



Towards understanding bugs in Python interpreters

Di Liu¹ · Yang Feng¹ · Yanyan Yan¹ · Baowen Xu¹

Accepted: 9 September 2022 / Published online: 13 December 2022

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

Python has been widely used to develop large-scale software systems such as distributed systems, cloud computing, artificial intelligence, and Web platforms due to its flexibility and versatility. As a kind of complex software, Python interpreter could also suffer from software bugs and thus fundamentally threaten the quality of *all* Python program applications. Since the first release of Python, more than 30,000 bugs have been discovered. While modern interpreters often consist of many modules, built-in libraries, extensions, etc, they could reach millions of code lines. The large size and high complexity of interpreters bring substantial challenges to their quality assurance.

To characterize the interpreter bugs and provide empirical supports, this paper conducts a large-scale empirical study on the two most popular Python interpreters – CPython and PyPy. We have comprehensively investigated the maintenance log information and collected 30,069 fixed bugs and 20,334 confirmed revisions. We further manually characterized and taxonomized 1200 bugs to investigate their representative symptoms and root causes deeply. Finally, we identified nine findings by comprehensively investigating bug locations, symptoms, root causes, and bug revealing & fixing time. The key findings include (for both interpreters): (1) the Library, object model, and interpreter back-end are the most buggy components; (2) unexpected behavior, crash, and performance are the most common symptoms; (3) *incorrect* algorithm logic, configuration, and internal call are the most common general root causes; *incorrect* object design is the most common Python-specific root cause; (4) some test-program triggering bugs are tiny (less than ten lines), and most bug fixes only involve slight modifications. Depending on these findings, we discuss the lessons learned and practical implications that can support the research on interpreters' testing, debugging, and improvements.

Keywords Bugs · Python interpreter · Empirical software engineering

Communicated by: Justyna Petke

✉ Yang Feng
fengyang@nju.edu.cn

¹ State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

1 Introduction

Python has become the top three popular programming languages in recent years Srinath (2017). As a dynamic programming language, Python is simple and flexible to write, which attracts plenty of developers and forms an extremely powerful software ecosystem Perez et al. (2010) and Raschka (2015). Besides, Python has several unique advantages: (1) Python has an extensive third-party library, which provides the support for the development of various scenarios Perez et al. (2010), (2) Python has a dynamic type system, it enables the attributes of objects flexibly changed according to the program context, (3) For code structure, Python also supports the dynamic insertion of modules and the dynamic construction of code, making the program structure change according to the user input Chen et al. (2016b) and Wang et al. (2015), (4) Python supports the hot deployment of code, which is conducive to maintaining and update of software Calmant et al. (2012). Because of these advantages, Python is widely used in the development of various large-scale software systems, such as distributed systems Leo and Zanetti (2010) and Dalcin et al. (2011), cloud computing Acuña et al. (2015), artificial intelligence Raschka (2015) and Pedregosa et al. (2011), and web platforms Forcier et al. (2008).

Unlike static language compilers, the execution of dynamic language programs often depends on the interactions between the interpreter and the run-time environment Reynolds (1972). The Python interpreter is responsible for converting Python source code into byte-code and executing on the virtual machine. Like any software, the Python interpreter is also affected by software defects. The bugs in the interpreter can cause the executed Python program to produce erroneous output or abnormal behavior, and the developers usually recognize the coding errors of the application. On the other hand, Python's flexible and dynamic characteristics make interpreter suffering from hidden risks. For example, Python's dynamic types and objects are implemented through dynamic references of heap space data. Compared with static languages, this implementation method is more susceptible to value manipulation errors and reference out of bounds, which can cause the interpreter to crash or produce wrong behavior Chen et al. (2014).

Although existing studies have investigated bug characteristics in various software domains, like software libraries Di Franco et al. (2017) and Islam et al. (2019), concurrency bugs Lu et al. (2008) and Leesatapornwongsa et al. (2016), performance bugs Jin et al. (2012) and Selakovic and Pradel (2016) and error-handling bugs Cacho et al. (2014) and Tian and Ray (2017), only a few of them involved interpreter or compiler bugs. Sun et al. (2016b) conducted the first empirical study on the compiler bugs of GCC&LLVM and discussed their characteristics. While these works have investigated buggy components and bugs' lifetime, they failed to analyze bugs' root causes and symptoms.

To fill this gap, we conducted a large-scale empirical study on two popular Python interpreters – CPython and PyPy to have a deep understanding of Python interpreter bugs. In total, we collected 27146 bugs of CPython and 2923 bugs of PyPy. To further understand the properties of interpreter bug fixes, we also examined 19732 CPython revisions and 602 PyPy revisions. By analyzing the data, we attempt to characterize the interpreter bugs from the following aspects:

- **Bug Distribution:** *Where are the bugs? What is the distribution of bugs in components and files?*
- **Bug Symptoms:** *To what extent do different bug symptoms occur?*
- **Root Causes:** *To what extent do different root causes of bug occur? What kinds of bug symptoms can each root cause produce?*

- **Bug Revealing & Fixing:** *What is the scale of bug-revealing test cases? What is the modification scale of the bugs that resulted from the different root causes?*

The remainder of this paper is organized as follows: Section 2 introduces the background knowledge of the Python interpreters. Section 3 overviews and motivates the research questions we investigate. Section 4 details the collecting and labeling process of source interpreter bugs. Section 5 covers the results of our empirical study and answers the research questions. Section 6 discuss the lessons learned and practical implications of our findings. Section 7 reviews prior related research on this topic. And finally, we draw conclusions in Section 8.

2 Background

In this section, we briefly overview the backgrounds. We first introduce the general architecture of Python interpreters, including the stages of Python source code execution. Then we highlight the two Python interpreters that we studied in this article.

2.1 Python Interpreter

Python is highly open source from the operating mechanism to the grammar specification, which makes its interpreters can run on almost all operating systems. According to different design purposes and implementation methods, many types of Python interpreters have emerged in recent years, such as CPython¹, PyPy², Jython³, and IronPython⁴, etc.

Although these interpreters differ in underlying implementation details, their structure of execution process and support modules are similar (Cao et al. 2015). Figure 1 shows the general architecture of Python, which contains the three major parts: Interpreter Core, Application Phase, and Implementation Phase.

- **Interpreter Core:** The Python interpreter core is designed with the source code as the input and the running results on the virtual machine as the output. Similar to traditional compilers, it consists of a front-end and a back-end. At the front end, the source program is first decomposed into a list of tokens by tokenizer and fed to the parser. Parsing is based on Python's specific syntax, and it transforms the token list to an Abstract Syntax Tree (AST). Then a Control Flow Graph (CFG) is constructed with the internal structure of AST nodes, which models program flow with a directed graph. The CFG blocks can represent the sequence of machine instructions executed in order, ie, the Bytecode. After that, the compiler will output the converted Bytecode to the back end and run it directly on the Python virtual machine (PVM) after optimization. Unlike binary machine code, Bytecode is a special instruction for PVM operations. It cannot be executed directly but only by the PVM specific to the target machine.
- **Application Phase:** The Python installers for most platforms (eg Windows and macOS) usually include the entire standard library and many additional modules, which can support Python programmers to achieve system functions in the application phase. In

¹<https://github.com/Python/cPython>

²<https://www.PyPy.org>

³<https://www.jython.org>

⁴<https://ironPython.net>

Fig. 1, we highlight the three most frequently used components in the application phase: Standard Library, Test Framework, and Extension Modules. The Standard Library contains built-in modules (written in C) that provide access to system functionality such as file IO and functional modules (written in Python) that provide standardized solutions for many problems during programming. The test framework targets both core committers and developers with needs. It supports test automation, sharing setup and shutdown code for tests, aggregating tests into collections, and independence from the reporting framework. In addition, Python also encourages developers to write extension modules in C or C++ to extend the interpreter. Those modules can define not only new functions but also new object types and their methods.

- Implementation Phase:** In the pre-execution phase, Python provides some auxiliary components, which act as the interpreter infrastructure: such as Build Module, Configuration Module, and Codecs (encoders and decoders). Build Module creates a virtual environment for the OS and provides the solutions for packaging and distribution solutions. In addition to the native tools, the Build Module also integrates mature packaging and dependency management tools in the Python ecosystem (eg pip and PyPI). Python allows developers to customize the Python environment using the Configuration Module, including performance options, debug options and security options in newer versions. Finally, because the default encoding formats of platforms are in different regions, Python needs to deal with exceptions caused by additional decoding requirements. Therefore, Python has many built-in codecs implemented through C functions or mapping tables. These codecs are constantly updated to increase the adaptation range and improve search performance.

2.2 Experiment Subjects

As mentioned previously, there are currently four primary implementation schemes of Python language specification: CPython, PyPy, Jython, and IronPython. All these interpreters are built with the similar architecture in Fig. 1. However, their update frequency and bug feedback vary significantly according to different application scenarios. Table 1 lists some basic information about these interpreters. We can observe that only CPython

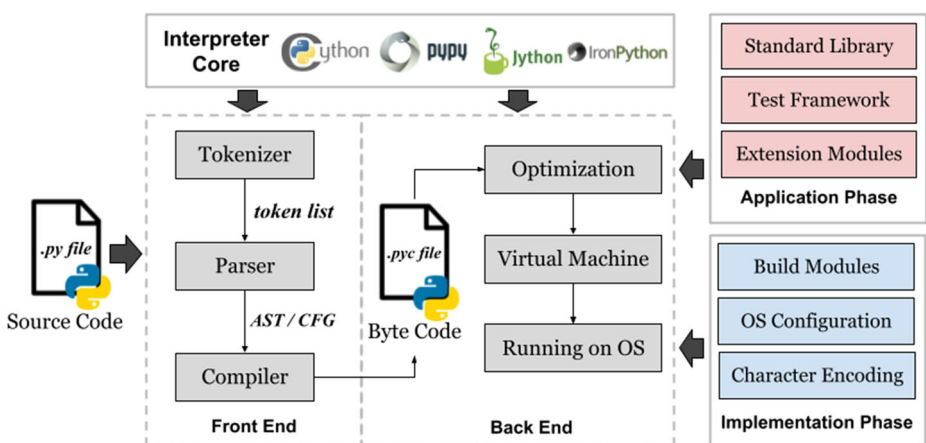


Fig. 1 The General Architecture of Python

Table 1 Information of Python interpreters

Interpreter	Latest Version	Latest Release	Bug Reports
CPython	3.10	Oct, 2021	50149
PyPy	7.3.7	Oct, 2021	2940
Jython	2.7.2	Mar, 2020	30
IronPython	3.4.0	Apr, 2021	231

and PyPy have recently released new versions and collected a considerable number of bug reports. Thus, we selected these two popular interpreters as experimental subjects in our study.

- **CPython:** CPython is a Python interpreter implemented by C and Python. As the official recommended interpreter for Python, CPython is the default and most widely used language implementation. CPython supports a variety of operating systems and has the richest extension modules. The original author of Python first published its initial release on Jan 26th, 1994. The latest stable version is 3.10, released on Oct 4th, 2021.
- **PyPy:** PyPy is another widespread implementation of Python. PyPy has been implemented in Python using a restricted subset of Python called RPython. Like CPython, PyPy also supports multiple platforms and is highly compatible with most existing Python code. PyPy usually runs faster than CPython because it includes a JIT(just-in-time) compiler. PyPy was initially a research project, and its mature version was released in about mid-2007. Its latest stable version is 7.3.7, which was released on Oct 25th, 2021.

3 Research Questions

To deeply understand bugs in Python interpreters, we design the following research questions concerning bug distribution, bug symptoms, the root causes, and bug revealing & fixing.

First, we examine the location and distribution of bugs in CPython and PyPy to investigate which components and files are most prone to bugs. This information allows researchers to know which parts of a Python interpreter require further effort in predicting, detecting, and repairing.

***RQ1:** Where are the bugs? What's the distribution of bugs in components and files ?*

Next, we focus on the frequency of different symptoms. Bug symptoms are the specific manifestation of the occurrence of the bugs, which can lead developers to triage the bugs. Thus, we have the second research question:

***RQ2:** To what extent do different bug symptoms occur?*

The kind and frequency of root causes are essential to understand CPython&PyPy bugs better. The extent to which a particular root cause may produce a specific symptom allows engineers to determine more actionable information about addressing a bug. This leads to our third research question:

***RQ3:** To what extent do different root causes of bug occur? What kinds of bug symptoms can each root cause produce?*

The immediate cause of a bug trigger is a crash or unexpected output caused by a test case, whose features can inspire developers to design modules to test new functionality. Meanwhile, developers usually follow a specific fix pattern when repairing the bugs with a similar root cause. Thus, we designed the last research question to extract the characteristics in bug revealing and fixes:

***RQ4:** What is the scale of bug-revealing test cases? What is the modification scale of the bugs that resulted from the different root causes?*

4 Methodology

This section introduces the preparations for our empirical study, including the data collection, the manual inspecting phase, and the relevant information to investigate.

