Yang *et al. Cybersecurity* (2023) 6:17
https://doi.org/10.1186/s42400-023-00153-0

Cybersecurity

RESEARCH

Open Access

Jeu de mots paronomasia: a StackOverflow-driven bug discovery approach

Yi Yang[1,2*], Ying Li[1,2], Kai Chen[1,2*] and Jinghua Liu[1,2]

Abstract

Locating bug code snippets (short for BugCode) has been a complex problem throughout the history of software security, mainly because the constraints that define BugCode are obscure and hard to summarize. Previously, security analysts attempted to define such constraints manually (e.g., limiting buffer size to detect overflow), but were limited to the types of BugCode. Recent researchers address this problem by extracting constraints from program documentation, which shows the potential for API misuse. But for bugs beyond the scope of API misuse, such an approach becomes less effective since the corresponding constraints are not defined in documents, not to mention the programs without documentation In this paper, inspired by the fact that expert programmers often correct the BugCode on open forums such as StackOverflow, we design an approach to automatically extract knowledge from StackOverflow and leverage it to detect BugCode. As we all know, the contexts in StackOverflow come from ordinary developers. Their writing tends to be loosely organized and in various styles, which are more challenging to analyze than program documentation. To address the challenges, we design a custom tokenization approach to segment sentences and employ sentiment analysis to find the Controversial Sentences (CSs) that typically contain the constraints we need for code analysis. Then we use constituency parsing to extract knowledge from CSs, which helps locate BugCode. We evaluated our system on 41,144 comments from the questions tagged with Java and Android. The results show that our approach achieves 95.5% precision in discovering CSs. We have discovered 276 pieces of BugCode proved to be true through manual validation including an assigned CVE. 89.3% of the discovered bugs remained in the current version of answers, which are unknown to users.

Keywords Bug detection, Natural language processing, Open forum

Introduction

One of the root causes of information system security threats is vulnerability. It usually takes the form of a bug, flaw, or omission in software and hardware that exists as a result of poor design or poor implementation and can be exploited by the attacker Joshi et al. (2015). To address this issue, security analysts search for and eliminate vulnerabilities in software and hardware. Dynamic and static methods are common approaches. Fuzzing is a dynamic method for detecting program crashes by analyzing a large number of inputs, which may overcome missing specific vulnerability types such as authentication bypass caused by API misuse Lv et al. (2020). Several static methods are developed to address this problem (Ren et al. 2020, 2020; Zhou et al. 2017), including extracting API-related rules/constraints from source code and official documents. Several reasons may lead to limited vulnerability types: (1) implicit API usage description resulting in missing part of the API-related bugs Lv et al. (2020; 2) incomplete API-related description in the official document, which yields missing specific types of bugs which

*Correspondence:
Yi Yang
yangyi@iie.ac.cn
Kai Chen
chenkai@iie.ac.cn
[1] SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

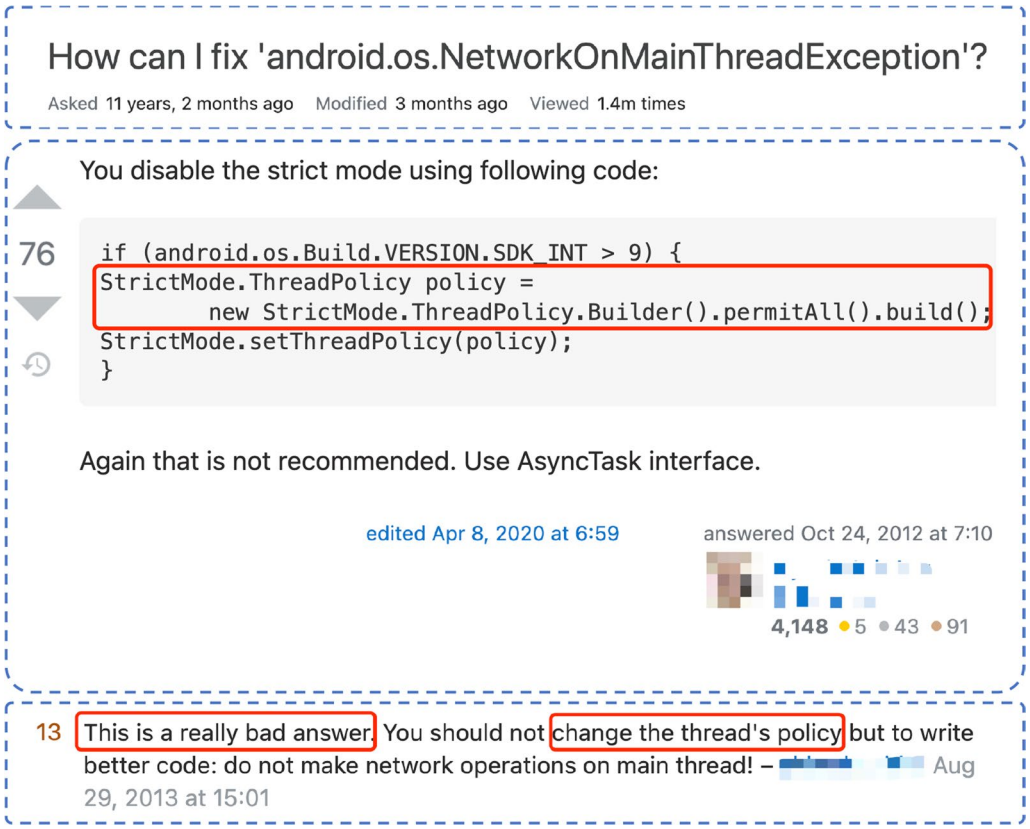

Fig. 1 The example case of StackOverflow

cannot be discovered merely through a document; (3) for other types of bugs such as API confusion, merely using the information in the documents is not enough since the documents rarely compare two similar APIs.

Finding bug in StackOverflow: challenges Natural Language Processing techniques have advanced in recent years. Unlike traditional research on official API documentation, we innovate on the data source onto StackOverflow, an open-source forum for developers discussing real-world practices. Our goal is to unearth knowledge buried in billions of question threads in order to locate bug code snippets that may influence real development. During the software development cycle, software engineers are expected to use official documents, GitHub, and open-source forums to solve problems. Due to the diverse API-usage practices and knowledge of errors in reality which are not included in official documents, people are more likely to use open-source forums like StackOverflow, to solve problems when faced. Shared knowledge proved its important role in developing software when facing bugs. In an open-source forum, for example, there will be multiple answers under one question thread, and users are free to show their opinions or suggestions under each answer directly. Figure 1 illustrates the complete context in the question page. We assume the vote-up of each answer indicates the current answer is followed by some of the developers in real practice and the other users may express their attitude towards this answer and point out in the comment if there exists a bug. We aim to explore if there exist security issues on shared knowledge, which may be adopted to affect the development. In Fig. 1, it is easy for users to learn that using method *"StrictMode.ThreadPolicy()"* according to the sentence *"You should not change the thread's policy"* in the comment is not recommended. While it is hard to do it automatically merely by analyzing documentation.

However, it is highly challenging to extract rules from open-source forums for bug detection due to the following reasons. **(C1) Loosely organized context.** Writings on StackOverflow vary a lot, and no patterns or keywords exist to guide the discovery of incorrect code descriptions. For example, in Fig. 1, the sentence *This is a really bad answer.* indicates the current answer is controversial, and the sentence "You should not change the thread's policy..." describes the incorrect operation in the answer. How to locate the sentences describing incorrect operations among numerous comments is the first challenge.

(C2) Hard to locate bugs in code. The descriptions of erroneous operations are in natural language. They usually do not include method names for direct comparison. How to detect bugs in the answer with the guidance of the description is another challenge. In Fig. 1, the description extracted in the comment is *"change the thread policy"* with which we should locate the piece of code using *StrictMode.ThreadPolicy()*. It is not straightforward to locate the buggy code from the writings.

The new discovery approach We design a new bug-discovery approach to exploit the hidden knowledge in discussions to discover bugs on StackOverflow and measure security issues in open-source software. Our approach is capable of conquering the several obstacles above by discovering the *Controversial Sentences*, locating the *Bug Code Context*, and discovering the *Bug Code Snippet* in the *Potential Controversial Answers*.

Based on our observation, though there are billions of APIs or method names that contain error codes and are also mentioned in the comments, it is still very hard to locate the wrongly-used APIs automatically due to the complicated writing styles on StackOverflow. Fortunately, we find that when criticizing errors in the answers, people tend to express negativity to get attention. For example, the sentence *This is a really bad answer* is recognized with a strong negative attitude indicating that there must exist some errors in the answer and the developers should give up following this solution. For sentences expressing negativity, we name them controversial sentences (CS for short), as they point out the controversy in the answer. To discover CSs, we perform a customized sentence tokenization of the comments since the proportion of attitude words may influence the sentiment analysis. After that, CSs are automatically located through sentiment analysis from the discussions in loosely-organized natural language. Then, considering that the clauses may contain the Bug Code Context (BugText for short) that contains the buggy code, we further extract verb phrases (VP) and noun phrases (NP) through constituency parsing and execute another round of sentiment analysis to accurately locate the phrases describing the bugs. Noticing the large gap between the phrases in natural languages and the buggy code in programming languages, we choose keywords that characterize bugs (called *anchors*) and utilize these anchors to identify the Bug Code Snippet in answers.

