The Journal of Systems and Software 207 (2024) 111868

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

COPS: An improved information retrieval-based bug localization technique using context-aware program simplification☆,☆☆

Yilin Yang [a], Ziyuan Wang [b,a,*], Zhenyu Chen [a], Baowen Xu [a]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[b] School of Computer Science and Technology, Nanjing University of Posts and Telecommunications, Nanjing, China

ARTICLE INFO

Keywords:
Bug localization
Information retrieval
Stack trace
Static analysis

ABSTRACT

Information Retrieval Based Bug Localization (IRBL) techniques are well suited for large-scale software debugging with fewer external dependencies and lower execution costs. However, existing IRBL techniques have several challenges, including localization granularity and applicability. First, existing IRBL techniques have not yet achieved statement-level bug localization. Second, almost all studies are limited to Java-based projects, while their effectiveness for other popular programming languages (e.g., Python) is unknown. The reason for these deficiencies is that existing IRBL techniques mainly rely on conventional NLP techniques to analyze the bug reports and have not yet fully utilized the stack traces attached to the bug reports. To improve the IRBL technique, we propose a context-aware program simplification technique – COPS – that can localize defective statements in suspicious files by analyzing the stack traces in bug reports, enabling statement-level bug localization for Python-based projects. Our experiment is based on 948 bug reports, and the results show that COPS can effectively localize buggy statements. First, compared to the original stack traces, Top@10 is improved by 102.6%, MAP@10 by 56.2%, and MRR@10 by 95.6%. We found that actual buggy code entities are more likely to appear in the first five frames of the stack trace. Second, COPS can achieve equally good localization performance compared to state-of-the-art statement-level bug localization techniques and achieve 92% buggy statement coverage with a full-scope search. Finally, experiments found that the stack trace's first two-thirds of information is more conducive to localizing buggy statements.

1. Introduction

Software debugging work consumes almost fifty percent of developers' time and effort (Britton et al., 2013; Wang et al., 2015; Youm et al., 2017). In particular, manual scanning of the project repository often relies on domain knowledge to establish semantic connections between bugs and related source files (Rath et al., 2018). This review process is highly dependent on the experience and preferences of the developers. In this regard, automatic debugging techniques have become increasingly popular among developers (Wotawa et al., 2012; Gulzar et al., 2018; Li et al., 2019; Zou et al., 2019; Huang et al., 2019; Wong et al., 2008; Gao and Wong, 2018; Xie et al., 2011; Wong et al., 2010), where bug localization (or fault localization) is considered an important step (Wong and Tse; Wong et al., 2016; Hong et al., 2015; Artzi et al., 2010a). Automatic bug localization techniques assist developers in saving debugging time (Xia et al., 2016) and are gaining importance in industrial practice (Kochhar et al., 2016).

Current automatic bug localization techniques can be divided into two main families of spectrum-based and information retrieval-based techniques (Le et al., 2015). Recent findings show that IR-based techniques perform as well as spectrum-based techniques, which are achieved through low-cost text analysis (Wang et al., 2015). IRBL techniques treat source files as plain documents and bug reports as queries and present the suspicious software entities (i.e., files or methods) to developers by analyzing the relevance or similarity of the two (Zhou et al., 2012; Sisman and Kak, 2012). In contrast to spectrum-based techniques, IRBL techniques do not require the execution of test cases to trigger bugs or even to work in a runnable target software system. Therefore, IRBL techniques can be applied at any stage of software development or maintenance and are practical for developers with limited runtime and resource costs.

However, there are still many challenges of IRBL techniques in practice (Murali et al., 2021). The first is that existing IRBL techniques can only localize bugs at the file level (Rahman and Roy, 2018; Koyuncu

☆ Editor: W. Eric Wong.
☆☆ This paper is an extended version of the conference paper Yang et al. (2022) presented by the same authors.
* Corresponding author at: School of Computer Science and Technology, Nanjing University of Posts and Telecommunications, Nanjing, China.
E-mail addresses: yilin.yang@smail.nju.edu.cn (Y. Yang), wangziyuan@njupt.edu.cn (Z. Wang).

https://doi.org/10.1016/j.jss.2023.111868
Received 1 February 2023; Received in revised form 8 August 2023; Accepted 27 September 2023
Available online 5 October 2023

et al., 2019) or method level (Rahman et al., 2016; Zhang et al., 2019) and cannot yet provide suspicious buggy statements for developers. The second is that almost previous studies are limited to Java-based projects, while their effectiveness for other popular programming languages (e.g., Python) has yet to be explored (Saha et al., 2014; Garnier and Garcia, 2016). The last and most important is how to extract defect-related information from bug reports effectively would directly impact the accuracy of the IRBL technique.

Although researchers have been actively trying to improve retrieval accuracy by using structured information such as stack traces, studies have found that the performance of IRBL techniques may be poor if bug reports lack structured information. On the other hand, having more structured information in bug reports does not necessarily help with automatic bug localization (Wang et al., 2015; Rahman and Roy, 2018; Rath and Mäder, 2019). Rahman and Roy (2018) explain this situation. Firstly, IRBL techniques could not provide valid queries if the bug reports lacked information about buggy program entities. Secondly, most existing IR-based techniques use conventional natural language preprocessing techniques (such as stop word removal and token segmentation) to process bug reports verbatim. This results in the data preprocessing process generating much non-bug related information, i.e., noisy data. A large amount of noisy input leads to a decrease in the accuracy of the IRBL technique. Murali et al. (2021) also showed that a bug report tends to have complex and unwieldy features when it contains a mixture of multiple stack traces from different threads. Therefore, we can improve the effectiveness of IRBL techniques by enhancing existing defect information mining methods. Firstly, we need to select an appropriate length of stack trace to extract suspicious buggy code entities (i.e., file/method names and code entities). Secondly, we should delve into the suspicious files and retrieve code entities related to the defect because the hints in the stack trace may not be the actual buggy statements.

In this paper, we propose COPS (COntext-aware Program Simplification), a novel approach for bug localization at the statement level.[1] Specifically, our technique includes the following steps. (1) Parse stack traces. The stack traces with highly structured information, containing program information directly or indirectly related to the bug, help us avoid noisy queries generated by natural language text. We start by parsing the stack frames one by one. Each stack frame contains an error path and a contextual code snippet. The error path shows the file and method to which the suspicious code snippet belongs. We use regular expressions to parse the file/method names from the error path and extract code entities from the code snippet by parsing the abstract syntax tree. We refer to these obtained suspicious objects as *suspicious_code_set* (Section 3.2). (2) Mining Buggy Entities. Sometimes, the suspicious statements provided in the stack traces are often not the actual buggy statements in the program, or even the position of the hints is far from the position of the actual buggy statements. Such information does not help developers localize bugs for informative or complex source files. Thus, we tried to trace the suspicious objects in known suspicious files to mine the potential bug entities. For an object *e* in *suspicious_code_set*. We retrieve the code entities related to *e* in the suspicious file by analyzing the data dependencies. Furthermore, we also add these bug-relevant code entities to *suspicious_code_set*. Tracing the suspicious objects allows us to determine the execution traces for subsequent bug localization analysis (Section 3.3) . (3) Localize buggy statements. The oversize fault space may increase the time of program repair. Following the previous step, the more suspicious program entities we mine, the more suspicious buggy statements may be involved. If we present all the suspicious statements to developers at once, bug review may remain a labor-intensive task. Here, we set the detection order and scope to address the problem of redundant suspicious objects. Finally, we remove the bug-irrelevant statements from the suspicious file. The simplified text is a collection of suspicious buggy statements. Ultimately, we present this collection to the developers (Section 3.4). This paper is an extended version of Yang et al. (2022). In the previous version, we discussed the effectiveness of COPS (*RQ1*), its comparison with state-of-the-art techniques (*RQ2*), and its coverage of buggy (*RQ3*), respectively. In this version, we add new experiments and discussions to evaluate which factors affect the effectiveness of COPS (*RQ4*). In addition, we have added a description of the research methodology and some technical extensions and discussions.

Overall, this paper makes the following contributions:

- A novel information retrieval technique. Our technique effectively utilizes stack trace information to track potential buggy code entities by analyzing data dependencies. Specifically, the highly structured stack trace information facilitates COPS to extract suspicious code entities directly, effectively avoiding the interference caused by verbatim text (i.e., hand-written bug reports). Furthermore, COPS applies context-aware program simplification techniques to identify bug-related statements in suspicious files and no longer merely sorts suspicious software entities (e.g., files or methods).
- A comprehensive evaluation. We evaluate COPS on the PyTraceBugs benchmark and compare it to state-of-the-art techniques using four widely used metrics. COPS is the first IRBL technique to achieve statement-level bug localization. In terms of accuracy and precision, COPS improves the accuracy of localizing bugs (i.e., Top@10) by 102.6%, precision (i.e., MAP@10) by 56.2%, and result ranking (i.e., MRR@10) by 95.6% compared to the original stack trace information (Section 5.1). We also compared COPS to other families of techniques. The experimental results show that COPS can achieve as good a performance as the state-of-the-art techniques in localizing buggy statements. In particular, COPS not only performs well in Top@10 but still shows better performance in Top@20 (Section 5.2).
- A novel bug localization technique. We attempt to avoid the search space explosion problem by setting the scope of defect detection (Scope@N). The experiments show that Scope@2 is a cost-effective setting, i.e., COPS achieves 64% coverage of buggy statements when the developer detects about 23 lines of COPS recommended text (Section 5.3). We also find that when employing stack traces to localize buggy statements, the closer the program entity is to the top of the stack traces, the more likely it is to introduce noisy data to the bug localization technique. Using the first two-thirds of the suspicious program entities in the stack traces is more beneficial to bug localization (Section 5.4).

Paper Organization. The rest of the paper is organized as follows. We describe the background and motivation in Section 2. Then we present the approach of this paper in Section 3, which consists of three steps: parse stack traces, mining buggy entities, and bug localization. In Section 4, we describe our experimental dataset and evaluation metrics and propose three research questions. We present our experimental results in Section 5 and discuss the findings in Section 6. After that, we describe related works in Section 7 and threats to the validity of this study in Section 8. Finally, we conclude with future work in Section 9.

