adoption of the Python programming language [1]. The built-in type bugs are the most common vulnerabilities in Python, and they can cause code to crash. Many researchers have proposed patterns for root causes and implemented these patterns in various detection methods, e.g., static analyzers or fuzzers [2]–[6].

Researchers have claimed that their tools identify particular bug types better than other tools. But they only tested the tools on handcrafted bugs or non-reproducible bugs derived from bug reports, and there is no prior work that accesses them on real-world bugs. Because of the differences in code structures and bug types between handmade and real-world defects, their evaluation results are untrustworthy when applied to modern Python projects. Thus, it is critical to understand how effective these tools are in detecting crash bugs in practice.

The number of bugs they can report on a collection of reproducible bugs from real-world projects is a frequently used metric for evaluating the performance in discovering defects. Some mature benchmarks are widely adopted in practice, such as Defects4J [7] and QuixBugs [8] for Java and BugsJS [9] for JavaScript. These benchmarks gathered reproducible crash bugs from real-world open-source projects. Meanwhile, they give root causes and repair information for the bugs.

Two benchmarks produced many Python defects [10], [11]. But when compared to other mature works, they fall short in tool evaluation. On the one hand, they only provide functional defects that do not cause a crash. Existing testing techniques, which identify flaws by monitoring crashes and exceptions during runtime, cannot detect these bugs without prior information. On the other hand, they use the entire project code as buggy code for each defect. Numerous lines of code in the project code do not affect bug triggering (We call them bug-relevant code). Because benchmarks do not filter the bug-relevant code, tools have no oracle when throwing warnings on them and may crash due to unsupported syntax.

We aim to fill this gap by providing a Python benchmark with real-world reproducible bugs. There are two challenges in constructing such a benchmark.

- **Bug reproduction.** Because variable types and values must be considered, reproducing a defect in Python is more complex than in other statically typed programming languages. Python has several language features that make it easier to use, such as dynamic type and dynamic attribute. With these features, users can change the type and attributes of an object during execution. Thus, choosing types and values to produce input data to trigger a Python bug can be challenging, especially in real-world projects with numerous self-defined types, leading to an enormously broad range of candidate types.

- **Bug-relevant code pruning.** Complete buggy code may contain numerous lines of Python code, the vast majority of which may be bug-relevant. Plenty of bug-relevant codes will reduce analyzer efficiency and make manual reduction time-consuming. Besides, validating the authenticity of bug reproduction on trimmed code is difficult with an automated pruning technique.
To address the first challenge, we choose to get input data from open-source projects. First, we eliminate bugs that require a specific environment to reproduce, such as systems or the Internet. Second, we extract code snippets from commit discussions and use the input data in these code snippets to generate test code. Third, we look at the newly added test cases to see whether developers provide new input data.

To deal with the second challenge, we except to automatically delete classes and modules that test code does not import during execution. To do this, we develop a tracer to follow test code execution and record a list of executed files, from which we can delete the unexecuted files. We re-run the test code after each deletion and compare the results to the original traceback. If the traceback changes after deleting a file, this file must be re-added.

We collect commits from the evolution of 300 Python projects on GitHub, and after manually reviewing 2966 commits, we obtain 180 reproducible crash bugs from 15 projects, covering 15 built-in bug types. To show all the code changes, the message describing the changes, and a list of executed files, we summarize the bugs, buggy code, and test scripts in a benchmark called EXCEPY.

To demonstrate the advantages of EXCEPY, we assess the benchmark in two ways: the efficacy of code pruning and the usefulness of the gathered bugs for tool evaluation. We employ five commonly used Python tools on EXCEPY, including three static analyzers and two fuzzers, and evaluate their performance on both the complete buggy code and the simplified code. According to the results, the simplified buggy code reduces execution time and eliminates the majority of timeouts and crashes. Besides, the gathered bugs can reveal the advantages and disadvantages of existing tools for detecting crash bugs in practice.

To summarize, we make the following contributions.

- We provide a Python benchmark EXCEPY, including 180 reproducible crash bugs spread over 15 Python built-in types. These bugs were gathered from well-known Python projects on GitHub and are illustrative of vulnerabilities encountered in daily use.
- We propose an automatic code pruning approach to eliminate bug-irrelevant code without affecting bug reproduction. We apply the method for pruning codes in EXCEPY to produce the simplified bug-relevant code. The experimental results show that reduced code can significantly reduce running time. Besides, by avoiding crashes without output, the reduced code is more effective at identifying the weaknesses of the tools.
- We conclude 26 root causes and fix patterns for 15 Python built-in bug types, which can facilitate bug detection and debugging studies of Python programs.

II. BACKGROUND

In this section, we will briefly introduce the Python built-in bug types, the common components in mature benchmarks, and the platform we use for bug collecting.

A. Python Built-in Bug Types

Python offers 64 bug types to deal with bugs that can result in crashes [12], which are all derived from BaseException, and divided into 16 categories based on their error conditions. Among various bug types, several bug types, such as TypeError (Typ) [2], [3], [13], [14], AttributeError (Att) [2], [13], [14], ValueError (Val) [15], and ZeroDivisionError (Zer) [2], are frequently studied in the literature.