4.1 Data collection

In this study, we used the two most popular Python interpreters as subjects. As Python language is entirely open-source, both CPython and PyPy chose Git to conduct distributed version control, in which CPython also deployed a synchronous issue tracker⁵ for bug submission and fix. This issue tracker contains most of the information on CPython that we needed for the study. The relevant information of PyPy can also be obtained through the issue directory in GitLab.

Note that we only focus on the *fixed* bugs because of two reasons: (1) these bugs have been confirmed and fixed by the core committers, and (2) the fixed bugs have relevant pull request information for a comprehensive understanding. A bug of CPython is fixed only if its *status* and *resolution* are set to *closed* and *fixed* respectively in the issue tracker. Similarly, a bug of PyPy is fixed only if its *status* is set to *resolved* in the issue repository.

To further investigate the fixes of these bugs, we also collected the corresponding revisions for them. In most instances, we can directly identify *Pull Requests*(*PR*) which *label* are set to *merged* as the revisions from the git repositories of CPython and PyPy. These *PRs* include the bug's id when they are named, which could help us trace the corresponding bug reports. However, CPython's Github bug repository can only be traced back to Feb 2017. Fortunately, its issue tracker still stores the earlier patch records of bug fixes as the revisions. Especially, if a CPython bug report contains both *PR* and *patch* information, we only collect the *PR* information in it.

⁵<https://bugs.Python.org>

In July 2020, as the Bitbucket team removed support for the Mercurial workflow⁶, the PyPy developers migrated their repositories to Gitlab.⁷ This caused many links in the issue tracker to fail, such as pull requests, fix history, and attachments. Therefore, we could not retrieve the revisions from these PyPy bugs reports.

Table 2 presents the number of bugs and their corresponding revisions that we used in this study. Finally, we obtained 27146 bugs and 19732 revisions for CPython, 2923 bugs, and 602 revisions for PyPy.

4.2 Data Preparation

To characterize the bugs in Python interpreters, we need to analyze the data that related to four research questions: **RQ1: Bug Distribution**, **RQ2: Bug Symptoms**, **RQ3: Root Causes**, and **RQ4: Bug Revealing and Fixing**. In following subsections we will introduce the data preparation of each RQ.

4.2.1 Data Collection for RQ1&RQ4

We could directly download the relevant information for RQ1 and RQ4 by running the crawler scripts. RQ1 focus on the bug location, which discusses how bugs are distributed in components and files. The buggy components responsible for issues can be obtained directly from the tags of reports. Also, we count the buggy files by analyzing the modifications of revisions in Section 4.1.

RQ4 discusses the bug revealing and their fixing scales. First, we collected relevant test cases from the attachments of issue reports to study how the bugs are triggered. Since the attachment list mainly contains files with two suffixes, the `.py` files that can reproduce bugs and the `.patch` files that the submitter suggests to fix the bugs, we only count the `.py` files as test cases. Take issue#12144⁸ of CPython as an example. Its attachments contain both the bug-revealing code and the fixing patches. We only collect the file named `cookielib-crash.py` as a test case. In this way, we collected 1730 test cases of CPython and 285 test cases of PyPy. Moreover, for the fixing scales, we could directly obtain the code changes from their revisions.

4.2.2 Classification and Labeling for RQ2&RQ3

Since the data we collected in Section 4.1 lacks original information on symptoms for RQ2 and root causes for RQ3, we need to complete the analysis through manual review.

As Section 4.1 shows, CPython bugs are several times more than PyPy. Considering the differences in their bug numbers and the coverage for multiple versions, in the experiment, we randomly selected 800 bugs for CPython and 400 bugs for PyPy. Although these bugs were marked as *fixed* when they were closed, many of them do not have substantial symptoms, eg, the enhancement of Python functions. We attempt to select bugs from different Python versions to ensure that the manual review could present the symptoms and root causes' general characteristics. In our investigation, nearly a quarter of the bugs belonged to the enhancement. The Python core committers encourage anyone to discuss and propose the

⁶<https://bitbucket.org/blog/sunsetting-mercurial-support-in-bitbucket>

⁷<https://doc.pypy.org/en/latest/faq.html?highlight=bug#why-doesn-t-pypy-use-git-and-move-to-github>

⁸<https://bugs.python.org/issue12144>

Table 2 The information of bugs and revisions

Interpreter	Start	End	Bug Reports	Revisions
CPython	2000.7	2021.10	27146	19732
PyPy	2005.5	2021.10	2923	602

enhancements through the mailing list. The solutions that have been approved will be publicized in the official documents, which are also known as PEP.⁹ Even though these reports can improve the Python ecosystem, we still filter them out because they could not represent any substantial symptoms. Finally, we have 667 labeled bugs for CPython and 327 labeled bugs for PyPy. We further understood the bugs by browsing the multiple information contained in reports and their revisions, including the comments, attachments, code changes, and documentation. To reduce the subjective bias of the labeling process, we adopted a cross-checking method. Each data set was assigned to two authors of this article separately, and a third author will check and discuss if their label results are inconsistent.

For symptoms, we followed the bug type officially recommended by CPython.¹⁰ The advantage is that these symptoms can be selected when submitting a bug report in the CPython issue tracker. Even if the submitter selects an inappropriate symptom, the core committers will complete the modification in time. Therefore, we only need to label the symptoms of PyPy bugs. The full list of *Symptoms* is as follows:

- **Behavior:** Unexpected behavior, results, or exceptions.
- **Compile error:** Errors reported by the compiler while compiling Python.
- **Crash:** Hard crashes of the Python interpreter, usually appears with a core dump or a Windows error box.
- **Performance:** Situations where too much time is necessary to complete the task. For example, a common task now takes significantly longer to complete.
- **Resource usage:** Situations where too many resources (eg, memory) are used.
- **Security:** Symptoms that might have security implications.

For root causes, prior research has summarized the root causes of different software defects and bugs Wan et al. (2017), Zhou et al. (2015), and Shen et al. (2021). In this paper, we adopted the taxonomy presented in Zhang et al. (2018) to categorize the root causes. At the start, we initialized the taxonomy of root causes (eg, coding and memory) derived from existing literature Seaman et al. (2008) and Thung et al. (2012). On finding a root cause that did not fit the initial taxonomy, each author conducting the manual analysis selected his label for the root cause. After this, all the authors would meet to resolve the labeling disagreements (about 6%) and mark them as an existing label or create a new root cause.

The disagreements partly arose between “Incorrect Algorithm Logic” and other root causes. Some bugs may involve large-scale code changes with obscure and complex logic structures, and they could be further subdivided into multiple root causes. We first consider the core developer’s comments in the reports when this happens. Take issue#23911¹¹ as an example: Although the fix for this issue contains thousands of code lines, the developer clarified it was caused by incorrect module initialization and building, so we marked it as

⁹<https://www.Python.org/dev/peps/>

¹⁰<https://devguide.Python.org/triaging/>

¹¹<https://bugs.Python.org/issue23911>

“Incorrect Configuration”. If their opinion of the bug cause is ambiguous and it cannot fit into any category we discussed, this bug will be marked as “Incorrect Algorithm Logic”.

After the first labeling round, we found that some root causes were too imprecise to describe bugs in Python (eg, typeobject design and GIL problem). Therefore, we conducted the second round of labeling to detail these root causes. We extracted the root causes with Python design features as *Python-specific root causes*. In contrast, the root causes summarized in the first round are referred to *general root causes*. In this way, we finally obtained 13 general root causes and 8 Python-specific causes.

The list of *General root causes* are as follows:

- **Incorrect Algorithm Logic (Logic):** The implementation of the algorithm’s logic is incorrect, and cannot be fixed by addressing only one of the other root causes.
- **Incorrect Numerical Computation (Num):** This cause involves incorrect numerical calculations and value usages (eg, Issue-I in Table 3).
- **Incorrect Assignment (Assi):** One or more variables is incorrectly assigned or initialized (eg, Issue-II in Table 3).
- **Incorrect Condition checks (ICC):** The necessary conditional checks are Incorrect.
- **Incorrect Regular Expression (RE):** This cause involves the misuse of the regular expressions.
- **Exception Handling Error (Exc):** This cause involves incorrect handling of exceptions and warnings.
- **Coding (Cod):** This cause involves the misuse of coding (eg, incorrect encoding and decoding of Unicode).
- **Memory (Mem):** This cause involves the misuse of memory (eg, improper memory allocation or de-allocation).

Table 3 Bug Examples for General Root Causes (CPython)

	Issue	Root Cause	Description	Critical Fix
I	#37315	Num	<i>math.factorial()</i> accepts integer-like objects (including objects with defined <code>index</code> .) as well as float instances with integral value.	made <i>factorial()</i> consistent with the integer-based <code>math</code> module functions
II	#35062	Assi	<code>io.IncrementalNewlineDecoder</code> gets a <i>translate</i> bitwise struct field, but it could be assigned arbitrary int value.	add a check function for validating argument ranges for <i>translate</i>
III	#32129	ExCall	<i>Tk</i> on macOS has implemented the <code>wm.iconphoto</code> command. <i>IDLE</i> that links to this version of <i>Tk</i> results in a highly blurry icon image in the Dock.	avoid blurry <i>IDLE</i> application icon on macOS with <i>Tk</i> 8.6
IV	#36374	InCall	If function <i>PyDict_SetDefault()</i> fails in <i>merge_consts_recursive()</i> , <i>Py_INCREF()</i> will be called on a null pointer.	modify the algorithm of <i>merge_consts_recursive()</i> when <i>PyDict_SetDefault()</i> fails
V	#33363	Version	<i>async</i> for statement is not a syntax error in sync context. Since 3.7, <i>async</i> generators can be used in non- <i>async</i> functions.	raise a <code>SyntaxError</code> for <i>async with</i> and <i>async for</i> statements outside of <i>async</i> functions.

We use emphasis to highlight these APIs and values that can help readers better understand this part

- **Incorrect External Call (ExCall):** This cause involves incorrect call of functions, methods, and interfaces from external systems or libraries (such as packages from PyPI). An example: Issue-III in Table 3.
- **Incorrect Internal Call (InCall):** This cause involves incorrect call of built-in functions, methods, and interfaces (such as modules from the standard library). An example: Issue-IV in Table 3.
- **Concurrency (Conc):** This cause involves the misuse of concurrency-oriented structures (eg, locks, critical regions, threads, etc).
- **Invalid Documentation (Doc):** This cause involves incorrect manuals, tutorials, and code comments.
- **Version (Ver):** This cause involves issues caused by differences in Python versions (eg, Issue-V in Table 3).
- **Incorrect Configuration (Config):** This cause involves modifications to files for building, compatibility, and installation.
- **Others:** Causes that occur highly infrequently and do not fall into any one of the above categories.

After the second round of labeling, the *Python-Specific Root Causes* are as follows. We provide each of them a bug example in Table 4:

- **Incorrect Object Design (Obj):** This category of bugs involves problems in the Python object system. According to the error characteristics of the object, we further divide

Table 4 Bug examples for Python-specific root causes (CPython)

	Issue	Root Cause	Description	Critical Fix
I	#12414	Obj	<code>sys.getsizeof()</code> on a code object returns on the size of the code struct, not the arrays and tuples which it references.	modify the return value of <code>sys.getsizeof()</code>
II	#42349	ByCo	The front-end of the bytecode compiler produces a broken CFG (control flow graph), which makes the back-end optimizations unsafe.	insert a check that the CFG is well-formed before optimizations, then fix up the front-end
II	#40048	VM	During VM executing, function <code>_PyEval_EvalFrameDefault()</code> does not reset <code>tstate→frame</code> if <code>_PyCode_InitOpccache()</code> fails.	reset the frame object properly
IV	#43751	Cor	Await <code>anext()</code> incorrectly return None when a default value is provided as the second argument.	modify the return value of <code>anext()</code>
V	#33021	Mut	Some <code>fstat()</code> calls do not release the GIL, possibly hanging all threads and cause the program to get stuck.	release the GIL during function <code>fstat()</code> calls
VI	#37424	Mup	The timeout check of <code>subprocess.run()</code> doesn't work if argument <code>shell=True</code> and <code>capture_output=True</code> .	modify the check timeout method of <code>subprocess.run()</code>
VII	#37990	GC	<code>gc.collect()</code> prints debug stats incorrectly, since <code>PySys_FormatStderr</code> doesn't support <code>"%.4f"</code> format	modify the print format of <code>gc.collect()</code>

We use emphasis to highlight these APIs and values that can help readers better understand this part

this category into four subcategories: (1) Incorrect underlying definition of objects (eg, wrong type, wrong inheritance, or incorrect return value), (2) Incorrect dynamic behavior of objects (eg, illegal property or function addition), (3) Incorrect operation of objects (eg, illegal call, or reference undefined methods), and (4) Unreasonable memory allocation of objects, including the destruction of objects.