2. Background and motivating example

2.1. Information retrieval-based bug localization

IRBL techniques treat bug reports as queries and source code files as plain files, presenting suspicious software entities (i.e., suspicious files or methods) to developers by calculating the relevance or similarity of the two. To further improve the effectiveness of the IRBL techniques, researchers attempt to extract feature information (e.g., version history Youm et al., 2017; Wang and Lo, 2016, similarity reports Wong et al., 2014; Moreno et al., 2014, code structure Jiang et al., 2012, stack traces Wang et al., 2015; Rahman and Roy, 2018, etc.) from the code repository and bug reports. Stack traces are regarded as the most valuable information for software debugging (Bettenburg et al., 2008),

[1] https://github.com/pilamula/COPS.

which is the exception information of the program execution when a bug occurs. Bug reports that include stack traces can be repaired more quickly (Zimmermann et al., 2010), and up to sixty percent of fixed bug reports that include stack traces involve modifications to one of the stack frames (Schroter et al., 2010).

Although many empirical studies have confirmed the value of stack traces for bug localization, stack trace-based IRBL techniques still have some weaknesses, and sometimes stack traces even reduce the effectiveness of IRBL techniques (Wu et al., 2014; Xin and Reiss, 2017; Rahman and Roy, 2018). We believe that the causes of this situation are fourfold: (1) the existing IRBL techniques have a coarse granularity of bug localization, (2) the positions provided by stack traces are not the actual bug positions, (3) employ conventional NLP techniques to process bug reports, (4) the excessive stack traces may lead to noisy queries. The details are described as follows.

First, existing IRBL techniques can only localize bugs at the file level or method level (Wong et al., 2016). IRBL techniques predict suspicious program entities by analyzing the program source code and referring to relevant programming rules (Dallmeier and Zimmermann; Rao and Kak, 2011). Since there is no need to execute test cases, IRBL techniques are inexpensive to execute and applicable to all stages of development. Unfortunately, the existing IRBL techniques have a coarse granularity in localizing bugs and have yet to be implemented at the statement level.

Second, the suspicious statements provided by the stack traces are often not the actual defective statements in the program (Wu et al., 2014; Xin and Reiss, 2017); Sometimes, the bug's position hinted at in the stack traces is far from the position of the actual buggy statement. Such hints cannot help the developer localize the bug for content-rich or complex source files. For example, bug report (#3473, pytorch/vision) shows that the most suspicious statement hinted by the stack traces is at line 213, while the actual buggy statement is at line 184. We can see from Fig. 1 that after the program declares the suspicious object `extra_compile_args` (at line 184), three more constraints are set, and any further operations on `extra_compile_args` need to meet these constraints first; This means that when the developer localizes the defect, the bug will not be actively exposed if the test case does not trigger the corresponding constraint. Even a manual review of the repository can be interrupted by a large amount of bug-irrelevant information.

Third, existing IRBL techniques mostly employ conventional NLP techniques to process text verbatim (e.g., stop word removal, token segmentation, etc.) without fully exploiting its structured information (Rahman and Roy, 2018). For example, we attempt to extract bug information from the bug report (#3473, pytorch/vision). After text processing,[2] we could only obtain bug-irrelevant keywords such as "`CPU, extensions, setup, file, modules, python, debug`". We consider that while stack traces share some characteristics with plain text bug reports, they should not simply be treated literally. In particular, conventional NLP techniques often rely on word frequency algorithms, which may lead to the automatic discarding of low-frequency but essential information in the results (Strzalkowski and Vauthey, 1992; Hofstätter et al., 2019). In addition, the order of stack traces text processing is also an issue ignored in the prior works. Contrary to traditional reading (i.e., top-down), information is more suspicious closer to the bottom of the stack.

Finally, excessive stack traces within bug reports can introduce a lot of noisy data into the input of IRBL techniques (Wang et al., 2015; Rath and Mäder, 2019). In particular, a bug report contains two or more stack traces from different threads, and this bug report tends to be complex and unwieldy features (Murali et al., 2021). For example, bug report (#5482, numba/numba)[3] has two stack traces attached that have 235 lines in text. Although the actual bug-relevant information is indeed in the stack traces (i.e., the third stack frame of the first stack trace); the 235 lines of stack trace text inevitably introduce a lot of noisy data into the input of the IRBL techniques, which seriously affects the execution efficiency of automatic bug localization.

Table 1
A bug report (#3473, pytorch/vision).

Field	Content
Title	Cannot be installed from source in debug mode if torch was built without CUDA
Description	1. Install torch without CUDA. 2. Run DEBUG = 1 python setup.py develop
Traceback	No CUDA runtime is found, using CUDA_HOME = '/opt/cuda' Building wheel torchvision-0.9.0a0+b266c2f Compile in debug mode Traceback (most recent call last): File "**setup**.py line 472, in <**module**> **ext_modules** = **get_extensions**()", File "**setup**.py line 213, in **get_extensions** **extra_compile_args**.append(−g") KeyError: 'cxx'

2.2. Motivating example

We describe our study motivation by using the bug report[4] in Table 1, which is extracted from pytorch/vision #3473. This bug is caused by developers not checking whether '`cxx`' is a key value of '`extra_compile_args`' before calling '`extra_compile_args['cxx'].append()`'. Fig. 1(a) shows the code snippets of the suspicious file *setup.py*. Buggy statements ( ) and stack trace information ( ) are highlighted with different colors. Fig. 1(b) shows the collection of suspicious buggy statements generated by the program simplification technique. Among these, the actual buggy statements are on lines 184, 199–202, and 207. Line 208 is a plausible but not buggy statement, and line 213 is an exception message in stack traces.

In this paper, we try to implement the statement-level bug localization technique. First, to avoid too much noisy data in the input text, we need to limit the length of the stack traces. According to the existing studies (Rahman and Roy, 2018; Murali et al., 2021; Schroter et al., 2010; Zimmermann et al., 2010), we set two restrictions: One is that the bug report contains only one stack trace; the other is that we only extract the first ten stack frames. Stack traces are highly structured text that can easily use regular expressions to identify suspicious program entities. For a stack frame, the first line is the error path, and the following lines are the contextual code snippets. We can extract the suspicious file/method name from the error path and the suspicious code entities from the code snippet.

Second, as shown in the bug report (#3473, pytorch/vision), the suspicious file *setup.py* has 477 lines of source code, among which the suspicious method `get_extensions` has 291 lines. With limited information, it is challenging to identify the buggy statements. Sometimes, the actual buggy statements are not shown in the stack traces. To address this issue, we analyze suspicious variables in suspicious files to localize buggy statements. For example, by parsing the first stack frame, we can extract the suspicious code entity `extra_compile_args`. In the `get_extensions()` method, we can determine that `extra_compile_args` is declared on line 184 by analyzing the data dependencies. Then, we find that `extra_compile_args` is reassigned on lines 199 and called on lines 207, 208, and 213.

Finally, we remove the text that is irrelevant to the bug from the suspicious file. The retained text, which consists of the collection of

[2] https://www.cortical.io/freetools/extract-keywords/.
[3] https://github.com/numba/numba/issues/5482.
[4] https://github.com/pytorch/vision/issues/3473.

```
@@ -181,7 +181,7 @@ def get_extensions():
181
182   define_macros = []
183
184   extra_compile_args = {}
185   if (torch.cuda.is_available() and ((CUDA_HOME is not None) \
186       or is_rocm_pytorch)) or os.getenv('FORCE_CUDA', '0') == '1':
187     extension = CUDAExtension
@@ -196,15 +196,11 @@ def get_extensions():
196     else:
197       define_macros += [('WITH_HIP', None)]
198       nvcc_flags = []
199     extra_compile_args = {
200       'cxx': [],
201       'nvcc': nvcc_flags,
202     }
203
204   if sys.platform == 'win32':
205     define_macros += [('torchvision_EXPORTS', None)]
206
207     extra_compile_args.setdefault('cxx', [])
208     extra_compile_args['cxx'].append('/MP')
209
210   debug_mode = os.getenv('DEBUG', '0') == '1'
211   if debug_mode:
212     print("Compile in debug mode")
213     extra_compile_args['cxx'].append("-g")
```

(a) The setup.py file is the suspicious file shown in the bug report (#3473 pytorch/vision); the above is the bug relevant code snippet.

```
184 : extra_compile_args = {}
199 : extra_compile_args = {
        'cxx': [],
        'nvcc': nvcc_flags,
    }
207 : extra_compile_args.setdefault('cxx', [])
208 : extra_compile_args['cxx'].append('/MP')
213 : extra_compile_args['cxx'].append("-g")

Actual buggy statement : Line 184,199,207
Statement is plausible but not buggy : 208
Stack trace information: 213
```

(b) Simplified text of the above code snippet

Fig. 1. Motivating example. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

suspicious buggy statements, can be presented to the developers for review. Program simplification techniques are often used to remove code that does not affect the program's output, also known as dead code elimination. We borrow this idea, treat bug-irrelevant source code as dead code, and remove it from the suspicious file.

3. The COPS approach

3.1. Overview

Fig. 2 shows our proposed technique-COPS schematic diagram. We employ a context-aware program simplification technique to localize bugs in the source code. First, we describe the extraction process of suspicious buggy entities (Section 3.2). Then, we identify potential suspicious buggy entities by data dependency analysis (Section 3.3). Finally, we retrieve the suspicious buggy statements in the buggy file (Section 3.4).

3.2. Parse stack traces

Stack traces record the active stack frames reported when the bug occurred. Its highly structured information not only helps us to avoid the noisy queries generated by natural language text but also contains program information directly or indirectly related to the bug, such as method calls, file names, code snippets, etc. In this paper, we employ a Python-based data set called PyTraceBugs. Python stack traces is also termed ***traceback***. Generally, a traceback starts with "*Traceback (most recent call last):*" and ends with "*TypeError*", with the stack frames in between. The first line shows the error path for a stack frame, and the following lines are the code snippets (i.e., suspicious buggy statement). The error path contains the suspicious file name, method name, and line number.

According to Schroter et al. (2010), if a bug report has two or more stack traces, the first stack trace is more helpful for fixing the bug. In addition, bug reports contain multiple stack traces or a stack trace with too many stack frames, which can cause noisy queries (Rahman and Roy, 2018). Therefore, we selected bug reports that contained only one stack trace and extracted the first ten stack frames from the stack trace. Considering the characteristics of the stack traces, we first use regular expressions (as described below) to extract suspicious methods `(?<=File ").+(?=.py") | (?<=line ).+(?=,) | (?<=, in ).+` from the error path of each stack frame in turn. Then, for the suspicious statements in the stack frames, we extract the suspicious code entities by parsing their abstract syntax tree (AST). These statements sometimes fail to be transformed into the AST due to incomplete syntax. We use text segmentation, stop word removal (here are Python's keywords and built-in function names), and regular expressions (based on punctuations, camel case splitting, and snake case splitting) to detect these statements to extract suspicious code entities. Ultimately, we refer to these obtained suspicious objects as *suspicious_code_set*. (**Step A**, Fig. 2)

3.3. Mining buggy entities

Stack traces in bug reports are essential information for bug localization. However, the information presented by the stack traces can only help us localize the bug at the file or method level, and the actual buggy statement sometimes does not appear in the stack traces. As shown in the bug report (Table 1), the stack traces hint that the most suspicious statement is at line 213, while the actual buggy statement is at line 184.

