Effectiveness of ChatGPT for Static Analysis: How Far Are We?

Mohammad Mahdi Mohajer
York University
Toronto, Canada
mmm98@yorku.ca

Reem Aleithan
York University
Toronto, Canada
reem1100@yorku.ca

Nima Shiri Harzevili
York University
Toronto, Canada
nshiri@yorku.ca

Moshi Wei
York University
Toronto, Canada
moshiwei@yorku.ca

Alvine Boaye Belle
York University
Toronto, Canada
alvine.belle@lassonde.yorku.ca

Hung Viet Pham
York University
Toronto, Canada
hvpham@yorku.ca

Song Wang
York University
Toronto, Canada
wangsong@yorku.ca

ABSTRACT

This paper conducted a novel study to explore the capabilities of ChatGPT, a state-of-the-art LLM, in static analysis tasks such as static bug detection and false positive warning removal. In our evaluation, we focused on two types of typical and critical bugs targeted by static bug detection, i.e., *Null Dereference* and *Resource Leak*, as our subjects. We employ Infer, a well-established static analyzer, to aid the gathering of these two bug types from 10 open-source projects. Consequently, our experiment dataset contains 222 instances of *Null Dereference* bugs and 46 instances of *Resource Leak* bugs. Our study demonstrates that ChatGPT can achieve remarkable performance in the mentioned static analysis tasks, including bug detection and false-positive warning removal. In static bug detection, ChatGPT achieves accuracy and precision values of up to 68.37% and 63.76% for detecting *Null Dereference* bugs and 76.95% and 82.73% for detecting *Resource Leak* bugs, improving the precision of the current leading bug detector, Infer by 12.86% and 43.13% respectively. For removing false-positive warnings, ChatGPT can reach a precision of up to 93.88% for *Null Dereference* bugs and 63.33% for *Resource Leak* bugs, surpassing existing state-of-the-art false-positive warning removal tools.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**; **Software defect analysis**; *Software testing and debugging*.

KEYWORDS

Static analysis, ChatGPT, Large language models

AIware '24, July 15–16, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0685-1/24/07
https://doi.org/10.1145/3664646.3664777

ACM Reference Format:
Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. 2024. Effectiveness of ChatGPT for Static Analysis: How Far Are We?. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware '24), July 15–16, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3664646.3664777

1 INTRODUCTION

Numerous static code analysis techniques have been utilized in the literature for the automatic detection of real-world software bugs [4, 7, 42, 44, 63]. These tools typically rely on predefined heuristic rules to scan and analyze the codebases or binaries of software projects [7, 42, 44]. During this analysis, any violations of these rules are categorized as a bug, leading the tools to flag the corresponding code artifact, such as a line or a group of lines, as buggy. However, employing static bug detectors presents specific challenges. One primary issue is that most static bug detectors generate numerous false-positive warnings [2, 41]. Consequently, additional manual review is essential to validate the reported potential bugs, resulting in a time-consuming and labor-intensive process [53].

Recently, Large Language Models (LLMs) such as ChatGPT have demonstrated significant potential in various reasoning and decision-making roles, serving as intelligent agents, especially in SE tasks such as code generation and understanding [25, 59, 65]. However, to date, there has yet to be research exploring the capabilities of ChatGPT for static code analysis tasks [19]. To address this gap, in this paper, we take a step toward conducting an empirical study on the effectiveness of ChatGPT for static code analysis. In our study, we employed ChatGPT on two different types of tasks: 1) static bug detection and 2) false positive warning removal. To evaluate ChatGPT, we select two typical and widely-studied bugs [5, 8, 16, 32, 33, 35, 40, 61] commonly targeted by static bug detection: *Null Dereference* and *Resource Leak* as our subjects. Following existing works [22, 33], we utilize Infer [26] to facilitate the collection of these two types of bugs from 10 open-source projects. As a result, our dataset includes 222 *Null Dereference* bugs and 46 *Resource Leak* bugs.

In this study, we use the latest versions of ChatGPT models at the time of the study (i.e., ChatGPT-3.5-Turbo and ChatGPT-4). Moreover, we adhere to the best practices of prompt engineering [6, 14] and create precise and context-aware prompts to effectively harness the language models' capabilities. We explore various prompting strategies, including zero-shot, one-shot, and few-shot prompting, to evaluate the performance of different ChatGPT models across various scenarios. By strategically adapting our prompts and methodologies, we aim to identify the most efficient and accurate ways of leveraging LLMs in static analysis.

Our experiments reveal that ChatGPT can achieve remarkable performance with significant improvements when compared to previous baseline methods for each respective task: 1) In static bug detection, they can achieve a precision rate that is 12.86% higher for *Null Dereference* bugs and 43.13% higher for *Resource Leak* bugs as improvements compared to the base results of Infer. 2) In false-positive warning removal, they can imp Infer's precision rate by 28.68% for *Null Dereference* bugs and 9.53% for *Resource Leak* bugs, surpassing the performance of existing baselines. Additionally, it can improve the precision of the results of static bug detection in the first step by 16.31% for *Null Dereference* bugs.

As a summary, this paper makes the following contributions:

- We present a novel empirical study on ChatGPT's capabilities on two static analysis tasks: 1) static bug detection and 2) false-positive warning removal.
- We show that ChatGPT can effectively improve static bug detection performance, improving the base results of the current state-of-the-art baseline tool, i.e., Infer.
- We demonstrate that ChatGPT can effectively eliminate false-positive warnings from the output of static bug detectors such as Infer and its extended static bug detector, thereby enhancing their precision and surpassing the performance of existing baseline methods in false-positive warning removal.
- We release the dataset and the source code for our experiments for future usage and the replication of our study[1].

2 BACKGROUND

2.1 Static Bug Detection

Static bug detection is an automated technique for inspecting and analyzing a program's source code, object code, or binaries, all without executing the program [3, 39]. This process identifies potential bugs by examining how the code's control and data flow align with specific bug patterns and rules [3, 64]. Multiple tools and methods have been developed in both research and industry for static bug detection [21]. Infer, created by Meta, is a static bug detection tool capable of being utilized across various programming languages, including Java, C, C++, Objective-C, and C#. It accomplishes this by utilizing a predetermined set of rules to identify potential bugs and conducting inter-procedural analysis as part of the project compilation process [26]. Google has also introduced ErrorProne, a static bug detector tailored for Java programs [17]. In this study, we employ Infer as a baseline for comparison with ChatGPT to assess the improvement and as a tool for generating warnings for our data collection.

