https://doi.org/10.1007/s11704-025-40672-2

REVIEW ARTICLE

Software defect detection using large language models: a literature review

Yu CHEN[1,2], Yi SHEN[1,2], Taiyan WANG[1,2], Shiwen OU[1,2], Ruipeng WANG[1,2], Yuwei LI[1,2], Zulie PAN[1,2]✉

1. College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China
2. Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei 230037, China

Received July 4, 2024; accepted February 24, 2025

E-mail: panzulie17@nudt.edu.cn

Abstract

As software systems grow in complexity, the importance of efficient defect detection escalates, becoming vital to maintain software quality. In recent years, artificial intelligence technology has boomed. In particular, with the proposal of Large Language Models (LLMs), researchers have found the huge potential of LLMs to enhance the performance of software defect detection. This review aims to elucidate the relationship between LLMs and software defect detection. We categorize and summarize existing research based on the distinct applications of LLMs in dynamic and static detection scenarios. Dynamic detection methods are categorized based on the different phases in which they employ LLMs, such as using them for test case generation, providing feedback guidance, and conducting output assessment. Static detection methods are classified according to whether they analyze the source code or the binary of the software under test. Furthermore, we investigate the prompt engineering and model fine-tuning strategies adopted within these studies. Finally, we summarize the emerging trend of integrating LLMs into software defect detection, identify challenges to be addressed and prospect for some potential research directions.

Keywords

software defect detection; large language models; prompt engineering; fine-tuning

1 Introduction

With the advancement of information technology, software has become integral to both our daily lives and professional activities. Nonetheless, software defects are inevitable and frequently arise from intricate factors throughout the development process. Nonetheless, the software defects are inevitable, which frequently arise from intricate factors throughout the development process. Such defects can undermine software performance, potentially causing system failures and security breaches, thereby posing significant inconveniences and risks to users. Therefore, the significance of software defect detection is increasingly valued, and efficient methods have been adopted by developers to enhance the quality, reliability, and security of software.

The rapid development of artificial intelligence, particularly in the realm of LLMs, is proceeding at an accelerated pace. These models have found extensive applications in various fields, demonstrating remarkable efficacy and transformative potential. In the field of software defect detection, when trained on extensive textual and code datasets, LLMs are capable of acquiring a deep understanding of syntactic rules and semantic knowledge. Such understanding is critical, enabling models to fully comprehend the semantics and context of codes, while also facilitating a thorough analysis of program structure and logic, thus improving the accuracy of potential defects identification. Furthermore, LLMs can produce high-quality test cases, ensuring thorough coverage of software functions to detect possible issues and defects. Consequently, LLMs demonstrate significant promise in improving software defect detection.

There have been several literature reviews summarizing the application of LLMs in the fields of software engineering and cybersecurity. Table 1 delineates the similarities and differences between our work and these existing reviews, highlighting distinctions in model network architectures and sizes, application domains of large language models, and the timeframe over which related research papers were collected. Hou et al. [1] conducted a systematic investigation into the application of LLMs throughout the entire software development life cycle, including requirements analysis, software design, implementation, testing, and maintenance. Wang et al. [2] surveyed relevant studies in the domain of software testing that involve LLMs, encompassing various language models with smaller-scale parameters. Furthermore, Zhang et al. [3] explored the use of LLMs in cybersecurity tasks, including the construction of cybersecurity-oriented LLMs and other applications. Zhou et al. [4] investigated the literature on vulnerability detection and repair using LLMs.

Table 1 Comparison of the differences between our work and other literature reviews that intersect

Literature	Published	Literature collection time period	Network architecture	LLM size	Task areas for LLM	Number of collected papers
Hou et al. [1]	2023.09	2020.01–2023.08	Decoder-only Encoder-decoder Encoder-only	117M+	Requirements engineering Software design Software development Software quality assurance Software maintenance Software management	395
Wang et al. [2]	2023.07	2019.01–2023.06	Encoder-decoder Decoder-only	117M+	Software test input generation Software repair	52
Zhang et al. [3]	2024.05	2023.01–2024.03	Decoder-only	1B+	Vulnerability detection Secure code generation Program Repair Binary IT operation Cybersecurity knowledge assistants	180
Zhou et al. [4]	2024.10	2021.01–2024.03	Decoder-only Encoder-decoder Encoder-only	60M+	Vulnerability detection Vulnerability repair	58
Our work	2024.11	2019.01–2024.11	Encoder-decoder Decoder-only	1B+	Software defect detection	72

1) **Network architecture:** [1] and [4] consider language models for Decoder-only, Encoder-decoder, and Encoder-only network architectures. However, models like BERT for Encoder-only architectures are not typically regarded as Large Language Models (LLMs) in much of the literature.