In order to implement statement-level bug localization techniques, we attempted to trace back the suspicious objects in the known suspicious files to mine potential buggy entities. In the bug report (Table 1), the buggy file is known to be `setup.py`. We transform `setup.py` into an AST and analyze the dependencies between program entities by traversing the AST nodes. As shown in Fig. 3, there are four sub-nodes under the root node (⬭), which correspond to the four methods in the `setup.py` file. Among them, `get_extensions` and `<module>` are known suspicious methods.

The program dependency graph (Chen and Rajlich, 2000; Moreno et al., 2014) abstracts the dependencies between statements into a graph, where nodes correspond to statements of the program and edges represent structural dependencies between statements, including control dependencies and data dependencies between statements (Ferrante et al., 1987; Hammer and Snelting, 2009). First, we determine whether the other two methods (i.e., `write_version_file` and `run`) are bug-relevant program entities by analyzing the dependencies of the suspicious file. If there are no call relationships or data dependencies between a method and a known suspicious method, we treat the method as bug-irrelevant. As shown in Fig. 3, `write_version_file` is treated as a bug irrelevant code entity (⬬). Instead, the `run` method is called in the `get_extensions` method and returns the value (i.e., `file`) to `extra_compile_args`. In this case, we treat the `file` object as a suspicious buggy code entity (⬬).

Then, we trace the suspicious code entities in `<module>` and `get_extensions` methods. By analyzing the program dependencies, we find that in `<module>` method, the `ext_modules` object is the return value from the `get_extensions` method, which is also shown in the second stack frame. Here, we treat it as, `<module>.ext_modules = get_extensions.ext_modules`. Also, we find that the other variables in the `<module>` method are not

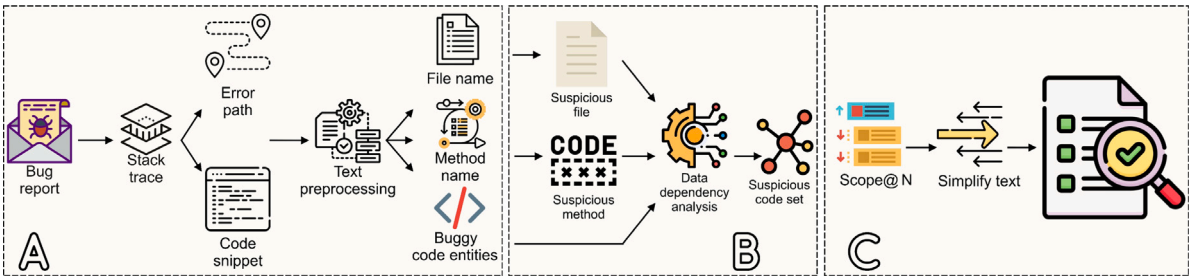

Fig. 2. Schematic diagram of the COPS technique.

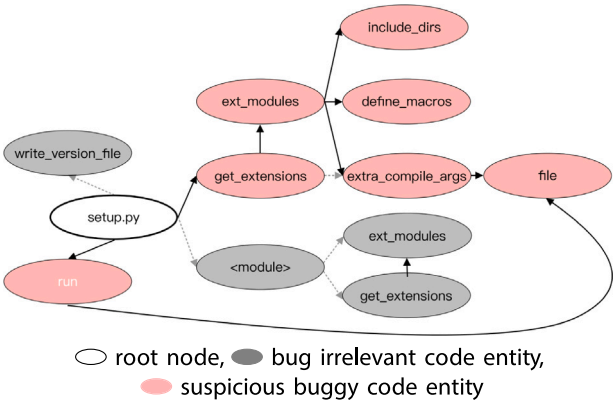

Fig. 3. The dependencies between the suspicious objects in "setup.py" file.

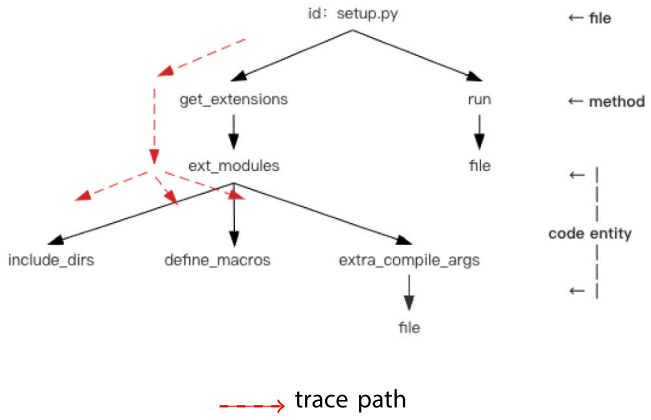

Fig. 4. Example for bug detection scope. The developer sets Scope@2 and detects only the first two level elements in the dependency of suspicious code entities.

associated with suspicious code entities. Thus, we remove `<module>` from *suspicious_code_set*. As shown in Fig. 3, we treat the `<module>` method as a bug irrelevant code entity (●). Similarly, after retrieving the `get_extensions` method, we find that the assignment of the `ext_modules` object is composed of three variables, that is `include_dirs`, `define_macros`, and `extra_compile_args`. Thus, we treat three variables as suspicious buggy code entities (●).

Finally, we identify bug relevant program entities by analyzing and mining potential buggy code entities in suspicious files. (**Step B,** Fig. 2)

Suspicious program entities in stack trace
file: setup.py
methods: get_extensions, ~~<module>~~
code entities: extra_compile_args, ext_modules

Mining buggy entities in suspicious file
methods: get_extensions, run
code entities: extra_compile_args, ext_modules
include_dirs, define_macros, file

3.4. Bug localization

Recent studies have shown that a fault space that is too large may increase the time it takes to fix a program (Qi et al., 2013; Wen et al., 2018). The more suspicious program entities we obtain, the more suspicious buggy statements there are. In real-world software development, bug reports often have complex descriptions with stack traces that contain many suspicious files and methods. We may extract many suspicious buggy code entities for programs with content-rich or complex code structures. However, presenting all suspicious buggy statements to developers at once can make bug review work more difficult and time-consuming. For example, consider the statements `b=a+1`, `c=b+1`, and `d=c+1`. If the stack trace shows that `d` is the buggy code entity, then `a`, `b`, and `c` are all tracked as suspicious objects. In this case, all statements are considered suspicious buggy statements. To reduce the size of the fault space, we set the detection order and scope to address the redundancy of suspicious objects.

First, we detect suspicious files and methods in the order of the stack framework. For example, in the previous section, we identified two suspicious methods, `get_extensions` and `run`, where `get_extensions` has a higher priority than `run`; This is because `get_extensions` is a suspicious program entity that appears in the first frame of the stack trace, while `run` is a potentially suspicious program entity derived from the analysis of program dependencies. Then, we define a *Scope@N* (default N = 2) that allows developers to set the scope of bug detection. Setting N = 2 is a relatively cost-effective parameter, and we discuss the impact of different detection scopes on bug localization in detail in Section 5.3. As shown in Fig. 4, the dependencies between the nodes are that "`get_extensions ← ext_modules`"; "`ext_modules ← {extra_compile_args, include_dirs, define_macros}`"; "`extra_compile_args ← file`". Scope@2 indicates that we only detect the first two level elements of this sequence, that is, `ext_modules` and `extra_compile_args`, `include_dirs`, `define_macros`.

This step is crucial because the more suspicious code entities there are, the more suspicious buggy statements COPS outputs. Therefore, we must balance the identification rate of buggy statements and the labor required to review the text. From the perspective of buggy statement coverage, a broader review scope can indeed improve the identification rate of buggy statements, but it also increases the developer's workload. Therefore, we first set the detection order, i.e., we ranked the suspicious program entities. The closer the program entity appears at the bottom of the stack traces, the more suspicious it is. Program entities that appear directly in the stack traces are more suspicious than those associated with them. Next, we set the detection scope, i.e., Scope@N. Finally, we parsed the abstract syntax tree of the suspicious file to determine the statement where the suspicious code entity is within. In

Table 2
Basic information of the experimental dataset.

#Projects	239	#Bugs	948
KLOC	417.74	#Violation types	131
Avg.# ST lines	19.49	Avg.# ST files	4.70
Avg.# ST methods	6.86	Avg.# ST code entities	23.75

* Stack Traces (ST).

the source file, we removed any statements irrelevant to the suspicious code entity, leaving only the simplified text, i.e., the suspicious buggy statement, for the developer to review. (**Step C**, Fig. 2)

4. Experiment setup

4.1. Experimental dataset

We evaluated the effectiveness of COPS on the benchmark dataset PyTraceBugs (Akimova et al., 2021), a large labeled dataset based on well-received algorithmic principles (Widyasari et al., 2020; Ferenc et al., 2020) to obtain relevant metadata from the GitHub. For this dataset, the projects are from diverse application domains, including infrastructure, cloud computing, and data science, and each project has well-maintained issue trackers. In addition, these bug reports contain full stack traces and source code at method granularity.

We selected 948 bug reports from PyTraceBugs. Each bug report should meet the following criteria: (1) It should only have one standard Python traceback. (2) The bug patch should be in a single file and small-sized, as small-sized bug patches are more conducive to automatic debugging (Sobreira et al., 2018). Following the approach of Schroter et al. (2010), we found that approximately 47.8% of stack traces showed the name of the buggy file in the first frame, and nearly 90% of stack traces showed the name of the buggy file in the first three frames. However, upon closer examination of the suspicious statements shown by the stack traces, we found that less than 30% of stack traces showed the actual buggy statement. This suggests that although stack traces can help us localize the buggy file, they may not always accurately localize the buggy statement. This observation prompted us to explore potential buggy code entities for achieving fine-grained bug localization.

Since COPS's current implementation can only localize buggy statements in a single file, we evaluate the effectiveness of COPS on the actual buggy file. In addition, we only detect the first ten frames of the stack traces and set the detection range to Scope@2. Table 2 presents the details of the experimental dataset; on average, one stack trace within 19.49 lines of text, 4.7 suspicious files, 6.86 suspicious methods, and 23.75 suspicious code entities.