- **Incorrect Bytecode Generation (ByCo):** This cause involves the problems during bytecode generation, such as disassembly errors and byte code injection errors.
- **Incorrect virtual machine execution (VM):** This cause involves problems during VM execution, such as the incorrect building of running or execution environment, misuse of stack memory, and operating system incompatibility.
- **Misuse of coroutine (Cor):** This cause involves the concurrency problems at the coroutine level (eg, context switching errors).
- **Misuse of multi-thread (Mut):** This cause involves the concurrency problems at the thread level (eg, release error of global interpreter lock, ie, *GIL*).
- **Misuse of multi-processing (Mup):** This cause involves the concurrency problems at the process level (eg, buffer exception and deadlock error caused by forking).
- **Incorrect garbage collection (GC):** This cause involves errors caused by garbage collection, such as unexpected object wipes and missing collection support.

5 Experimental Results

In this section, we analysis the experimental results and answer the four research questions.

5.1 RQ1: bug distribution

The first research question focus on the bug location. We mainly discuss how bugs are distributed in components and files.

5.1.1 Distribution of Bugs in Components

Some reporters choose to check the component list provided by the Python maintainer when submitting bugs. In our study, 91% of the CPython bugs were tagged with component information. However, only 15% of PyPy bugs are tagged, partly due to indefinable descriptions of the components.

Table 5 presents the statistics of buggy components of CPython. Since the CPython issue tracker allows developers to tag multiple components for one bug report, we also present the bugs which involve multiple components in Table 6a. As Table 6a shows, 88% of the CPython bugs involve only one component, 10% bugs contain two components, and less than 3% bugs involve more than three components. This indicates that Python's component structure is highly independent, bugs rarely involve multiple components. Only two bugs involve five components (issue#24245 and issue#6007), both proposed by core committers. We provide the details of issue#24245¹² in Table 7, the committers removed the do-nothing exception handlers to improve the readability of the code, so it involves multiple components like Distutils, IDLE, Library, etc. These handlers are unnecessary because they are at the end of the exception handler list. Issue#6007¹³ involves similar components.

¹²<https://bugs.Python.org/issue24245>

¹³<https://bugs.Python.org/issue6007>

Table 5 The description of components in CPython

Component	Description
Library(Lib)	Python modules in Lib
Interpreter Core	Contains the built-in objects in Objects, the miscellaneous source files for Python, Grammar, and Parser
Tests	Contains the unittest framework, the test runner and the test support utilities
Extension Modules	Contains the standard library extension modules, or former extension modules that are now built-in modules
Windows	Contains specific implementations and configurations for running Python on Windows
Build	The build process of Python
IDLE	Python's Integrated Development and Learning Environment
macOS	Contains specific implementations and configurations for running Python on macOS

The committers refined the documentation for distutil. They added a disclaimer about some extensions not compiling to prevent outdated documents from misleading developers.

Finding 1: 88% of the CPython bugs involve only one component.

Table 6b presents the most buggy components of CPython. Table 5 details the functionality and structure of these components. We observe that the library is the most buggy component in CPython, accounting for a quarter of all the bugs. The most buggy components after that are Interpreter Core (13%) and Tests Module (8%). Together, these three

Table 6 Statistics of buggy components in CPython

(a) The number of components in each tagged bug report		
Num	Reports	%
1	21768	88.13
2	2623	10.62
3	278	1.13
4	27	0.11
5	2	0.01
(b) The Top 8 buggy components of CPython		
Component	Reports	%(of all)
Library(Lib)	6215	25.16
Interpreter Core	3186	12.90
Tests	2013	8.15
Extension Modules	1415	5.73
Windows	1104	4.47
Build	1064	4.31
IDLE	555	2.25
macOS	466	1.89

Table 7 The information of issue#24245

Title: Eliminate do-nothing exception handlers	
Resolution: fixed	Version: Python 3.5 Priority: normal
Components: Distutils, IDLE, Library (Lib), Tests, Tkinter	
<div>  <div> vadmium commented on 20 May 2015 <div>Member Author</div> </div> </div> <p>These changes remove exception handlers that simply reraise the exception. They are not needed because either they are at the end of the exception handler list, or the exception being reraised would not be caught by any other handler (e.g. no need to reraise SystemExit if you are only interested in Exception). I think they make the code more confusing to read.</p> <p>I noticed the tkinter/font.py dead code when reading the code, Victor Stinner pointed out the distutils case in bpo-21259, and I found the rest by searching for similar cases.</p> <pre> Lib/distutils/core.py 2 -- Lib/idlelib/idle.pyw 26 ++++++++----- Lib/idlelib/rpc.py 5 +---- Lib/test/regtest.py 4 ---- Lib/test/test_queue.py 10 +----- Lib/test/test_urllib2net.py 2 -- Lib/tkinter/font.py 2 -- Lib/unittest/test/support.py 4 ---- </pre> <div>Bug Description</div>	
<div>  <div> python-dev mannequin commented on 20 May 2015 <div>Mannequin</div> </div> </div> <p>New changeset 004c689d259c by Serhiy Storchaka in branch 'default': Issue bpo-24245: Eliminated senseless expect clauses that have no any effect. https://hg.python.org/cpython/rev/004c689d259c</p> <p>New changeset 56e1d24806b3 by Serhiy Storchaka in branch '2.7': Issue bpo-24245: Eliminated senseless expect clauses that have no any effect in https://hg.python.org/cpython/rev/56e1d24806b3</p> <p>New changeset f10ba5313fbb by Serhiy Storchaka in branch '3.4': Issue bpo-24245: Eliminated senseless expect clauses that have no any effect in https://hg.python.org/cpython/rev/f10ba5313fbb</p> <p>New changeset 5deb169ebb22 by Serhiy Storchaka in branch 'default': Issue bpo-24245: Eliminated senseless expect clauses that have no any effect in https://hg.python.org/cpython/rev/5deb169ebb22</p> <div>Comments of Fix</div>	

components are responsible for nearly half of the bugs. Possible reasons for this can be found in Table 5. The library for CPython is officially released with the Python standard library, which is extensive and offers a wide range of facilities. The library contains built-in modules that provide access to system functionality, such as file I/O and modules that provide standardized solutions programming requirements. Although Python has become one of the most widely used languages, the existing research or industrial efforts rarely focus on the quality assurance of its library. The Interpreter Core becomes the second bug-prone component. According to Chen et al.'s survey of compiler testing, the Interpreter Core has a more complex implementation and optimization in most interpreter architectures (Chen et al. 2020). Compared with statically typed languages, Python interpreter testing faces difficulties like unpredictable dynamic behavior and generating reasonable test-bytecode (Chen et al. 2017). At present, there is still a lack of test tools that are specifically designed for Python interpreters. Although some compiler testing techniques (eg, EMI (Le et al. 2014)) can be reused, most of them have not been completed for interpreters already. Comparing the data in Table 6 with the architecture of Python execution in Fig. 1, we find the CPython bugs mainly come from interpreter core and application phase. However, which particular

stage is prone to bugs is not explicit (such as the front-end or back-end of the interpreter), so we further analyzed the most buggy files in the following subsection.

Table 8 presents the statistics of buggy components in PyPy. PyPy moved the repository from BitBucket to GitLab in mid-2020. The new bug tracker does not support checking components when submitting, so the data in the table is derived from the repository. PyPy's Bug Tracker does not provide a clear division of its components, and most reporters submitted bugs without specifying their components. In the bugs we collected, the component that showed up most frequently was PyPy3, which is the primary implementation of Python3. Python3 is an upgrade from earlier versions of Python. As the Python development team discontinued support for Python2, Python3 became the most crucial feature in PyPy. We note that, as a standalone component, RPython accounts for 8% of the bugs. RPython is a static subset of Python, it follows Python's syntax specification but removes the dynamic feature. RPython writes most of the functionality of the PyPy interpreter. This allows PyPy's compilation toolchain to perform type inferences on RPython code statically and achieve a faster running speed than CPython.

Finding 2: The most buggy CPython component is library, accounting for more than 25% of the bugs. The second buggy component is Interpreter Core. The most buggy PyPy component is PyPy3, which accounting for 46% of the bugs.

5.1.2 Distribution of bugs in files

To investigate how bugs are distributed in different source files, we compute statistics of the most modified files in revisions and present the results in Table 9.

As Table 9a shows, the distribution of buggy files and components in CPython presents a similar central tendency. The most buggy files are mainly from four components: Interpreter Core, Library, Extension Modules, and Tests. Among the files related to the Interpreter Core, we note that the definition for objects appeared three times. This reflects the most significant feature of Python: *everything is an object*. Unlike static languages, classes and functions in Python are also built by instantiation of the type objects. Figure 2 briefly demonstrates the partly architecture of the Python object system with files in Table 9a. *Object* is the header information of all objects, and it maintains a pointer to the *TypeObject*. By instantiating the methods and properties of *TypeObject*, Python can build specific types, such as *Numeric Objects* (int, float), *Sequence Objects* (list, tuple, Unicode) and *Mapping Objects* (dict, set). The design of these objects needs to implement language features and interact with the frequent calls in the application stage. The wider the application scenario,

Table 8 Statistics of buggy components in PyPy

Component	Reports	%
PyPy3(RunningPython3.x)	200	46.19
PyPy2(RunningPython2.7)	180	41.57
RPython	36	8.31
WindowsRunningPyPy3	12	2.77
WindowsRunningPyPy2	5	1.15

Table 9 The top 20 buggy files of CPython and PyPy

File	Cnt	Component	Description
(a) CPython			
Modules/posixmodule.c	330	Extension Modules	POSIX module implementation
Objects/unicodeobject.c	293	Interpreter Core	Unicode object design
Python/ceval.c	242	Interpreter Core	Compiled code execution
Python/py/lifecycle.c	229	Interpreter Core	Interpreter top-level routines
configure.ac	216	Build	Configure scripts generation
Makefile.pre.in	216	Build	Top-level Makefile for Python
Objects/typeobject.c	174	Interpreter Core	Type object design
Python/compile.c	169	Interpreter Core	Compiling AST into bytecode
test/test_os.py	166	Tests	Testing the OS module
test/test_ssl.py	161	Tests	Testing the SSL & sockets
Modules/_ssl.c	161	Extension Modules	SSL socket module
Python/sysmodule.c	159	Library	System module interface
_test_multiprocessing.py	155	Tests	Tests for the multiprocessing
_testcapimodule.c	152	Tests	Testing interpreter C APIs
Lib/idlelib/configdialog.py	151	Library	Displaying configuration dialogs
Objects/dictobject.c	150	Interpreter Core	Dictionary object design
Modules/socketmodule.c	138	Extension Modules	Socket module implementation
Lib/inspect.py	133	Library	Inspecting live objects
Lib/unittest.py	131	Tests	Unit test framework
Python/import.c	129	Library	Importing interface

Table 9 (continued)

(b) PyPy			
cpyext/typeobject.py	21	Type object for C-API	
rPython/rlib/rposix.py	19	POSIX implementation for RPython	
cpyext/api.py	18	API implementation for C	
cpyext/slotdefs.py	17	Slot implementation in Type object	
tool/release/package.py	16	PyPy packaging	
cpyext/test/test_typeobject.py	14	Testing the Type object for C-API	
posix/test/test_posix2.py	14	POSIX Testing	
rPython/annotator/test/test_annotPython.py	12	Testing the RPython annotator	
posix/interp-posix.py	12	POSIX interpolating	
interpreter/baselibspace.py	12	Object space implementation	
interpreter/app_main.py	11	Main entry point handling	
micronumpy/test/test_ndarray.py	9	Testing ndarray for micronumpy	
rPython/annotator/unaryop.py	9	Unary operations of annotator	
objspace/std/bytearrayobject.py	8	The built-in bytearray implementation	
micronumpy/ndarray.py	8	ndarray implementation of micronumpy	
cpyext/unicodeobject.py	8	Unicode implementation for C-API	
micronumpy/interp_numarray.py	8	numarray interpolating of micronumpy	
posix/_init_.py	8	POSIX initialization	
cpyext/test/test_cpyext.py	7	Testing C-API	
rPython/memory/gc/incminimark.py	7	Incremental version of the MiniMark GC	