Findings In this work, we leverage sentiment analysis, constituency parsing and dependency parsing to implement our discovery approach on question threads from StackOverflow focusing on java and android, and further evaluate the effectiveness compared with the other state-of-the-art tools on CS discovery and bug discovery. Our method shows an average accuracy of 95.5% on CS discovery and 92.0% on bug discovery.

Through our discovery of 1,000 threads, our approach automatically discovers 1088 pieces of BugCode. After manual validation on randomly selected 300 pieces of BugCode, we find that 276 of them (92.0%) are real bugs, many of them are security-related bugs including program crash, memory leak, stack overflow and null dereference, etc.[1] The erroneous writings causing these problems are mostly incomplete answers, incorrect answers, out-of-date answers and potentially incorrect answers, etc. To the best of our knowledge, this is the first bug-discovery approach for uncovering hidden knowledge in StackOverflow discussions, a step forward to automatically exploiting information extracted from expert knowledge to find bugs. We also find that even the answers with 183 upvotes under 439k view counts remained incorrect for nearly 5 years and resulted in a vulnerability CVE (2022). This vulnerability allows an attacker with access to the machine to potentially access data in a temporary directory created by this deprecated API. One of the answers contains the bug which has already been reported before, however, the developers still believed in their own experience rather than public bug report Bug (2022). After manual confirmation, we find that only 10.7% of the potentially controversial answers in SO which contain BugCode according to our approach are revised by the author, which means that 89.3% answers still contain bugs in the current version, continuing to mislead future readers.

Contributions We summarize our main contributions as follows.

- *New approach for bug discovery* We develop a new bug discovery approach that leverages loosely organized textual descriptions in StackOverflow. To extract useful knowledge from comments written in natural languages, we design a set of new techniques, such as anchor-based bug code discovery. To the best of our knowledge, we are the first to leverage discussions on StackOverflow to help detect bugs.
- *New findings* Working on 1000 threads from StackOverflow, our approach discovered 1088 pieces of BugCode through automatic processing discussions on StackOverflow. We also find that among the potentially controversial answers, 89.3% of them still contain errors, which may mislead readers to write flawed code.

[1] https://anonymous.4open.science/r/SO-bugs-discovered-BF8A/

Background

Vulnerability detection

The existing methods of vulnerability detection usually include code-based and text-based ones.

Code-based vulnerability detection method The code-based detection methods, classified as dynamic methods, static methods and the ones combining both static and dynamic methods, are to detect vulnerability directly on the target program. Fuzzing is one of the dynamic methods through feeding abnormal or random inputs into the program to trigger unexpected behaviours, which can be mainly classified as coverage-based fuzzing and directed fuzzing (Böhme et al. 2017, Chen et al. 2018, Li et al. 2018, Zong et al. 2020). Although static methods are supposed to figure out the suspicious part of the program without running the code, dynamic methods could discover the exception which could be the potential root cause for the bug, including potential security violation, run-time error and logic inconsistency (Ayewah et al. 2008). And also the detection methods combining both the dynamic analysis and static analysis can not only detect the vulnerabilities which can only be detected by static analysis but also detect the vulnerabilities when implementing dynamic analysis (Aggarwal 2006; Amin et al. 2019; Kiss et al. 2015). Such methods combine the advantages of both methods, but are also limited by the current techniques and are in need of further development.

Text-based vulnerability detection method With the increasing development of Natural Language Processing techniques and the abundant context information, using official documentation for vulnerability detection is becoming popular (Chen et al. 2019; Zhou et al. 2017; Ren et al. 2020, 2020). Software engineers are supposed to follow the official documentation to properly use APIs. However, knowledge asymmetry exists between the writers and the users of the documentation, including implicit API usage and misunderstanding of the API constraints, which may result in bugs during software development. The majority of the research is analyzing API-related context to detect bugs. We also unveil the hidden knowledge in discussions on an open-source forum which does not focus merely on API.

Natural language processing

To process the comments from StackOverflow, we leveraged several NLP techniques, as illustrated here.

Sentiment analysis Sentiment analysis is one of the NLP techniques, designed to output a score indicating the given sentence's attitudes, opinions or emotions (Medhat et al. 2014). The major task of sentiment analysis is to classify the input sentences with a score indicating a negative attitude, positive attitude or neutral. In our research, we utilize the existing open-source tool Google Language API (GoogleNLP 2022) to process the comments on StackOverflow to select the sentences describing the controversy of the answers and the verb phrases illustrating the incorrect operations. We also utilize sentiment analysis to classify the CSs to narrow the scope of PCAs and help locate bug code snippets according to the negative sentences and verb phrases.

Constituency parsing Constituency parsing is a sentence-level syntactic analysis, aiming to unveil the internal relations between constituents of the sentence (Zhang 2020). Noun phrases, verb phrases and the relations between them are shown in the constituent parsing tree. We utilize the constituency parser from AllenNLP (Gardner et al. 2017) to help extract each constituent of the sentence to locate the VPs under answers.

Dependency parsing Dependency parsing is designed for both syntactic analysis and semantic analysis (Chen and Manning 2014). It describes the binary relations between each word in a sentence, which is clearly illustrated according to the dependency tree. In our work, we utilize the dependency parser to locate the noun words related to a given VP, which turn out to be the anchors in finding BugCode.

Design

In this section, we elaborate on the observation and design of our new approach for automatically discovering buggy code through analysis on StackOverflow. We first give an overview of the whole design, its measurement pipeline and an example that shows how it works, and then move to how its individual component is designed.

Overview

Discovery approach We aim to dig out the knowledge buried in comments to locate the bug code snippets in the answer. Figure 2 illustrates the design of our discovery approach: Controversial Sentence(CS) Discovery, Bug Code Context(BugText) Location and Bug Code Snippet(BugCode) Discovery. For discovered bug code snippets, we manually validate them with third-party resources, the process of which is illustrated in Sect. 6 by introducing a case study. During the workflow, we merely use the comments under each answer on StackOverflow (2022) as our input dataset.

StackOverflow answers are made up of code snippets and natural language descriptions. CSs are automatically located in Controversial Sentence Discovery via sentiment analysis from comments under answers in loosely-organized natural language. The Potential Controversial Answers are confirmed after being filtered by customized rules and containing at least one CS in the comments below the answer. Then, using syntactic analysis

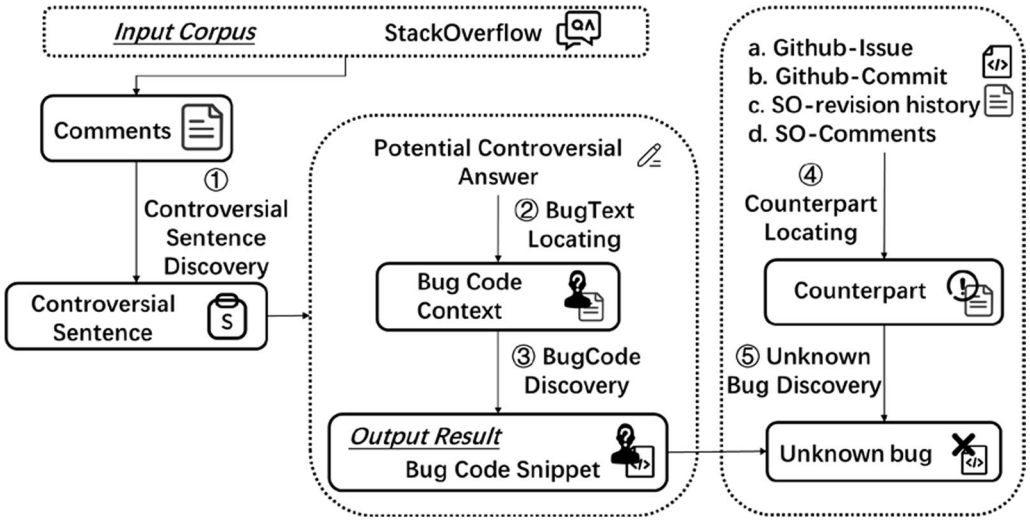

Fig. 2 The workflow of our discovery approach

and linguistic parsing, BugText is located among the sentences from comments under each answer. Following the extraction of the anchors from BugText, our approach can detect Bug Code Snippets in the answers by matching the anchors with the code snippets in the given answers. Finally, we manually validate the discovered BugCode for real bugs using third-party resources.

Example Figure 3 illustrates the workflow of our approach with an example. The comment illustrates the incorrect API selection without mentioning the method name, which is "This is a really bad answer. You should not change the thread's policy...". The comment delivers the information that the answer may be controversial. Note that StackOverflow is very different from the documentation that usually introduces API usage. Previous studies (Lv et al. 2020; Pandita et al. 2016; Zhong et al. 2011) can extract rules from the documents. However, comments in StackOverflow tend to be more loosely organized, which renders previously studied methods ineffective. In other words, generating rules or fixed patterns related to specific APIs from StackOverflow seem impossible. So we put in the effort to unveil the expertise buried in discussions and exploit them to contribute to bug discovery on StackOverflow.