4.2. Evaluation metrics

To measure the effectiveness of COPS, we used the following four widely used metrics in our study (Wong et al., 2016).

Top@K: In some literature, Top@K is also referred to as Hit@k (Rahman and Roy, 2018), Top-K Accuracy (Rahman and Roy, 2017) and Recall@Top-K (Saha et al., 2013). This metric returns the percentage of queries for which at least one buggy entity (i.e., the buggy statement in this study) is correctly returned within the Top K results. When a technique has a high Top@1 accuracy, its result can often be used automatically for downstream applications without human intervention (Murali et al., 2021). The higher the Top@K value, the less effort it takes for the developer to localize bugs.

Mean Average Precision@K (MAP@K): This metric measures the ranking of correct results in a list (Manning, 2008; Lawrie, 2012). First, the Precision@K is calculated for each occurrence of the buggy statement in the list. Then, the average of Precision@K is calculated for all buggy statements in the ranked list, i.e., *AP@K*. *Mean Average Precision@K* is the average of *AP@K* for all queries (q). Here, *buggy(k)* represents whether this statement is buggy; it returns 'true' if it is or 'false' if it is not. P_k is the precision at the kth result, and D is the number of total results. The higher the value of MAP@K, the better the performance.

$$AP@K = \frac{\sum_{k=1}^{D} P_k \times buggy(k)}{|S|} \tag{1}$$

$$MAP@K = \frac{\sum_{q \epsilon Q} AP@K(q)}{|Q|} \tag{2}$$

Mean Reciprocal Rank@K (MRR@K): This metric is the average reciprocal rank@k of a set of queries (Voorhees, 2001). Reciprocal Rank@K is the multiplicative inverse of the rank of the first faulty statement within the top K results. Here, $Rank_{best}(q)$ returns the rank of the first buggy statement within a ranked list. MAP@K and MRR@K are often used to evaluate the ability of a given technique to localize bugs. As with MAP@K, the higher the MRR@K value, the better the performance of bug localization (Wen et al., 2016, 2019).

$$MRR@K(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{Rank_{best}(q)} \tag{3}$$

Average Coverage Ratio (ACR): This metric measures the percentage of the number of correct results that a bug localization approach can return from its results (Tantithamthavorn et al., 2013; Artzi et al., 2010b). Since the result always presents multiple suspicious statements, *Coverage* is the percentage of the result. $\#P_s$ is the number of successfully predicted statements. $\#actual$ is the number of the actual buggy statements. *ACR* is the average coverage ratio of all results.

$$Coverage = \frac{\#P_s}{\#actual}, ACR = \frac{\sum_{i=1}^{M} Coverage_i}{|Q|} \tag{4}$$

4.3. Research questions

Our evaluation addresses the following research questions:

RQ1. How does COPS take advantage of existing information in the stack traces? We attempt to extract suspicious buggy code entities from the stack traces and localize the buggy statements in the suspicious files. Therefore, we first need to evaluate the potential of COPS in exploiting stack trace information.

RQ2. How does COPS perform in bug localization? We compare COPS with the current state-of-the-art techniques to evaluate the effectiveness of COPS in localizing bugs at the statement level.

RQ3. How many buggy statements does COPS cover? In this study, we performed an in-depth analysis to evaluate the effectiveness of COPS for localization by using coverage rates.

RQ4. How do suspicious buggy code entities impact the effectiveness of COPS? We evaluate the impact of the number of stack frames and the suspiciousness of buggy code entities on the effectiveness of the bug localization algorithm.

5. Experimental results

5.1. Answering RQ1: Effectiveness of COPS

To investigate RQ1, we measured the effectiveness of COPS and compared its performance metrics with the original stack traces. We first define an approach for identifying suspicious buggy statements based on the study by Schroter et al. (2010). This approach returns a list of suspicious buggy statements in the stack traces. The Nth frame is given a suspiciousness score of *1/N*. The score of a suspicious buggy statement is the maximum suspiciousness score among all frames in the stack traces. The principle of COPS is to extract and mine suspicious objects from the stack traces and localize the buggy statements. Our purpose is to evaluate the potential of COPS in exploiting stack trace information. In our experimental dataset, 90% of the bugs reported in the first three frames of the stack traces show the positions of their

Table 3
Performance of COPS and stack traces in bug localization.

Metric	ST_{CS}	$COPS_{RT}$		Impr (%)
TOP@1	0.114	Scope@2	0.227	99.1%
		Scope@ALL	0.168	47.4%
TOP@5	0.230	Scope@2	0.426	85.2%
		Scope@ALL	0.402	74.8%
TOP@10	0.273	Scope@2	0.553	102.6%
		Scope@ALL	0.560	105.1%
MAP@10	0.188	Scope@2	0.293	55.9%
		Scope@ALL	0.269	43.1%
MRR@10	0.130	Scope@2	0.254	95.4%
		Scope@ALL	0.220	69.2%

* ST_{CS}: the suspicious buggy code snippets in stack traces.
* $COPS_{RT}$: the returned text of COPS.

buggy files; this indicates that the stack traces help identify the files where the bugs are localized. Less than 30% of the stack traces in total show the actual buggy statements, which contradicts the former result that stack traces can provide detailed information about the specific lines of code causing the bugs.

> **Finding 1.1:** Stack traces can effectively show the position of files with bugs, but do not provide fine-grained bug localization information.

On average, each stack trace contains 19.49 lines of text; that is, a stack trace contains almost ten suspicious buggy code snippets, as shown in Table 2. We compare the top ten statements of the suspicious list returned by COPS with the suspicious code snippets of the stack traces to evaluate their effectiveness in localizing bugs. Here, we refer to the suspicious buggy code snippets in stack traces as ST_{CS}, and the text returned by COPS as $COPS_{RT}$. Table 3 shows the results of Top@K, MAP@10 and MRR@10. ST_{CS} results at Top@1, Top@5, and Top@10 are 0.114, 0.230, and 0.273, respectively, with a significant slowdown from Top@5 onwards. This performance suggests that the actual buggy statements tend not to appear in the stack trace's first five frames. ST_{CS} also performs poorly for MAP@10 and MRR@10.

> **Finding 1.2:** The buggy statements are more likely to appear in the top five frames of the stack traces rather than in the first frame.

Compared with ST_{CS}, $COPS_{RT}$ achieves a better improvement. When set Scope@2, the results for Top@1, Top@5, and Top@10 are 0.227, 0.426, and 0.553 respectively. When set Scope@ALL, the results for Top@1, Top@5, and Top@10 are 0.168, 0.402, and 0.560 respectively. In terms of MAP@10 and MRR@10, MAP@10 improved from 0.188 to 0.293, an increase of 55.9%, and MRR@10 improved from 0.130 to 0.254, an increase of 95.4%. Overall, the metrics perform better at Scope@2. This improvement shows that our technique significantly exploits the potential of stack trace information. Also, developers only need to detect the first ten statements of $COPS_{RT}$ to have a 55.3% probability of localizing an actual bug statement.

> **Finding 1.3:** Extracting suspicious buggy code entities from the stack traces can help retrieve buggy statements.

5.2. Answering RQ2: Comparing with other families techniques

To evaluate the effectiveness of COPS, we compared its performance with two techniques: HSFL (based on history spectrum) (Wen et al., 2019) and StackTrace (based on stack trace) (Zou et al., 2019). We selected HSFL because it has shown good performance compared to other techniques, and we selected StackTrace as a baseline because COPS is also a bug localization technique based on stack trace information.

Following the results of RQ1, we evaluate the effectiveness of COPS at Scope@2. Tables 4 and 5 present the Top@K results of the three techniques. The results indicate that COPS outperforms both baselines in terms of Top@K results. Specifically, COPS shows improvements of 26.4%, 4.1%, and 11.6% over HSFL for Top@1, Top@5, and Top@10, respectively. Notably, COPS performs well not only on Top@10 but also shows a significantly better subsequent improvement than the two benchmarks. Compared to HSFL, the improvement of COPS on Top@20 increases from 57.4% to 66.5%. This finding suggests that developers only need to identify the first 20 statements of $COPS_{RT}$ to have a 66.5% probability of localizing an actual bug statement. Additionally, Table 4 shows the results of MAP@10 and MRR@10, demonstrating that COPS significantly outperforms StackTrace in both metrics. On average, COPS achieves a 65.1% improvement in MAP@10 and a 75.6% improvement in MRR@10 compared to StackTrace. Furthermore, COPS achieves an average improvement of 19.1% over HSFL in MAP@10 and slightly lower than HSFL in MRR@10.

In addition, we compared the performance of COPS with three coverage-based bug localization techniques in terms of Top@K. We selected Ochiai (Abreu et al., 2007) and DStar (Wong et al., 2013), two classical spectrum-based approaches, and SmartFL (Zeng et al., 2022), a state-of-the-art semantics-based probabilistic fault localization technique. Performance data for all three techniques was obtained from existing studies (Zou et al., 2019; Zeng et al., 2022). Table 6 presents the number and percentage of bugs localized at the statement level using different techniques. SmartFL outperforms the other two coverage-based bug localization techniques for all values of k. COPS significantly outperformed Ochiai and DStar at Top@1 and slightly outperformed SmartFL. At Top@5, COPS slightly outperformed Ochiai and DStar and was comparable to SmartFL. At Top@10, COPS slightly outperformed the other three techniques.

> **Finding 2.1:** COPS achieves good performance compared to different families of bug localization techniques. In particular, COPS not only performs well in Top@10, but also maintains a good improvement in performance improvement thereafter.

According to existing research (Pearson et al., 2017; Zhang et al., 2017; Campos et al., 2012; Wong et al., 2016; Gao and Wong, 2018; Xie et al., 2011), spectrum-based bug localization techniques can perform well at the statement level but also suffer from some limitations. Specifically, the information used to construct the spectrum does not directly correspond to the root cause (the actual defective statement), and there is an inability to distinguish well between buggy and non-buggy code entities.

The core idea of HSFL is that suspicious code entities evolved from bug-inducing commits are more likely to be the faulty statements, and potential noise data can be filtered out by executing bug-revealing tests.

Table 4
Performance of COPS, StackTrace and HSFL in bug localization.

Metric	StackTrace	HSFL	COPS	Impr (%)	
Top@1	23/357	64/357	215/948	↑ ***251.7%***	↑ ***26.4%***
Top@5	46/357	146/357	404/948	↑ ***230.4%***	↑ ***4.1%***
Top@10	59/357	177/357	525/948	↑ ***234.7%***	↑ ***11.6%***
MAP@10	0.149	0.246	0.293	↑ ***65.1%***	↑ ***19.1%***
MRR@10	0.164	0.288	0.254	↑ ***75.6%***	↓ ***11.8%***

* 357: the number of bugs in HSFL and StackTrace.
* 948: the number of bugs in our experimental dataset.