Bugs of built-in types use traceback to indicate the causes, including the method calls along the execution path. Every two lines in a traceback indicate a method call, with the first marking the file location and line number and the second containing the source code corresponding to the line number.

As stated in Section I, creating input data to trigger bugs necessitates the use of the proper types and values. In Figure 1, we present a code snippet with two bugs to demonstrate the relevance of types in triggering bugs, one of which is of type Typ and the other of type Att. Both the two bugs happen in method test at line 7 when this method extracts two attributes from the input data and performs operation addition.

To trigger the bug of type Typ, the input data must have two attributes whose types are not supported by the add operation, and a frequent situation is between the types str and int. We give an example input data from line 11 to 13 for triggering a Typ bug, where we change the type of attribute attri_2 from str to int, and provide the object obj_a to the method. The add operation between an int variable and a str variable will cause a bug of Typ because the interpreter does not know whether to convert int 1 to str '1' and concatenate two strings, or convert str '2' to int 2 and conduct addition.

The bug of type Att can only be triggered if one of the attributes does not exist. We provide an example input data that can trigger this bug from lines 15 to 17 in Figure 1. Supported by the language feature dynamic attribute, we can delete the attribute attri_2 from the object obj_a. When we call method test at line 17, the interpreter cannot find attribute attri_2 in object input_data at line 7, and reports a bug of type Att.

B. Bug Collection Source: GitHub Platform

GitHub is an open-source community with over 65 million developers [16], where developers can freely update and release their code. GitHub adopts the term project to refer to a deployed project and applies the term pull request to refer to code changes submitted by a developer to the origin project. A pull request contains multiple commits, each of which stores the code changes, the message describing the changes, and a particular timestamp.

GitHub provides two mechanisms for users to report bugs: issues and pull requests. The primary distinction between the two is that issues do not require users to fork the project and make code changes. Users can utilize the two techniques to report defects and attach their test code, traceback, and fix
recommendations to the original projects, as well as discuss with other users.

C. Common Components in Existing Benchmarks

There are three major components in existing benchmarks to provide bug data and make bug reproduction easier [7]-[9], [11], [17], which are bug information, buggy code, and test scripts.

- **Bug information.** It offers users the bug type, reports, and fix patches. This part is critical for determining the correctness of reproduction and analyzing the root causes of the bugs.
- **Buggy code.** Buggy code describes a piece of code that includes bugs.
- **Test scripts.** The goal of test scripts is to make the process of reproducing bugs simpler. A test script includes instructions for switching to the buggy version of the code, installing the buggy code, and running the Python test code. Users can reproduce the bug by directly executing the script.

III. THE PROPOSED BENCHMARK: EXCEPY

The architecture of EXCEPY is depicted in Figure 2, where,

- **The dataset** stores the data we gather for ExcePy, including the test scripts, buggy code, and git information about the projects.
- **The execution framework** takes over the process of reproducing bugs, as well as providing unified APIs.
- **Application** allows users to load their tools on EXCEPY.

EXCEPY now is publicly available¹. EXCEPY is constructed with two phases. The first phase involves gathering bugs from real-world projects, while the second is responsible for pruning bug-relevant code.

A. Bug Collection

Our goal is to collect reproducible built-in type bugs from real-world projects. The bugs must fulfill the following characteristics to achieve the purpose.

```python
class A:
    def __init__(self):
        self.attr1 = '1'
        self.attr2 = '2'

    def test(input_data):
        input_data.attr1 + input_data.attr2

    if __name__ == '__main__':
        obj_a = A()
        obj_a.attr2 = 2
        # Trigger a bug of type TypeError
        test(obj_a)
        del obj_a.attr2
        # Trigger a bug of type AttributeError
        test(obj_a)
```

Fig. 1: An example of two bugs that rely on specific input data to trigger.

- **Reproducible.** We only collect reproducible bugs and exclude those that rely on specific hardware or network to trigger.
- **Python-Only.** We exclusively gather Python source code-related bugs. We exclude the bug caused by the interaction of several programming languages, such as those found in Python native libraries.
- **Isolated.** We require that bugs be repaired by modifying just one Python source file, and we exclude bugs that are fixed by surrounding a try-catch block or by simply removing the buggy code.

Figure 3 depicts the phases of the collection workflow with four steps. First, we collect the most popular Python projects on GitHub. Second, we collect commits that may attempt to fix bugs from the evolution of the projects and choose those projects with sufficient bug-fixing commits. Third, we manually verify the changes and attempt to reproduce the bugs recorded in the commits. At last, we utilize the reproducible bugs to create the dataset of EXCEPY.

1) **Python Project Selection:** As indicated in Section II-B, GitHub is a popular platform for sharing open-source projects. Thus we choose to gather bugs from projects hosted on GitHub. We rank Python projects by star rating and select the top 300 as bug collection sources to catch more generic bugs. These projects can represent the concerns of most Python users.

2) **Candidate Commit Collection:** The 300 projects have numerous commits, and manually going through them is laborious. We must automatically filter out valuable commits to lessen the load of human inspections in the following phases. We utilize a popular filtering technique that matches keywords in commit message parts, and the keywords we use in this process are the names of built-in bug types.