Our approach first automatically identifies CS, which is "You should not change the thread's policy.", indicating the answer is incorrect with a negative attitude. Then, we extract the VP ("change the thread's policy") from the sentence. "change the thread's policy" is confirmed as the BugText according to our approach by describing the bug code operation. Note that merely using BugText to locate BugCode in the answer is not straightforward. Our approach automatically discovers the anchors "change", "thread's", "policy", and locates the piece of BugCode *StrictMode.ThreadPolicy policy = new StrictMode.ThreadPolicy.Builder(). permitAll().build(); StrictMode.setThreadPolicy(policy);* which contains the anchors. We further confirm that the discovered BugCode is a real bug according to the Github Commit.[2]

Controversial sentence discovery

Unlike the API documentation designed to introduce usages of APIs, loosely-organized context on open-source forums such as StackOverflow is hard to be processed with fixed patterns or rules. Therefore, after manually identifying and analyzing a set of sentences, we observe that CSs are in a strong negative attitude, which could be utilized to discover such sentences. We find these sentences can be classified into two types: (1) Bare CS: this type of CS only includes a negative attitude towards the answer without describing the bug code snippet of the answer. From this kind of CS, users could only sense that the answer may be incorrect. However, it is hard for users to tell which part of the code snippet is buggy. (2) CS with objects: this type of CS describes the details of the bug code snippet in the answers. From this kind of CS, users could explicitly locate the BugCode of the answer. In Table 1, we illustrate each category with examples.

Due to the huge amount of answers on StackOverflow, it is hard to locate the potential controversial answers. Therefore, we pre-process the attributes related to both the comments and the answers to narrow the scope of

[2] https://github.com/avalax/FitBuddy/commit/065e035c3a5592f08ffb409de312e8bee1048779

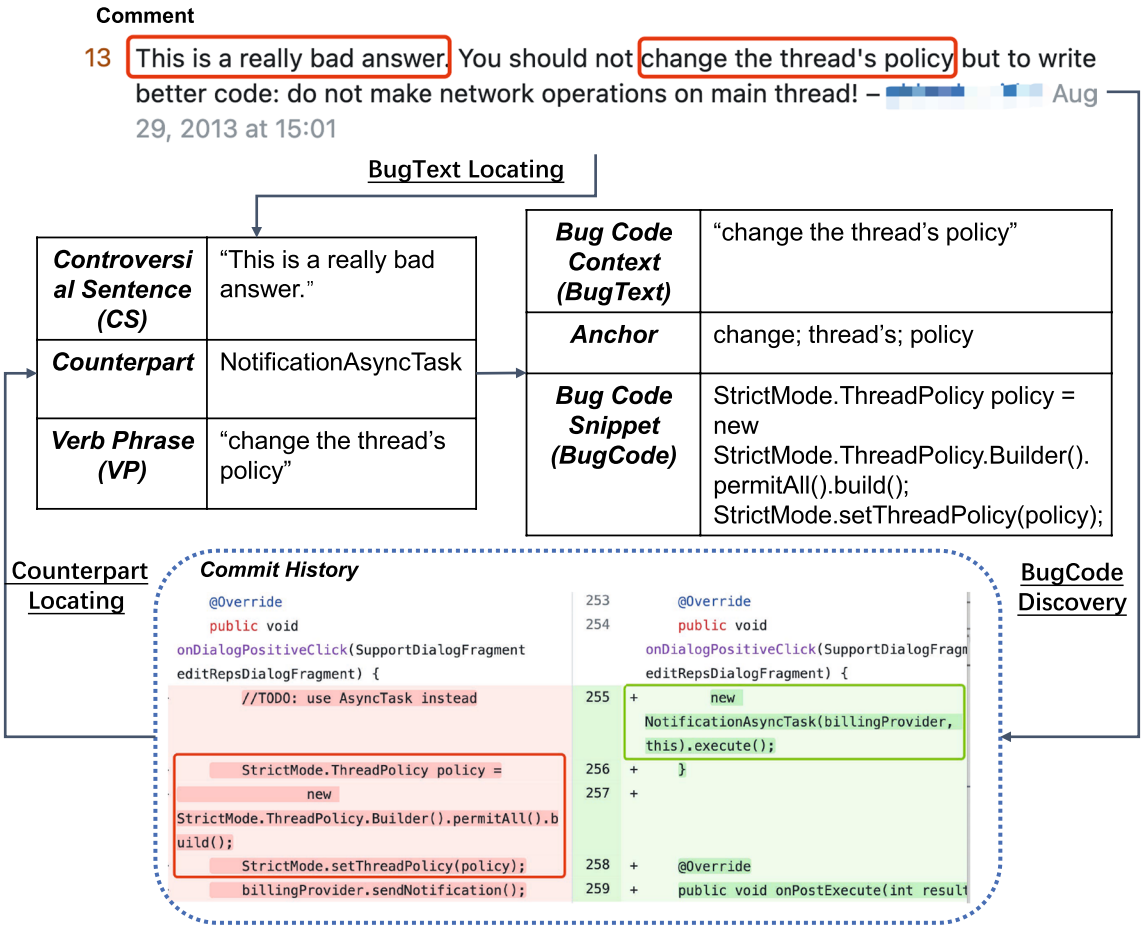

Fig. 3 The example of knowledge extracted from StackOverflow

Table 1 Category of controversial sentence

Category	Description	Example
a. Bare CS	Only expressing the negative attitude to the answer.	"This is a really bad answer."
b. CS with objects	Illustrating the problematic operations in the answer with negative attitude.	"Directly calling setSeed is very dangerous."

PCAs to be processed. We design several customized constraints from four dimensions to filter comments: (a) $Votes_{answer}$ should be over 0: the vote-up for each answer indicates the current answer is useful or appropriate to the community (sovoteup 2022) and is believed to be followed by other developers in practice, thus we drop all the answers with votes under 0, which means this answer is probably not used by actual developers. (b) $Votes_{comment}$ should be top 2: the vote number of each comment indicates the level of agreement by other users. In order to achieve higher precision and recall in extracting CS, we select the comments whose vote number ranks top 2 among all the comments under the given answer. (c) $Author_{comment}$ is not the same as $Author_{answer}$: we need to select the comment which is criticized by other users, which means the author of the comment should not be the same as the author of the answer. (d) "@" is not supposed to appear in the comment: users are supposed to use "@" to reply to other users on StackOverflow. For such comments starting with "@", little has been found useful for CS discovery. Therefore, we drop the comments starting with "@" to achieve higher accuracy.

After the preparation of the dataset, we utilize the existing sentiment analysis tool to score each sentence

extracted from the questions tagged with [java] and [android]. In Sect. 4, we discuss the selection process of the threshold. In the evaluation, we show our comparison results which achieve a higher precision rate.

Bug code context locating

In this part, we aim to locate the Bug Code Context in the comment, which describes the bug code through natural language in the answer and belongs to "CS with objects" in Table 1. After extracting the CSs, we confirm the scope of Potential Controversial Answers (PCAs) and the comments which constitute the dataset for further BugText locating.

After analyzing the sentences identified in Sect. 3.2, we chose the comments containing both negative attitudes and bug code descriptions to help locate BugText. Negative attitudes indicate the current answer is incorrect from the users' aspect, and bug code descriptions are written in natural language. We define 2 types of CSs to cover the maximum scope of PCAs. After that, we need to extract BugText, which describes the bug code snippet of the given PCA. We propose a solution that combined constituency parsing, sentiment analysis and dependency parsing altogether to locate the BugText in the comments under PCA. Details will be illustrated in the following.

Sentiment-based BugText locating We observe that BugText comes along with a verb phrase indicating a negative attitude. For example, "Directly calling setSeed" is the BugText in "Directly calling setSeed is very dangerous.", and "is very dangerous" expresses a relatively strong negative attitude. Therefore, we extracted VPs from each sentence by utilizing constituency parsing(see Fig. 4) and did sentiment analysis of each extracted VP. Sampling on the classified VPs, we set the threshold to select the qualified VPs and the selection process is illustrated in Sect. 4. However, not all the qualified sentences contain specific incorrect operations. For example, the sentence "This test is flawed." contains a negative attitude and the NP extracted is "This test". The NP does not describe any specific incorrect operations but a demonstrative pronoun referring to the incorrect answer. Therefore, we construct a keyword list containing the demonstrative pronouns (such as "it","that","this",etc) to filter out the NPs which describe specific incorrect operations.

We find that a sentence with a detailed description may affect the results of sentiment analysis. For example, "This test is flawed." is correctly classified as CS after being separated from the sentence *"This test is flawed as it runs all 3 tests in the same JVM instance."*. However, the longer sentence can not be correctly classified by sentiment analysis. The context directly referenced using "`" or the inline code examples using <code> </code> do not need to be analyzed, thus we directly extract the context referenced or tagged. For example, "`double`" from *"the entire detour with* <code>double</code> *"* and *"provided the implementation doesn't change"* from the sentence *"'provided the implementation doesn't change' - and there's the problem."*.