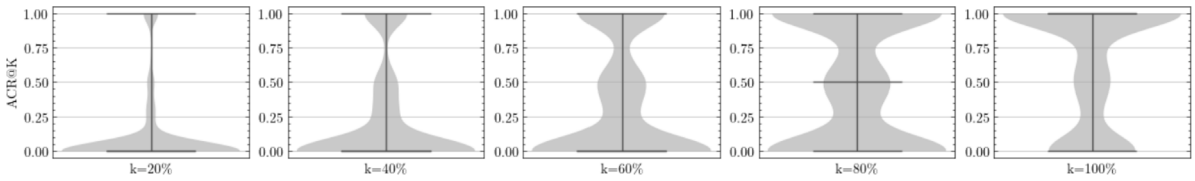

Fig. 5. Performance of COPS (Scope@2) in terms of ACR.

Table 5
Results of Top@N of COPS, StackTrace and HSFL.

Top@K	COPS	StackTrace	HSFL
k = 1	22.7%	6.4%	17.9%
k = 2	28.9%	8.7%	27.7%
k = 3	33.6%	10.1%	32.5%
k = 4	39.4%	11.8%	37.3%
k = 5	42.6%	12.9%	40.9%
k = 6	46.7%	13.7%	43.4%
k = 7	48.7%	13.7%	44.3%
k = 8	50.9%	13.7%	46.5%
k = 9	52.8%	14.3%	48.5%
k = 10	55.3%	16.5%	49.6%
k = 11	56.7%	17.9%	52.1%
k = 12	58.1%	18.5%	52.4%
k = 13	59.5%	19.6%	53.8%
k = 14	61.1%	20.2%	54.3%
k = 15	62.1%	20.7%	54.6%
k = 16	63.8%	21.0%	54.9%
k = 17	64.6%	21.6%	55.5%
k = 18	65.4%	21.8%	55.5%
k = 19	65.6%	22.4%	56.0%
k = 20	66.5%	22.7%	57.4%

In our technique, the closer the code entity (and the entities it depends on) is to the bottom of the stack traces, the more likely it is to be the one causing program failures. By setting the defect detection scope, potential noisy data can be filtered out, and the fault space can be effectively managed. The performance of StackTrace once again proves that using only stack trace information is not very effective in helping fine-grained bug localization techniques. In our approach, this issue is effectively solved by analyzing the dependencies between program entities. Finally, experiments show that our approach is as good as the state-of-the-art statement-level defect localization techniques.

> **Finding 2.2:** Analyzing the dependencies between program entities helps to mine potential buggy code entities. Potentially noisy data can be filtered out by setting the defect detection range. These contribute to improving the effectiveness of stack traces for bug localization.

5.3. Answering RQ3: Buggy statements coverage ratio of COPS

We use the Top@K metric in RQ1 and RQ2 to evaluate the effectiveness of our approach, which measures the percentage of successfully localized bugs among the top K results. However, in practice, many bugs involve multiple statements with errors. Therefore, we perform an analysis to verify the effectiveness of COPS on bug localization by calculating the average coverage ratio (ACR) (Tantithamthavorn et al., 2013). We report the results of ACR and the average number of lines of $COPS_{RT}$. On average, a buggy file contains 417.74 lines of statements. In COPS, we define Scope@N, which allows developers to set the scope of bug detection (see Section 3.4). Here, the default value of N is 2. A larger value of N indicates a more significant number of suspicious buggy code entities and a richer content of the $COPS_{RT}$. To further analyze the impact of the defect detection scope on bug localization, we evaluated the number of lines of $COPS_{RT}$ and the average coverage of buggy statements for different detection scopes.

Table 7 shows the results for Scope@N, where N = 2, 3, 5, and all, respectively. Scope@ALL indicates that we retrieved all suspicious code entities. Since the number of lines in $COPS_{RT}$ is not a static value, we calculated ACR@10% to ACR@100%. The results of Scope@2 at ACR@50%, ACR@80%, and ACR@100% are 0.31, 0.49, and 0.64, respectively. This performance indicates that COPS can achieve 64% buggy statement coverage when the developer detects about 23 lines of $COPS_{RT}$. The results of Scope@ALL at ACR@50%, ACR@80%, and ACR@100% are 0.44, 0.73, and 0.92, respectively. This performance indicates that COPS can achieve 92% buggy statement coverage when the developer detects about 84 lines of $COPS_{RT}$.

In addition, we also evaluated Scope@3 and Scope@5. As shown in Table 7, at ACR@50%, Scope@2 performed similarly to Scope@3 and Scope@5; This suggests that although we detected more text, no significant increase in defective statements was found. On ACR@100%, Scope@2 performed similarly to Scope@3. Although Scope@5 improved by ten percent over Scope@2 on ACR@100%, the amount of text detected was three times that of Scope@2.

We compared the results of different detection scopes and found that setting N = 2 is a reasonable choice. Firstly, when N is equal to 2, the number of lines in $COPS_{RT}$ is reduced to 5.5% of the original file, significantly reducing the scope of code review for developers. Additionally, the overall performance of Scope@3 is very close to that of Scope@2. Although Scope@5 has better overall performance, it requires more text content to be detected than Scope@2. In comparison, setting Scope@ALL allows for a defect coverage of 92% in $COPS_{RT}$. Therefore, Scope@2 is more suitable for quickly localizing buggy statements, while Scope@ALL is more appropriate for full-scope detection of buggy statements.

> **Finding 3.1:** COPS significantly narrows the code review scope of developers. On average, the number of lines of $COPS_{RT}$(`Scope@2`) is 5.5% of the buggy file; the number of lines of $COPS_{RT}$(`Scope@ALL`) is 19.87% of the buggy file.

Table 6
Results of Top@N of COPS and three coverage-based bug localization techniques.

Metric	Ochiai	DStar	SmartFL	COPS	Impr (%)		
Top@1	11/222	12/222	47/222	215/948	↑ ***357.7%***	↑ ***319.6%***	↑ ***7.1%***
Top@5	86/222	86/222	97/222	404/948	↑ ***10.0%***	↑ ***10.0%***	↓ ***2.5%***
Top@10	118/222	117/222	118/222	525/948	↑ ***4.2%***	↑ ***5.1%***	↑ ***4.2%***

* 222: the number of bugs in SmartFL.
* 948: the number of bugs in our experimental dataset.

Table 7
Results of average coverage of COPS.

Metric	Scope@2	Scope@3	Scope@5	Scope@ALL
Avg.# lines	22.78	32.92	64.23	84.37
ACR@10%	0.04	0.04	0.06	0.08
ACR@20%	0.10	0.11	0.12	0.16
ACR@30%	0.16	0.17	0.20	0.25
ACR@40%	0.23	0.25	0.29	0.35
ACR@50%	0.31	0.33	0.36	0.44
ACR@60%	0.35	0.38	0.42	0.54
ACR@70%	0.42	0.44	0.51	0.63
ACR@80%	0.49	0.53	0.62	0.73
ACR@90%	0.54	0.60	0.67	0.82
ACR@100%	0.64	0.65	0.74	0.92

Comparing Scope@2 with Scope@ALL, the buggy statement coverage is very close in the first 50% of the $COPS_{RT}$, and in the second 50% of the $COPS_{RT}$, the coverage rate of Scope@ALL is still steadily increasing. This result indicates that nearly half of the buggy statements can be detected quickly. In contrast, some buggy statements are deeply hidden and require constant analysis of the dependencies between program entities to be detected. As shown in Fig. 5, Scope@2 has achieved optimum performance with a median of ACR@80% of 0.5 and ACR@100% of 1.0, which indicates that COPS can detect the vast majority of buggy statements except for some exceptional cases. Although we prefer to advocate a small scope of bug detection, it is very promising that COPS can identify 92% of buggy statements when we retrieve them in full scope. This motivates us to further optimize the bug localization algorithm in the future to improve the detection and coverage rates.

> **Finding 3.2:** $COPS_{RT}$(`Scope@2`) can achieve 64% buggy statement coverage. $COPS_{RT}$(`Scope@ALL`) can identify 92% buggy statements when we retrieve them in full scope.

5.4. Answering RQ4: Factors affecting bug localization

Recent studies (Qi et al., 2013; Le Goues et al., 2013; Wen et al., 2018) have shown that the size of the fault space directly affects the effectiveness of search-based automatic program repair techniques. As the size of the fault space increases, the repair time increases, but the probability of a successful repair decreases. The experimental results of RQ1 and RQ2 show that setting the scope of defect detection can effectively solve the problem of fault space explosion. To further improve the effectiveness of COPS, we try to reduce the number of program entities with low suspiciousness and adapt the order of $COPS_{RT}$, which is based on the following two intuitions.

First, Schroter et al.'s study shows that about 80% of buggy filenames can be found in the first six frames of a stack trace, and about 90% can be found in the first ten frames. Combined with our experimental results in RQ1 (i.e., the buggy statements are more likely to appear in the first five frames of the stack traces rather than in the first frame), we believe that we can attempt to improve its effectiveness for bug location by extracting part of the stack trace information. In this case, we evaluate the impact of the first one-third (ST@1/3), the first two-thirds (ST@2/3), and the full information (ST@3/3) of the stack traces on the effectiveness of buggy localization, as shown in Table 8. We use COPS default Scope@2. This practice necessarily reduces the number of non-buggy objects and narrows the scope of potentially buggy objects, thus filtering out the noise data in $COPS_{RT}$. Second, the order of the text present in $COPS_{RT}$ no longer follows the order of the original buggy file. Instead, $COPS_{RT}$ prioritizes the output of highly suspicious statements. Specifically, we not only prioritize retrieval of methods with high suspiciousness (in this case, the order of stack frames). In a method, referring to the suspicious code entities list, we record the line number where the statement that includes the suspicious code entity is located by parsing the bug file AST. Based on this retrieval order, we finally generate $COPS_{RT}$. This practice is done to improve the accuracy of Top@K.

Table 8
Performance of different stack traces for bug localization.

ST@K	TOP@1	TOP@5	TOP@10	MAP@10	MRR@10
ST@(1/3)	0.198	0.455	0.590	0.291	0.241
ST@(2/3)	0.245	0.507	0.640	0.330	0.290
ST@(3/3)	0.213	0.446	0.570	0.295	0.251

* ST@K: We evaluate the effectiveness of the first K stack frames for bug localization.