The keywords matching approach has limits when filtering out some commits that do not meet our requirements. One limitation is that some commits try to fix the bugs in obsolete test cases by simply removing the test cases. Another is that developers frequently include bug types in the commit only when improving warning outputs. Even if these commits do not fulfill our requirements, the keywords matching technique cannot filter them out.

With the limitations mentioned above, we enhance the automatic filtering approach. To address the first limitation, we analyze the changed files in commits and limit all changes

---

¹https://github.com/Stardust1225/ExcePy_Present
to Python files. Besides, because all files whose filenames begin with test_ are test files according to Python coding standards, we require at least one changed file with a filename that does not begin with test_. We noticed that commits aiming at upgrading warning outputs typically contain both the original and changed types. Thus, to overcome the second limitation, we add the keywords fix and repair to the matching strategy and filter out commits containing multiple bug type names in the message part.

For the selected commits (called candidate commits) after the automatic filtering process, we discover 42 types in the commits of the 300 projects, indicating that bugs of these types are common in Python projects. Figure 4 depicts the distributions of candidate commits in the 42 bug types, where we rank the types according to the number of related candidate commits and show the names of the top 15 types. The remaining categories are categorized as other. Four of the 42 types have a high frequency in the message parts and are involved in 1548 commits. Types Typ, Val, Att, and Key are the most common. Furthermore, there are four kinds with just one associated commit: Ari, Flo, Con, Int. To some extent, these bug types are less likely to be encountered in daily use or rely on specific platforms to trigger.

We have 5150 candidate commits from the 300 projects. After grading the projects based on the number of candidate commits and deleting those with fewer than 20 commits or aiming at education, we select 2966 candidate commits from the top 40 projects for the following phases. Each of the 40 projects has more than 30 candidate commits.

3) Bug Reproduction: A successful reproduction depends on two factors: (1) appropriate input data for triggering and (2) a traceback to identify the correctness of reproduction. We collect reported traceback from commits and pull requests that reference them. Besides, we look for code snippets and fill them with extra code for importing libraries and initializing objects to build test code.

After collecting the reported traceback and constructing a test script, we run the test code on the buggy code to check that the test code can produce the same traceback as the reported one. First, we swap the project code to the required version, and then we install the project in development mode. We search the development documentation for additional steps because most projects need specific dependencies throughout the installation process. After executing the test code, we compare the output traceback to the previously recorded traceback. If we can produce the identical traceback, we save the test script for the next phase.

One challenge in bug reproduction is that developers only report a traceback without source code. The other is that the bug requires some external files to be triggered, such as a specific picture file or a markdown file. To address these challenges, we endeavor to extract the code snippets from the pull request conversation and search for code snippets from test cases in a later version of the source code. We can then add code for importing modules and generating objects based on the code snippets to generate an executable test code.

We can reproduce 15 of the 42 bug types we have collected. We cannot reproduce bugs in the collected commits because of the three factors stated below.

- A candidate commit simply modifies the bug types or changes the output message.
- Changes in a commit are just to catch a bug or to throw a new bug.
- A commit solves a bug in the test cases after some APIs are changed.

In this part, we have manually inspected 2966 commits in 17 projects and successfully reproduced 183 bugs with 15 bug types. The proportion of candidate commits corresponding to the 15 types in the inner black circle is depicted in Figure 4.

4) Fix Patches Validation: A bug resolved with a try–catch block does not fulfill our requirements. As a result, we must double-check the fix patches to ensure that they meet our requirements. We inspect the patches to exclude the cases that the bugs are fixed with a try–catch structure. We eliminate three bugs from the 183 bugs in 17 projects and include 180 bugs from 15 projects in our dataset.

B. Code Reduction

As indicated in Section I, in addition to the complete project code, we provide simplified buggy code to help evaluate the tools. Because Python does not fully compile before execution, for a piece of test code, we can delete files that have not been executed and ensure that the test code will not crash when executed again. Thus, the fundamental idea behind developing simplified flawed code is to reduce unexecuted files while guaranteeing the accuracy of bug reproduction.

Figure 5 depicts the workflow of the method for pruning bug-irrelevant code. The method takes a piece of test code and its related complete buggy code as input and returns the simplified buggy code. Tracing the executed files and trimming
bug irrelevant code are two main phases in code pruning. We run the test code and record the executed Python files in the first phase. The second phase aims at eliminating as many unexecuted files as possible while ensuring bug reproduction.

1) Executed Files Tracing: We build a tracer and register it in the test code using API sys.settrace() to store the files necessary for test code execution. The Python Interpreter will invoke the tracer and provide execution information such as file locations and line numbers when executing the test code. After running, we can collect the executed files from the tracer.

2) Bug-irrelevant Files Pruning: One Python file can include one or more class definitions, and these classes may depend on the other. Because no mature tool is capable of creating a complete call graph [18], pruning code within a Python file can be adventurous if the dependencies between the methods and classes are not thoroughly understood. As a result, we choose to prune code at the file level.