By utilizing the qualified VPs on the former method, we could locate BugText, shown as two types of forms: (1) "modal verb. + negative words + verb phrase". This type of BugText always appears after a negative word (such as "not","never", etc) and is followed by the bug code snippet description. For example, the extracted VP from the sentence "You cannot change the size of the list!" is *"cannot change the size of the list"*, which is classified as qualified VP after analysis and the verb phrase after the negative word "not" indicating the error in the answer. (2) "noun phrase + verb phrase". For example, the qualified VP extracted from the sentence "the entire detour with <code> double <code> values makes no sense at all" is *"makes no sense at all"*, after a fine-grained sentence tokenization and the BugText is *"the entire detour with <code>double</code>"*. In this way, no *modal verb.* is detected in the sentence. Therefore we try to find the NP in relation to the given VP after dependency parsing, according to the dependency tree. Besides the above, there rest types of BugText in the dataset, which are referenced directly by the comment. For example, *"'provided the implementation does not change' - and there is the problem."* shows that the referenced context *"provided the implementation does not change"* directly rewriting from the answer is incorrect. We drop such cases since they are pointing out the incorrect part written in natural language rather than code.

Bug code Snippet discovery

The BugText extracted from Sect. 3.3 is used to find BugCode in this stage. Different from BugText, BugCode is in the form of a programming language. We come up with a solution by extracting *anchors* to help locate BugCode since anchors could connect BugText to BugCode. In this stage, we split the procedure into two steps: Anchor Discovery and Bug Code Snippet Locating.

Anchor discovery In this part, we aim to extract anchors which could be used to locate BugCode. We define the anchors as keywords which can be found in BugText, describing the operation of BugCode. The example in Fig. 4 shows that the qualified BugText is either a noun phrase or a verb phrase according to constituency parsing. We tagged the part-of-speech of each BugText to extract the noun words accompanied by its modifier, and the process is shown in Fig. 5. On the left, after PoS tagging for each word in NP, `detour` and <code>double<code> values is tagged `NN`. from the tagger, since `double` is the inline code examples from the

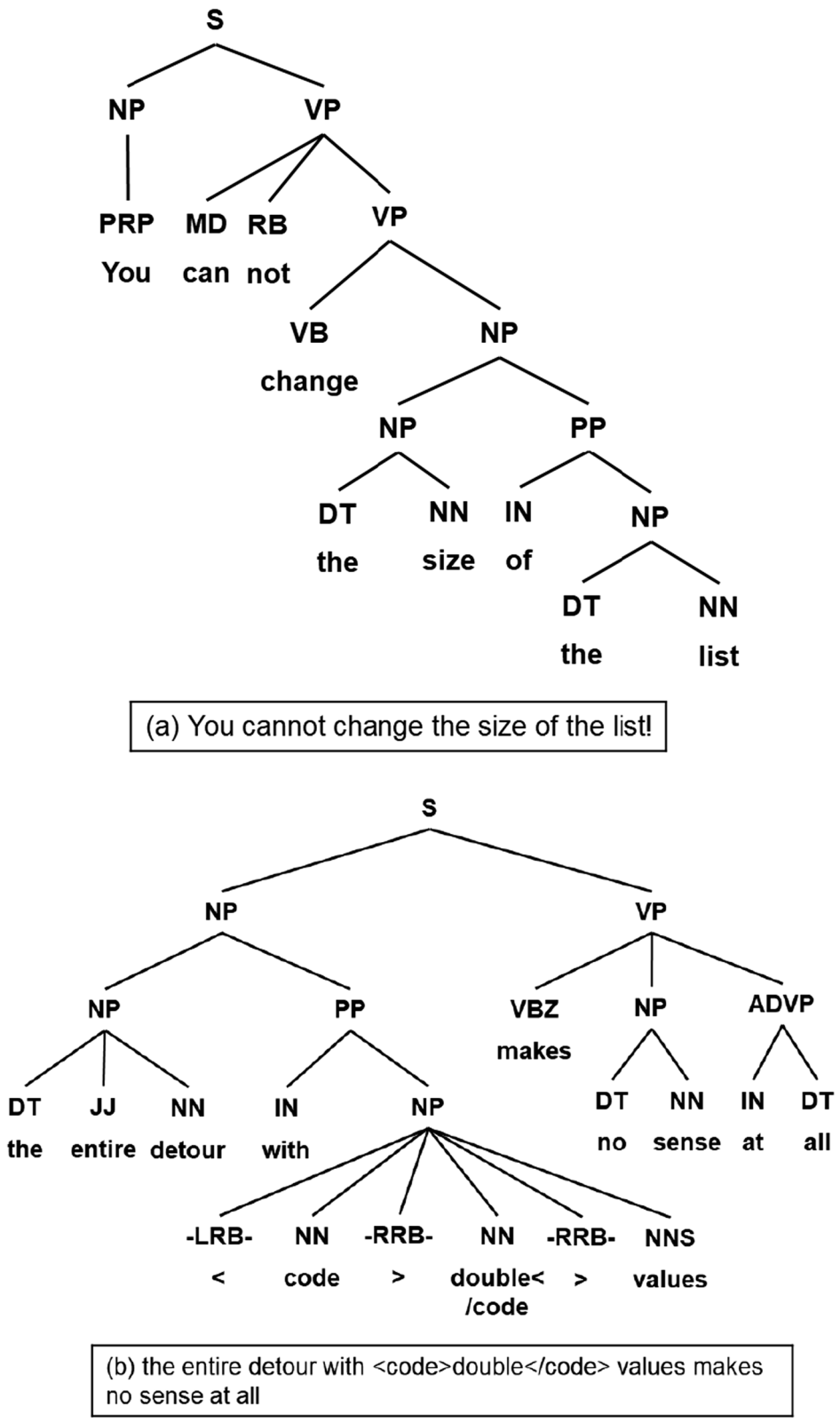

Fig. 4 The example of BugText

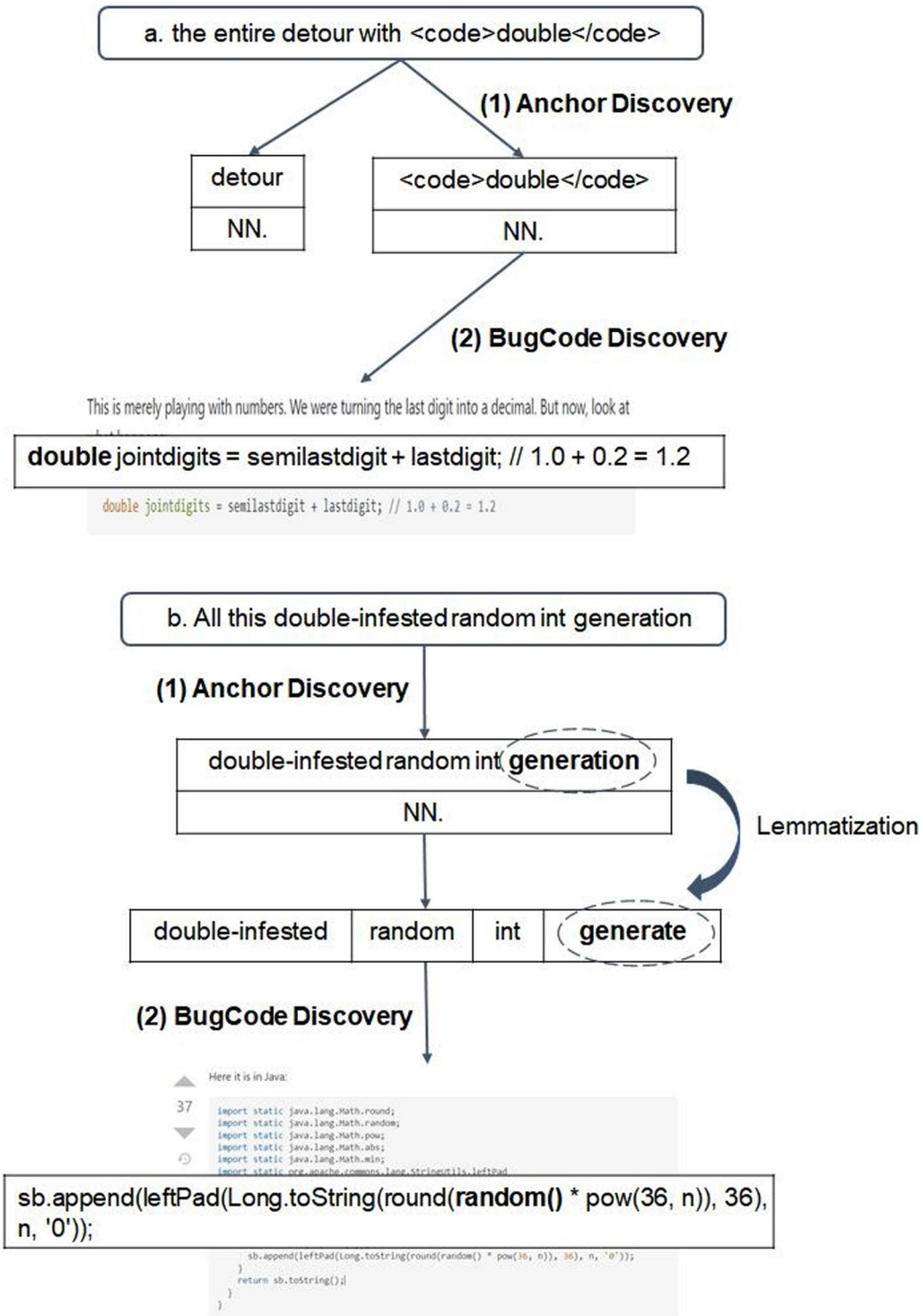

Fig. 5 The discovery process of Anchor and BugCode

source data, we directly extract "`detour`" and "`double`" as the anchors to discover BugCode.