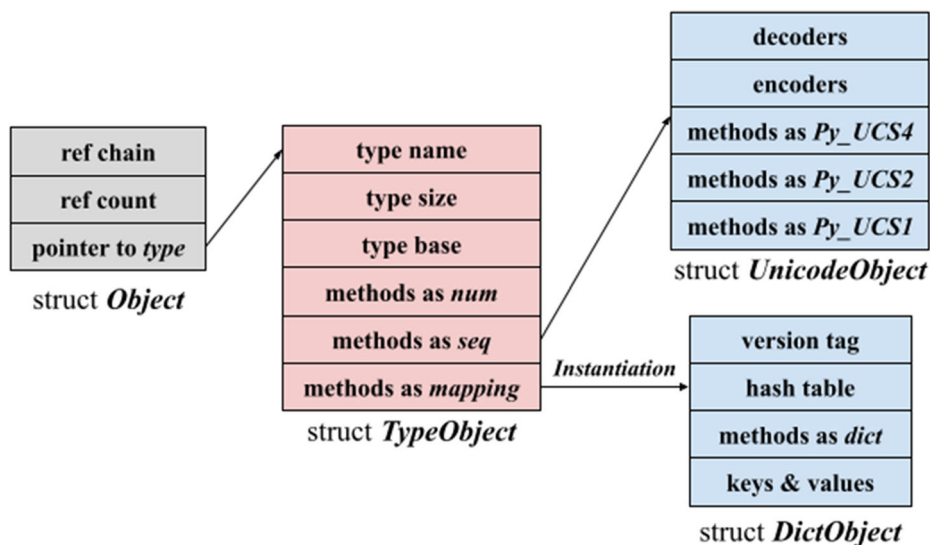


Fig. 2 The Object System of Python

the easier the object model is to contain bugs. For example, *TypeObject* defines a large number of C function pointers to determine the specific behavior of different type objects at runtime, which is easy to cause reference logic errors and memory errors. Similarly, *DictObject* undertakes the critical tasks of running Python virtual machine, while *UnicodeObject* needs to handle the encoding and decoding for dozens of character systems.

The other files related to the Interpreter Core all come from the back-end. Such as *pylife-cycle.c* controls the initialization and exit of the interpreter, *compile.c* is responsible for generating bytecode objects, and *ceval.c* is responsible for executing bytecode objects on the virtual machine. However, we did not find any front-end files in Table 9a. One reason is that the essence of front-end design is text processing, in which the implementation process and algorithm are relatively mature. For example, although CPython completely rewrites the Parser (LL(1) to PEG) in version 3.9¹⁴, we hardly find many bugs in it. Nevertheless, the back-end implementation needs to be more consistent with the language features, including resource allocation and optimization, making it more complex and unpredictable.

As mentioned before, Library is the most buggy component in CPython, but we cannot find many library files in Table 9a. This proves that the division of Library is clear, the functions corresponding to each file (or module) are independent. Once a file contains a bug, the impact on other files is relatively limited. In our investigation, 64% of files in the Library contain only one bug. However, its test module always needs to test the multiple functions of different source files, eg, *testcapimodule.c* tests multiple functions for C-API, which also proves why the files in the Tests component contain more bugs.

For PyPy, according to Table 9b the most buggy files are mainly from five modules: cpytext, rPython, interpreter, micrumpy, and POSIX. Among them, rPython and micrumpy are unique modules of PyPy. cpyext (CPython Extension Modules) appears six times in Table 9b, becoming the module with the most occurrences. cpyext is the compatibility

¹⁴<https://www.Python.org/dev/peps/pep-0617/>

layer of PyPy to the c extension packages, which implements the call to C-API in CPython, allowing developers to call various extension packages like NumPy and pandas expediently. In recent years it has undergone several changes and improvements in version updates. Frequent modification causes it to contain the most buggy files in PyPy.

By summarizing the files that simultaneously appear in Table 9a and 9b, we could list some files prone to bugs during the different Python implementations. First, the object design with complex instantiation functions and dynamic features are easier to contain bugs, such as *TypeObject* and *UnicodeObject*. We also found that the POSIX (Portable Operating System Interface) module has the highest and the second bug frequency in CPython and PyPy, respectively. The POSIX module provides standardized access to multiple operating systems. Its bugs mainly come from the incompatibility of operating system updates, such as file transfer exceptions, OS API support problems, and localized coding errors. Another common buggy files are from the Test module, which bugs are generally led by large-scale source file modification. Maintainers need to perform regression tests on the modified source files, and many test cases will be rewritten at this stage. Therefore, they always appear in pairs in Table 9, such as *_ssl.c* with *testssl.py* for CPython, and *_ndarray.py* with *test_ndarray.py* for PyPy.

Finding 3: The most buggy CPython files are from the Interpreter Core, Library, and Tests. The Interpreter Core related files mainly relate to the object design and back-end (which is contrary to GCC/LLVM). The most buggy PyPy files are from cpytext, rpython, micronumpy, and POSIX. The most buggy files in both interpreters are from object design, POSIX, and test modules.

5.2 RQ2: bug Symptoms

The following research question we discuss covers the symptoms that bugs exhibit. Since we followed the standard of the CPython issue tracker to define the symptoms (564 of 667 are tagged), we only need to complete the labeling for PyPy. Table 10 illustrates the statistics of bug symptoms. Although the data size and determination methods are different, the symptom distribution of CPython and PyPy are similar. Their three most frequent symptoms are behavior, crash, and performance.

Behavior is the most common symptom, with 394 cases in CPython and 251 cases in PyPy. CPython Developer's Guide provides a clear definition of behavior, "*Unexpected*

Table 10 Symptoms of bugs in CPython and PyPy

Symptoms	CPython	PyPy	$Total_{Sym}$	%
Behavior	394	251	645	72.39
Compile Error	23	11	34	3.82
Crash	61	34	95	10.66
Performance	42	15	57	6.39
Resource Usage	30	9	39	4.38
Security	14	7	21	2.35
Total	564	327	891	-

Table 11 The information of issue#38293

Title: Deepcopying <i>property</i> objects results in unexpected TypeError	
Resolution: fixed	Version: Python 3.5
Components: Library (Lib)	Priority: normal

GudniNatan commented on 28 Sep 2019 • edited by miss-islington

Contributor

Copying property objects results in a TypeError. Steps to reproduce:

```
>>> import copy
>>> obj = property()
>>> copy.copy(obj)
```

This affects both shallow and deep copying.
My idea for a fix is to add property objects to the list of "atomic" objects in the copy module.
These already include types like functions and type objects.

Bug Description

brandtbucher left a comment • edited

Member

Thanks @GudniNatan, and welcome to CPython. This looks good!

I think this should have a NEWS entry. It's a new feature (and there's precedent for it when `copy / deepcopy` support was added for other objects in the past). [Creating one is super quick!](#)

Maybe just something like:

```
Add :func:`copy.copy` and :func:`copy.deepcopy` support to :func:`property` objects.
```

Just out of curiosity, what was your use case for wanting to copy these?

Comments of Core Developer

Lib/copy.py

@@ -107,7 +107,7 @@ def copy(x):

107 107 def _copy_immutable(x):

108 108 return x

109 109 for t in (type(None), int, float, bool, complex, str, tuple,

110 - bytes, frozenset, type, range, slice,

110 + bytes, frozenset, type, range, slice, property,

111 types.BuiltinFunctionType, type(Ellipsis), type(NotImplemented),

112 types.FunctionType, weakref.ref):

113 113 d[t] = _copy_immutable

@@ -195,6 +195,7 @@ def _deepcopy_atomic(x, memo):

195 195 d[types.BuiltinFunctionType] = _deepcopy_atomic

196 196 d[types.FunctionType] = _deepcopy_atomic


197 197 d[weakref.ref] = _deepcopy_atomic

198 + d[property] = _deepcopy_atomic

Changes of Code

behavior, result, or exception.” Table 10 shows that more than half of the symptoms were classified as unexpected behavior, such as lack of function/improvement of function, triggering of exceptions/warnings, and failure of the building process. The bugs of functions possibly come from the errors in extensive libraries provided by CPython and PyPy. Developers may encounter unexpected results when calling these functions, which occurs even for very commonly used functions. For the instance in Table 11, someone found that performing either a shallow copy or a deep copy to a property object would result in a type error.¹⁵ This bug can be reproduced on multiple Python versions (3.7-3.9). Another unexpected behavior, triggering exceptions/warnings, refers to failing to properly catch and handle the exceptions or issuing a warning. For instance, the factorial function in the math module accepts

¹⁵<https://github.com/Python/cPython/pull/16438>

 Springer

non-integral decimal numbers without giving any exception message.¹⁶ The building errors may come from the various operating systems or running environments for CPython, which involves the lack of dependencies, the illegal system calls, the wrong identification of the file directory, and the version incompatibilities.

Crash is the second common symptom, with 65 occurrences across CPython and PyPy. This symptom differs from triggering an assertion or exception and usually appears with a core dump (Unix) or an error box (Windows). When the program crashes, most operating systems will report the exceptions in error messages. However, there are only a few cases where this information can be directly reviewed in the reports that we have classified. Rather than marking the bug as a crash, some developers prefer to directly report the crash in the comments, and mark it as another symptom.

In addition, due to the complexity of crash trigger conditions, many modules' exception prompts need to be improved.¹⁷ In our observation, the most common crash errors are displayed as illegal access to memory, such as invalid dereferences to null-pointers, writes to read-only memory areas, and access to protected memory areas, etc.

Performance is another notable symptom that occurs—totaling 57 instances for CPython and PyPy. The situation of this symptom differs significantly between CPython and PyPy. For CPython, performance issues are mostly developers' challenges to program execution time. In many cases, they optimize the algorithm logic to achieve a faster running speed of the original program, such as someone used a pre-calculated hash method to increase the speed of type creation by 20%. For PyPy, performance issues are mainly the comparison with CPython running speed. Most developers use PyPy in pursuit of higher compilation speed. When they find that some modules PyPy runs slower than CPython, they report them as performance issues. However, there are two types of modules involved in these bugs: built into PyPy, or from third parties. Such as numppypy (provide by PyPy) and numpy (mainly face the optimization of CPython). For the performance issues caused by modules like numpy, the developers will admit that there is no good solution to fix them.¹⁸ The slow speed of PyPy's NumPy is due to many C-API calls, and these APIs are only optimized for CPython.

Finding 4: Unexpected behavior, Crash, and Performance are the most common symptoms in both CPython and PyPy, which accounting for 72%, 11%, and 6% of bugs, respectively.

5.3 RQ3: root causes

To answer RQ3, we first discuss the bug distribution by root causes. After that, we investigate the relationship between root causes and symptoms by discussing which particular root cause may produce a specific symptom.

¹⁶<https://bugs.Python.org/issue33083>

¹⁷<https://bugs.Python.org/issue33871>

¹⁸<https://foss.heptapod.net/PyPy/PyPy/-/issues/1784>

5.3.1 Bug distribution by root causes

Table 12 present the bug numbers by different root causes. The upper part of the table revolves around the general root causes which also exists in other software systems, and the lower part revolves around the Python-specific root causes.