As shown in Table 8, ST@(2/3) outperforms ST@(1/3) and ST@(3/3) in all metrics . In contrast, the poor performance of ST@(1/3) can be attributed to two aspects. Firstly, as shown in Fig. 6, the first third of the stack trace contains, on average, less than 40% of the suspicious buggy files. The effectiveness of bug localization is reduced because enough suspicious program entities are not available. Secondly, the performance of ST_{CS} in RQ1 indicates that buggy statements are more likely to appear in the first five frames of the stack traces rather than at the bottom. Therefore, ST@(2/3) is more effective than ST@(1/3). The reasons for the poor performance of ST@(3/3) can be attributed to the following two aspects. First, our algorithm for mining potentially buggy code entities relies on depth-first traversal. Although we set Scope@2 to limit the fault space, it still introduces noisy data as the number of retrievals increases. Second, the stack frames near the top, even if they contain actual buggy code entities, are ranked lower in $COPS_{RT}$ for the buggy statements associated with them due to the retrieval order, which leads to poor performance in each metric. Based on the above analysis, the information in the first two-thirds of the stack trace is more useful for localizing buggy statements.

> **Finding 4:** Lacking or containing too many buggy code entities can both affect the effectiveness of bug localization. Therefore, the first two-thirds of the frames of the stack trace are more helpful for bug localization.

6. Discussion

Software bug localization is widely recognized as one of the most time-consuming and costly activities in software debugging. As software systems become increasingly complex, bugs are almost inevitable. These bugs can significantly affect the performance and development of software, leading to considerable economic losses and even loss of life. In reality, large projects receive hundreds or even thousands of

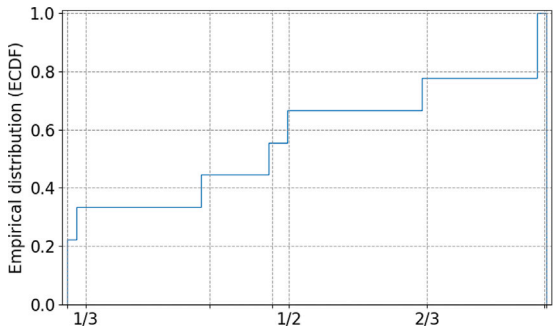

Fig. 6. The percentage of suspicious buggy code entities included in the first X out of Y stack traces.

bug reports on a daily basis. For instance, Mozilla Firefox receives an average of 307 new bug reports per day, as reported in Fan et al. (2018). Manual bug localization heavily relies on developers' expertise, and it can be time-consuming and costly. Existing IRBL techniques prioritize suspicious bug files by the relevance of the source code to the bug reports and provide developers with a guide for bug fixes. This technique generates a ranking list based on the source files, without localizing bugs on more granular program entities (e.g., statements, blocks, etc.) (Wong et al., 2016).

Initially, researchers attempted to directly use different retrieval models (e.g., VSM Chowdhury, 2010, LDA Blei et al., 2003) to improve the effectiveness of bug localization. Later, they found that the code base and bug reports were not composed of plain natural language but contained some domain-related structured information and features. By mining these features (mainly including version history Chen et al., 2008; Wen et al., 2019, similar reports Zhou et al., 2012; Saha et al., 2013; Youm et al., 2017, code structure Dilshener et al., 2018; Swe and Oo, 2018, and stack traces Schroter et al., 2010; Moreno et al., 2014; Wong et al., 2014), the accuracy of retrieval results can be improved. Among these, stack traces are one of the most valuable pieces of information for software debugging (Zimmermann et al., 2010). Stack traces show the active function calls and the position of program crashes when a system exception occurs, providing a ranked list of suspicious buggy files (Wang and Lo, 2016).

Moreno et al. (2014) combined text similarity and stack traces similarity to localize bugs. Based on the program dependency graph, they calculated the distance between each code element (class) and the corresponding code element in the stack traces. The closer the distance was, the more suspicious the code entity was. Wong et al. (2014) extracted all the file names and method names from the stack traces. Additionally, after obtaining a set of suspicious files, the files corresponding to the classes directly called in the methods corresponding to the suspicious files were added to the suspicious files set. Finally, the results were improved by improving the ranking of these files in the retrieval results. In addition, Wang and Lo (2016) proposed a stack trace score to measure the suspiciousness of each file that appears in the stack frames. They first extracted all file names in the stack traces in order and removed any duplicates. The reciprocal of each file's ranking was then used as a measure of its suspiciousness score concerning the bug report on the stack frames. Rahman and Roy (2018) extracted the code entities (class names and method names) from the stack traces and constructed a weighted graph based on the program order. Then, the PageRank algorithm was applied to calculate the weight of each code entity to reformulate the queries. Despite many efforts have been made in IRBL techniques, most existing approaches do not fully utilize suspicious buggy code entities in the stack traces and fail to achieve statement-level bug localization.

A Novel Information Retrieval Technique. Our approach differs from existing studies in that we first extract suspicious program entities from stack traces and parse buggy code entities using regular expressions. Next, we analyze the program dependencies of code entities in the suspicious file to trace back potential buggy code entities, which we refer to as "potential buggy code entities". We construct a list of suspicious code entities for each bug by traversing the program's dependencies in a depth-first traversal. To avoid the issue of an explosion of the fault space, we also set a defect detection scope, which we refer to as Scope@N. Finally, we adopt program simplification techniques to treat source code irrelevant to the bug as dead code and remove it from the suspicious file. We conducted our experiment on a Python dataset, but our methodology is not limited to a specific programming language. The main idea is to extract suspicious bug information from stack traces and to further explore potential buggy code entities by analyzing the dependency relationships of suspicious code. In future work, we would continue to extend related research to validate the effectiveness of this method in other languages and improve existing methods.

We also found that unlike the study by Schroter et al. (2010) and Wang and Lo (2016), the original stack trace information does not directly contribute to fine-grained bug localization. In our experimental dataset, nearly 90% of the bugs reported in the first three frames of the stack traces showed the positions of their buggy files. However, less than 30% of the total stack traces showed the actual buggy statements; This indicates that stack traces do not directly provide fine-grained bug localization information but can effectively show the position of buggy files. Furthermore, we observed that the buggy statements are more likely to appear in the first five frames of the stack traces than in the first frame. The first two-thirds of the suspicious program entities in the stack traces are more helpful for bug localization. Additionally, mining potentially buggy code entities and setting the scope of defect detection can effectively improve the effectiveness of bug localization using stack traces. Specifically, ranking the final text according to the suspiciousness of the suspicious buggy code entities can improve the accuracy of Top@K.

Statement-level Bug Localization. Techniques for localizing bugs at the statement level often involve obtaining execution information about the program and using dynamic and static analysis. According to Kochhar et al. (2016), developers prefer method-, statement-, and basic block-level bug localization techniques over file-level bug localization. Currently, the most advanced statement-level bug localization technique is Spectrum-Based Fault Localization (SBFL), which analyzes the execution behavior and runtime results of test cases to determine the actual fault localization in the tested program with finer granularity (Xie et al., 2011). However, this approach often requires more running time and resources, and the number and quality of test cases significantly impact bug localization performance. Existing studies show that Information Retrieval-Based Localization (IRBL) techniques perform as well as spectrum-based techniques achieved by low-cost text analysis. Unfortunately, current IRBL techniques have a coarse granularity for bug localization and can only localize bugs at the file or method level (Wang et al., 2015).

To implement fine-grained bug localization techniques, we evaluated the effectiveness of structured information for IRBL and explored how to exploit the full potential of structured information. Existing studies have shown that using conventional NLP techniques to process bug reports tends to generate a large amount of noisy data, which affects the query results (Rahman and Roy, 2018). Our strategy is identifying the relevant buggy code entities by analyzing the error paths and suspicious code snippets shown in the stack trace information. Then, by identifying the relevant suspicious buggy statements in the buggy file, thus implement bug localization. The stack trace records the active stack frames reported when a bug occurs. Its highly structured information helps us avoid noisy data generated by natural language text and contains program elements directly or indirectly related to the bug.

We evaluate the effectiveness of COPS by comparing the performance of $COPS_{RT}$ with the original stack trace information on bug

localization. Experiments show that $COPS_{RT}$ achieves better improvement compared to ST_{CS}. Namely, the improvements in Top@1,Top@5 and Top@10 are 99.2%,85.0% and 102.6%, respectively. For MAP@10 and MRR@10, the optimal improvement is also achieved for $COPS_{RT}$ compared to ST_{CS}. These results show that COPS significantly improves the effectiveness of stack trace information on IRBL.

Comparison with Existing Techniques. We compare COPS with HSFL and StackTrace (Wen et al., 2019; Zou et al., 2019), which is done to evaluate whether COPS can achieve the same performance as conventional SBFL techniques. The experiments show that COPS performs as well as the current state-of-the-art bug localization techniques. Regarding Top@K, COPS can not only perform well at Top@10, but the increase afterward is significantly better than the two benchmarks. This result shows that developers detecting the first 20 statements of $COPS_{RT}$ have at least a 66.5% probability of localizing an actual buggy statement.

We also evaluated the coverage of buggy statements in $COPS_{RT}$. We found that the assumption that the broader the code review scope, the better the bug localization performance is not always true. As we described in Section 3.4, in actual software development, most bug reports have complex descriptions, especially when their stack traces involve more suspicious files and methods. Recommending all the suspicious objects at once to the developers makes the code review work difficult and time-consuming. Therefore, it is necessary and significant to consider the appropriate scope of bug detection. Experiments show that COPS can achieve better localization performance when the Scope@N is set to 2, retrieving the first two elements of a data dependency.

Bug Localization for Python. Despite the widespread popularity of Python-based programs, there has been little research on automated debugging tools for Python (Widyasari et al., 2022); This is because the bug characteristics of Python programs are different from those of other popular programming languages like Java and C/C++. The current research results are not fully applicable to Python programs. For instance, the correct execution of Python programs is affected by the interpreter (Wang et al., 2022), and improper use of its dynamic features and type system can introduce bugs into programs (Åkerblom et al., 2014; Chen et al., 2020). Additionally, the convenience of the Python programming language has contributed to its popularity among newcomers. However, due to the limited experience of junior programmers, it can be challenging to localize program bugs (Cosman et al., 2020) accurately. Our proposed stack trace-based bug localization technique is highly practical as it uses stack traces as input without relying on test cases and executables. Furthermore, our approach is not affected by the quality of the bug report description and can even be directly applied to crash bugs encountered during development.