[1] https://doi.org/10.5281/zenodo.10828316

2.2 False-Positive Warning Removal

A significant issue associated with static bug detectors is their tendency to generate a considerable volume of inaccurate warnings, which are essentially alerts that are not genuine indicators of actual bugs [20, 23, 30, 33, 49, 54, 57, 72]. Recent research demonstrates that the false-positive warning rate can escalate to as high as 91% [30]. Recent studies have addressed this issue by providing various techniques for detecting and eliminating false-positive warnings. Wang et al. [66] have proposed a "Golden Features" set to detect actionable warnings and eliminate the unactionable ones. Recently, Kharkar et al. [33] introduced distinct tools that leverage state-of-the-art neural models, mostly transformer-based models, which are widely regarded as the most effective approach for eliminating false-positive warnings. In this work, we opt to utilize the tools outlined in this study as our baselines for comparison. Specifically, we use the feature-based approach, DeepInferEnhance, and a GPT-C powered approach [33] as our baselines for false positive removal.

2.3 Large Language Models

Large Language Models (LLMs) have gained significant popularity in recent research and industrial applications. Numerous recent studies are investigating the utilization of LLMs in the field of Software Engineering (SE), driven by the significant progress and advancements achieved by LLMs [13, 58, 70, 74]. ChatGPT [51], one of the most renowned LLMs, has recently gained widespread recognition for performing software engineering tasks [6, 14, 18, 70]. ChatGPT is accessible throughout an API[2] and has been created by harnessing the capabilities of two state-of-the-art GPT models, specifically, GPT-3.5 Turbo [52] and GPT-4 [1]. Utilizing LLMs like ChatGPT as decision-making components introduces a novel approach to systematically interact with instruction-tuned LLMs, a method known as prompt engineering. Prompt engineering is the practice of creating tailored input queries that effectively communicate with LLMs [6, 14]. Numerous investigations leverage prompt engineering in their utilization of LLMs [6, 31, 46, 70, 73]. Prompt engineering as a practice offers the flexibility to utilize various strategies, including the zero-shot approach, where the LLM is prompted without any prior input/output examples; the one-shot method, involving an additional example; and the few-shot strategy, denoted as K-shot, which provides K examples as previous input/output pairs for the LLM [14, 31]. Moreover, in prompt engineering, techniques like Chain-of-Thought (COT) are employed to enhance the correctness of generated output. This is achieved by either including the thinking steps in examples or requesting an explanation of the decision-making process from the LLM [14, 34, 68].

3 DATA COLLECTION

In this work, we take two types of typical and critical bugs that are targeted by static bug detection, i.e., *Null Dereference* and *Resource Leak*, as our subjects. To accelerate the data collection, we first applied Infer [26] to our experimental projects with a focus on detecting *Null Dereference* and *Resource Leak* bugs. The rationale behind selecting Infer is its extensive adoption in various companies, including Microsoft. Furthermore, Infer exhibits higher precision

[2] https://platform.openai.com/docs/api-reference

Table 1: Summary of analyzed projects. Projects highlighted in ● are from Kharkar et al. [33], and projects highlighted in ● are collected by us. The warnings reported in this table are generated by Infer [26] and manually verified. The column "Verified Warnings" has the number of verified warnings, including total, true positives (#TP), and false positives (#FP), for corresponding projects.

Project	Version	LOC	Repository Group	Verified Warnings		
				#Total	#TP	#FP
nacos	2.0.2	217,653	Alibaba	58	35	23
azure-maven-plugins	2.2.2	53,025	Microsoft	45	29	16
playwright-java	1.13.0	67,548	Microsoft	5	5	0
java-debug	0.47.0	22,852	Microsoft	2	1	1
dolphinscheduler	2.0.9	215,808	Apache	100	77	23
dubbo	3.2	350,957	Apache	193	74	119
bundletool	1.15.1	135,711	Google	51	14	37
guava	32.1.1	698,201	Google	35	12	23
jreleaser	1.7.0	114,914	Community	30	19	11
jsoup	1.16.1	33,689	Community	33	2	31
Total Number of Verified Warnings				**552**	**268**	**284**

in comparison to alternative static analysis tools, leading to the generation of more valid warnings [33]. We apply Infer to a selection of seven prominent GitHub projects (with at least more than 200 stars), along with three projects featured in prior research, to generate warnings [33]. These projects are shown in Table 1.

Note that, as Infer can report false positives [22, 33], we further manually check whether it is a true bug or a false positive for each reported warning. This manual labeling process involves three authors with at least four years of development experience, each independently reviewing all the reported warnings by Infer. For each warning, they assign a binary label, i.e., zero indicating a "false positive", signifying that the warning generated by Infer is incorrect and does not represent a true bug, and one indicating a "true positive", indicating that the warning is accurate and demonstrates a real bug. Following this individual labeling, the authors then collaborate to identify and resolve any discrepancies or conflicts in their assessments. After resolving these conflicts, we have our comprehensive dataset containing warnings generated by Infer, their corresponding ground truth labels, and the method by which the warning occurred. Eventually, we have 268 true-positive warnings and 284 false-positive warnings in our dataset.

4 STUDY SETUP

4.1 Overview

Figure 1 provides an overview of the pipeline of our study, which contains two main steps: 1) evaluating ChatGPT for static bug detection (Section 5.1) and 2) evaluating ChatGPT on identifying false-positive bugs in the results of Infer and step 1 for improving the detection accuracy (Section 5.2).

4.2 Research Questions

To evaluate the performance of ChatGPT, we design experiments to answer the following research questions (RQs):

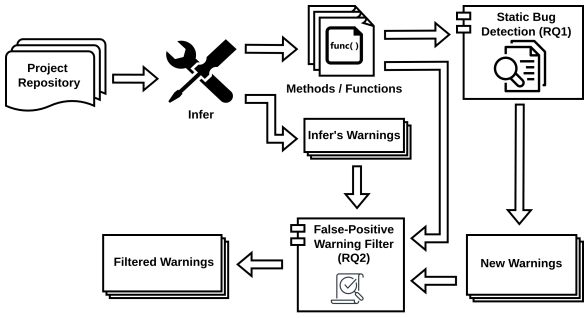

Figure 1: The overview of our study pipeline.