The algorithm of pruning code is presented in Algorithm 1, where we use c_code to represent the complete buggy code, s_code to refer to the simplified code, exec_files to represent the executed files recorded by the tracer. From line 1 to line 5, we extract the unmarked files from the whole code and store them in del_list, which we use to prune the code. From lines 6 to 11, we remove the unexecuted files while ensuring the correctness of reproduction via comparing the traceback with the reported one.

C. Unified APIs Development

To provide users with consistent APIs, we build the execution framework in four parts: version switch, data release, installation, and execution. The version switch component handles the mapping from bug ID to commit ID. The data release section supplies users with the necessary test scripts. The installation section is in charge of installing dependencies and building projects from the source code. Finally, the execution section executes the test code and compares the results to the recorded traceback.

Test code in EXCEPY can be run directly with a python3 command or imported by third-party testing frameworks such as pytest [19] and unittest [20]. It is simple to track test code execution and collect executed statements for calculating coverage. Users can also set up personal tracer to help with analysis and testing.

IV. ANALYSIS OF THE BUGS AND BUGGY CODE

In this part, we will discuss the data in EXCEPY from two perspectives: projects and test scripts. Then, based on the debates in the commits and other similar studies [10], [13], we provide some root causes and fix patterns that we manually conclude for the 15 built-in bug types.

A. The Distributions of the Python Projects and Bugs

Table I displays basic information about the 15 Python projects in EXCEPY, including their number of stars and commits, as well as the fields to which they belong. As indicated in the table, the gathered projects cover several popular areas, such as machine learning and scientific computing, attract numerous third-party developers, and are well maintained.

<table>
<thead>
<tr>
<th>ID</th>
<th>Project</th>
<th>#Star</th>
<th>#Commit</th>
<th>Field(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>CPython</td>
<td>40.4K</td>
<td>111K</td>
<td>Compiler &amp; Testing</td>
</tr>
<tr>
<td>P2</td>
<td>IPython</td>
<td>15K</td>
<td>25K</td>
<td>Compiler &amp; Testing</td>
</tr>
<tr>
<td>P3</td>
<td>Jedi</td>
<td>10.1K</td>
<td>1K</td>
<td>Program Analysis</td>
</tr>
<tr>
<td>P4</td>
<td>Matplotlib</td>
<td>14.3K</td>
<td>40K</td>
<td>Image Processing</td>
</tr>
<tr>
<td>P5</td>
<td>NumPy</td>
<td>18.3K</td>
<td>27K</td>
<td>Scientific computing</td>
</tr>
<tr>
<td>P6</td>
<td>Pandas</td>
<td>31.2K</td>
<td>27K</td>
<td>Machine learning</td>
</tr>
<tr>
<td>P7</td>
<td>Pelican</td>
<td>10.6K</td>
<td>3K</td>
<td>Compiler &amp; Testing</td>
</tr>
<tr>
<td>P8</td>
<td>Pillow</td>
<td>9K</td>
<td>12K</td>
<td>Image Processing</td>
</tr>
<tr>
<td>P9</td>
<td>Pytest</td>
<td>7.8K</td>
<td>13K</td>
<td>Compiler &amp; Testing</td>
</tr>
<tr>
<td>P10</td>
<td>requests</td>
<td>25.3K</td>
<td>2K</td>
<td>Network</td>
</tr>
<tr>
<td>P11</td>
<td>SciPy</td>
<td>8.6K</td>
<td>26K</td>
<td>Scientific computing</td>
</tr>
<tr>
<td>P12</td>
<td>Scikit-image</td>
<td>4.5K</td>
<td>12K</td>
<td>Image Processing</td>
</tr>
<tr>
<td>P13</td>
<td>Scikit-learn</td>
<td>47.4K</td>
<td>27K</td>
<td>Machine learning</td>
</tr>
<tr>
<td>P14</td>
<td>SynPy</td>
<td>8.5K</td>
<td>48K</td>
<td>Scientific computing</td>
</tr>
<tr>
<td>P15</td>
<td>Tensorflow</td>
<td>159K</td>
<td>118K</td>
<td>Machine learning</td>
</tr>
</tbody>
</table>
TABLE II: Bugs in EXCEPY (with the number of bugs of each type and each project).

<table>
<thead>
<tr>
<th>ID</th>
<th>Bug Type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Attribute</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>T2</td>
<td>Import</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>T3</td>
<td>Index</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>23</td>
</tr>
<tr>
<td>T4</td>
<td>Key</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>T5</td>
<td>Memory</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>T6</td>
<td>Name</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>T7</td>
<td>NotImplement</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>T8</td>
<td>OS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>T9</td>
<td>Overflow</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>T10</td>
<td>Recursion</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>T11</td>
<td>Runtime</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>T12</td>
<td>Type</td>
<td>3</td>
<td>-</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>21</td>
<td>62</td>
</tr>
<tr>
<td>T13</td>
<td>UnboundLocal</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>T14</td>
<td>Value</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>25</td>
</tr>
<tr>
<td>T15</td>
<td>ZeroDivision</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>sum</td>
<td></td>
<td>11</td>
<td>5</td>
<td>3</td>
<td>11</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>33</td>
<td>180</td>
</tr>
</tbody>
</table>

In Table II, we show the total number of bugs for each project, where we use the ID in Table I to represent the projects. Since all bug type names end with an error, we omit the error in the bug type in Table I. As it shows, most of the bugs concentrate on a few types, e.g., type TypeError, AttributeError, ValueError and IndexError, which suggests that developers are more likely to make the four types of errors. Some bug types only exist in one project, such as types MemoryError and RuntimeError. These types are hard to trigger as they rely on the exceptions in some system calls.