Bug code matching In this stage, we utilize the anchors to locate BugCode, the process is to discover the piece of code snippets containing any forms of the anchors. We first did lemmatization to acquire the original verb form of the anchors, and the anchors which do not have verb form remained in the noun form. For example, the

anchor of *"The assertion to check for overflow is wrong."* is "assertion", it should be lemmatized to "assert" to match the BugCode "assert(Integer.MAX_VALUE -a>=b:)" in the answer.

Two examples are shown in Fig. 5 to describe the process in detail. For sentence-a, the anchors are "detour" and "<code> double </code>", thus we use them as keywords to locate the BugCode shown in the answer which is *double jointdigits = semilastdigit + lastdigit; 1.0 + 0.2 = 1.2*. For sentence-b, the extracted anchors are "double-infested", "random", "int" and "generation", using each anchor to match the line of code snippets which contains the most anchors, we finally find the BugCode is *sb.append(leftPad(Long.toString(round(random() *pow(36,n)),36),n,'0'));*. For multiple lines of code containing the same anchor, we manually validate them. Our work discovered 1,088 pieces of BugCode in total. After manually checking on the 300 random pieces of them, we find 276 pieces of BugCode are true. 89.3% of them remain incorrect in the current version of answers, still unknown to SO users and may mislead users to write flawed code

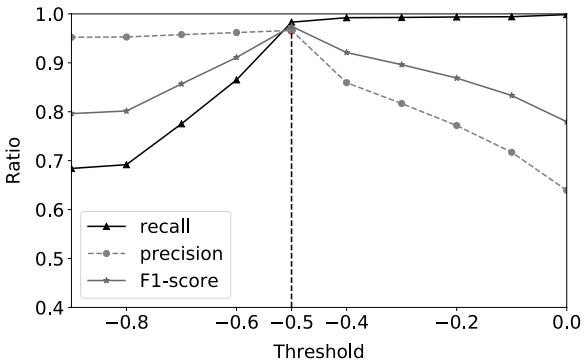

Fig. 6 The Performance of CS Discovery under different thresholds

Table 2 Statistics for selecting comments

Range	Precision	Recall	F1 score
Top 1 comments	0.98	0.78	0.87
Top 2 comments	0.96	0.89	0.92
Top 3 comments	0.86	0.96	0.91

Implementation

Controversial sentence discovery In this stage, we designed a sentiment-based approach to discover CSs. First, we built the dataset for our research. Due to the special mechanism of StackExchange API Stackexchange (2022), people are not allowed to crawl the questions directly from StackOverflow. We queried a question_id first and then got the content under each question, including several answers and comments stored in JSON format. Since our goal is to detect bugs in the answers, therefore we delete the answers without code segments. To confirm the selected comments are accurate, we performed a statistical analysis of the top 3 comments under each answer. Table 2 shows the statistical analysis of the comments in terms of the number of votes. We selected the top 2 comments as our dataset as it has the highest F1 score. For fine-grained sentence tokenization, we utilized NLTK (2022) with custom constraints in order to acquire accurate results after sentiment analysis: a. Deleting the context tagged with "" and "()"; b. Segmenting subordinate clauses and using "," to split complex sentences into simple ones, to fine-grained tokenize sentences. In the following, we deployed Google Natural Language API (2022) to analyze our pre-processed sentences. After doing sentiment analysis, we used 1,000 random sentences to figure out their accuracy, recall, and F1 score. We drew the broken line graph with the results. The point where the lines meet is supposed to be the right point of balance for the indicators. In Fig. 6, when the threshold is set to −0.5, after which the precision and F1 score are going down and the growth rate for the recall is prominent slowly. Therefore, we set α to −0.5 to achieve better performance for our approach.

Bug code context locating In this stage, we implement four steps to locate the BugText among the comments. We first utilized NLTK (2022) to tokenize each comment into sentences, reserving the detailed description of BugText, including subordinate clauses and context rewritten directly. Then, we leveraged the Constituency Parser (2022) from AllenNLP (2017) to extract verb phrases and fed them to the Google Natural Language API (2022) to implement sentiment analysis. After analyzing randomly selected 1000 pieces of sentences, Fig. 7 shows the growth rate is limited while the threshold is bigger than −0.5 and the precision and F1 score reached the peak in the meanwhile. Therefore, we set β to −0.5 for our approach. At the final stage, we proposed two solutions to locate BugText. For qualified VPs containing negative words (such as "not", "never", etc) following a modal verb, we directly extracted the phrase (always in the form of VP) after the negative word as the BugText. For other situations, we utilized the dependency parser from SpaCy (2022) to build a dependency tree to extract the BugText related to the qualified VPs.

Bug code Snippet discovery

In this part, we utilized different tools for these two steps. First, we utilized the NLTK (2022) to tokenize

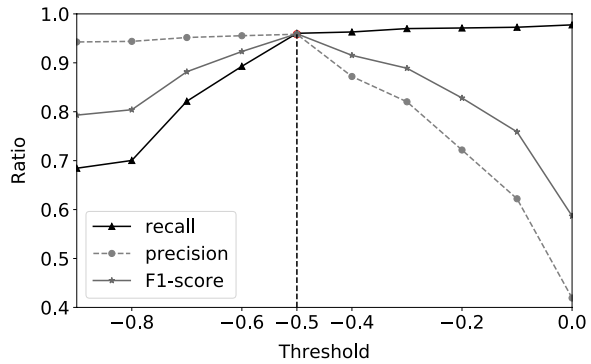

Fig. 7 The performance of VP extraction under different thresholds

each word from the BugText, tagging the Part-of-Speech of them with StanfordNLP (2022) to recognize the noun words and their modifiers. Besides this, the information referenced directly and the inline code examples using "<code> </code>" from the source data are extracted directly to serve as anchors. For example, in Fig. 5 "`double`" will be extracted from "<code>double</code>" as the anchor to locate the BugCode directly. In the following, we did lemmatization with NLTK (2022) to transform the forms of anchors into the original verb form. For example, in Fig. 5 one of the anchors in *"double-infested random int generation"* is "`generation`" and after lemmatization, we get "`generate`" as the anchor. For all the anchors, we located the code snippets containing the most anchors to be the BugCode in our work. There may exist some cases in which anchors appear in multiple lines of code in the answer. For such situations, our approach will extract all the pieces of code snippets for manual validation. And the results show that our approach covers the majority of BugCode on StackOverflow.

Evaluation

In this section, we will introduce our experiment setting including the platform and dataset, and evaluate the bug discovery approach including the end-to-end effectiveness and the effectiveness of each stage. And we will illustrate the comparison of our individual stage with the state-of-the-art methods since there exists no similar end-to-end approach for comparison.

Experiment setting

Dataset To evaluate the process of bug discovery, we leverage 2 datasets for our approach. First, we will introduce the corpora of StackOverflow and the Ground_Truth dataset for our approach. Then, we will illustrate the dataset for each stage evaluation in Sect. 5.2.

- *Corpora of StackOverflow for experiment* Based on the statistics from StackOverflow (2022), the total number of question threads focusing on java and android ranked top one among all the tags on the forum. Thus we selected these question threads as our research objects which could be found through tags. 1000 question threads constitute the corpora of Stack Overflow for experiment and validation, with 21,713 answers, 41,144 comments, and 82,718 sentences in total.
- *Ground_truth dataset* To measure the feasibility of the design, we manually analyzed a set of known BugCode validated from the revision history of StackOverflow, which is confirmed by two authors within 3 weeks. For questions under [java] and [android], we manually checked 2000 sentences from 72 question threads beyond the 1000 threads for the experiment, with 502 answers and 848 comments. Detecting 249 pieces of Controversial Sentences distributed in 190 answers. 63 pieces of BugCode are confirmed in total.

Platform All the experiments in our study were conducted on the macOS with 2.3 GHz CPU, 16 GB memory and 256 GB hard drive.

Effectiveness

End-to-end effectiveness. In our experiment, we ran our discovery approach on randomly selected 1000 threads from StackOverflow tagged with [java] and [android] to show the performance. Our approach discovered 1088 pieces of BugCode in total. We randomly selected 300 pieces of BugCode for manual check and it took two researchers one week to validate the results. 276 pieces of BugCode were proved to be accurate with third-party validations with a precision rate of 92.02%, 24 cases are false positives.