Table 2: Prompt templates used in this work

RQ	Prompt Template
1	You are an advanced static analyzer for programs and source codes written in Java. Your duty is to detect **<BugType>** bugs in the given Java codes delimited by #### **REMEMBER**: Each of these Java codes contains a method implementation, and you are supposed to analyze the body of this method. **REMEMBER**: **<Description and Context Related to the BugType>** **REMEMBER**: Only report the objects or method and function calls and invocations you are sure and confident are highly likely a **<BugType>** potential bug. Please also explain your decisions in the output. **<FORMATTING INSTRUCTIONS>**
2	You possess expertise in the examination of Java programs and their source code. We've employed an external static analyzer on Java programs and their source code, which has produced warnings regarding potential **<BugType>** issues. Your responsibility is to assess the correctness of the warnings generated by the external static analyzer. Some of these warnings are false-positive predictions, which means that these warnings are not a correct indicator of null dereference bugs. You need to ensure that these warnings indicate a potential **<BugType>** bug. For your analysis, you will receive the following items as input: 1. The Java code enclosed by #### in the input. 2. The warning generated by the external static analyzer is delimited by **$$$$** in the input. **REMEMBER**: **<Description and Context Related to the BugType>** **REMEMBER**: Each of these Java codes contains a method implementation, and you are supposed to analyze the body of this method. Please also explain your decisions in the output. **<FORMATTING INSTRUCTIONS>**

RQ1 (Static Bug Detection): How effective is ChatGPT in improving the performance of static bug detectors?
RQ2 (False Positive Warning Removal): What is the effectiveness of ChatGPT in filtering false positive warnings?

We explore the potential of ChatGPT in the realm of static bug detection in RQ1. In RQ2, we delve into the effectiveness of ChatGPT in identifying and eliminating false-positive warnings from the output of a static bug detector.

4.3 ChatGPT Versions

We choose the latest variations of ChatGPT models [51], ChatGPT-4 [1] and ChatGPT-3.5 Turbo [52], as our study subjects. For ChatGPT-4 and ChatGPT-3.5 Turbo, we utilize the default settings, which provide maximum token support of 8,192 and 4,097, respectively [1, 52]. Due to this limitation in the maximum input token, we constrain the code inputs to the method scope, implying that we only provide the methods themselves as input code snippets to the LLMs.

4.4 Prompting Strategies

In each of our experiments, we use different prompt engineering strategies such as zero-shot, one-shot, and few-shot strategies. In the zero-shot strategy, the LLM is prompted without preceding examples from the previous LLM input and output pairs. In contrast, the few-shot and one-shot strategies incorporate examples from

the previous LLM input and output pairs. The difference between the one-shot and few-shot strategies lies in the number of examples included: the one-shot strategy employs a single example in the prompt, while the few-shot strategy encompasses multiple examples. In RQ1 (static bug detection) and RQ2 (false-positive warning removal), we use zero-shot, one-shot, and few-shot strategies. In this paper, for the few-shot strategy (K-shot), we input the model with three examples ($K = 3$). The rationale for selecting $K = 3$ is based on the consideration that opting for values exceeding three could potentially violate the maximum token limit constraint imposed on our inputs for ChatGPT models (details in Section 4.3). Furthermore, in the prompts for all of our experiments, we request the LLM to explain its decision-making process and the steps it takes to arrive at its conclusions. According to the literature, this approach, known as zero-shot Chain-of-Thought reasoning, can enhance the output's robustness and validity [34, 68]. Eventually, our prompt templates for each of our research questions are depicted in Table 2. Our prompt templates offer detailed descriptions and context for bug types such as Null Dereference and Resource Leak. This contextual information helps us to use the models' full capabilities for detection performance. This includes defining the criteria that classify these bugs. Additionally, we guide LLMs using delimiters to identify specific inputs within the given prompt. For instance, in the prompt template for RQ1 (static bug detection), we use #### to specify the location of the input Java code snippet, ensuring that the LLM focuses solely on this section for analysis. Moreover, at the end of each prompt template, two elements are incorporated: 1) a request for the model to explain its decision-making process and the rationale behind the output, and 2) a directive for the model to generate the output in a specified format, following the provided formatting instructions. For example, in our experiments, we instruct the models to produce the output in JSON format, facilitating easier parsing of the output into our program variables for subsequent analysis.

4.5 Data Sampling

Our experiments employ N-fold cross-validation to remove potential data sampling bias, with N set to 5 in this study [69]. This approach involves splitting the dataset into five subsets, where one-fifth of the data serves as the validation set, and the remaining four-fifths of functions are used to select examples in prompting the LLMs for one-shot and few-shot strategies. We select the examples for these strategies randomly in a uniform distribution. Also, in each of the examples utilized in one-shot and few-shot strategies, we incorporate one true-positive record and one false-positive record to prevent any bias towards a particular group of examples when applying these strategies to the LLM.

4.6 Evaluation Metrics

We employ the following evaluation metrics to assess our experiments concerning each of our research questions:

For **Static Bug Detection (RQ1)** and **False-Positive Warning Removal (RQ2)**, we use Accuracy, Precision, and Recall metrics since we have a ground truth for the warnings we generated for our dataset for evaluation. Also, to choose the best combination of strategy and model, we use the F1-score metric. Also, it is important to highlight that our dataset relies on the warnings produced by Infer, containing both accurate warnings (true positives) and inaccurate warnings (false positives). As a result, precision is the only metric available for evaluating Infer because other values needed for calculating recall and accuracy, such as true negative rate and false negative rate, are not available. Consequently, in our experiments measuring improvements relative to Infer, we prioritize precision enhancement as the primary factor.

5 RESULT ANALYSIS

5.1 RQ1: Performance of ChatGPT on Improving Static Bug Detectors

Approach: In the case of static bug detection, we input the code snippet of each record in our dataset to the ChatGPT models. We expect that ChatGPT models will identify the issue previously described and detected by Infer and produce a valid warning. We perform our experiments under different prompting strategies such as zero-shot, one-shot, and few-shot strategies (more details in Section 4.4) by using two different ChatGPT models, i.e., ChatGPT-3.5 Turbo and ChatGPT-4 (more details in Section 4.3). The models have been given a specialized prompt with specifications for each bug type to increase its detection validity. For example, to address *Null Dereference* bugs, we collect common bug patterns for this type of bug, like not having null checks before dereferencing an object. We then provide this information to the LLM in the initial prompt. This specialization helps the models understand the task at hand. Additionally, for Resource Leak bugs, a similar approach is used. Moreover, specific structured output requirements are defined to facilitate easy parsing and extracting necessary information from the models' responses. Given that we possess a ground truth for the provided buggy code snippet, we expect ChatGPT models to recognize the problem previously identified by Infer and issue a valid warning for it. Furthermore, we ask ChatGPT models to offer an additional explanation for each potential bug it detects and the warnings it generates.