B. Statistics of the Test Scripts in EXCEPY

```bash
# Commit version: e4b9f6ec0868c3211c1da716d76248115c11feec
# Dependencies scripts
python3 -m pip install Cython==0.22
# Installation scripts
pip3 install -e .
# Test code
dump_from_pkl loads (dumps (bunch) )
```

Fig. 6: A script for triggering a bug of type Key in project scikit-learn.

Figure 6 shows a test script we created for a defect in the project scikit-learn, including instructions, test code, and a traceback for the fault. Lines 3–7 provide instructions for installing dependencies and compiling the project from source code. Lines 9 through 15 include the test code. To make the test code executable, we excise lines 14 and 15 from the bug report and insert lines 9 to 13. From line 17 through line 22, we capture the reported traceback.

Figure 7 depicts the distributions of the number of lines of test code and traceback.

C. Root causes and Fix Patterns

By manually examining bugs, execution paths, tracebacks, and repair patches, we summarize several root causes and fix patterns for each bug type and present the four common types in Table III. We provide a detailed report for the 15 bug types on our website.2

There are three parts for each bug type. The Root causes column offers a brief explanation of the root cause, the Example buggy code column displays some pseudo-code samples or descriptions that will cause the problem, and the Fix patterns column shows the most popular fix patches. One bug type may have many root causes in this table.

In the investigation, we further extract three highly frequent root causes and fix patterns among these types.

- **Cause 1: Receiving an unexpected type of returned variable.** The type of the returned variable is not definite since Python is a dynamic typing language. Directly processing the returned value without further judgment when executing a method can easily result in a crash, especially if developers regularly set the default type of returned values for specific methods. In EXCEPY, bugs related to unexpected returned type can be of bug types Att, Typ, Val, and Zer.

- **Fix 1: Adding type judgment.** Including code that validates the type of returned values, or limiting the input data to certain types, can help developers prevent these bugs.

TABLE III: The root causes, examples buggy code, and fix patterns for three most frequent bug types.

<table>
<thead>
<tr>
<th>Type</th>
<th>ID</th>
<th>Root causes</th>
<th>Example buggy code</th>
<th>Fix patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Att</td>
<td>T1-1</td>
<td>Get unexpected type of return variable.</td>
<td><code>real_frames.shape[-1].value</code></td>
<td>Add an examination to the type.</td>
</tr>
<tr>
<td></td>
<td>T1-2</td>
<td>Get value from an unexisting class member.</td>
<td></td>
<td>Add an examination to the class member.</td>
</tr>
<tr>
<td>Typ</td>
<td>T12-1</td>
<td>Concatenate strings and numbers with a plus.</td>
<td><code>a=&quot;hello&quot;; b=1; a+b</code></td>
<td>Choose to change one type to the same as another.</td>
</tr>
<tr>
<td></td>
<td>T12-2</td>
<td>Call an operation on an object that does not implement it.</td>
<td><code>*</code></td>
<td>Add code to support the types.</td>
</tr>
<tr>
<td></td>
<td>T12-3</td>
<td>Operate with an unexpected type.</td>
<td><code>f1/f2</code> and <code>f1/f2.</code></td>
<td>Add a check for the type of return value before referencing it.</td>
</tr>
<tr>
<td></td>
<td>T12-4</td>
<td>Converse type.</td>
<td>The targeted type does not consider the input type.</td>
<td>Change the targeted type or implement conversion code for the uncovered type.</td>
</tr>
<tr>
<td>Val</td>
<td>T14-1</td>
<td>Operate on a variable with an unsupported type.</td>
<td></td>
<td>Pass type information to the operation to help the operation make a judgment, or use type conversion on the arguments.</td>
</tr>
<tr>
<td>Key</td>
<td>T4-1</td>
<td>API misuses.</td>
<td><code>keras</code> changes its implementation and <code>tensorflow</code> has not adapted to it.</td>
<td>Add a judgment at the reading process to ensure that the layer of the model is correct.</td>
</tr>
<tr>
<td></td>
<td>T4-2</td>
<td>Split a string on an empty string.</td>
<td>Empty parts would appear after the first splitting, and the second splitting on an empty part would lead to it.</td>
<td>Add length check for each part.</td>
</tr>
</tbody>
</table>

- **Cause 2: Unexpected input data.** Some input data might lead to an endless loop or infinite recursion if developers fail to control particular boundary values or set wrong conditions in the code. An example of this root cause is a bug in the `random` module of project `CPython`\(^3\), where it raises a bug of type `Zer` when taking a specific `int` value as input.

**Fix 2: Adding code to exclude illegal input data.** Developers can include code at the beginning of the method to filter certain input data, or they may include checking code to avoid any illegal inputs. In the benchmark, bugs of types `Zer`, `Mem`, `Rec`, and `Not` are often triggered by some unexpected input data.