We analyze the 24 false positives from two aspects: tools and contexts. First, both the sentiment analysis tool and the dependency parser used in Sect. 3.3 could introduce incorrect results. We found that 16 of 24 false positives were mainly introduced by the NLP tools. For example, the VP extracted from the sentence *"Somehow the branch prediction only has a 25% miss rate, how can it do better than 50% miss?"* is *"has a 25% miss rate"* and the VP was recognized as qualified after sentiment analysis. However, it did not contain a negative attitude, this VP would be misclassified and yielded a false positive in locating BugCode. Another reason is the complicated descriptions, they confused both the users and tools by including both positive attitudes while pointing out the incorrect part of the answers. For example, in comment *"+1 This I agree with. You should never return half-initialized objects."*, the first sentence shows a positive

Table 3 Accuracy of CS extractor

Tags	S#	CS	Non-CS	OUR WORK			S-HAN			Keywords		
				Acc	FNR	FPR	Acc	FNR	FPR	Acc	FNR	FPR
Java	1000	266	734	0.96	0.04	0.08	0.26	0.74	0.66	0.08	0.92	0.14
Android	1000	263	737	0.95	0.05	0.07	0.27	0.73	0.31	0.09	0.91	0.03
C++	1000	238	762	0.93	0.07	0.06	0.24	0.76	0.44	0.03	0.97	0.01

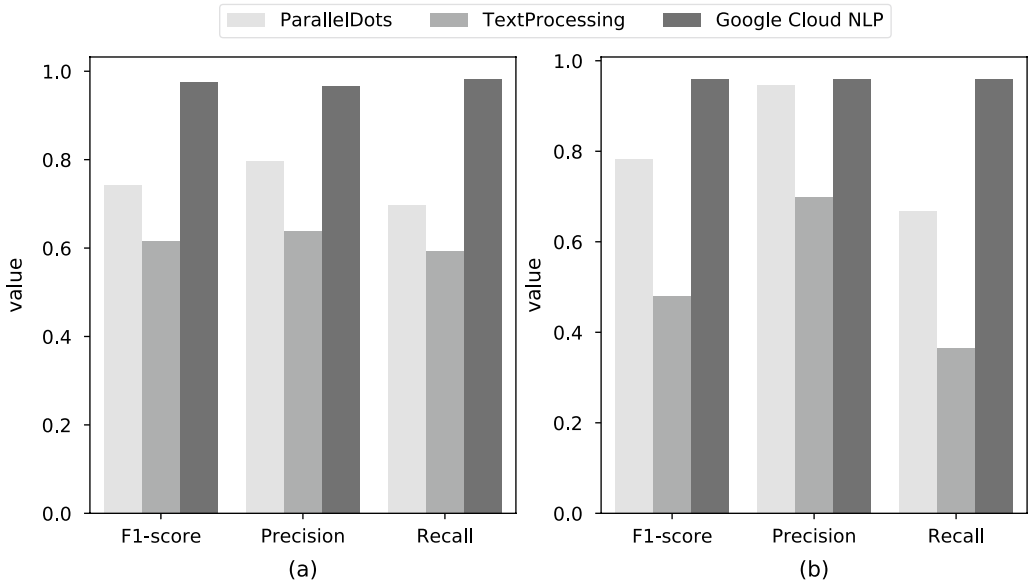

Fig. 8 The performance of different sentiment analysis tools for **a** CSs and **b** VPs

attitude, however, it is confusing for both the users and tools since it does not explicitly mention which object it agrees with. The tools cannot correctly classify the sentences since the comments' attitude is unclear.

Effectiveness of CS discovery We randomly selected 1000 sentences from questions under [java] and [android] to evaluate our approach. To prove the feasibility of our approach to discovering CSs on all the question threads on StackOverflow, we also included question threads tagged with [c++]. Our approach found 266, 263 and 238 CSs, respectively. The results show an average accuracy of 94.6% with a relatively low false negatives rate of 4% on [java] compared to evaluations on [android] and [c++] shown in Table 3, which proved our approach could be applied to discover CS which are not limited to specific tags.

We will discuss the methods utilized to handle both false negatives and false positives. First, the proportion of negative words may affect the results of sentiment analysis and introduce false negatives. For example, *"Try not to use DISPLAY, HOST or ID - these items could change."* scored − 0.4 showing a lower negative attitude, while it indicates the answer may have an error in *using DISPLAY, HOST OR ID*. However, the sentence *"- these items could change."* reduces the effect of negative words and affects the result. Most of the false positives were introduced by the tools, since the sentiment analysis tool is black-boxed, we did fine-grained tokenization for each sentence by deleting clauses and tokenising the sentence using custom rules to decrease the false positives.

Then, we compared the tool utilized in our approach which is Google NLP with the other open-source sentiment analysis tools to show remarkable effectiveness. We manually annotated 2,000 pieces of sentences and VPs according to our definition of Controversial Sentences and Qualified Verb phrases. Statistics of F1-score, precision and recall for each selected tool were done for evaluation. We chose the open source tools ParallelDots, TextProcessing and Google Natural Language API (2022) to compare on our dataset. Figure 8 illustrates the performance of these tools from three dimensions. On the left, Google API performed the best on CS discovery from all three dimensions. When implemented on extracting qualified VPs, ParallelDots and Google API demonstrate

an equal precision rate while ParallelDots has a lower recall. In general, Google Natural Language API performs the best and shows that it is the proper choice for our research.

Then, we further evaluate the effectiveness of our tool and the tools proposed by top conference papers. Since there does not exist the exact same tools or models aiming to extract CSs as we did, we chose S-HAN from Advance Lv et al. (2020) and the keywords extracted from Ren et al. (2019). Because of the tool developed by Ren et al. (2019) is not publicly available to access, therefore we utilize the keywords listed in their work for comparison. In Table 3, compared with the method using keywords-matching, S-HAN showed a relatively higher accuracy rate and lower false negative rate. The reason for such a dissatisfied performance is that S-HAN is designed to catch the sentence in a strong tone, while merely one-quarter of sentences in CSs may have a strong negative attitude and most of the CSs do not score with an absolutely high score due to the small proportion of negative words in each sentence. Not to mention the keyword-matching methods, one of the obstacles to processing the discussions between users is that there do not exist any fixed patterns or keywords to match. According to the comparison results, our method shows better performance on CS discovery.

Effectiveness of Bug code context locating From Sect. 3.2, 2155 PCAs were located, after which 5074 pieces of VPs were extracted and classified as qualified VPs for BugText Locating. We located 4718 pieces of BugText with an accuracy rate of 93.2% and a false positive rate of 14.1%. This stage includes three steps of evaluation: VP extraction, sentiment analysis for VP and dependency parsing for NP. First, constituency parsing is utilized to extract VPs from sentences. Looking into the false negatives, we did fine-grained tokenization before extracting VP, the only reason for some false negatives is the various expressing characters may not exist in our custom rules of tokenization which affected the results. And the false positives are introduced by the model itself. Second, another round of sentiment analysis on the extracted VPs to further located the BugText. Finally, 5074 VPs scored below -0.5 are fed into a dependency tree to locate the VPs or NPs who serve as the BugText in the sentence. When looking into the false negatives, we found they may be introduced by the limited categories of BugText. For any false positives, they could be introduced by the tools utilized in former steps by classifying incorrectly the sentences in a positive attitude.

Effectiveness of Bug code Snippet discovery After locating the BugText from Sect. 3.3, we tried to discover anchors to help locate the BugCode and we got 11,121 anchors in total with 1088 pieces of BugCode being detected. This stage is completed by a result filtration after PoS-tagging with a selection of noun words. All the anchors will be fed into the next step to search the BugCode in the answers. In our evaluation, nearly 40% of the anchors are demonstrative pronouns, indicating the answer was incorrect while no BugCode could be matched. With another small portion of anchors directly referencing the text in PCA; the rest of the anchors could precisely locate at least one piece of BugCode. After manually checking all the discovered BugCode, we found that some of the BugCode cannot be found in the answers but in the revision history which means the answer has been revised according to the comments. The correctness of BugCode is proved by other third-party resources via manual check.

Comparison with the state-of-the-art

We ran three state-of-the-art static analysis tools (SP'17 2017, Infer 2022 and Advance 2020) to evaluate the performance of our work. We chose 100 cases found by our work to evaluate their performance. Each of the tools can detect 22 bugs, 7 bugs and 24 bugs, respectively. We detailed 20 cases including all the types of cases in supplementary material, including bugs with API keywords, non-API keywords, API selection issues and security issues found by our work to further evaluate the performance and discuss details. SP'17 (2017) focuses on crypto-API-related bugs while for other non-API bugs, it barely detected any. Infer (2022) mainly detects null-dereference and memory leak bugs and it is incapable of detecting other types of security issues such as stack overflow. And Advance (2020) seems to have a similar discovering path as ours, except that they rely on well-defined API constraints from documentation (we discussed it in Sect. 5.2), when implementing on loosely-structured text from StackOverflow, it lost its power.

Note that we did not run our approach on the bugs found by other tools to cross-evaluate the performance, since our work is in need of sentences expressing attitudes and descriptions of bug code operations to guide the bug discovery approach. Apparently, the other tools are incapable of providing such resources for our approach. However, our work is capable of detecting various bug types which could show our performance in bug discovery.

Measurement

In this part, we illustrate the process of Counterpart Discovery and Unknown Bug Discovery by introducing a case. We give the bugs' details below with examples, including their category and impact. We also summarize lessons from our work.

Finding

In this part, we illustrate our findings on the discovered bugs by our approach and the context of SO related to bugs.