Baselines: As a baseline for this RQ, we select Infer, a state-of-the-art static bug detector. Using our collected dataset, we compare ChatGPT models' performance in static bug detection with Infer's base results to see if it can improve Infer's precision.

It is important to note that we did not compare ChatGPT models' performance directly to Infer as a standalone static bug detector. In fact, we evaluated their performance only on warnings generated by Infer, not the entire body of source code in the examined projects in our dataset. Our main goal is to determine whether ChatGPT models can help improve the base results produced by Infer.

Result: Table 3 summarizes the results of the static bug detection task under different prompting strategies with different ChatGPT models using Infer on our collected dataset and the projects used in the prior study by Kharkar et al [33]. We also show the optimal combination of model and strategy for the static bug detection task for each of the datasets, which is the one that has the highest F1-Score for a specific bug type. Notably, in our collected dataset, the most effective combination for both *Null Dereference* and *Resource Leak* bugs involves utilizing ChatGPT-4 with the zero-shot strategy. Considering this, in static bug detection, ChatGPT models can achieve accuracy, precision, and recall rates of 68.37%, 63.76%, and 88.93%,

Table 3: Summary of ChatGPT models' results for each of the model-strategy combination in Static Bug Detection (RQ1) using two datasets. In this table, rows highlighted in ● indicate the most effective combination of model and strategy for *Null Dereference* bugs, and rows in ● indicate the most effective one for *Resource Leak* bugs.

Dataset	Bug Type	Infer's Precision	Strategy	Model	Accuracy	Precision	Recall	F1-Score
Our collection	Null Dereference	50.9%	**Zero Shot**	GPT-3.5-Turbo	50.66%	52.32%	40.64%	45.75%
				GPT-4	**68.37%**	**63.76%**	**88.93%**	**74.27%**
			One Shot	GPT-3.5-Turbo	60.67%	62.49%	59.76%	61.09%
				GPT-4	64.54%	62.20%	79.18%	69.67%
			Few Shot	GPT-3.5-Turbo	60.47%	66.14%	48.63%	56.05%
				GPT-4	64.31%	65.21%	64.96%	65.09%
	Resource Leak	39.6%	**Zero Shot**	GPT-3.5-Turbo	56.31%	44.34%	40.44%	42.30%
				GPT-4	**76.95%**	**82.73%**	**55.11%**	**66.15%**
			One Shot	GPT-3.5-Turbo	34.87%	35.03%	72.22%	47.18%
				GPT-4	72.78%	68.39%	63.55%	65.88%
			Few Shot	GPT-3.5-Turbo	49.31%	41.49%	65.55%	50.82%
				GPT-4	75.25%	74.12%	59.55%	66.04%
Projects from [33]	Null Dereference	65.2%	**Zero Shot**	GPT-3.5-Turbo	50.31%	69.85%	41.53%	52.09%
				GPT-4	**77.90%**	**81.87%**	**85.51%**	**83.65%**
			One Shot	GPT-3.5-Turbo	58.25%	70.90%	59.87%	64.92%
				GPT-4	70.51%	80.95%	72.56%	76.53%
			Few Shot	GPT-3.5-Turbo	56.47%	75.60%	51.28%	61.11%
				GPT-4	68.52%	80.57%	67.56%	73.50%
	Resource Leak	53.8%	**Zero Shot**	GPT-3.5-Turbo	61.66%	50.0%	50.0%	50%
				GPT-4	**84.61%**	**100%**	**71.42%**	**83.32%**
			One Shot	GPT-3.5-Turbo	46.66%	50%	70%	58.33%
				GPT-4	73.33%	60%	50%	54.54%
			Few Shot	GPT-3.5-Turbo	48.33%	36.66%	40%	38.26%
				GPT-4	68.33%	60%	40%	48%

respectively, for *Null Dereference* bugs. Likewise, for *Resource Leak bugs*, these metrics can attain values of 76.95%, 82.73%, and 55.11%, respectively. This indicates that ChatGPT models exhibit precision levels that are 12.86% and 43.13% higher than Infer, enhancing the performance of the state-of-the-art static bug detector baseline. Furthermore, it is noteworthy that some other model-strategy combinations also demonstrate significant enhancement after applying to Infer's base results. For instance, ChatGPT-3.5-Turbo utilized with the one-shot strategy continues to enhance Infer's base results in the detection of *Null Dereference* bugs, achieving a rate of 62.49% compared to Infer's 50.9%. We can also see a similar result in the dataset from the projects used by Kharkar et al. [33]. In this scenario, the most effective strategy and model combination is the zero-shot strategy coupled with the ChatGPT-4 model. Compared to Infer's base results on this dataset depicted in Table 3, we have a 16.6% and 46.2% boost in precision for *Null Dereference* and *Resource Leak* bugs, respectively.

Answer to RQ1: ChatGPT can significantly improve the base results of the state-of-the-art static bug detection tool (i.e., Infer) on detecting *Null Dereference* and *Resource Leak* bugs.

5.2 RQ2: Performance of ChatGPT on False-Positive Warning Removal

Approach: To answer this RQ, we input both a code snippet and the warning linked to the code snippet generated by a bug detector to the ChatGPT models. This step aims to improve the precision of static bug detection by eliminating false-positive warnings. These warnings can originate from various static bug detectors, such as Infer's output or the results of RQ1 by ChatGPT models. To examine the generalizability of ChatGPT models in removing false positives, we use the warnings generated by both Infer and ChatGPT models in RQ1 (as described in Section 5.1). Subsequently, the models, previously instructed with specialized guidelines, evaluate the code snippet and its corresponding warning. We also perform our experiments under different prompting strategies such as zero-shot, one-shot, and few-shot strategies (more details in Section 4.3) by using two different ChatGPT models, i.e., ChatGPT-3.5 Turbo and ChatGPT-4 (more details in Section 4.3).

Baselines: We select the state-of-the-art false-positive removal approach proposed in the recent study conducted by Kharkar et al. [33], i.e., GPT-C and two other baselines to evaluate GPT-C, which are a feature-based logistic regression model [33] and DeepInferEnhance (based on CodeBERTa) [33]. However, due to confidential issues, the authors neither disclosed the source code for their tool nor their experiments. They also did not release their collected dataset. Thus, to enable a meaningful comparison with the baseline tools outlined in their study, we gathered data that closely mirrored the one described in their paper, including the same projects and similar versions. Ultimately, we conducted our experiments on a dataset akin to the one they utilized, allowing for a fair and direct comparison between ChatGPT models and their baseline tools.