- **Cause 3: Ignoring API changes.** When developers neglect changes in particular APIs, it might result in bugs in the caller. Modifying the type range of the returned value, adding extra restrictions to the input data, or even altering the function name are all possible code modifications. This root cause can lead to bugs of types `Not` and `Key`.

**Fix 3: Adjusting code to the new APIs.** Developers keep up with changes in the new code and modify their imported code, or they restrict the version of particular packages in the configuration files.

V. EVALUATION OF EXCEPY

In this section, we demonstrate the effectiveness of EXCEPY via answering the following three research questions.

- **RQ1:** Is the code pruning method effective in reducing buggy code size?
- **RQ2:** Is the benchmark effective in evaluating Python static analyzers?
- **RQ3:** Is the benchmark effective in evaluating Python fuzzers?

We evaluate the performance of static analyzers on both the complete buggy code and the simplified code, including execution time and outputs, to assess the efficacy of code pruning in answering RQ1.

We use state-of-the-art static analysis and fuzzers to answer RQ2 and RQ3 and evaluate their capacity to discover bugs gathered in EXCEPY. To check if the analyzers can correctly report the bugs, we compare the warning positions to the collected bugs in RQ2. Because all of the bugs in EXCEPY can cause crashes, we compare the tracebacks from fuzzers to the recorded tracebacks in EXCEPY in RQ3 to determine if the fuzzers can find the bugs.

A. Setup

All experiments are carried out on a PC with an Intel Core i7 3.60GHz CPU and 16 GByte RAM. The collected projects from GitHub are built on Ubuntu 18 with Python 3.7.

We adopt five state-of-the-art tools for the experiments, including three static analyzers and two fuzzers. We list the details of the five tools below.

- **Pytype** [21]. It is a Google-maintained static analyzer. It can infer variable types, find bugs in Python bytecode via inter-procedural analysis.
- **Pylint** [22]. This tool is also a static analyzer, released by Python Code Quality Authority (PyCQA). It can check for defects with variable types, suggest restructuring blocks, and offer information about the length of each line of code.
- **Pyflakes** [23]. Pyflakes is another tool produced by PyCQA. This tool parses the source file and examines the constructed syntax tree to detect errors.
- **pythonfuzz** [24]. It is a coverage-guided fuzzing tool maintained by Google, which detects unhandled exceptions and memory limitations in Python libraries.

862
• python-afl [25]. This tool applies American Fuzzy Lop (AFL) for pure Python code. Currently, it is an experimental module.

We set a time restriction of 60 minutes for three static analyzers and 12 hours for fuzzers for each bug. The comparison experiments in their original papers or supporting materials are referred to determine this setting.

B. RQ1: Evaluation of Code Pruning

To see how much code our approach can prune, we count the lines in the complete buggy code and compare them to the number in the simplified code. Figure 8 illustrates a comparison of the number of lines of code for 180 bugs, where we order the complete codes by the number of lines. Notice that while some bugs are from the same project, they are from different versions, which means they have varying quantities of lines of code. We use the terms complete and simplified to refer to complete and simplified code, respectively, and a red line to denote the half-size of the complete code. As seen in the diagram, our approach may remove over half of the project code.

We examine the situations where our approach can only prune less than 10% of the complete code, and we find code features that prevent the approach from removing more files. First, some Python files import a lot of modules and classes, but not all of them are used in the called classes or methods during test code execution. However, the Python Interpreter must go through all of the imported modules and classes during execution. Second, when importing a module, all Python files for initialization (file __init__.py) in the sub-modules are also executed. Though some sub-modules may not be used for triggering the bugs, we cannot delete them automatically. Third, many class definitions may exist in the same Python file. Though the test code only uses one or a few classes in this file, all the code in this file is regarded as being executed. Because we only prune code at the file level, these unexecuted lines within Python files can influence the number of pruned lines.

To show the benefits of code pruning for evaluating the tools, for each bug, we run three static analysis tools on both the complete buggy code and simplified code, and record the execution time using Linux command time. We classify the execution results in three categories: Finish within 60 minutes, Timeout, and Crash in 60 minutes, where Timeout refers to the situations that the tools do not stop within 60 minutes, and Crash in 60 minutes refer to the tools throw exceptions or stop due to errors. For every tool, we count the number of bugs belonging to these three categories, respectively.

Figure 9 shows the numbers of bugs for the three categories in each tool, under two versions of code. As it shows, tools using simplified code can not only avoid many failures but also considerably reduce running time. We conclude two causes for the crashes on the complete buggy code: (1) a tool may not be capable of handling some unsupported encoding formats or some Python syntax; (2) when a tool attempts to cover all execution paths, it crashes for the exponential number of paths in the unexecuted code.

However, for tool Pytype, the results in category Timeout increase rapidly with the simplified code. We manually check the logs generated during the execution and discover that (1) many bugs can be analyzed with the simplified code, instead of code crash; (2) the tool traverses lots of modules when inferring the types of some variables; and (3) the tool needs to deal with too many variables and does not complete the type inferring phase within 60 minutes.