Discovered Bugs From 1000 question threads, we found 1088 pieces of BugCode. Some of the bugs are security-related, including stack overflow, memory leak and null dereference. After manually checking, we find 32 cases to be known bugs only took up 10.7% and the unknown bugs consist of 268 cases (89.3%). In our work, we define the BugCode as having been revised as known bugs for users; the other pieces of BugCode remain incorrect in the current version of answers are unknown bugs for users. For example, *Files.createTempDir()* is deprecated before appeared in the answer SO (2022), the answer with 183 upvotes under 439k view-counts was not revised to the correct one after the BugText has shown up in the comment. The delay of modification may result in vulnerability, since the attacker with access to the machine may potentially access data in a temporary directory CVE (2022) assigned by CVE. Directly using code snippets from StackOverflow without checking could introduce security issues in real-world development and the answers on StackOverflow may be a little bit "out-of-date" for fast development nowadays.

Problematic answers Problematic answers may mislead users. The issues from each case can be classified as (1) incomplete answer: the validity of the answer is confirmed while part of the question is not answered; (2) incorrect answer: part of the answer contains incorrectness when answering the question introducing issues such as lower performance; (3) out-of-date answer: the answer was right when it was first written but with the lapse of time it is outmoded for answering the question; and (4) potential insecure answer: the answer contains bugs which will have a security impact, including program crash, stack overflow, memory leak, null-dereference, etc. For those answers containing bugs proved to be true by third-party resources, some of the authors revised the answer within one day, however, the others may take several months or years to revise the answer. And for those answers containing bugs without revision, two of them remained incorrect for nearly 10 years till now, four answers remained incorrect for at least 5 years. Such updating frequency of answers may influence the security of software development.

Case study

In this section, we pick one case found in our work that could not be detected by other works based on documentation analysis Lv et al. (2020) and illustrate the process of counterpart discovery through third-party resources (e.g., GitHub Issue, documentation, GitHub Commit, etc.) and bug validation. Figure 9 illustrates the complete process of counterpart discovery and bug validation. First, we locate the CS from the comment which is *"Directly calling setSeed is dangerous."*. The BugText extracted from the CS is *"directly calling setSeed"*. The impact of this dangerous operation is *"It may replace the (really random) seed with the date."*. For counterpart discovery, we refer to the documentation of the correct usage. And in the middle of the figure, the documentation does not mention the related information on security issues. Therefore, we search the GitHub Issue for any extra information which could support the comment. The Issue (2022) clearly summarizes the bugs found related to when using setSeed and the specific rules generated from practices. We confirm the BugCode *"rand.setSeed(new Date.getTime())"* in the PCA after calling SecureRandom is buggy. We check the dates of the answer and the comment, the bug is still unknown to users. In total, we found nearly 89.3% BugCode remained in the current version of answers on StackOverflow. Note that we have reported all the BugCode found in our work to users. This will help them when using StackOverflow to help solve developing issues.[3] We will release our tool in the future.

Lessons learnt from our work

In Sect. 6.1, we summarize the overall finding from our work related to answer issues and potential impact. We have the unique opportunity to summarize the lessons learned from our work.

For StackOverflow community

Control the spreading of incorrect code snippets It is the StackOverflow community's responsibility to control the spreading of bug code snippets, these may either mislead users looking for proper answers or influence the development of open-source software. The community should either highlight the bug code snippet to users or advise users to revise their answers to improve the answers' quality. And also the community should remind the users to carefully deal with the answers containing incorrect code snippets.

For users sharing knowledge on SO

Write complete and explicit comments for answers improvement When leaving comments about the doubts or incorrectness of the answers, users are required to provide the correct and explicit information to help improve the answers. Based on our observation, a certain amount of users would not correct the answers and just indicate the current answer may not be correct, which may result

[3] https://anonymous.4open.science/r/SO-bugs-discovered-BF8A/.

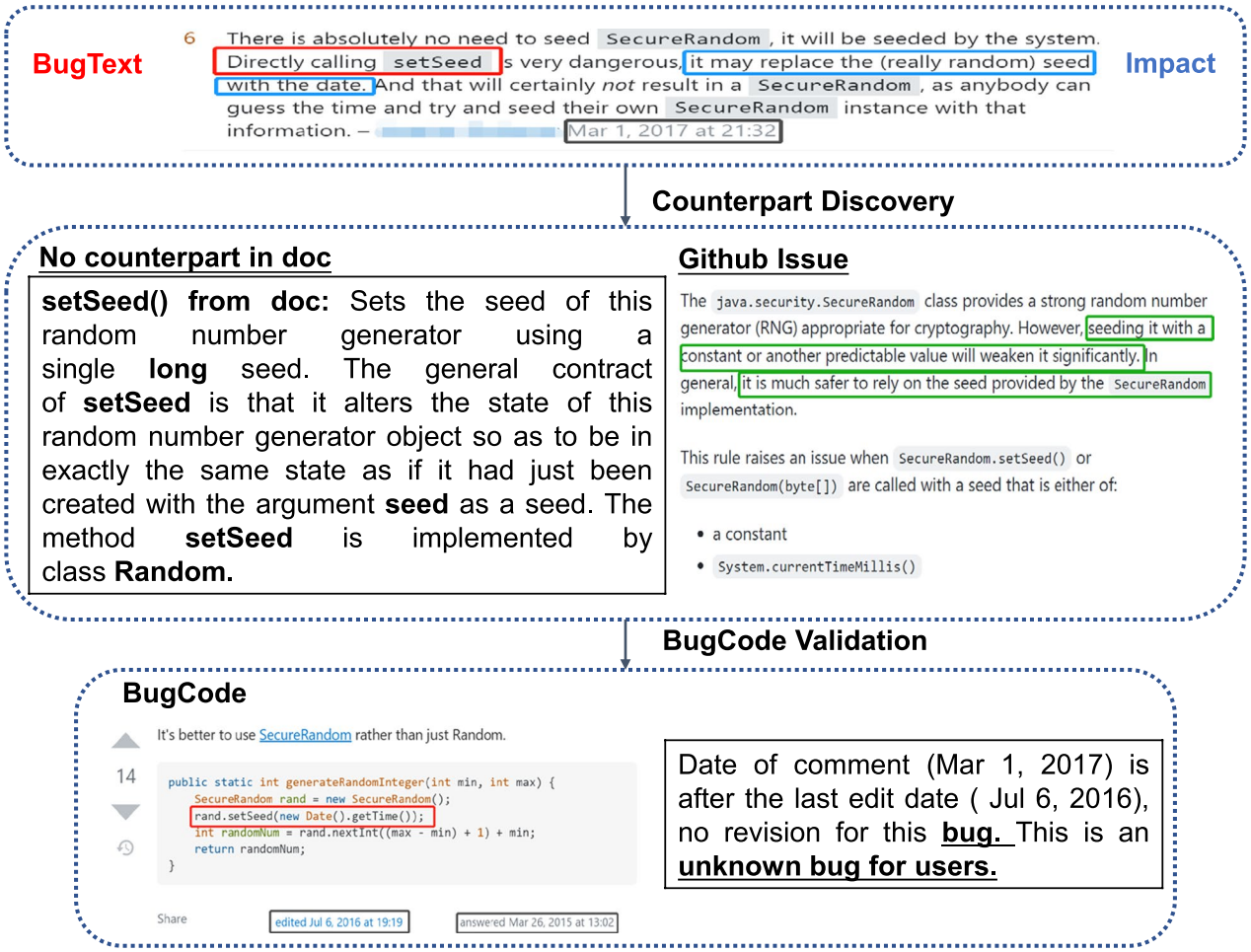

Fig. 9 The API-selection case for counterpart discovery and unknown bug discovery and validation

in severe impact when the buggy answer remained incorrect. Lack of useful information to correct the answer or implicit information from the comments may influence the time period of answer updating, since the author of the answer may not know how to locate and fix the buggy part of the answers by reading the comments.

For users requiring knowledge from SO

Use SO as a supplementary resource for software development After our manual check, we find that the bug found in SO could be validated from GitHub Commits or Issues, which indicates that the buggy code snippets from SO may have been utilized to develop software. The accepted answers or the relatively higher votes-up of each answer may influence the users when choosing the solutions. Since SO is a combination of knowledge resources including expertise, official documentation links related to the question and third-party links used to validate the answer. Therefore, it is proper to utilize SO as a supplementary resource to develop software rather than directly copying the code snippets from answers.

Discussion

In this section, we illustrate the limitations of our work as follows.

Problematic answers In Sect. 3.2, Google Natural Language API is adopted to complete sentiment analysis. Since the model inside is black-box, only the output of the tool is available for users, thus any false positives or false negatives yielded are inextricable. These errors introduced by the tool may influence our results in bug discovery.

False positives introduced by the context of SO The complicated content of comments also yielded false positives for our work. We found some of the comments included both positive attitudes towards the answer and the description of the incorrect operation. Natural language understanding techniques may have the potential to avoid such cases, however, they may need a great

amount of high-quality corpus to build a solid knowledge database. The current version of our work is incapable of handling such a situation.