Result: As we explained, we have two options for the static bug detector utilized for false-positive warning removal:

Table 4: ChatGPT models' performance in False-Positive Warning Removal (RQ2) on warnings generated by Infer and ChatGPT. In this table, $P_{Original}$ is the precision of the static bug detector for the corresponding bug type and P_{After} is the precision of the static bug detector after applying False-Positive Warning Removal process. "Imp." indicates the amount of precision improvement. The records that have no improvement in precision are shown with "–".

Static Bug Detector	Bug Type	Strategy	Model	$P_{Original}$	P_{After}	Imp.	Recall	Accuracy	F1-Score
Infer	Null Dereference	Zero Shot	GPT-3.5-Turbo	65.2%	40%	-	13.07%	37.91%	19.71%
			GPT-4		95%	**+29.8**	27.30%	51.41%	42.42%
		One Shot	GPT-3.5-Turbo		59.32%	-	43.71%	43.91%	50.33%
			GPT-4		95%	**+29.8**	51.66%	66.29%	66.93%
		Few Shot	GPT-3.5-Turbo		53.63%	-	52.17%	41.40%	52.89%
			GPT-4		**93.88%**	**+28.68**	**64.23%**	**73.46%**	**76.27%**
	Resource Leak	**Zero Shot**	**GPT-3.5-Turbo**	53.8%	**63.33%**	**+9.53**	**80%**	**75%**	**70.69%**
			GPT-4		60%	**+6.2%**	60%	80%	60%
		One Shot	GPT-3.5-Turbo		40%	-	60%	50%	48%
			GPT-4		60%	**+6.2%**	50%	75%	54.54%
		Few Shot	GPT-3.5-Turbo		50%	-	80%	50%	61.53%
			GPT-4		60%	**+6.2%**	60%	80%	60%
ChatGPT	Null Dereference	Zero Shot	GPT-3.5-Turbo	81.87%	70%	-	19.27%	32.93%	30.22%
			GPT-4		86%	**+4.13%**	26.72%	37.45%	40.78%
		One Shot	GPT-3.5-Turbo		67%	-	30.72%	32.72%	42.26%
			GPT-4		97.5%	**+15.63**	52.72%	59.92%	68.44%
		Few Shot	GPT-3.5-Turbo		73.97%	-	38.18%	38.90%	50.36%
			GPT-4		**98.18%**	**+16.31**	**77.45%**	**80.25%**	**86.59%**

Table 5: Performance of False-Positive Warning Removal of baseline tools proposed by [33]. "Precision Imp." indicates the precision improvement of Infer after applying each of the baseline tools. "ChatGPT Recall Imp." shows ChatGPT's maximum improvement in Recall for each bug type mentioned in Table 4 compared to the baseline tool.

Baseline	Bug Type	Precision Imp.	Recall	ChatGPT Recall Imp.
Feature-based	Null Dereference	+8.26%	65.1%	+12.35%
	Resource Leak	–	–	–
DeepInferEnhance	Null Dereference	+15.13%	88.3%	-10.85%
	Resource Leak	–	–	–
GPT-C	Null Dereference	+17.47%	83.7%	-6.25%
	Resource Leak	+5.56%	64.5%	+15.5%

5.2.1 Option 1 – Infer. Table 4 shows the results of applying ChatGPT as a false-positive warning removal tool on warnings generated by Infer and ChatGPT models in RQ1.

As demonstrated, when dealing with *Null Dereference* bugs, we can enhance Infer's precision by 28.68% by employing the zero-shot strategy alongside the ChatGPT-4 model. In the case of *Resource Leak* bugs, Infer's precision can be improved by up to 9.53% when utilizing the zero-shot strategy combined with the ChatGPT-3.5-Turbo model. Furthermore, in comparison to the current baselines [33], as indicated in Table 5, our findings reveal that ChatGPT models can surpass the existing baselines in precision improvement, with a margin of at least 11.21% and 3.97% for *Null Dereference* and *Resource Leak* bugs, respectively. Also, it is worth noting that Kharkar et al. [33] did not provide the results of the feature-based logistic regression model and DeepInferEnhance for *Resource Leak* bugs. Therefore, we specified them with "–" in Table 5.

5.2.2 Option 2 – ChatGPT's results from RQ1. Table 4 also shows the result of false-positive warning removal on the warnings generated by ChatGPT models in Section 5.1. Given that these warnings may arise from various model-strategy combinations, we selected the most effective model-strategy combination for warning generation to further enhance precision through the false-positive warning removal process. Therefore, we opted for the zero-shot strategy with the ChatGPT-4 model.

Moreover, it is worth noting that no false-positive warnings were associated with *Resource Leak* issues in the warnings generated by this specific model-strategy combination. Consequently, our primary focus remains improving the false-positive warnings related to *Null Dereference* issues.

As shown in Table 4, by selecting a proper model-strategy combination, which is, in this case, the few-shot strategy with the ChatGPT-4 model, we can improve the precision by removing false-positive warnings of the warnings generated in Section 5.1 by 16.31%.

Answer to RQ2: ChatGPT can effectively remove false-positive warnings and outperform the previous state-of-the-art false-positive warning removal baselines.

6 DISCUSSIONS

6.1 Reasons for Missing Detecting Bugs

ChatGPT also has limitations, such as missing detection of bugs in RQ1. In this section, we explore the underlying reasons behind the bugs that ChatGPT models cannot detect. For our analysis, we first collect all the missing bugs that ChatGPT models cannot detect. Then, we manually examine all the missing bugs and observe the possible recurring and common patterns among them.

```
1  public DependentParameters getDependency() {
2  if (this.dependency == null) {
3   Map<String, Object> taskParamsMap =
4   JSONUtils.parseObject(this.getTaskParams(), new
5   TypeReference<Map<String, Object>>() {});
6    this.dependency = JSONUtils.parseObject((String)
7   taskParamsMap.get(Constants.DEPENDENCE),
8   DependentParameters.class);
9  }
10 return this.dependency;}
```

Ground Truth Warning: object "taskParamsMap" last assigned on line 3 could be null and is dereferenced at line 7.

ChatGPT's Explanation: The code checks if "this.dependency" is null before assigning a value to it. Therefore, there is no potential null dereference bug in this code.