For both interpreters, the top three general root causes that lead to most bugs are Incorrect Algorithm Logic (Logic), Incorrect Configuration (Config), and Incorrect Internal Call (IncCall). Incorrect Algorithm Logic caused the most bugs, including 260, accounting for a quarter of all bugs. Such root cause has been proved to be the most common cause of bugs in other software systems (Shen et al. 2021; Garcia et al. 2020). It can be logic errors

Table 12 Bug distribution by root causes in CPython and PyPy

General Root Cause	CPython	PyPy	$Total_{Rc}$	%
Incorrect algorithm logic (Logic)	171	89	260	26.16
Incorrect numerical computation (Num)	6	1	7	0.7
Incorrect assignment (Assi)	9	12	21	2.11
Incorrect condition checks (ICC)	19	8	27	2.72
Incorrect regular expression (RE)	11	2	13	1.31
Exception Handling Error (Exc)	51	13	64	6.44
Coding (Cod)	18	15	33	3.32
Memory (Mem)	37	17	54	5.43
Incorrect external call (Ex-call)	19	39	58	5.84
Incorrect internal call (In-Call)	102	48	150	15.09
Concurrency (Conc)	47	12	59	5.94
Invalid Documentation (Doc)	60	3	63	6.34
Version (Ver)	35	11	46	4.63
Incorrect Configuration (Config)	76	51	127	12.78
Others	6	6	12	1.21
Total	667	327	994	-
Python-Specific Root Cause	CPython	PyPy	$Total_{Rc}$	%
Incorrect object design (Obj)	43	17	60	34.09
Incorrect bytecode generation (ByCo)	12	8	20	11.36
Incorrect virtual machine execution (VM)	15	18	35	19.89
Misuse of Coroutine (Cor)	11	-	11	6.25
Misuse of Multi-thread (Mut)	21	2	23	13.07
Misuse of Multi-processing (Mup)	15	5	20	11.36
Incorrect Garbage Collection (GC)	6	1	7	3.97
Total	125	51	176	-

in the algorithm implementation, logic modifications during code optimization, or even human errors in the development.¹⁹ Repairing its resulting errors require non-trivial modifications, including extensive changes of code in multiple files. Therefore, studies show that accurately locating and automatically fixing this kind of bug remains severe challenges (Ghanbari et al. 2019; Motwani et al. 2018; Xia et al. 2016).

Incorrect Configuration (Config) and Internal Call (In-Call) also lead to a substantial number of bugs, which appeared 127 and 150 times, respectively. The configuration phase constructs the Python runtime environment, including initialization, directory building, system compatibility, and interactive input editing. How to run Python on disparate devices is an unavoidable problem for users. One solution is the constant update and testing of the diversified operating environment by Python developers. Incorrect Internal Call involves wrong calls of internal functions, methods, and interfaces. The most common incorrect call is to reference an internal function to a null pointer.²⁰ The results are unpredictable, producing abnormal output, performance degradation, or even an interpreter crash.

Not all general root causes present similar distribution trends in CPython and PyPy, eg, Invalid Documentation (Doc) and Incorrect External Call (Ex-Call). Invalid Documentation involves incorrect manuals, tutorials, and code comments, and it causes 63 bugs in CPython but only one in PyPy. One possible reason is that, unlike PyPy, CPython maintains many reStructuredText files as manuals for developers. Among them, the files contained in the library alone exceed 300.²² Ensuring the accuracy of these documents is an arduous task. The CPython module's improvements and developments will mislead developers to write invalid code if it cannot be updated in time.

For Incorrect External Call—which involves incorrect call of functions, methods, and interfaces from other systems or libraries, caused 39 bugs in PyPy but only 19 in CPython. A typical external call is access to operating system functions. PyPy and CPython have specific differences in the underlying mechanisms, such as implementing JIT and garbage collection strategy, which will lead to different system calls. However, some Unix-like systems, such as FreeBSD, will follow the PEP (modification standard of CPython) when upgrading the system port, leading to PyPy compatibility issues when performing the incorrect external calls.

Finding 5: The top three most common general root causes for both interpreters are Incorrect Algorithm Logic, Incorrect Configuration, and Incorrect Internal Call. The general root cause that only causes many bugs in CPython is Invalid Documentation, partly due to the complex document structure. The general root cause that causes more bugs in PyPy is Incorrect External Call, partly due to the compatibility of operating systems.

Although Finding 5 illustrates the bug distribution by general root causes, it does not detail the circumstances caused by Python's design characteristics. Therefore, in the lower part of Table 12, we present the number of bugs caused by Python-specific root causes. Python's design draws on the mature implementation strategies of many modern compilers,

¹⁹<https://bugs.Python.org/issue31673>

²⁰<https://bugs.Python.org/issue34987>,

²¹<https://bugs.Python.org/issue36374>

²²<https://github.com/Python/cPython/tree/master/Doc/library>

so the specific root causes are not unique to the Python interpreter. Even if Global Interpreter Lock (GIL) is a memory solution independently proposed by the early designers of CPython, it is eliminated in other Python interpreters such as Jython with advanced operating systems. In addition, some Python-specific root causes may be included in the general root causes. For example, Concurrency contains the interpreter concurrence problems at different memory resource levels, and Incorrect Internal Calls may contain illegal calls in Incorrect Object Design, etc.

From Table 12, 176 bugs are marked as triggered by Python-specific root causes. Among them, the most common root cause is Incorrect Object Design, accounting for one-third of the bugs. Note that all functions and classes in Python are object-oriented and constructed with underlying objects. These objects are designed throughout the program lifecycle, from basic data type objects to code objects executed on the virtual machine. Once there is a problem with the object design, the high-level implementation will inevitably cause a bug. In Table 13 we present a sample bug for each subcategory of Incorrect Object Design that we divided in Section 4.2.1. From Table 13, Incorrect Object Design may come from various running stages of Python, such as basic data type construction, object method implementation, runtime environment, and bytecode generation. In our investigation, subcategories (3) and (4) occur more often, partly due to the frequent internal calls and releases of objects. This result indicates that when building Python object system, the context information of calling methods and runtime environment should be carefully considered.

Concurrency problems are the second common Python-Specific root cause. We divide them into three levels with Python resource allocation methods: coroutine (Cor), multi-thread (Mut), and multi-processing (Mup). This method explicitly triaged all Concurrency bugs of CPython in Table 12 (47 out of 47), and 21 of them are from *threading* module. The *threading* module not only implements the creation, scheduling, and destruction of threads but also maintains a global interpreter lock (GIL). The GIL mechanism ensures that only one thread executes Python bytecode at a time. It simplifies the implicit security of CPython concurrent access but sacrifices the concurrency on multiprocessors. Some third-party libraries pursuing performance will release GIL when completing computing-intensive tasks such as compression or hashing. The complex interpreter state and release timing will make this costlier to maintain. For all Python-specific root causes, Misuse of Coroutine is the only one we did not find bugs in PyPy. This is because coroutine is still an unimplemented feature of PyPy²³. Coroutine has no thread switching overhead and can be interrupted by program control at any time during function execution, which has higher execution efficiency. *asyncio* module in CPython was formally introduced in version 3.5 to realize the construction of coroutines. This module also faces some unique challenges, such as context switching, internal detection mechanisms, and threads cooperation.

Surprisingly, although the PyPy bugs we labeled are less than half of CPython, there are more bugs caused by Incorrect Virtual Machine Execution (VM). One explanation is the different implementations of the two interpreters. Compared with CPython, PyPy's VM is much more complex. After the bytecode is generated at the interpreter front end, CPython VM directly converts it into stack frame objects and executes them one by one. Since the front end of PyPy is written by RPython (a static subset of Python), a trace-based JIT compiler can be introduced to convert bytecode into machine code, resulting in huge performance improvement. The JIT compiler contains some independent components, such as tracker, optimizer, and back end, which may bring more difficulties to the VM maintenance.

²³<https://doc.pypy.org/en/latest/stackless.html?highlight=Coroutines#unimplemented-features>

Table 13 Bug examples for incorrect object design

Subcategory / Count	Example	Buggy File	Issue Description
Incorrect underlying definition / 4	#12414	<i>codeobject.c</i>	Incorrect return value calculation method of <i>getsizeoff()</i>
Incorrect dynamic behavior / 3	#24257	<i>namespaceobject.c</i>	Illegal dynamic property modification of <i>Isinstance()</i>
Incorrect operation / 31	#3471	<i>classobject.c</i>	Incorrectly call <i>PyObject_GetAttr()</i> to get special method from objects
Incorrect allocation / 9	#3369	<i>floatobject.c</i>	Memory leak when a nan or inf is generated in <i>PyFloat_FromString()</i>

Figure 3 shows a bug caused by the PyPy JIT compiler, where the test case can correctly run without JIT, but it will cause a segfault when JIT is used. The core committer confirmed it as a severe issue and quickly fixed it by modifying the optimized algorithm.

Finding 6: The most common Python-specific root cause is Incorrect Object Design, accounting for 30% of the bugs. The specific causes that only in CPython are Misuse of Coroutine and Multi-thread. The notable cause of PyPy is Incorrect Virtual Machine Execution, which is partly due to the additional JIT compiler.

5.3.2 Root causes and symptoms

To further understanding the root causes, we discuss the relationship between root causes and the symptoms they produced.

Table 14 illustrates the extent to which a particular root cause resulted in a specific symptom across CPython and PyPy. As mentioned above, Incorrect Algorithm Logic was the most common general root cause. Unsurprisingly, it resulted in all bug symptoms in

Segfault related to JIT

Created originally on Bitbucket by [btlachance](#) (Brian Lachance)

After translating the attached program with `-Ojit` the resulting binary segfaults. Without the JIT, and when run untranslated, it correctly exits with status code 13.

If you uncomment the hint on `reverse` and translate with `-Ojit`, this binary also correctly exits with status code 13.

Are there any other builds I should try to help diagnose the issue? Is there anything else I should attach? I'm not that familiar with hg, but when I run `hg log` it says the most recent changeset is `a1608b11c5da`.



Bitbucket Importer @bitbucket_importer · 4 years ago
Created originally on Bitbucket by [fjja1](#) (Maciej Fijałkowski)

Author Maintainer

Ok, looks like a very legit JIT bug. The trace is bogus



Carl Friedrich Bolz-Tereick @cfbolz · 4 years ago
Fixed in [0f364935cfcf](#). Thanks a lot for this report, this was quite a serious issue that we somehow missed (I suspect with some thought it's can probably be turned into a segfault in pypy-c).

Owner

Fig. 3 A PyPy Bug Sample Caused by JIT Compiler (#2650)

Table 14 Symptoms that general root causes exhibit across CPython and PyPy

Symptoms	Root Cause														
	Logic	Num	Assi	ICC	RE	Exc	Cod	Mem	ExCall	InCall	Conc	Doc	Ver	Config	Others
(a) CPython															
Behavior	125	6	5	13	7	45	12	6	12	54	39	60	28	50	5
Compile error	2	0	0	1	0	3	0	0	6	3	0	0	0	20	0
Crash	11	0	2	3	0	3	1	8	1	33	1	0	0	5	0
Performance	22	0	1	1	3	0	5	3	0	7	2	0	4	0	1
Resource Usage	9	0	1	0	0	0	0	19	0	1	5	0	3	0	0
Security	2	0	0	1	1	0	0	1	0	4	0	0	0	1	0
Total	171	6	9	19	11	51	18	37	19	102	47	60	35	76	6
(b) PyPy															
Behavior	76	1	10	5	1	9	12	1	29	37	9	3	10	48	5
Compile error	1	0	0	1	0	1	0	0	2	1	0	0	0	0	0
Crash	2	0	2	1	0	3	3	8	4	4	2	0	0	2	1
Performance	9	0	0	1	1	0	0	5	1	6	0	0	1	0	0
Resource Usage	0	0	0	0	0	0	0	2	1	0	1	0	0	0	0
Security	1	0	0	0	0	0	0	1	2	0	0	0	0	1	0
Total	89	1	12	8	2	13	15	17	39	48	12	3	11	51	6

Table 14, 6 out of 6 for CPython and 5 out of 6 for PyPy. The most severely affected by Incorrect Algorithm Logic is the behavior of interpreters. More than 75% of symptoms caused by it are manifested as unexpected behavior. Given that the unexpected behavior performances during Python's execution, dynamically analyzing the program and automatically repairing the Incorrect Algorithm Logic are urgent challenges for Python core developers.

As the second most common root cause in both interpreters, Incorrect Internal Call caused all CPython bug symptoms and related to half of the crash bugs. However, PyPy only covers four types of symptoms, even if the number of its bugs is second to Incorrect Algorithm Logic. In contrast, Incorrect External Call covers all symptoms in PyPy, but only three symptoms in CPython. This result further illustrates that incorrect calls in CPython are more from the functions provided by itself, such as C-API and build-in modules. At the same time, PyPy is more affected by external circumstances, such as incompatible updates to Unix-like systems.

Some general root causes lead to many bugs but covered a few symptoms, eg, Incorrect Configuration and Exception Handling Errors. Incorrect Configuration causes 127 bugs (3rd out of 15 root causes) but is only responsible for four CPython symptoms and three in PyPy. As described in Section 4.2.1, Incorrect Configuration mainly contains the incorrect steps for building Python runtime environment, more likely producing the errors of unexpected running behavior and compilation. Analogously, Exception Handling Error causes a total of 64 bugs (4th out of 15 root causes), with only three symptoms. Exception handling error involves in-correct handling of exceptions and warnings, including missing detection conditions, wrong feedback information, and capture of extreme cases. Such errors frequently result in unexpected behavior but rarely cover performance or security issues.