We observed the experimental results and found that the top three bug statements were: assignment statements with 2311, function call statements with 1199, and If statements with 1075. In the top five statements of $COPS_{RT}$ (i.e., Top@5), there were 651 assignment statements, 142 function call statements, and 142 If statements. This result shows that bugs in assignment statements are more easily localized in time than API calls and constraint checking. We believe this may be attributed to two reasons (Theisen et al., 2015; Gong et al., 2014). The first reason is that the causes of program exceptions may be related to some private APIs, and the suspicious objects related to bugs are difficult to directly reflect in the stack trace. The second reason is that the actual fault position of bugs caused by incorrect constraints is far from the position indicated in the stack traces with respect to program dependencies. Recent research proposed an approach for predicting the presence of crash bugs in stack traces (Gu et al., 2019). In future work, we could build on this research and fully explore the characteristics of Python program stack traces. By predicting whether bug-related information is present or absent in the stack trace, it can help developers to localize bugs quickly, as well as prioritize debugging efforts.

7. Related work

IR-based Bug Localization. Many advanced IRBL techniques use stack trace information since bug localization is likely in the error path. Moreno et al. (2014) applied stack traces to their technique (Lobster). They used stack traces from bug reports and program dependency graphs from software source code to find structurally similar code elements. Wang and Lo (2016) proposed the hypothesis that the closer a class is to the top of the stack trace (i.e., the bug's position), the more likely the file contains the bug. They designed a score to measure the suspiciousness of each file appearing in the stack trace. Rahman and Roy (2018) extracted the source code entities (i.e., class and method names) from the stack traces. They constructed a weight graph based on the execution order and applied the PageRank algorithm to calculate the weight of each source code entity to reformulate the query. Wang et al. (2015) showed that bug reports containing too many stack traces could underperform IRBL. Rath and Mäder (2019) analyzed the impact of structured information in IRBL and showed that source text generally improves bug localization performance, but stack traces have mixed results. Dilshener et al. (2018) propose an IRBL approach that does not require historical information and uses only structural information. Swe and Oo (2018) divided the code structure into class names, method names, and variable names to avoid the impact of large files on the results.

Compared to existing IRBL techniques, our approach achieves fine-grained bug localization, i.e., bug localization at the statement level. In addition, we do not use conventional NLP techniques to process the stack trace information, effectively avoiding the interference caused by a large amount of structured information. We collect the buggy code entities directly from the stack trace and apply context-aware program simplification techniques to identify the relevant buggy statements in suspicious files. Experiments show that our technology can help developers localize bugs efficiently.

Bug Localization in Other Languages. Since structured IRBL employs syntactic features of the source code, different types of program structures may affect the effectiveness of bug localization (Garnier and Garcia, 2016). The existing IRBL techniques are mainly applied to Java-based software projects, and less research has been done on software projects in other programming languages.

Saha et al. (2014) conducted a large-scale study to verify the effectiveness of IRBL for C systems. They created a large experimental dataset containing over 7500 bug reports and tested BLUiR on this dataset. Their experimental results showed that the IRBL was equally effective in five popular C software projects. However, some software features, such as program structural information, were less beneficial for C software projects. Garnier and Garcia (2016) evaluated the BLUiR (Saha et al., 2013), BLUiR+ and AmaLgam (Wang and Lo, 2014) techniques on 20 C# projects and found that IR-based bug localization still has limited effectiveness when applied in realistic settings. They also adapted AmaLgam to explicitly consider the full set of available C# program constructs, resulting in an average increase in effectiveness of 18%. In this paper, we first evaluate the effectiveness of IRBL techniques in Python. The experimental results show that the appropriate exploitation of structured information can effectively help developers to localize bugs.

8. Threats to validity

One potential threat to validity is the generalizability of the bug reports used in our evaluation, meaning that our experimental results may not generalize to other datasets. To reduce this threat, we employed 948 actual bug reports to evaluate our bug localization technique. To minimize experimental errors, we carefully reviewed the experimental data. Specifically, each bug report should have a small-sized (i.e., ten frames or less) stack trace, and at least one frame of the stack trace should show the actual buggy file.

The other threat is that COPS's current implementation can only localize buggy statements in a single file. COPS's suggestion ranking is also quite elementary, relying solely on the order of the stack frames. Although existing research has confirmed that in stack traces, the closer a suspicious file is to the bottom of the stack trace, the more likely it is to be buggy. For some lower-ranked buggy code entities, COPS may lower the ranking of the actual buggy statements.

9. Conclusion

Recent studies showed that IRBL techniques could perform as well as spectrum-based bug localization techniques, given fewer external dependencies and lower implementation costs. However, existing IRBL techniques have not yet achieved statement-level bug localization, and their effectiveness in Python-based programs is unknown due to the existing approach depending on syntactic features. In addition, bug report preprocessing approaches in existing IRBL techniques are not always helpful for bug localization as they might introduce a lot of noisy data due to excessively structured information. This paper proposes a novel technique – CPOS – that can localize bugs using the context-aware program simplification technique. COPS first mines potential buggy code entities in the buggy file by analyzing the stack trace information. Then, COPS applies context-aware program simplification techniques to identify relevant buggy statements in the files.

We evaluated the effectiveness of COPS in 948 bug reports in Python projects employing four widely used performance metrics. Compared with the original stack trace information within the bug report, COPS achieves the optimum improvement, that is, Top@10 improved by 102.6%, MAP@10 improved by 56.2%, and MRR@10 improved by 95.6%. Compared to the most advanced statement-level bug localization techniques, COPS performs as well as the two techniques and achieves a good coverage of buggy statements. Also, we found that the first two-thirds of the stack trace frames are more conducive to localizing buggy statements. Overall, COPS improves on the Top@K, MAP@10, and MRR@10 metrics and achieves good coverage of buggy statements.

In the future, we plan to design better ranking and retrieval algorithms specifically for COPS to improve its effectiveness in bug localization. We would mine more structural features to validate the effectiveness of stack traces against IRBL techniques through different combinations and novel retrieval algorithms. In addition, we plan to invite developers and draw on their expertise and practical experience to extract suspicious information from the limited information in bug reports, thus facilitating bug localization techniques at the statement level.

CRediT authorship contribution statement

Yilin Yang: Conceptualization, Methodology, Software, Data curation, Writing – original draft. **Ziyuan Wang:** Investigation, Writing – review & editing. **Zhenyu Chen:** Validation, Writing – review & editing. **Baowen Xu:** Supervision, Project administration, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

We thank the anonymous reviewers for their constructive comments. This research was partially funded by the the Science, Technology and Innovation Commission of Shenzhen Municipality (No. CJGJZD- 20200617103001003, 2021Szvup057).

References

Abreu, R., Zoeteweij, P., van Gemund, A.J., 2007. On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007). pp. 89–98. http://dx.doi.org/10.1109/TAIC.PART.2007.13.

Åkerblom, B., Stendahl, J., Tumlin, M., Wrigstad, T., 2014. Tracing dynamic features in python programs. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 292–295.

Akimova, E.N., Bersenev, A.Y., Deikov, A.A., Kobylkin, K.S., Konygin, A.V., Mezentsev, I.P., Misilov, V.E., 2021. PyTraceBugs: A large python code dataset for supervised machine learning in software defect prediction. In: 2021 28th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 141–151.

Artzi, S., Dolby, J., Tip, F., Pistoia, M., 2010a. Directed test generation for effective fault localization. In: Proceedings of the 19th International Symposium on Software Testing and Analysis. pp. 49–60.

Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D., 2010b. Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE Trans. Softw. Eng. 36 (4), 474–494.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., Zimmermann, T., 2008. What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 308–318.

Blei, D.M., Ng, A.Y., Jordan, M.I., 2003. Latent dirichlet allocation. J. Mach. Learn. Res. 3 (Jan), 993–1022.

Britton, T., Jeng, L., Carver, G., Cheak, P., Katzenellenbogen, T., 2013. Reversible Debugging Software. Tech. Rep., Judge Bus. School, Univ. Cambridge, Cambridge, UK, p. 1.

Campos, J., Riboira, A., Perez, A., Abreu, R., 2012. Gzoltar: an eclipse plug-in for testing and debugging. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 378–381.

Chen, I.-X., Jaygarl, H., Yang, C.-Z., Wu, P.-J., 2008. Information retrieval on bug locations by learning co-located bug report clusters. In: Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 801–802.

Chen, Z., Li, Y., Chen, B., Ma, W., Chen, L., Xu, B., 2020. An empirical study on dynamic typing related practices in python systems. In: Proceedings of the 28th International Conference on Program Comprehension. pp. 83–93.

Chen, K., Rajlich, V., 2000. Case study of feature location using dependence graph. In: Proceedings IWPC 2000. 8th International Workshop on Program Comprehension. IEEE, pp. 241–247.

Chowdhury, G.G., 2010. Introduction to Modern Information Retrieval. Facet publishing.

Cosman, B., Endres, M., Sakkas, G., Medvinsky, L., Yang, Y.-Y., Jhala, R., Chaudhuri, K., Weimer, W., 2020. Pablo: Helping novices debug python code through data-driven fault localization. In: Proceedings of the 51st ACM Technical Symposium on Computer Science Education. pp. 1047–1053.

Dallmeier, V., Zimmermann, T., Automatic extraction of bug localization benchmarks from history. In: Proc. Int'l Conf. on Automated Software Eng. Citeseer, pp. 433–436.

Dilshener, T., Wermelinger, M., Yu, Y., 2018. Locating bugs without looking back. Autom. Softw. Eng. 25 (3), 383–434.

Fan, Y., Xia, X., Lo, D., Hassan, A.E., 2018. Chaff from the wheat: Characterizing and determining valid bug reports. IEEE Trans. Softw. Eng. 46 (5), 495–525.

Ferenc, R., Gyimesi, P., Gyimesi, G., Tóth, Z., Gyimóthy, T., 2020. An automatically created novel bug dataset and its validation in bug prediction. J. Syst. Softw. 169, 110691.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) 9 (3), 319–349.

Gao, R., Wong, W.E., 2018. Mseer: an advanced technique for locating multiple bugs in parallel. In: Proceedings of the 40th International Conference on Software Engineering. pp. 1064–1064.

Garnier, M., Garcia, A., 2016. On the evaluation of structured information retrieval-based bug localization on 20 C# projects. In: Proceedings of the 30th Brazilian Symposium on Software Engineering. pp. 123–132.

Gong, L., Zhang, H., Seo, H., Kim, S., 2014. Locating crashing faults based on crash stack traces. arXiv preprint arXiv:1404.4100.