Figure 2: An example of a missing *Null Dereference*.

```
1private static void registerContextBeans(ConfigurableListableBeanFactory
2beanFactory, DubboSpringInitContext context) {
3    registerSingleton(beanFactory, context);
4    registerSingleton(beanFactory,
5   context.getApplicationModel());
6    registerSingleton(beanFactory,
7   context.getModuleModel());
8}
```

Ground Truth Warning: object returned by "context.getApplicationModel()" could be null and is dereferenced by call to "registerSingleton(...)" at line 6.

ChatGPT's Explanation: The values of "context.getApplicationModel()" and "context.getModuleModel()" are being checked for null before being registered as singletons, so there is no null dereference bug in this code.

Figure 3: An example of a missing *Null Dereference* bug by ChatGPT. In this case, the "registerSingleton" is a function declared as outside of the method where the *Null Dereference* is detected. Consequently, ChatGPT attempts to establish a connection between the registering for singleton and the *Null Dereference* issue, even though they are not directly related.

6.1.1 Null Dereference. We have identified three patterns from *Null Dereference* bugs that ChatGPT models missed. Firstly, ChatGPT models struggle to distinguish *Null Dereferences* of objects within a null check for another object. This is often observed when ChatGPT models overlook objects that might become null and dereferenced later within a null check for another object. This issue becomes more pronounced when these objects have some form of relationship with one another, such as when the first object serves as an argument to a method call of the second object. Figure 2 shows an example bug that is missing by ChatGPT models. Second, we have observed that ChatGPT models sometimes make random assumptions about the method, API call, class variable, and instance variable that are usually outside the current method's scope. ChatGPT models' inability to access the implementation details of these entities hinders its ability to determine whether they can return null or not. Consequently, ChatGPT models occasionally make incorrect assumptions about the return values of such entities, leading to missing potential *Null Dereference* bugs. Figure 3 shows an example bug that ChatGPT models cannot detect for the above reason. Third, another prevalent pattern is ChatGPT models' difficulty in detecting *Null Dereferences* that occur in a single line through a chain of method invocations. While ChatGPT models may be able to detect the first object invoking a method, it often fails to identify the subsequent methods chained through method calls to the initial object, which could also return null. As an example, we can look at Figure 4.

```
1private void cacheNormalInvokers(BitList<Invoker<T>>
2  invokers) {
3    BitList<Invoker<T>> clonedInvokers = invokers.clone();
4   clonedInvokers.removeIf((invoker) ->
5   invoker.getUrl().getProtocol().equals(MOCK_PROTOCOL));
6   normalInvokers = clonedInvokers;}
```

Ground Truth Warning: object returned by "invoker.getUrl().getProtocol()" could be null and is dereferenced at line 5.

ChatGPT's Explanation: The 'normalInvokers' object is assigned the value of 'clonedInvokers', which is a cloned version of 'invokers'. However, there is no null check for 'clonedInvokers' before assigning it to 'normalInvokers'. If 'invokers.clone()' returns null, it will result in a null dereference bug.

Figure 4: An example of a missing *Null Dereference* bug by ChatGPT. In this example, ChatGPT missed the *Null Dereference* issue in the method call chain and did not report anything about it.

6.1.2 Resource Leak. The prevalent pattern of overlooked *Resource Leak* bugs involves cases where the code partially handles the release of resources. This implies that while some resources are properly released after their use, there are still other resources that remain unreleased. Figure 5 is an example of a missing *Resource Leak* bug due to this issue.

7 THREATS TO VALIDITY

This research focuses solely on Java projects, and we do not consider projects developed in other programming languages. Therefore, conducting a similar analysis on additional languages (e.g., C, C#, and Python) can contribute to the validity of our research. Another potential challenge to the validity of our work is that the evaluation is conducted exclusively on a dataset created from warnings generated by Infer, without taking into account warnings generated by alternative static analysis tools like SpotBugs [60] or Error-Prone [17]. In this study, we opted not to include SpotBugs and Error-Prone as static bug detectors. This decision was made

```
1  private void executeForSdkArchive() {
2  try (TempDirectory tempDir = new
3  TempDirectory(getClass (). getSimpleName())) {
4   ZipFile asarZip = new
5        ZipFile (getSdkArchivePath (). get (). toFile ())
6  Path modulesPath = tempDir.getPath (). resolve (EXTRACTED_SDK);
7  try ( ZipFile modulesZip = getModulesZip(asarZip, modulesPath)) {
8        // long try-body codes ... (no release of resources)
9  } catch (Exception e) {
10       // long exception codes ... (no release of resources)
11 }}
```

Ground Truth Warning: resource of type "java.util.zip.ZipFile" acquired by call to "ZipFile(...)" at line 4 is not released after line 20.

ChatGPT's Explanation: The resources "tempDir" and "asarZip" are allocated using the try-with-resources syntax, which means they will be automatically released and closed after the try block finishes. Therefore, there is no resource leakage bug in this code.

Figure 5: An example of a missing *Resource Leak* bug by ChatGPT. Although ChatGPT correctly identifies "tempDir" as not buggy, it cannot detect the "asarZip", which is a true *Resource Leak* bug. The code does not use Java 7 try-with-resources syntax for the "asarZip" object. It only uses it for "tempDir" and also "modulesZip" in the next lines.

because these tools categorize bug types differently, and we aimed to maintain consistency in our bug classification [22]. Additionally, Infer has better precision and outperforms them in accuracy [33]. Note that, in our investigation, we examined ChatGPT models using a dataset derived from the warnings generated by Infer. While we demonstrated that ChatGPT models can improve Infer's detection and precision, we did not directly compare ChatGPT to Infer because our evaluation only covers a subset of the files and methods processed by Infer. To ensure an unbiased and equitable comparison, establishing a detection pipeline akin to Infer, with ChatGPT serving as the core detection mechanism, can enhance the credibility and validity of our research. Also, we exclusively employed LLM models from OpenAI, specifically ChatGPT-4 and ChatGPT-3.5 Turbo. It is essential to recognize that other companies have also introduced their LLM models, such as Meta's Llama2 and Google's PaLM2 and Bard. These alternative models may bring their unique features and performance characteristics, which could potentially impact the validity of our findings. Furthermore, it is important to acknowledge that the performance of ChatGPT models may exhibit variations across different sets of projects. To account for this variability and enhance the generalizability of our results, we have included a diverse array of projects from various repositories and backgrounds. Moreover, during our investigation, we refrained from conducting inter-procedural analysis due to ChatGPT token limitations, whereas Infer performs such an analysis.