Related work

Recent years have witnessed the emerging demand for knowledge sharing and its impact on security-related code practice. In this section, we discuss the related work on bug detection through documentation and code, and the research related to StackOverflow. We classify them into three types: bug detection through document analysis, bug detection through code comparison and fuzzing, and research on StackOverflow.

Bug detection through document analysis In recent years, the amount of research on document analysis for bug detection are increasing rapidly. Researchers attempt to exact API constraints from official manuals to help with bug detection. Advance (2020) is designed to extract rules from C++ official documentation, and generate verification code from API descriptions to detect unknown bugs on open-source projects. Zhong et al. (2011) and Pandita et al. (2016) propose different ways to extract API constraints and API call-order rules, respectively, to detect the inconsistency between the context information and source code. Different from them, our work detects bugs by analyzing comments in StackOverflow which has more loosely organized writings.

Bug detection through code comparison and fuzzing Another general way to detect bugs is code analysis. Code comparison and fuzzing are two of the common methods. Ahmadi et al. (2021) propose a method to compare the structure of similar code snippets to detect bugs. Yamaguchi et al. (2013) propose to do a taint analysis on code to extract security-related objects exceptions and missing conditions to speed up the manual check. You et al. (2017 propose SemFuzz to extract vulnerability-related information to guide fuzzing. Zong et al. (2020) propose FuzzGuard to filter the seed for fuzzing to increase the efficiency of fuzzers.

Research on StackOverflow Security research on StackOverflow includes several research areas: Meldrum et al. (2020) propose a measurement work on evaluating the answers' quality on StackOverflow, and the result shows users need to be cautious when reusing the code snippets from StackOverflow. Fischer et al. (2017) utilize a static code analysis tool to detect the code similarity between code snippets from SO and applications from the Android market to label the insecure code snippets on SO. Chen et al. (2019) conduct a measurement work on the security-related posts on StackOverflow and found that insecure posts had more view counts and higher scores. Ren et al. (2019) propose a similar work to ours, they proposed an approach to find controversial answers among all the accepted answers on StackOverflow and combined the extracted information with official API documentation to help users better understand the controversies. In our work, we propose a new discovery approach to dig out the buried knowledge from comments with no guidance of any API keywords and discover the bugs on SO.

Conclusion

In this paper, we propose an automatic approach to exploit the knowledge from discussions on the guidance of bug discovery on StackOverflow. Utilizing NLP techniques, we leverage sentiment analysis to discover CSs from discussions, constituency parsing and dependency parsing to assist the process of BugText Locating. Then a fine-grained tokenization method is adopted to extract anchors which help the discovery process of BugCode. We applied our approach on 1000 threads from StackOverflow and discovered 1088 pieces of BugCode in total, achieving a precision rate of 95.5% in CS Discovery. In randomly selected 300 pieces of BugCode, 276 real bugs were discovered by our approach and 89.3% of them remained in the current version of answers without revision which may further mislead the users.

Acknowledgements
We would like to thank the anonymous reviewers for their detailed comments and useful feedback.

Author contributions
YY: investigation, conceptualization, methodology, materials, writing, editing, experiment, validation, review, resources. YL: experiment, validation, review. KC: discussion, review, supervision. JL: experiment, review. All authors read and approved the final manuscript.

Funding
Not applicable.

Availibility of data and materials
Not applicable.

Declarations

Competing interests
We confirm that none of the authors has any competing interests in the manuscript.

Received: 28 December 2022 Accepted: 19 March 2023
Published online: 03 April 2023

References

Aggarwal A, Jalote P (2006) Integrating static and dynamic analysis for detecting vulnerabilities. In: 30th annual international computer software and applications conference (COMPSAC'06), vol 1, pp 343–350. IEEE
Ahmadi M, Farkhani RM, Williams R, Lu L (2021) Finding bugs using your own code: detecting functionally-similar yet inconsistent code. In: 30th USENIX security symposium (USENIX Security 21), pp 2025–2040

Amin A, Eldessouki A, Magdy MT, Abdeen N, Hindy H, Hegazy I (2019) Androshield: automated android applications vulnerability detection, a hybrid static and dynamic analysis approach. Information 10(10):326
Ayewah N, Pugh W, Hovemeyer D, Morgenthaler JD, Penix J (2008) Using static analysis to find bugs. IEEE Softw 25(5):22–29
Böhme M, Pham V-T, Nguyen M-D, Roychoudhury A (2017) Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2329–2344
Bug-java (2022) http://bugs.java.com/bugdatabase/view_bug.do?bug_id=5003595
Chen M, Fischer F, Meng N, Wang X, Grossklags J (2019) How reliable is the crowdsourced knowledge of security implementation? In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE), pp 536–547 (2019). IEEE
Chen D, Manning CD (2014) A fast and accurate dependency parser using neural networks. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), pp 740–750
Chen H, Xue Y, Li Y, Chen B, Xie X, Wu X, Liu Y (2018) Hawkeye: Towards a desired directed grey-box fuzzer. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, pp 2095–2108
Constituency Parser (2022) https://github.com/nitishgupta/constituency-parse-predictor
CVE-2020-8908. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8908 (2022)
Fischer F, Böttinger K, Xiao H, Stransky C, Acar Y, Backes M, Fahl S (2017) Stack overflow considered harmful? the impact of copy & paste on android application security. In: 2017 IEEE symposium on security and privacy (SP), pp 121–136. IEEE
Gardner M, Grus J, Neumann M, Tafjord O, Dasigi P, Liu NF, Peters M, Schmitz M, Zettlemoyer LS (2017) Allennlp: a deep semantic natural language processing platform arXiv:1803.07640
Google NLP API (2022) https://cloud.google.com/docs/
Infer (2022) https://github.com/facebook/infer/
Issue-3 (2022) https://github.com/rakcy/code-scanner-demo/issues/3/
Joshi C, Singh UK, Tarey K (2015) A review on taxonomies of attacks and vulnerability in computer and network system. Int J 5(1)
Kiss B, Kosmatov N, Pariente D, Puccetti A (2015) Combining static and dynamic analyses for vulnerability detection: illustration on heartbleed. In: Hardware and software: verification and testing: 11th international Haifa verification conference, HVC 2015, Haifa, Israel, November 17–19, 2015, Proceedings 11, pp 39–50. Springer, Berlin
Li J, Zhao B, Zhang C (2018) Fuzzing: a survey. Cybersecurity 1(1):1–13
Lv T, Li R, Yang Y, Chen K, Liao X, Wang X, Hu P, Xing L (2020) Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection. In: Proceedings of the 2020 ACM SIGSAC conference on computer and communications security, pp 1837–1852
Medhat W, Hassan A, Korashy H (2014) Sentiment analysis algorithms and applications: a survey. Ain Shams Eng J 5(4):1093–1113
Meldrum S, Licorish SA, Owen CA, Savarimuthu BTR (2020) Understanding stack overflow code quality: a recommendation of caution. Sci Comput Program 199:102516
NLTK (2022) https://nltk.org/
Pandita R, Taneja K, Williams L, Tung T (2016) Icon: inferring temporal constraints from natural language API descriptions. In: 2016 IEEE international conference on software maintenance and evolution (ICSME), pp 378–388. IEEE
Ren X, Sun J, Xing Z, Xia X, Sun J (2020) Demystify official API usage directives with crowdsourced API misuse scenarios, erroneous code examples and patches. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 925–936
Ren X, Xing Z, Xia X, Li G, Sun J (2019) Discovering, explaining and summarizing controversial discussions in community q &a sites. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE), pp 151–162. IEEE
Ren X, Ye X, Xing Z, Xia X, Xu X, Zhu L, Sun J (2020) API-misuse detection driven by fine-grained API-constraint knowledge graph. In: 2020 35th IEEE/ACM international conference on automated software engineering (ASE), pp 461–472 . IEEE
SO-617414 (2022) https://stackoverflow.com/questions/617414/how-to-create-a-temporary-directory-folder-in-java/6403880#6403880
SO-vote-up (2022) https://stackoverflow.com/help/privileges/vote-up
Spacy (2022) https://spacy.io/
Stackexchange (2022) https://api.stackexchange.com/
Stackoverflow. https://stackoverflow.com/ (2022)
Stanfordnlp (2022) https://nlp.stanford.edu/software/
Yamaguchi F, Wressnegger C, Gascon H, Rieck K (2013) Chucky: exposing missing checks in source code for vulnerability discovery. In: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security, pp.499–510
You W, Zong P, Chen K, Wang X, Liao X, Bian P, Liang B (2017) Semfuzz: semantics-based automatic generation of proof-of-concept exploits. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2139–2154
Zhang M (2020) A survey of syntactic-semantic parsing based on constituent and dependency structures. Sci China Technol Sci 63(10):1898–1920
Zhong H, Zhang L, Xie T, Mei H (2011) Inferring specifications for resources from natural language API documentation. Autom Softw Eng 18(3):227–261
Zhou Y, Gu R, Chen T, Huang Z, Panichella S, Gall H (2017) Analyzing APIs documentation and code to detect directive defects. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE), pp 27–37. IEEE
Zong P, Lv T, Wang D, Deng Z, Liang R, Chen K (2020) {FuzzGuard}: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: 29th USENIX security symposium (USENIX security 20), pp 2255–2269

Publisher's Note