2) **Model parameter scales:** [1–3] collected language models with model sizes exceeding 117M, 117M, and 60M parameters, respectively. According to the literature [5], LLMs are defined as models with parameter counts above one billion. such as BART [6], which has 140 million parameters but fewer than 1 billion.

3) **Large language modeling application task domains:** [1,3] cover all tasks within the software engineering and cybersecurity domains, respectively. [2] focuses on software test input generation and remediation, while [4] primarily addresses vulnerability detection and remediation tasks. Notably, [2] and while [4] discuss nearly half of the literature on LLM applications in software defect or vulnerability remediation but do not emphasize techniques related to defect or vulnerability detection.

4) **Time frame of paper collection:** [1,2] collected papers through June and August 2023, respectively, while [3,4] collected papers through March 2024. We included the latest papers up to November 2024 to reflect the most relevant techniques applied in recent research, providing readers with novel insights. The in-depth summary encompasses both dynamic and static detection of software defects. In the LLM screening, we selected language models with a parameter count exceeding 1B, consistent with the definition of LLMs in the literature.

We have observed that LLMs are rarely used independently in software defect detection, which is related to the contradiction between the low interpretability of LLMs and the high accuracy requirements for software defect detection. A significant majority of researchers have integrated LLMs into traditional software defect detection methodologies. Consequently, this review is structured around the ways in which LLMs can augment and enhance traditional detection approaches. Dynamic and static analyses are fundamental methods in software defect detection. Upon reviewing the research conducted over the past two years on applying LLMs to this domain, it becomes evident that these studies predominantly fall into two categories: dynamic and static detection of software defects using LLMs.

This review collects research from the past few years concerning the application of LLMs in software defect detection, reviews current advancements in this field, and examines key technologies along with their evaluation results. The paper introduces the background knowledge about software defect detection and LLM as well as the methodology of collecting related literature. First, the dynamic detection methods of software defects utilizing LLM are presented based on the different phases in which LLM are employed, encompassing test case generation [7–43], feedback guidance [14,17,21,32,35,41,44–46], and output assessment [20,25,45,47]. Then, static detection methods are categorized according to the type of their targets, which include defect detection for software source code [48–71] and for binary [72–78]. The prompt engineering and fine-tuning techniques employed in each study are detailed to demonstrate how LLMs are integrated into their workflows. Next, we analyze the LLMs utilized in the reviewed papers, the datasets employed [52–54,57–61,79–82], the target software addressed, and the technical performance outcomes. A mind map summarizing the collected literature is presented in Fig. 1. Finally, we summarize the overall concepts and trends of existing research and propose potential directions for future LLM-based software defect detection studies. We hope this review provides a valuable reference for future LLM-based software defect detection studies.

2 Background

2.1 Software defect detection

Software defects, which are issues, errors, or hidden functional flaws in computer software or programs that impede normal operation, can