8 RELATED WORK

Recently, there has been a considerable volume of research dedicated to exploring the capabilities of Large Language Models (LLMs) within the domain of Software Engineering (SE). For example, several studies focus on automated program repair [6, 12, 13, 15, 27–29, 45, 55, 70, 71]. For instance, Xia et al. [70] conducted the first empirical study to evaluate nine recent state-of-the-art LLMs for automated program repair tasks on five different repair datasets. Also, numerous studies in software testing and fuzzing utilize LLMs [9, 10, 31, 36]. For example, Deng et al. [10] proposed FuzzGPT, a novel LLM-based fuzzer that can produce unusual programs for fuzzing real-world systems. Kang et al. [31] proposed LIBRO, a framework that uses LLMs to automate test generation from general bug reports. Furthermore, some studies focus on Oracle generation [11, 50, 62]. For example, Tufano et al. [62] proposed a novel assertion generation approach using a BART transformer model. Nie et al. [50] proposed an approach for predicting the next statement in test methods that need reasoning about the code execution by utilizing CodeT5 model. There also studies that used LLMs in Requirements Engineering (RE) [24, 43, 48, 56, 67]. For example, Hey et al. [24] fine-tuned the BERT model for different requirement classification tasks. Luo et al. [43] proposed a prompt learning technique in BERT-based pre-trained models for requirement classification. In static analysis, Li et al. [37, 38] applied ChatGPT using prompt engineering to prune false-positive warnings produced by a static analyzer to improve the tool. As far as we know, this is the only study that utilized ChatGPT for static analysis and improving a static analyzer. However, we took a different approach by evaluating various versions of ChatGPT models and prompt strategies and testing them on two different static analysis tasks. In our study, we aimed to comprehensively assess ChatGPT's capabilities in various static analysis tasks (e.g., LLM-powered static bug detection), not just focusing on filtering false-positive warnings.

9 CONCLUSION

This paper conducts an empirical study that evaluates ChatGPT models for static code analysis. Our experiments can showcase the capabilities of LLMs like ChatGPT models in carrying out code analysis tasks, including static bug detection and false-positive warning removal. To generate warnings, we employed Infer, a well-established static analysis tool, on prominent open-source Java projects and projects from prior research. Subsequently, we meticulously labeled each of the generated warnings for two types of bugs: *Null Dereference* and *Resource Leak*, thereby creating a new dataset for our analytical work. We then harnessed the power of different ChatGPT models (i.e., ChatGPT-3.5 Turbo, ChatGPT-4) under different prompting strategies. Our experiments reveal that ChatGPT models can improve or outperform baseline counterparts, all while offering significant advantages in terms of reduced costs and complexity.

In the future, our research endeavors will broaden in scope as we aim to explore a wider array of LLMs, such as Meta's Llama, Google's Bard, and PaLM2. We also intend to delve into a comparative analysis between fine-tuned LLMs and LLMs that are specialized for specific tasks through prompt engineering.

10 DATA AVAILABILITY

We release the dataset and the source code [47] for our experiments for future usage and to support the replication of our study.

REFERENCES

[1] 2023. GPT-4 Technical Report. ArXiv. https://arxiv.org/abs/2303.08774 Accessed 17 Oct. 2023.

[2] Sharmin Afrose, Ya Xiao, Sazzadur Rahaman, Barton P. Miller, and Danfeng Yao. 2023. Evaluation of Static Vulnerability Detection Tools With Java Cryptographic API Benchmarks. *IEEE Transactions on Software Engineering* 49, 2 (2023), 485–497. https://doi.org/10.1109/TSE.2022.3154717

[3] Qirat Ashfaq, Rimsha Khan, and Sehrish Farooq. 2019. A comparative analysis of static code analysis tools that check java code adherence to java coding standards. In *2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE)*. IEEE, 98–103.

[4] Rohan Bavishi, Hiroaki Yoshida, and Mukul R Prasad. 2019. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *FSE 2019*. 613–624.

[5] Bhargav Nagaraja Bhatt and Carlo A. Furia. 2022. Automated repair of resource leaks in Android applications. *Journal of Systems and Software* 192 (2022), 111417. https://doi.org/10.1016/j.jss.2022.111417

[6] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-chi Cheung. 2023. A study on prompt design, advantages and limitations of chatgpt for deep learning program repair. *arXiv preprint arXiv:2304.08191* (2023).

[7] Antônio Carvalho, Welder Luz, Diego Marcílio, Rodrigo Bonifácio, Gustavo Pinto, and Edna Dias Canedo. 2020. C-3PR: A Bot for Fixing Static Analysis Violations via Pull Requests. In *SANER 2020*. 161–171. https://doi.org/10.1109/SANER48275.2020.9054842

[8] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023. SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 82 (may 2023), 21 pages. https://doi.org/10.1145/3542948

[9] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via large language models. *arXiv preprint arXiv:2212.14834* (2022).

[10] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014* (2023).

[11] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2130–2141.

[12] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *ICSE 2023*. 1469–1481.

[13] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Improving automatically generated code from Codex via Automated Program Repair. *arXiv preprint arXiv:2205.10583* (2022).

[14] Sidong Feng and Chunyang Chen. 2023. Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models. *arXiv preprint arXiv:2306.01987* (2023).

[15] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-Based Automated Software Vulnerability Repair *(ESEC/FSE 2022)*. 935–947.

[16] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. 2020. Memory and resource leak defects and their repairs in Java projects. *Empirical Software Engineering* 25, 1 (2020), 678–718. https://doi.org/10.1007/s10664-019-09731-8

[17] Google. 2023. *ErrorProne*. https://errorprone.info/index Accessed on Date.

[18] Qi Guo and et al. 2023. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. *arXiv* (2023). arXiv:2309.08221 https://arxiv.org/abs/2309.08221 Accessed 19 Oct. 2023.

[19] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. 2023. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5154–5188. https://doi.org/10.1109/TSE.2023.3329667

[20] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. 2014. Finding Patterns in Static Analysis Alerts: Improving Actionable Alert Ranking. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 152–161. https://doi.org/10.1145/2597073.2597100

[21] Nima S. Harzevili et al. 2023. Automatic Static Bug Detection for Machine Learning Libraries: Are We There Yet? *ArXiv* (2023). https://arxiv.org/abs/2307.04080 Accessed 18 Oct. 2023.

[22] Nima Shiri Harzevili, Jiho Shin, Junjie Wang, and Song Wang. 2022. Characterizing and Understanding Software Security Vulnerabilities in Machine Learning Libraries. *arXiv preprint arXiv:2203.06502* (2022).