Gu, Y., Xuan, J., Zhang, H., Zhang, L., Fan, Q., Xie, X., Qian, T., 2019. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. J. Syst. Softw. 148, 88–104.

Gulzar, M.A., Wang, S., Kim, M., 2018. Bigsift: Automated debugging of big data analytics in data-intensive scalable computing. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 863–866.

Hammer, C., Snelting, G., 2009. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Secur. 8 (6), 399–422.

Hofstätter, S., Rekabsaz, N., Eickhoff, C., Hanbury, A., 2019. On the effect of low-frequency terms on neural-IR models. In: Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 1137–1140.

Hong, S., Lee, B., Kwak, T., Jeon, Y., Ko, B., Kim, Y., Kim, M., 2015. Mutation-based fault localization for real-world multilingual programs (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 464–475.

Huang, Q., Xia, X., Lo, D., 2019. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empir. Softw. Eng. 24 (5), 2823–2862.

Jiang, S., Li, W., Li, H., Zhang, Y., Zhang, H., Liu, Y., 2012. Fault localization for null pointer exception based on stack trace and program slicing. In: 2012 12th International Conference on Quality Software. IEEE, pp. 9–12.

Kochhar, P.S., Xia, X., Lo, D., Li, S., 2016. Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. pp. 165–176.

Koyuncu, A., Bissyandé, T.F., Kim, D., Liu, K., Klein, J., Monperrus, M., Traon, Y.L., 2019. D&c: A divide-and-conquer approach to ir-based bug localization. arXiv preprint arXiv:1902.02703.

Lawrie, D., 2012. Discussion of appropriate evaluation metrics. In: 1st Workshop on Text Analysis in Software Maintenance.

Le, T.-D.B., Oentaryo, R.J., Lo, D., 2015. Information retrieval and spectrum based bug localization: Better together. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 579–590.

Le Goues, C., Forrest, S., Weimer, W., 2013. Current challenges in automatic software repair. Softw. Qual. J. 21 (3), 421–443.

Li, X., Li, W., Zhang, Y., Zhang, L., 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 169–180.

Manning, C.D., 2008. Introduction to Information Retrieval. Syngress Publishing.

Moreno, L., Treadway, J.J., Marcus, A., Shen, W., 2014. On the use of stack traces to improve text retrieval-based bug localization. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 151–160.

Murali, V., Gross, L., Qian, R., Chandra, S., 2021. Industry-scale ir-based bug localization: A perspective from facebook. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, pp. 188–197.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, pp. 609–620.

Qi, Y., Mao, X., Lei, Y., Wang, C., 2013. Using automated program repair for evaluating the effectiveness of fault localization techniques. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. pp. 191–201.

Rahman, S., Rahman, M., Sakib, K., et al., 2016. An improved method level bug localization approach using minimized code space. In: International Conference on Evaluation of Novel Approaches to Software Engineering. Springer, pp. 179–200.

Rahman, M.M., Roy, C.K., 2017. STRICT: Information retrieval based search term identification for concept location. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 79–90.

Rahman, M.M., Roy, C.K., 2018. Improving IR-based bug localization with context-aware query reformulation. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 621–632.

Rao, S., Kak, A., 2011. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th Working Conference on Mining Software Repositories. pp. 43–52.

Rath, M., Lo, D., Mäder, P., 2018. Analyzing requirements and traceability information to improve bug localization. In: Proceedings of the 15th International Conference on Mining Software Repositories. pp. 442–453.

Rath, M., Mäder, P., 2019. Structured information in bug report descriptions—influence on IR-based bug localization and developers. Softw. Qual. J. 27 (3), 1315–1337.

Saha, R.K., Lawall, J., Khurshid, S., Perry, D.E., 2014. On the effectiveness of information retrieval based bug localization for c programs. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 161–170.

Saha, R.K., Lease, M., Khurshid, S., Perry, D.E., 2013. Improving bug localization using structured information retrieval. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 345–355.

Schroter, A., Schröter, A., Bettenburg, N., Premraj, R., 2010. Do stack traces help developers fix bugs? In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE, pp. 118–121.

Sisman, B., Kak, A.C., 2012. Incorporating version histories in information retrieval based bug localization. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR). IEEE, pp. 50–59.

Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., de Almeida Maia, M., 2018. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 130–140.

Strzalkowski, T., Vauthey, B., 1992. Information retrieval using robust natural language processing. In: 30th Annual Meeting of the Association for Computational Linguistics. pp. 104–111.

Swe, K.E.E., Oo, H.M., 2018. Bug localization approach using source code structure with different structure fields. In: 2018 IEEE 16th International Conference on Software Engineering Research, Management and Applications (SERA). IEEE, pp. 159–164.

Tantithamthavorn, C., Ihara, A., Matsumoto, K.-i., 2013. Using co-change histories to improve bug localization performance. In: 2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. IEEE, pp. 543–548.

Theisen, C., Herzig, K., Morrison, P., Murphy, B., Williams, L., 2015. Approximating attack surfaces with stack traces. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, pp. 199–208.

Voorhees, E.M., 2001. The TREC question answering track. Natural Lang. Eng. 7 (4), 361–378.

Wang, Z., Bu, D., Sun, A., Gou, S., Wang, Y., Chen, L., 2022. An empirical study on bugs in python interpreters. IEEE Trans. Reliab. 71 (2), 716–734.

Wang, S., Lo, D., 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In: Proceedings of the 22nd International Conference on Program Comprehension. pp. 53–63.

Wang, S., Lo, D., 2016. Amalgam+: Composing rich information sources for accurate bug localization. J. Softw.: Evol. Process 28 (10), 921–942.

Wang, Q., Parnin, C., Orso, A., 2015. Evaluating the usefulness of ir-based fault localization techniques. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 1–11.

Wen, M., Chen, J., Tian, Y., Wu, R., Hao, D., Han, S., Cheung, S.-C., 2019. Historical spectrum based fault localization. IEEE Trans. Softw. Eng. 47 (11), 2348–2368.

Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.-C., 2018. Context-aware patch generation for better automated program repair. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, pp. 1–11.

Wen, M., Wu, R., Cheung, S.-C., 2016. Locus: Locating bugs from software changes. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 262–273.

Widyasari, R., Haryono, S.A., Thung, F., Shi, J., Tan, C., Wee, F., Phan, J., Lo, D., 2022. On the influence of biases in bug localization: Evaluation and benchmark. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 128–139.

Widyasari, R., Sim, S.Q., Lok, C., Qi, H., Phan, J., Tay, Q., Tan, C., Wee, F., Tan, J.E., 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.

Wong, W.E., Debroy, V., Choi, B., 2010. A family of code coverage-based heuristics for effective fault localization. J. Syst. Softw. 83 (2), 188–208.

Wong, W.E., Debroy, V., Gao, R., Li, Y., 2013. The dstar method for effective software fault localization. IEEE Trans. Reliab. 63 (1), 290–308.

Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.

Wong, W.E., Shi, Y., Qi, Y., Golden, R., 2008. Using an RBF neural network to locate program bugs. In: 2008 19th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 27–36.

Wong, W.E., Tse, T., Handbook of software fault localization: foundations and advances. Wiley-IEEE Press.

Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H., 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 181–190.

Wotawa, F., Nica, M., Moraru, I., 2012. Automated debugging based on a constraint model of the program and a test case. J. Logic Algebr. Program. 81 (4), 390–407.

Wu, R., Zhang, H., Cheung, S.-C., Kim, S., 2014. Crashlocator: Locating crashing faults based on crash stacks. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 204–214.

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.

Xie, X., Wong, W.E., Chen, T.Y., Xu, B., 2011. Spectrum-based fault localization: Testing oracles are no longer mandatory. In: 2011 11th International Conference on Quality Software. IEEE, pp. 1–10.

Xin, Q., Reiss, S.P., 2017. Leveraging syntax-related code for automated program repair. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 660–670.

Yang, Y., Wang, Z., Chen, Z., Xu, B., 2022. Context-aware program simplification to improve information retrieval-based bug localization. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS). pp. 252–263. http://dx.doi.org/10.1109/QRS57517.2022.00035.

Youm, K.C., Ahn, J., Lee, E., 2017. Improved bug localization based on code change histories and bug reports. Inf. Softw. Technol. 82, 177–192.

Zeng, M., Wu, Y., Ye, Z., Xiong, Y., Zhang, X., Zhang, L., 2022. Fault localization via efficient probabilistic modeling of program semantics. In: Proceedings of the 44th International Conference on Software Engineering. pp. 958–969.
Zhang, W., Li, Z., Wang, Q., Li, J., 2019. FineLocator: A novel approach to method-level fine-grained bug localization by query expansion. Inf. Softw. Technol. 110, 121–135.
Zhang, M., Li, X., Zhang, L., Khurshid, S., 2017. Boosting spectrum-based fault localization using pagerank. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 261–272.
Zhou, J., Zhang, H., Lo, D., 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: 2012 34th International Conference on Software Engineering (ICSE). IEEE, pp. 14–24.
Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A., Weiss, C., 2010. What makes a good bug report? IEEE Trans. Softw. Eng. 36 (5), 618–643.
Zou, D., Liang, J., Xiong, Y., Ernst, M.D., Zhang, L., 2019. An empirical study of fault localization families and their combinations. IEEE Trans. Softw. Eng. 47 (2), 332–347.

Yilin Yang is a Ph.D candidate in the Software Institute, Nanjing University, under the supervision of Prof. Zhenyu Chen and Prof. Baowen Xu. Her research interest is software testing and automatic program debugging.

Ziyuan Wang received the B.S. degree in mathematics and the Ph. D. degree in computer science from Southeast University, Nanjing, China, in 2004 and 2009, respectively.,From 2009 to 2012, he worked as a Postdoctoral Researcher with the Department of Computer Science and Technology, Nanjing University, Nanjing, China. He is currently an Associate Professor with the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications, Nanjing, China. His research interests include software testing and program analysis.

Zhenyu Chen received the bachelor and Ph.D. degrees in mathematics from Nanjing University. He is currently a professor in Software Institute, Nanjing University. His research interests focus on software analysis and testing. He has more than 80 publications at major venues including TOSEM, TSE, ICSE, FSE, ISSTA, ASE, ICST, etc. He has served as a PC co-chair of QRS 2016, QSIC 2013, AST 2013, IWPD 2012, and a PC member of many international conferences. He has more than 30 patents and some have been used in Baidu, Alibaba, Huawei, etc. He is the founder of mooctest.net.

Baowen Xu was born in 1961. He is a professor at the department of Computer Science and Technology, Nanjing University. His research areas are programming languages, software engineering, concurrent software and web software.