In Table 15 we further present the symptoms that each Python-specific root cause may exhibit and observed some crucial facts. First, Incorrect Object Design covers the widest symptoms and is responsible for 33% unexpected behaviors. As mentioned before, defects in object design will affect higher-level applications. Object System in Python covers every stage of the language implementation, so the bugs detected in the object application phase will appear as various symptoms. Second, Incorrect Bytecode Generation and Incorrect Virtual Machine Execution cover most compiler error bugs. This finding verifies our view that the bugs in the Python interpreter core are mainly from its back end. Also, 60% of compiler errors were caused by Incorrect Virtual Machine Execution. According to Table 14 we speculate that these errors concentrate in the building phase of the virtual machine environment, such as Incorrect Configuration, which caused 65% of the total compiler errors. Finally, for the concurrency problems, Misuse of Multi-thread frequently causes most program crashes.

Table 15 Symptoms that Python-specific root causes exhibit

Symptoms	Root Cause							
	Obj	ByCo	VM	Cor	Mut	Mup	GC	Total
Behavior	45	13	20	7	11	6	3	105
Compile error	2	5	12	0	1	2	0	22
Crash	5	2	2	1	7	1	1	19
Performance	6	0	0	1	0	0	1	8
Resource Usage	2	0	1	2	4	11	2	22
Security	0	0	0	0	0	0	0	0
Total	60	20	35	11	23	20	7	176

The incorrect release of GIL will easily cause segmentation faults. In addition, we find that most Resource Usage bugs in Table 14 come from Multi-processing. The incorrect allocation strategy of *multiprocessing* module will lead to inefficient use of shared memory and unnecessary leakage.

Finding 7: For general root causes: Incorrect Algorithm Logic caused the widest symptoms. Incorrect Call covers all symptoms in CPython, while Incorrect External Call covers all symptoms in PyPy. Incorrect Configuration and Exception Handling Error led to a substantial number of bugs but only covered a few types of symptoms.

For Python-specific root causes: Incorrect Object Design covers the widest symptoms. Incorrect Bytecode Generation covers most Compiler errors, while Misuse of Multi-thread and Multi-processing causes most Crash bugs and Resource Usage problems, respectively.

5.4 RQ4: Revealing And Fixing Bugs

In the final research question, we discuss the bug revealing and their fixing scales.

5.4.1 Size of bug-revealing test cases

Table 16 presents the distribution over the sizes of test cases. As shown, more than 65% of the test cases are smaller than 50 lines of code, and more than 85% of the test cases are smaller than 100 lines of code. This result can genuinely assist the fuzzing of Python interpreters. It proves that when randomly generating the under-test program, the test cases should be designed as short and complex as possible. For example: causing heap/stack overflow of the interpreter through deep recursion or nested loops²⁴, checking program boundaries through uncommon argument combinations.²⁵

Finding 8: Most of the program that trigger the bugs are very simple. More than half of the test cases are smaller than 25 lines of code, and 86.8% of the test cases are smaller than 100 lines.

5.4.2 Fix Scale for Different Root Causes

In this part, we investigate the relationship between fixing effort and bugs produced by different root causes. Due to the PyPy Git repository migration, its most fixing information is unavailable, so we only conduct related analyses on the CPython data set.

Table 17 illustrates the size of code changes to fix the CPython bugs produced by different root causes. Except for some bugs with no associated repair version, we statistics 414 bugs and their related revisions. As shown in the *Total* column, more than half of the bug fixes have less than 25 lines of code modification, and 84% of the bug fixes contain fewer than 100 lines. This result shows that most of the bugs in CPython can be fixed with only

²⁴<https://bugs.Python.org/issue41697>

²⁵<https://bugs.Python.org/issue37424>

Table 16 The Size of Test Cases (lines of code)

<i>lines</i>	1-10	11-25	26-50	51-100	101-500	500+
CPython	348	585	355	205	177	60
PyPy	97	83	49	27	20	9
Total	445	668	404	232	197	69
%	22.08	33.15	20.06	11.51	9.78	3.42

minor code changes, such as update function logic, rewriting conditional branches, modifying parameters and return values, etc. Referring to the bug example in Fig. 3, the maintainer fixed this serious compiler segfault only by moving the function position (as Fig. 4 shows) in the *if* statement. Understanding this can guide engineers working on the automatic repair of CPython bugs, which should focus on precise defect location and the mapping of the fix patterns.

Table 17 also shows the relationship between the bug fixes and different root causes. The bug fixes caused by the version contain the most extensive code modification, with 119 lines. Version changes have a more significant impact on the original code, and its fixes generally involve multiple files containing thousands of code lines. For example, the new release of a file in the CPython library requires updating all parameters involved in related functions²⁶. Incorrect Algorithm Logic is another root cause that needs a large scale of code modification (eg, 91 lines on average). As discussed in Section 5.3, repairing algorithm logic errors is an arduous task, which requires non-trivial modifications in multiple files. On the contrary, some root causes only require minor code changes to fix the bugs they produced. Such as Exception Handling Error, Incorrect Numerical Computation, Incorrect Configuration, and Coding— with 34, 37, 38, 39 lines of code on average, respectively.

Finding 9: Most fixes of CPython bugs contain minor code changes. More than half of the bug fixes contain fewer than 50 lines of code. The root causes that require relatively large-scale modifications are Version and Incorrect Algorithm Logic (with 119 and 91 lines of code on average, respectively).

6 Discussion

This section discusses the lessons learned and practical implications based on our experimental results, which can guide future work for researchers and developers in related areas.

6.1 Bug location

According to Finding 2, 38% of the CPython bugs originate from the Standard Library and Interpreter Core. Although PyPy does not divide the components in detail, Finding 3 indicates that its most buggy files are also from libraries (eg, *micronumpy*) and interpreter

²⁶<https://bugs.Python.org/issue38121>

issue #2650 testing

Functions that write immutable fields don't need to invalidate the heap cache for such fields. However, they *do* need to force the lazy sets of such fields!

▼ rpython/jit/metainterp/optimizeopt/heap.py

View file @f364935

```
... @@ -445,13 +445,13 @@ class OptHeap(Optimization):
445     if effectinfo.check_readonly_descr_field(fielddescr):
446         cf.force_lazy_set(self, fielddescr)
447     if effectinfo.check_write_descr_field(fielddescr):
448 +    cf.force_lazy_set(self, fielddescr, can_cache=False)
449     if fielddescr.is_always_pure():
450         continue
451     try:
452         del self.cached_dict_reads[fielddescr]
453     except KeyError:
454         pass
455 -    cf.force_lazy_set(self, fielddescr, can_cache=False)
456     #
457     for arraydescr, submap in self.cached_arrayitems.items():
458         if effectinfo.check_readonly_descr_array(arraydescr):
459             ...
```

Fig. 4 A Fix Sample for a Serious Segfault in PyPy JIT Compiler (#2650)

(eg, RPython). In Interpreter Core there are two bug-prone stages: (1) the construction of *TypeObjects* and their instantiated concrete objects (eg, *DictObject*), and (2) the back end, especially the execution of virtual machine. This suggests that more efforts should be spent on testing the object model and interpreter back end. Thus, based on Table 9, we can take gradual steps toward testing the files like *typeobject.c* and *ceval.c*.

Simultaneously, Finding 1 shows that 87% of the bugs only involve a single component. Most library files only contain a few bugs, implying that the testing should focus on independent units. In addition, the POSIX and the Test module also deserve attention due to

Table 17 The size of changes to fix the bugs

Root Cause	1~10	11~25	26~50	51~100	101~500	500+	Mean
Logic	21	26	23	14	19	3	91
Num	0	1	1	1	0	0	37
Assi	1	1	2	1	1	0	84
ICC	5	4	1	2	2	0	60
RE	0	2	1	1	2	0	88
Exc	11	6	10	5	1	0	34
Cod	2	4	4	1	1	0	39
Mem	11	4	2	4	3	0	42
ExCall	5	1	4	0	2	1	84
InCall	16	13	16	11	8	2	85
Conc	7	8	6	3	3	1	54
Doc	21	9	3	3	1	1	59
Ver	6	7	5	3	0	1	119
Config	15	6	9	9	4	0	38
Total	121	92	87	58	47	9	—

their occurrence in the most buggy files. This indicates that core maintainers should update relevant modules in time with the operating systems updates and source file repair.

6.2 Root Cause And Symptoms

As corroborated by Finding 7, Incorrect Algorithm Logic is the most common root cause and result in most bug symptoms (all symptoms in CPython, 5 out of 6 in PyPy). Besides, Table 14 shows that 75% of bugs caused by Incorrect Algorithm Logic is performed as undefined behavior, which is also the most common symptom. This results confirm that the maintainers need a specific automated testing technique to detect undefined behavior errors caused by the Incorrect Algorithm Logic. Since the lengthy source code for such bugs usually contains obscure and complex logic structures, eg, the interpreter's complex optimization strategy, black-box testing is a reasonable choice. Black-box testing can detect bugs by providing unexpected input to the interpreter and monitoring abnormal results. Classical black-box testing techniques such as fuzzing and randomized testing have successfully detected thousands of bugs in the compilers (eg, GCC/LLVM) and interpreters (eg, Java, PHP) Ye et al. (2021), Livinskii et al. (2020), and Koroglu and Wotawa F (2019). As a popular programming language, many designs of Python are similar to modern compilers. Therefore, existing mature tools can be transplanted and improved according to Python's characteristics when conducting black-box testing.

Finding 7 reveals that Incorrect Object Design is the most common root cause and covers the widest symptoms. The object system makes Python concise and flexible. But incorrect dynamic modification and illegal internal calls will lead to abnormal behavior (as Table 13 shows). One solution could be conduct fuzzing by mutating test case fragments without changing the results. Such techniques have been proved to find many bugs in compilers of other object-oriented language Le et al. (2015b), Sun et al. (2016a), and Delgado-Pérez et al. (2017).

According to Finding 7, most compiler errors are caused by Incorrect Virtual Machine Execution. Investigating these errors, most of them are segmentation faults and a few stack trace errors. Although the VM designs of CPython and PyPy are pretty different, they adopt the bytecode as an intermediate representation. Chen et al. (2019) and Chen et al. (2016a) conducted testing on Java Virtual Machine (JVM) by generating illegal bytecode files. Their methods are especially suitable for the PyPy JIT compiler, which is designed based on tracing JIT and recently studied for Java and JavaScript.²⁷

6.3 Bug Detection and Fixing

Finding 8 reveals that most of the test cases that trigger the bugs are tiny, and 55.23% of them are smaller than 25 lines of code. This result indicates the generated program should be designed short but complex, ie, they should contain fewer lines of code and cover more functions simultaneously. For example, the design of the generation algorithm can be based on the results in Table 12. We can consider the coverage of conditional branches, multiple variable assignments, and as many internal calls as possible.

However, current technologies may be less mature for helping researchers generating test cases for Python. As shown in Table 9, *TypeObject* is the most buggy file in Python Object System. This result reveals that the correct type information is crucial for automatic test

²⁷<https://rPython.readthedocs.io/en/latest/jit/overview.html#id4>

generation Lukasczyk et al. (2021). Type information can provide appropriate parameter types for method calls or to assemble complex objects. In the absence of type information, the test generation can only guess the appropriate parameter types for new function calls. This problem can be facilitated in the recent version of CPython. Since Python 3.5, the types can be directly annotated in the implementing source code similar to statically typed languages.²⁸ For PyPy, the collaboration with the type inference studies may be a solution for the lack of type information.

According to Finding 9, although the internal architecture of Python is complicated, most bug fixes only contain small code changes. The average size of all fixes is only 70 lines, and 51.44% of bug fixes have fewer than 50 lines of code. Studies have shown that the fix patterns extracted or the patch representations generated during the automatic repairing are also small Le Goues et al. (2012). Also, the ecosystem of Python contains rich historical repairing information, which may be a hopeful attempt to automate fixing interpreter bugs Koyuncu et al. (2020) and Liu et al. (2018).