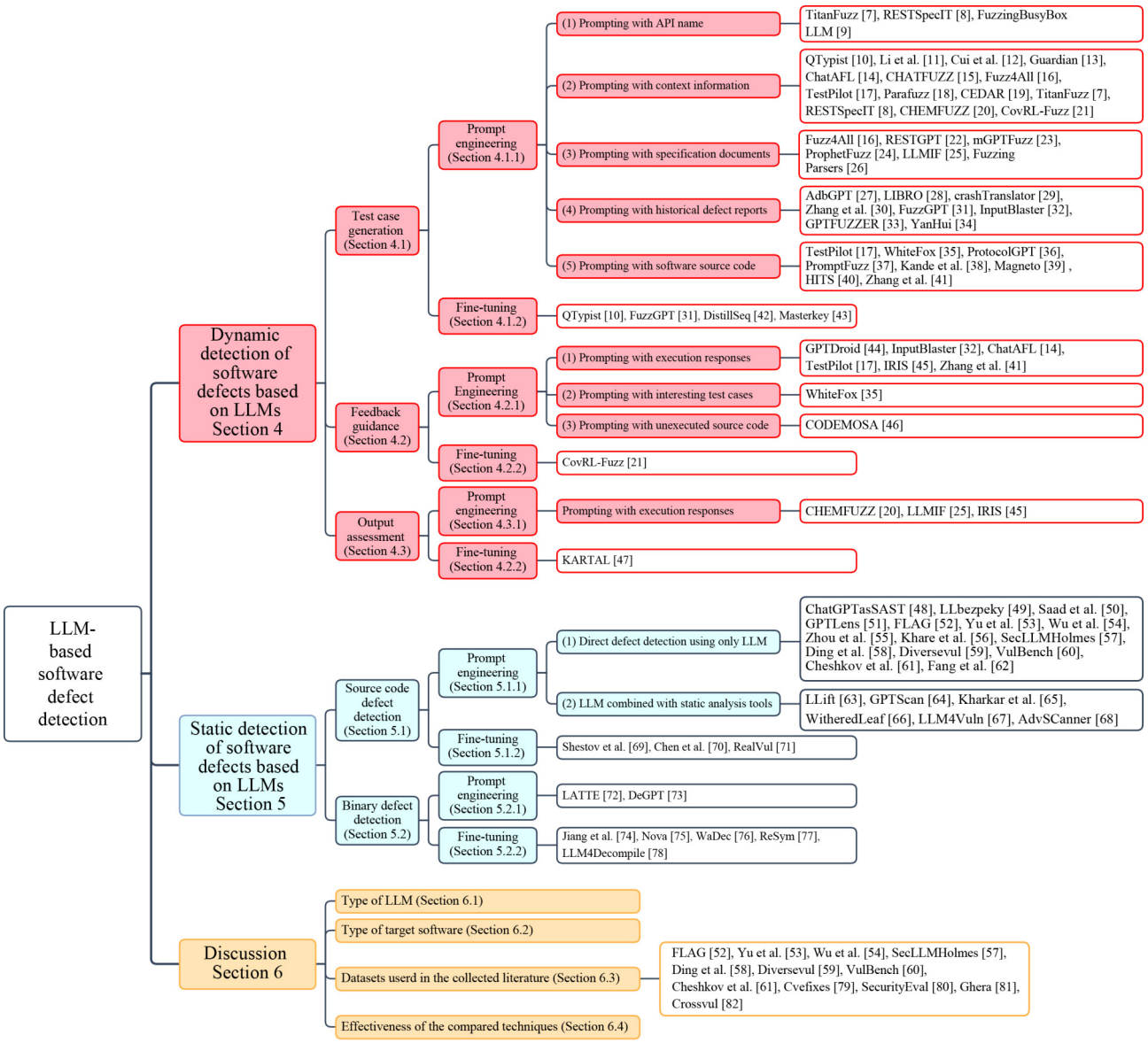

Fig. 1 Classification of software defect detection studies using LLM

have severe consequences. First, they may degrade system performance, negatively impacting user experience and potentially causing system crashes, resulting in significant losses for users. Second, software defects can be exploited maliciously, compromising user privacy and data security. Additionally, they may lead to legal issues, such as violations of user rights and interests. Therefore, software defect analysis is an essential component of the software development process, encompassing both defect detection and repair. Effective defect detection can enhance software quality and mitigate potential risks.

Software defect detection pertains to the identification and localization of flaws in computer programs or systems. There are two principal methods for this task: dynamic detection and static detection. Dynamic detection involves generating and mutating test cases during software execution to expose potential defects, whereas static detection entails analyzing the code itself to pinpoint possible issues and propose solutions.

The primary challenge encountered in dynamic defect detection is its limited coverage, which often results in the underreporting of numerous defects. Conversely, static defect detection faces challenges due to its high reliance on expert knowledge, the propensity to generate false positives, and the struggle to analyze complex and diverse code.

2.2 Large language model

LLM is a deep learning-based natural language processing technique designed to train general-purpose language generation models from large-scale textual data. The goal of such models is to understand and generate human language, thereby achieving superior performance in various natural language processing tasks. To achieve this goal, the LLM employs an architecture called Transformer [83]. The core idea of the Transformer architecture is the Self-Attention Mechanism, which allows the model to attend to information at different locations

while processing the input sequence. This mechanism enables the model to capture long-distance dependencies, improving context understanding. Additionally, Transformer introduces Multi-Head Attention and Residual Connections to further enhance the model's performance. Although there is no universally accepted figure for the dataset size required to train a large language model, these models typically have at least a billion or more parameters.