Conclusion. According to the results of the experiments, the code pruning method can reduce the size of project code by half while not affecting bug reproduction. This approach can help to evaluate tools on large-scale projects.

C. RQ2: Evaluation with Three Static Analyzers

We employ three dimensions for further comparison of the analyzers in addition to reporting bugs.

• File level. This level shows that a tool can locate the file containing the bug.
• Method level. This level indicates that a tool is capable of locating the method that contains the bug.
• Line level. This level demonstrates that a tool can cover the lines where the bug arises.
• Bug level. This level means that the tool can report bugs correctly.

We use the simplified code to run the three tools and count the results on these four dimensions. We evaluate the outcomes within 60 minutes for this evaluation since tools may crash due to defective code.

Fig. 8: Number of lines of code in the complete buggy code and the simplified code.

Fig. 9: Performance of three static analyzers on the complete code and simplified code.
The number of detected bugs for each dimension is presented in Table IV. As the results indicate, Pylint can report warnings within the same files and same lines. Although the three tools can throw warnings in the same method and even on the same line as the bugs in EXCEPY, they can only report ten of 180 bugs collectively. All of the tools can detect these ten bugs in 3 minutes on code s_code. But some reports from tools Pylint and Pyflakes are not accurate enough.

To understand the capabilities and limits of these tools, we check the successfully reported bugs in detail. For the ten bugs, we list the related project, the type of the bug, and the detection result of three tools in the first five columns in Table V. In the last two columns in this table, we present the number of lines of code in the complete buggy code and the simplified code.

As demonstrated in the table, the three static analysis tools can detect correctly three bugs of type Nam. The first two tools can detect four bugs of type Att. When investigating bugs of type Att, we find that the three tools scan only the class definition code, without considering language feature dynamic attribute. For the bugs of type Nam, the tools check whether a variable has been claimed previously in the current scope. Furthermore, the ten bugs are triggered in a single Python file. However, most of the bugs in EXCEPY are caused by boundary input data or irregular execution paths, where the three tools cannot obtain enough information to infer the variable type and inspect. Therefore, the three tools may be good at locating bugs within a single Python file.

**Conclusion.** The three static analysis tools can still discover 10 out of 180 bugs based on the simplified code, even if the projects in EXCEPY are large-scale. They are, however, less efficient at identifying cross-file bugs or bugs linked to third-party packages.

**TABLE IV:** Distributions of the reports from three static analyzers.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Pytype</th>
<th>Pylint</th>
<th>Pyflakes</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Level</td>
<td>46</td>
<td>138</td>
<td>57</td>
</tr>
<tr>
<td>Method Level</td>
<td>18</td>
<td>96</td>
<td>7</td>
</tr>
<tr>
<td>Line Level</td>
<td>12</td>
<td>43</td>
<td>7</td>
</tr>
<tr>
<td>Bug Level</td>
<td>8</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

**D. RQ3: Evaluation with Two Fuzzers**

The two tools adopt code coverage to guide the generation of input data. To compute the coverage, they build their tracers to record the execution paths. Consequently, we cannot know if the input data generated by the tools can cover the buggy file or buggy function. Our solution is to compare whether the tracebacks generated by the tools match the tracebacks stored in EXCEPY. Because the two fuzzers need users to specify the entry point, the execution paths on both simplified and complete codes are the same. Hence, we do not compare the performance of the tools on the two kinds of code.

We limit the running time to 12 hours. The two fuzzers fail to trigger the bugs correctly, even though they can achieve more than 95% of the statement coverage on the buggy code. We execute the test code and analyze each execution path to see how much code needs to execute before the bug is triggered to figure out why the fuzzers failed. We count the number of lines and calls along the execution path. Among the 15 projects, project IPython replaces our tracer with its own, as the project needs to use the tracer for debugging. Therefore, we cannot receive data on this project from our tracer. For other projects, because we can track the execution of each statement and expression, we count each loop based on the times of execution and count all init methods that are invoked while importing packages and initializing objects.

The number of lines and method calls is presented in two aspects: distributions on each bug type and each project. Table VI provides the average number of calls and lines for each project, whereas the data for each bug type is shown in Table VII. In the two tables, the average number of method calls is shown in the second column, and the average number of lines for code being executed is shown in the third column. The project name and the type name are the same as in Table I and Table II.

We infer that there are two causes for their inability to discover bugs in EXCEPY. The bugs rely on certain types and values of input data. The two fuzzers are unable to generate the needed types since they only include default types. Second, because the fuzzers are coverage-driven in input data generation, the adopted techniques for triggering the bugs may prefer short paths over long.

**Conclusion.** In this evaluation, we found that a single criterion is insufficient to guide the generation of input data.