[23] Sarah Heckman and Laurie Williams. 2011. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology* 53, 4 (2011), 363–387.

[24] Tobias Hey, Jan Keim, Anne Koziolek, and Walter F. Tichy. 2020. NoRBERT: Transfer Learning for Requirements Classification. In *RE 2020*. 169–179.

[25] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620* (2023).

[26] Infer. [n. d.]. *Infer official website*. https://fbinfer.com/

[27] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1430–1442.

[28] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263* (2023).

[29] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. *Proceedings of the AAAI Conference on Artificial Intelligence* 37, 4 (Jun. 2023), 5131–5140.

[30] Hong Jin Kang, Khai Loong Aw, and David Lo. 2022. Detecting False Alarms from Automatic Static Analysis Tools: How Far Are We? *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 698–709.

[31] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *ICSE 2023*. IEEE, 2312–2323.

[32] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. 2021. Lightweight and modular resource leak verification *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 181–192.

[33] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin Clement, and Neel Sundaresan. 2022. Learning to reduce false positives in analytic bug detectors. In *Proceedings of the 44th International Conference on Software Engineering*. 1307–1316.

[34] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 22199–22213.

[35] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2022. NPEX: repairing Java null pointer exceptions without tests. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1532–1544. https://doi.org/10.1145/3510003.3510186

[36] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[37] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2107–2111. https://doi.org/10.1145/3611643.3613078

[38] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models. *arXiv preprint arXiv:2308.00245* (2023).

[39] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95.

[40] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: memory leak detection using partial call-path analysis *(ESEC/FSE 2020)*. 1621–1625.

[41] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection *(ISSTA 2022)*. 544–555.

[42] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyandè. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *SANER 2019*. 1–12. https://doi.org/10.1109/SANER.2019.8667970

[43] Xianchang Luo, Yinxing Xue, Zhenchang Xing, and Jiamou Sun. 2023. PRCBERT: Prompt Learning for Requirement Classification Using BERT-Based Pretrained Language Models *(ASE '22)*. Article 75, 13 pages.

[44] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube *(ICPC'19)*. 209–219.

[45] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *MSR 2021*. 505–509.

[46] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2023. Recent Advances in Natural Language Processing via Large Pre-Trained Language Models: A Survey. *ACM Comput. Surv.* 56, 2, Article 30 (sep 2023), 40 pages.

[47] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. 2024. Replication Package for "Effectiveness of ChatGPT for Static Analysis: How Far Are We?". https:

//doi.org/10.5281/zenodo.10828316
[48] Ambarish Moharil and Arpit Sharma. 2022. Identification of Intra-Domain Ambiguity using Transformer-based Machine Learning. In *NLBSE 2022*. 51–58.
[49] Tukaram Muske and Alexander Serebrenik. 2020. Techniques for Efficient Automated Elimination of False Positives. In *SCAM 2020*. 259–263.
[50] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. *arXiv preprint arXiv:2302.10166* (2023).
[51] OpenAI. 2023. *ChatGPT*. https://openai.com/blog/chatgpt Accessed on Date.
[52] OpenAI. 2023. *ChatGPT-3.5*. https://platform.openai.com/docs/models/gpt-3-5 Accessed on Date.
[53] Ya Pan, Xiuting Ge, Chunrong Fang, and Yong Fan. 2020. A Systematic Literature Review of Android Malware Detection Using Static Analysis. *IEEE Access* 8 (2020), 116363–116379. https://doi.org/10.1109/ACCESS.2020.3002842
[54] Zachary P. Reynolds, Abhinandan B. Jayanth, Ugur Koc, Adam A. Porter, Rajeev R. Raje, and James H. Hill. 2017. Identifying and Documenting False Positive Patterns Generated by Static Code Analysis Tools. In *SER&IP 2017*. 55–61.
[55] Francisco Ribeiro. 2023. Large Language Models for Automated Program Repair *(SPLASH 2023)*. 7–9.
[56] Kimya Khakzad Shahandashti, Mithila Sivakumar, Mohammad Mahdi Mohajer, Alvine B Belle, Song Wang, and Timothy C Lethbridge. 2024. Evaluating the Effectiveness of GPT-4 Turbo in Creating Defeaters for Assurance Cases. *arXiv preprint arXiv:2401.17991* (2024).
[57] Haihao Shen, Jianhong Fang, and Jianjun Zhao. 2011. EFindBugs: Effective Error Ranking for FindBugs. In *ICST 2011*. 299–308.
[58] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. 2023. Prompt Engineering or Fine Tuning: An Empirical Assessment of Large Language Models in Automated Software Engineering Tasks. *arXiv preprint arXiv:2310.10508* (2023).
[59] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2998–3009.
[60] David A. Tomassi. 2018. Bugs in the Wild: Examining the Effectiveness of Static Analyzers at Finding Real-World Bugs *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 980–982.
[61] David A. Tomassi and Cindy Rubio-González. 2021. On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions. 292–303.
[62] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating Accurate Assert Statements for Unit Test Cases Using Pretrained Transformers *(AST '22)*. 54–64.
[63] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*. 151–162.
[64] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25 (2020), 1419–1457.
[65] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291* (2023).
[66] Junjie Wang, Song Wang, and Qing Wang. 2018. Is There a "Golden" Feature Set for Static Warning Identification? An Experimental Evaluation *(ESEM '18)*. Association for Computing Machinery, New York, NY, USA, Article 17, 10 pages.
[67] Yawen Wang, Lin Shi, Mingyang Li, Qing Wang, and Yun Yang. 2020. A Deep Context-wise Method for Coreference Detection in Natural Language Requirements. In *RE 2020*. 180–191.
[68] Jason Wei and et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *arXiv* (2022). arXiv:2201.11903 Accessed 19 Oct. 2023.
[69] Tzu-Tsung Wong and Po-Yang Yeh. 2020. Reliable Accuracy Estimates from k-Fold Cross Validation. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2020), 1586–1594.
[70] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models *(ICSE '23)*. IEEE Press, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129
[71] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971.
[72] Xueqi Yang, Jianfeng Chen, Rahul Yedida, Zhe Yu, and Tim Menzies. 2021. Learning to Recognize Actionable Static Code Warnings (is Intrinsically Easy). *Empirical Softw. Engg.* 26, 3 (may 2021), 24 pages. https://doi.org/10.1007/s10664-021-09948-6
[73] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large Language Models Meet NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Toronto, Canada, 7443–7464.
[74] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *ISSTA*. 39–51.

Received 2024-04-05; accepted 2024-05-04