6.4 Actionable Suggestion

The above findings can also be summarized into several actionable suggestion for developers:

Testing Target Finding 2 & 3 reveals that the bugs of CPython and PyPy are mainly concentrated in *the Library, object model, and interpreter back-end*. We suggest that the interpreter testing should pay more effort into such components. For the Library, since most of the files contain bugs, the test deserves to be covered for every single module. For the object model, the test can focus on the files related to type object design (Table 9). For the interpreter back-end, the test can focus on *ceval.c*, which is responsible for implementing the virtual machine (Table 9a).

Testing Method Finding 4 reveals that unexcepted behavior is the most common symptom of Python interpreter bugs. This indicates that interpreter testing also faces test-oracle problems. To avoid the test program exposing any undesired behavior, we suggest two popular methods:

Differential testing detects bugs by comparing the execution results from multiple compilers with the same specification. We suggest *conducting differential testing between multiple Python interpreters*. Due to the numerous bugs caused by Version (Table 12), it could also be *conducted on different interpreter versions*.

EMI(equivalence modulo inputs, Le et al. (2014)) is another testing method we suggest. The main idea of EMI is to mutate the test program without changing the behavior. It can efficiently generate semantic-preserving mutants relying on Python's specifications.

Library Testing Finding 2 and Table 9a reveals that the Python interpreter bugs distributed in most files of the Library. Also, Finding 5 reveals that incorrect internal call is a common root cause lead to bugs. We suggest that *the test of interpreter library should cover function calls and files in the Library comprehensively*. One potential method could be *conduct difference testing through function-level skeletal program enumeration* (SPE, (Zhang et al.

²⁸<https://www.Python.org/dev/peps/pep-0484/>

2017)). SPE triggers various compiler optimization passes with different variable usage patterns. It's not complicated and theoretically possible to replace variable usage patterns with function calls to cover the entire Library.

Object Model Testing Finding 6 and Table 13 reveal that incorrect allocation&operation occur more frequently in the object design. We suggest that *the test of Python object models could focus on the object instantiating and calling stage*. As long as we replace variable usage patterns with the properties/functions of objects, *SPE may also be suitable for testing the object model*. Another potential solution could be *mutate the test-program based on the idea of EMI*. For example, we can modify unused properties during object instantiation and call them with uncovered methods through control flow analysis.

Back-end Testing Table 9 shows that the most buggy file of interpreter back-end comes from bytecode execution. Table 12 shows that bytecode generation is a common Python-specific root cause that leads to back-end bugs. Therefore, we suggest that *the test of interpreter back-end could start with the more buggy parts, such as bytecode optimization*. One potential method could be using the cross-optimization strategy. As a widely-used testing strategy (Chen et al. 2020), it could detect interpreter bugs by comparing results produced using different optimizations implemented in a single interpreter.

Test-Program Finding 8 reveals that the programs that trigger the bugs are tiny. This strongly suggests that *the test-program of interpreters should be designed short but complex*. Similar insights have also been made by some C compiler researchers (Zhang et al. 2017). For example, the test-program should consider the coverage of conditional branches, multiple variable assignments, and as many internal calls as possible.

Fuzzing Finding 7 and Table 15 reveal that a specific automated testing technique is needed to detect undefined behavior bugs caused by the incorrect algorithm logic. We *suggest fuzzing could be a reasonable choice* since the lengthy code for such bugs usually contains obscure and complex logic structures. Unlike regular software systems, fuzzing of the interpreter takes program fragments or bytecode sequences as input and detects bugs that cause crashes.

As a vital part of conducting fuzzing, we recommend the following strategies: (1) *mutate the program by modifying unexecuted branches*; (2) *extract the syntactic program skeleton, replacing identifier holes in it with object (eg, variables, properties, and functions) enumerations*; (3) *change program semantics by inserting/deleting fragments (eg, function calls, conditional branches, and arithmetic expressions), then select mutations that are more likely to trigger interpreter crashes with dynamic programming(eg, Markov Chain Monte Carlo)* (Chen et al. 2016a).

Usage of Pattern Finding 9 shows that 22% of the bug fixes fewer than 10 lines of code. This reveals that some interpreter bugs could be fixed by applying simple patterns such as operator modifying, expressions rewriting, or conditional statement adding. We believe *it is valuable to extract fine-grained patterns from these fixes*. In the future, these patterns could be used for automatic interpreter repairing or support IDE development (eg, code recommendations and error warnings).

Developer Coordination Finding 1 and Table 12 indicate that Python interpreter suffers from developers' lack of coordination. Outdated documentation, conflicting external

modules, and version differences can lead to bugs in the interpreter. We suggest *the developers strengthen the communication and discipline of team coordination*. For example, they should analyze whether test code needs co-evolution when changing critical codes, and they should also be aware of the differences between local environments when developing external modules.

Interpreter Extension Finding 5 reveals that the Python interpreter is accessible to produce bugs by incorrect external calls. We suggest the developers *provide more standard extension APIs in the Library, such as low-level type conversion and memory management*, to make the external calls more reliable.

GIL Alternative Finding 7 reveals that the wrong release of GIL frequently results in crash bugs. The author of Python confirmed that removing GIL from Python still faces formidable challenges of performance and compatibility. If developers forcibly need to implement concurrency avoiding GIL, we suggest two alternatives: (1) *use the interpreter version²⁹ without GIL implemented by SMT (software transactional memory)*; (2) *use the multiprocessing module to implement process-level concurrent operations*.

Shared Memory Finding 7 reveals that Multi-processing covers most of the resource usage issues. Some of them were caused by shared memory. In the multi-process module, the memory leakage is usually caused by the shared memory between processes. Until all processes exit, resource tracking will release the tracked objects. Once a sub-process is accidentally killed, a leak of shared memory may occur. Although shared memory can provide a better performance, we suggest that *the developer should carefully consider its usage scenarios*.

6.5 Threats to Validity

As with any empirical studies, our study is not free of threats to validity. we briefly discuss them from two perspectives.

Internal Threats The primary internal threat to validity involves subjective bias or errors in the classification of bugs. To reduce this threat, we initiate our labeling process with classification schemes from existing literature Islam et al. (2019), Di Franco et al. (2017), Vasilescu et al. (2015), and Zhang et al. (2018). We selected only closed and fixed issues or merged pull requests to focus on actual bugs and fixes. Each bug is inspected and labeled by two authors independently. Any discrepancy is discussed until a consensus is reached.

Another internal threat is that we only focus on two Python interpreters –CPython and PyPy. CPython and PyPy are currently the two most popular Python interpreter projects, which support compilation and execution in multiple operating environments. They have efficient performance optimization algorithms and complete back-end architecture, and well reflect most characteristics of the Python interpreters. However, due to differences in the underlying language and the functional advantages, these two interpreters may not reflect other Python interpreters (eg, Jython and IPython).

²⁹<https://doc.pypy.org/en/latest/stm.html>

External threats One external threat is the generalizability of the data set we collected. We have adopted several strategies to mitigate this threat. First, the raw data we collected includes all pull requests and issues from CPython and PyPy's git repository until Oct 2021. This strategy ensures that this study covers a comprehensive set of data. Second, we have adopted a method similar to those used in existing bug studies Vasilescu et al. (2015), Di Franco et al. (2017), and Islam et al. (2019) to identify as many bug-fix pull requests as possible in the data pre-processing step. Third, we have only studied the merged pull requests that fix bugs to ensure that developers accept the studied bugs and their corresponding fixes.

7 Related work

7.1 Empirical Study on Bugs

Previous empirical studies have examined various software systems and characterized their defects and associated information on various dimensions. Islam et al. (2019), Thung et al. (2012), and Zhang et al. (2018) studied the bugs in machine learning and deep learning frameworks such as Caffe, Tensorflow, Torch, OpenNLP, etc. Di Franco et al. (2017) studied the scientific computing libraries that these frameworks rely on (eg, NumPy, SciPy, and LAPACK). Tanakorn et al. Leesatapornwongsa et al. (2016), Lu et al. (2008), Gao et al. (2018) studied on the bugs in large-scale distributed systems such as ZooKeeper, Hadoop, and HBase. Guo and Engler (2009) and Xiao et al. (2019) analyzed the bugs in the Linux kernel. Chen et al. (2017) studied the changes of dynamic feature code when fixing bugs of Python program. Some researchers focus on the benchmarks of empirical studies, Orrú et al. (2015) firstly released a curated collection of metrics for Python software systems.

However, there are not many related studies that focus on programming languages and their compilers/interpreters Sun et al. (2016b), Zhang et al. (2019), and Wang et al. (2017), which inspired us perform an empirical study on bugs in popular Python interpreters. In our study, we followed the bug characteristics on multiple dimensions, including bug location, bug symptoms, root causes, and bug revealing & fixing that widely adopted by existing empirical studies Di Franco et al. (2017), Islam et al. (2019), Leesatapornwongsa et al. (2016), Selakovic and Pradel (2016), Thung et al. (2012), and Zhang et al. (2018).

7.2 Compiler & Interpreter Testing

Compilers or interpreters are critical and complex software. Testing is an important method to ensure their correctness and stability. At present, the most successful testing technique for compilers/interpreters is fuzzing testing. Fuzzing is an automated software testing technology, first proposed by Miller et al. (1990). Fuzzing automatically generates a large amount of input data for the under-test program and detects program abnormalities by monitoring its executing status. Its representative tools are AFL³⁰, libfuzzer³¹ and Honggfuzz³², they generate new input data by randomly editing the bytes of the input file. Such a technique is highly

³⁰<https://lcamtuf.coredump.cx/afl>

³¹<https://lvm.org/docs/LibFuzzer.html>

³²<https://honggfuzz.dev/>

universal and supports fuzzing programs written in multiple languages on different operating systems. Veggalam et al. (2016) proposed an automated evolutionary fuzzing technique to find bugs in JavaScript interpreters. Holler et al. (2012) discovered security bugs in the Mozilla JavaScript interpreter and PHP interpreter by fuzzing with code fragments that partially from programs known to have caused invalid behavior before.

Another successful compiler testing technique is randomized testing. Randomized testing explores the program's defects by comparing the running results of multiple theoretically equivalent programs. As a classic randomized testing tool, Csmith Yang et al. (2011) can randomly generate the C programs by avoiding introducing undefined behaviors and checking the final sum of all global variables to verify whether the program was miscompiled. Orange3 proposed by Nagai et al. (2014) also detected bugs in GCC/LLVM by generating more extended arithmetic expressions and accommodating multiple expressions in test programs. Le et al. (2014) dynamically analyzed the C compilers' execution process and randomly deleted unexecuted statements to generate a variant equivalent to the source program. Since then, they have improved the random strategy and found more than 1000 bugs in GCC/LLVM Le et al. (2015b) and Le et al. (2015a). In addition to the C language compiler, randomized testing is also adopted in other languages, such as OpenCL Lidbury et al. (2015) and Cummins et al. (2018) and OCaml Midtgaard et al. (2017).

Current studies have proposed numerous compiler testing techniques for static programming languages like C/C++, but few of them directly focus on the programming languages. However, the related studies of Python projects may contribute to code generation and code analysis for its interpreter testing. Biswas et al. (2019) built a Boa dataset of Data Science Software in Python for mining software repository research. Similarly, Widyasari et al. (2020) designed BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. Gharibi et al. (2018) introduced a prototype Python tool named code2graph that can automatically generate static call graphs for Python source code to assist its interprocedural analyses and software comprehension. Atwi et al. (2021) presents a tool that can automatically detect method-level refactoring operations in Python projects. Refactoring detection can help understand the software development process and the relationship between various system versions.

8 Conclusion

Python has become one of the most widely used programming languages. The Python interpreter is also complex and strict software, suffering from software defects, which will fundamentally threaten the quality of all Python program applications. To prudently understand the bugs in Python interpreter, we present a large-scale study of more than 30,000 fixed bugs and 20,000 confirmed revisions in two famous interpreters – CPython&Pypy. We further characterized and taxonomized 1200 bugs to analyze their representative symptoms and root causes deeply. This article answers four research questions (bug location, symptoms, root causes, and bug revealing & fixing) by identifying nine findings. Based on them, we discuss the lessons learned and practical implications that guide the theory and method of Python interpreter testing and improvements in the future.

For future work, we plan to continue concerning the quality assurance of Python interpreters. Achieving this requires overcoming numerous challenges brought by language characteristics, including the oracle problem resulting by dynamic behavior, and the impact of type information on execution. Also, we aim to construct automatic test case generation

and test-oracle construction techniques and assess which testing method is more applicable for Python interpreters. Lastly, we plan to refine our study's results to produce a benchmark dataset for facilitating automatic program repair for Python interpreters.