**TABLE V:** Information of the bugs that are successfully reported by three static analysis tools.

<table>
<thead>
<tr>
<th>Project</th>
<th>Bug Type</th>
<th>Pytype</th>
<th>Pylint</th>
<th>Pyflakes</th>
<th>#C_code</th>
<th>#S_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Att</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>273K</td>
<td>3K</td>
</tr>
<tr>
<td>P1</td>
<td>Unb</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>291K</td>
<td>62</td>
</tr>
<tr>
<td>P1</td>
<td>Att</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>274K</td>
<td>3K</td>
</tr>
<tr>
<td>P3</td>
<td>Att</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>97K</td>
<td>9K</td>
</tr>
<tr>
<td>P9</td>
<td>Att</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>47K</td>
<td>2K</td>
</tr>
<tr>
<td>P14</td>
<td>Typ</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>359K</td>
<td>202K</td>
</tr>
<tr>
<td>P14</td>
<td>Imp</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>455K</td>
<td>250K</td>
</tr>
<tr>
<td>P14</td>
<td>Nam</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>287K</td>
<td>181K</td>
</tr>
<tr>
<td>P14</td>
<td>Nam</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>408K</td>
<td>212K</td>
</tr>
<tr>
<td>P14</td>
<td>Nam</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>287K</td>
<td>181K</td>
</tr>
</tbody>
</table>

**TABLE VI:** Average number of function calls and lines of code required to trigger a bug in each project.

<table>
<thead>
<tr>
<th>Project</th>
<th>#Calls</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>404</td>
<td>2734</td>
</tr>
<tr>
<td>P3</td>
<td>403</td>
<td>2532</td>
</tr>
<tr>
<td>P5</td>
<td>1572</td>
<td>11K</td>
</tr>
<tr>
<td>P7</td>
<td>123</td>
<td>916</td>
</tr>
<tr>
<td>P9</td>
<td>663</td>
<td>58K</td>
</tr>
<tr>
<td>P11</td>
<td>1738</td>
<td>87K</td>
</tr>
<tr>
<td>P13</td>
<td>441</td>
<td>3083</td>
</tr>
<tr>
<td>P15</td>
<td>101K</td>
<td>556K</td>
</tr>
</tbody>
</table>

**TABLE VII:** Average number of function calls and lines of code required to trigger a bug in each project.

<table>
<thead>
<tr>
<th>Project</th>
<th>#Calls</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>P4</td>
<td>48K</td>
<td>243K</td>
</tr>
<tr>
<td>P6</td>
<td>5066</td>
<td>28K</td>
</tr>
<tr>
<td>P8</td>
<td>545</td>
<td>2915</td>
</tr>
<tr>
<td>P10</td>
<td>164</td>
<td>1757</td>
</tr>
<tr>
<td>P12</td>
<td>246</td>
<td>5153</td>
</tr>
<tr>
<td>P14</td>
<td>786K</td>
<td>3147K</td>
</tr>
</tbody>
</table>

864
### Related Work

**Benchmarks for Bug Collection.** For Python, there are a few benchmarks for bugs. The closest similar work is BugsInPy [10], which collected 493 bugs from 17 real-world Python projects by running newly added test cases on the previous version. Another Python benchmark is ManyTypes4Py [11]. Its purpose is to assess the performance of type inference. QuixBugs [8] collects Java and Python defects from small programs, as well as patches that change only one line of code. Projects gathered in QuixBugs are quite small, with 17 to 48 lines of code, and the bugs are manually entered, as opposed to BugsInPy and EXCEPY.

**Studies on Bugs in Python.** Many works concentrate on detecting bugs caused by using Python language features. For example, Rao and Chimalakonda [26] consider the potential problems caused by adopting Lambda expressions in Python applications, as well as some typical pitfalls. Hu and Zhang [3] investigate issues that occur when Python programmers use C-language APIs and provide some bug patterns when Python interacts with the C language. Chapman and Stolee [27] focus on the problems when using regular expressions in Python projects. The bug types in these researchers have been covered in EXCEPY.

**Analysis Techniques Towards Python Type System.** There are also several studies focusing on static analysis of the types in Python. Chen et al. [28] investigate the use of dynamic typing in nine real-world Python systems. Xu et al. [14] employ dynamic symbolic execution to infer type information and produce variable type ranges. Monat et al. [2] propose a method for applying abstract interpretation in Python code, which might assist in the analysis of type information. As many bugs collected in EXCEPY are type-related, these researchers can apply EXCEPY to evaluate the performance of their techniques in inferring types and finding conflicts in operations between various types.

### Threats to Validity

Some threats may influence the validity of EXCEPY, and we discuss these threats in this section.

**VIII. Conclusion and Future Work**

In this paper, we have proposed a Python benchmark named EXCEPY, which includes 180 bugs of Python built-in types from 15 real-world open-source Python projects. Moreover, we have also provided a buggy code reduction method for code simplification. EXCEPY provides both test scripts and buggy code (both the original and the simplified) for bug reproduction. Meanwhile, it also provides unified APIs and application to facilitate the evaluation of various tools. Apart from the benchmark, we have extracted 26 novel patterns for root causes and fix patterns of the collection types of bugs. To demonstrate the effectiveness of EXCEPY, we have also evaluated three static analyzers and two fuzzy testing tools on the ability of bug detection. Based on the evaluation results, we have provided some suggestions for improving the five state-of-the-art Python tools. In the future, we plan to enrich the database with more bugs and new built-in bug types.

IX. Acknowledgment

This work is supported in part by the Key Research Program of Frontier Sciences, Chinese Academy of Sciences under grant No. QYZDJ-SSW-JSC036.
REFERENCES