Acknowledgments This work was partially supported by the National Natural Science Foundation of China (No.61932012, No.62172209), the Science, Technology and Innovation Commission of Shenzhen Municipality (No.CJGJZD20200617103001003), the Fundamental Research Funds for the Central Universities (No. 2022300295), and the Cooperation Fund of Huawei-Nanjing University Next Generation Programming Innovation Lab (No.YBN2019105178SW37).

Declarations

Conflict of Interests To the best of our knowledge, all the named authors have no conflict of interest, financial or otherwise.

References

- Acuña R, Lacroix Z, Bazzi RA (2015) Instrumentation and trace analysis for ad-hoc Python workflows in cloud environments. In: 2015 IEEE 8th international conference on cloud computing. IEEE, pp 114–121
- Atwi H, Lin B, Tsantalis N, Kashiwa Y, Kamei Y, Ubayashi N, Bavota G, Lanza M (2021) PYREF: refactoring detection in Python projects. In: 2021 IEEE 21st international working conference on source code analysis and manipulation (SCAM). pp 136–141
- Biswas S, Islam M, Huang Y, Rajan H (2019) Boa meets Python: a boa dataset of data science software in Python language. In: 2019 IEEE/ACM 16th international conference on mining software repositories (MSR). pp 577–581
- Cacho N, Barbosa EA, Araujo J, Pranto F, Garcia A, Cesar T, Soares E, Cassio A, Filipe T, Garcia I (2014) How does exception handling behavior evolve? an exploratory study in java and c# applications. In: 2014 IEEE international conference on software maintenance and evolution. IEEE, pp 31–40
- Calmant T, Americo JC, Gattaz O, Donsez D, Gama K (2012) A dynamic and service-oriented component model for Python long-lived applications. In: Proceedings of the 15th ACM SIGSOFT symposium on component based software engineering, pp 35–40
- Cao H, Gu N, Ren K, Li Y (2015) Performance research and optimization on cPython's interpreter, IEEE, FedCSIS
- Chen Z, Chen L, Zhou Y, Xu Z, Chu WC, Xu B (2014) Dynamic slicing of Python programs. In: 2014 IEEE 38th annual computer software and applications conference. IEEE, pp 219–228
- Chen Z, Ma W, Lin W, Chen L, Li Y, Xu B (2017) A study on the changes of dynamic feature code when fixing bugs: towards the benefits and costs of Python dynamic features. *Sci China Inf Sci* 61:1–18
- Chen Z, Ma W, Lin W, Chen L, Xu B (2016b) Tracking down dynamic feature code changes against Python software evolution. In: 2016 third international conference on trustworthy systems and their applications (TSA). IEEE, pp 54–63
- Chen J, Patra J, Pradel M, Xiong Y, Zhang, Hao D (2020) A survey of compiler testing, vol 53
- Chen Y, Su T, Su Z (2019) Deep differential testing of jvm implementations. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE). IEEE, pp 1257–1268
- Chen Y, Su T, Sun C, Su Z, Zhao J (2016a) Coverage-directed differential testing of jvm implementations. In: Proceedings of the 37th ACM SIGPLAN conference on programming language design and implementation, pp 85–99
- Cummins C, Petoumenos P, Murray A, Leather H (2018) Compiler fuzzing through deep learning. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 95–105
- Dalcin LD, Paz RR, Kler PA, Cosimo A (2011) Parallel distributed computing using Python. *Adv Water Resour* 34(9):1124–1139
- Delgado-Pérez P, Medina-Bulo I, Segura S, García-Domínguez A, José J (2017) Gigan: evolutionary mutation testing for c++ object-oriented systems. In: Proceedings of the symposium on applied computing, pp 1387–1392

- Di Franco A, Guo H, Rubio-gonzález C (2017) A comprehensive study of real-world numerical bug characteristics. In: 2017 32Nd IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 509–519
- Forcier J, Bissex P, Chun WJ (2008) Python web development with Django. Addison-Wesley professional
- Gao Y, Dou W, Qin F, Gao C, Wang D, Wei J, Huang R, Zhou L, Wu Y (2018) An empirical study on crash recovery bugs in large-scale distributed systems. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 539–550
- Garcia J, Feng Y, Shen J, Almanee S, Xia Y, Chen Q (2020) A comprehensive study of autonomous vehicle bugs. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 385–396
- Ghanbari A, Benton S, Zhang L (2019) Practical program repair via bytecode mutation. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, pp 19–30
- Gharibi G, Tripathi R, Lee Y (2018) Code2graph: automatic generation of static call graphs for Python source code. In: Proceedings Of The 33rd ACM/IEEE international conference on automated software engineering, pp 880–883
- Guo PJ, Engler DR (2009) Linux kernel developer responses to static analysis bug reports. In: USENIX annual technical conference, pp 285–292
- Holler C, Herzig K, Zeller A (2012) Fuzzing with code fragments. In: 21st {USENIX} security symposium ({USENIX} security 12), pp 445–458
- Islam MJ, Nguyen G, Pan R, Rajan H (2019) A comprehensive study on deep learning bug characteristics. In: Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 510–520
- Jin G, Song L, Shi X, Scherpelz J, Lu S (2012) Understanding and detecting real-world performance bugs. ACM SIGPLAN Not 47(6):77–88
- Koroglu Y, Wotawa F (2019) Fully automated compiler testing of a reasoning engine via mutated grammar fuzzing. In: 2019 IEEE/ACM 14th international workshop on automation of software test (AST). IEEE, pp 28–34
- Koyuncu A, Liu K, Bissyandé T, Kim D, Klein J, Monperrus M, Le Traon Y (2020) Fixminer: mining relevant fix patterns for automated program repair. *Empir Softw Eng* 25:1980–2024
- Le V, Afshari M, Su Z (2014) Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Not* 49(6):216–226
- Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012) A Systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each. In: 2012 34th international conference on software engineering (ICSE). IEEE, pp 3–13
- Le V, Sun C, Su Z (2015a) Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Not* 50(10):386–399
- Le V, Sun C, Su Z (2015b) Randomized stress-testing of link-time optimizers. In: Proceedings of the 2015 international symposium on software testing and analysis, pp 327–337
- Leesatapornwongsa T, Lukman JF, Lu S, Gunawi HS (2016) Taxdc: a taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In: Proceedings of the twenty-first international conference on architectural support for programming languages and operating systems, pp 517–530
- Leo S, Zanetti G (2010) Pydoop: a Python mapreduce and hdfs api for hadoop. In: Proceedings of the 19th ACM international symposium on high performance distributed computing, pp 819–825
- Lidbury C, Lascu A, Chong N, Donaldson AF (2015) Many-core compiler fuzzing. *ACM SIGPLAN Not* 50(6):65–76
- Liu K, Kim D, Bissyandé T, Yoo S, Le Traon Y (2018) Mining fix patterns for findbugs violations. *IEEE Trans Software Eng* 47:165–188
- Livinskii V, Babokin D, Regehr J (2020) Random testing for c and c++ compilers with yarpgen. *Proc ACM Program Language* 4(OOPSLA):1–25
- Lu S, Park S, Seo E, Zhou Y (2008) Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th international conference on architectural support for programming languages and operating systems, pp 329–339
- Lukasczyk S, Kroiß F, Fraser G (2021) An empirical study of automated unit test generation for Python. [arXiv:2111.05003](https://arxiv.org/abs/2111.05003)
- Midtgaard J, Justesen MN, Kasting P, Nielson F, Nielson HR (2017) Effect-driven quickchecking of compilers. *Proc ACM Program Language* 1(ICFP):1–23
- Miller BP, Fredriksen L, So B (1990) An empirical study of the reliability of unix utilities. *Commun ACM* 33(12):32–44

- Motwani M, Sankaranarayanan S, Just R, Brun Y (2018) Do automated program repair techniques repair hard and important bugs? *Empir Softw Eng* 23(5):2901–2947
- Nagai E, Hashimoto A, Ishiura N (2014) Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions. *IPJS Trans Syst LSI Design Methodol* 7:91–100
- Orrú M, Tempero E, Marchesi M, Tonelli R, Destefanis G (2015) A curated benchmark collection of Python systems for empirical studies on software engineering. *Proceedings Of The 11th international conference on predictive models and data analytics in software engineering*, pp 1–4
- Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V (2011) Scikit-learn: machine learning in Python. *J Mach Learn Res* 12:2825–2830
- Perez F, Granger BE, Hunter JD (2010) Python: an ecosystem for scientific computing. *Comput Sci Eng* 13(2):13–21
- Raschka S (2015) Python machine learning. Packt publishing Ltd
- Reynolds JC (1972) Definitional interpreters for higher-order programming languages. In: *Proceedings of the ACM annual conference-Volume 2*, pp 717–740
- Seaman CB, Shull F, Regardie M, Elbert D, Feldmann RL, Guo Y, Godfrey S (2008) Defect categorization: making use of a decade of widely varying historical data. In: *Proceedings of the second ACM-IEEE international symposium on Empirical software engineering and measurement*, pp 149–157
- Selakovic M, Pradel M (2016) Performance issues and optimizations in javascript: an empirical study. In: *Proceedings of the 38th international conference on software engineering*, pp 61–72
- Shen Q, Ma H, Chen J, Tian Y, Cheung SC, Chen X (2021) A comprehensive study of deep learning compiler bugs. In: *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp 968–980
- Srinath K (2017) Python—the fastest growing programming language. *Int Res J Eng Technol (IRJET)* 4(12):354–357
- Sun C, Le V, Su Z (2016a) Finding compiler bugs via live code mutation. In: *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*, pp 849–863
- Sun C, Le V, Zhang, Su Z (2016b) Toward understanding compiler bugs in gcc and llvm. In: *Proceedings of the 25th international symposium on software testing and analysis*, pp 294–305
- Thung F, Wang S, Lo D, Jiang L (2012) An empirical study of bugs in machine learning systems. In: *2012 IEEE 23rd international symposium on software reliability engineering. IEEE*, pp 271–280
- Tian Y, Ray B (2017) Automatically diagnosing and repairing error handling bugs in c. In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pp 752–762
- Vasilescu B, Yu Y, Wang H, Devanbu P, Filkov V (2015) Quality and productivity outcomes relating to continuous integration in github. In: *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pp 805–816
- Veggalam S, Rawat S, Haller I, Bos H (2016) Ifuzzer: an evolutionary interpreter fuzzer using genetic programming. In: *European symposium on research in computer security. Springer*, pp 581–601
- Wan Z, Lo D, Xia X, Cai L (2017) Bug characteristics in blockchain systems: a large-scale empirical study. In: *2017 IEEE/ACM 14th international conference on mining software repositories (MSR). IEEE*, pp 413–424
- Wang B, Chen L, Ma W, Chen Z, Xu B (2015) An empirical study on the impact of Python dynamic features on change-proneness. In: *SEKE*, pp 134–139
- Wang J, Dou W, Gao Y, Gao C, Qin F, Yin K, Wei J (2017) A comprehensive study on real world concurrency bugs in node.js. In: *2017 32Nd IEEE/ACM international conference on automated software engineering (ASE). IEEE*, pp 520–531
- Widyasari R, Sim S, Lok C, Qi H, Phan J, Tay Q, Tan C, Wee F, Tan J, Yieh Y 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*, pp 1556–1560
- Xia X, Bao L, Lo D, Li S (2016) “automated debugging considered harmful” considered harmful: a user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In: *2016 IEEE international conference on software maintenance and evolution (ICSME). IEEE*, pp 267–278
- Xiao G, Zheng Z, Jiang B, Sui Y (2019) An empirical study of regression bug chains in linux. *IEEE Trans Reliab* 69(2):558–570
- Yang X, Chen Y, Eide E, Regehr J (2011) Finding and understanding bugs in c compilers. In: *Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation*, pp 283–294

- Ye G, Tang Z, Tan SH, Huang S, Fang D, Sun X, Bian L, Wang H, Wang Z (2021) Automated conformance testing for javascript engines via deep compiler fuzzing. In: Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation, pp 435–450
- Zhang, Chen B, Chen L, Peng X, Zhao W (2019) A large-scale empirical study of compiler errors in continuous integration. In: Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 176–187
- Zhang, Chen Y, Cheung SC, Xiong Y, Zhang (2018) An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, pp 129–140
- Zhang, Sun C, Su Z (2017) Skeletal program enumeration for rigorous compiler testing. In: Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation, pp 347–361
- Zhou H, Lou JG, Zhang, Lin H, Lin H, Qin T (2015) An empirical study on quality issues of production big data platform. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering. IEEE, vol 2, pp 17–26

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.