LLMs play a pivotal role in the contemporary domain of Artificial Intelligence, exhibiting their capability to comprehend and generate natural language by assimilating vast amounts of textual data. These models can be segregated into distinct categories based on their usage licenses and application areas. First, from a usage licensing perspective, LLMs can be bifurcated into commercial LLMs and open-sourced LLMs. Commercial LLMs are typically developed by enterprises, possessing model structures and algorithms that remain undisclosed to the public, and are furnished to users as a paid service, exemplified by OpenAI's GPT [84] series. Conversely, open-sourced LLMs, such as Meta's Llama 2 [85] and its derivatives, permit users to access their code and model structure, facilitating local deployment and fine-tuning of models. Second, in terms of application scope, LLMs can be classified into general-purpose LLMs and domain-specific LLMs. General-purpose LLMs are engineered to execute a broad spectrum of linguistic tasks and are not confined to any specific topic or domain, as exemplified by the GPT family of models mentioned above. In contrast, domain-specific LLMs are optimized for a particular domain to deliver more precise prediction and generation capabilities within that domain, exemplified by the code-focused models Codex [86], StarCoder [87], etc. These models, through training on large-scale code datasets, acquire knowledge of the syntactic rules, code structures, and common programming patterns of the programming language, thereby gaining the capability to interpret and generate code.

Nevertheless, LLMs encounter challenges with instability and relevance in both form and content when addressing complex task scenarios. To enhance the output quality of LLMs, researchers recommend prompt engineering and fine-tuning the model to fully leverage its learning and reasoning capabilities.

2.2.1 Prompt engineering

Prompt engineering is dedicated to the development and optimization of prompts, enabling users to utilize LLMs in a variety of scenarios and research domains. Through prompt engineering, researchers can enhance the capability of LLMs to solve intricate task situations. Common methods of prompt engineering include few-shot prompting [88], Chain-of-Thought (CoT) prompting [89], and retrieval augmented generation (RAG) [90].

Few-shot prompting. Few-shot prompting is a technique that facilitates in-context learning (ICL) by incorporating demonstrations within prompt words, thereby guiding the LLM toward enhanced performance. These demonstrations act as conditions for subsequent examples, with the expectation that the model will produce a corresponding response. This feature of few-shot prompting becomes evident when the model's size is sufficiently large.

Chain-of-thought prompting Wei et al. [89] introduced CoT prompts, which enable complex reasoning capabilities by prompting LLMs to perform intermediate reasoning steps. This can be combined with few-shot prompts to achieve better results for more complex tasks that require reasoning before answering.

Retrieval augmented generation The parameterized knowledge in LLMs is static, potentially limiting their effectiveness for complex, knowledge-intensive tasks. To address this, Meta AI researchers developed Retrieval Augmented Generation (RAG), a method that accesses external knowledge sources to supplement LLM capabilities. RAG processes input and retrieves pertinent documents, integrating them with initial prompt words before feeding them into the LLM. This process enables the LLM to access current information without necessitating retraining, thereby producing dependable output through retrieval.

2.2.2 Fine-tuning

Utilizing the aforementioned prompt engineering techniques on user prompts is intended to enhance model output consistency with user preferences. However, these techniques are not always effective, particularly for the smaller LLMs. Moreover, incorporating examples into the prompt can occupy valuable context window space, thereby limiting room for additional pertinent information. When these methods fail to address the pertinent issues, fine-tuning the LLM becomes essential. Unlike the pre-training phase that employs extensive unstructured text data, fine-tuning is a supervised learning process that involves updating the LLM's weights using a labeled dataset of examples. These labeled instances typically comprise prompt-response pairs, enhancing the model's proficiency in completing specific tasks.

3 Literature collection

This section outlines our systematic methodology for collecting literature on software defects related to large language models. The approach ensures comprehensive coverage and minimizes the risk of overlooking significant publications.

3.1 Search strategy

We reference paper [5], which introduces a statistical LLM initiated at T5 in October 2019. Consequently, our investigation concentrates on publications from 2019 onwards. To ensure a comprehensive analysis, we identified the leading conferences and journals in the fields of software engineering, artificial intelligence, and network and information security. This selection includes 16 prominent conferences such as FSE, OOPSLA, ASE, ICSE, ISSTA, CCS, S&P, USENIX Security, NDSS, ACSAC, RAID, AAAI, IJCAI, ICML, NeurIPS, and ACL, as well as four highly regarded journals, namely TOSEM, TSE, TDSC, and TIFS.

To augment our search comprehensively, automated queries were conducted across IEEE Xplore, ACM Digital Library, SpringerLink, ScienceDirect, Web of Science, and arXiv databases. A stringent series of filters was subsequently applied to extract the most pertinent studies. Additionally, snowballing techniques were utilized to broaden the scope of our findings. This meticulous methodology ensured both efficiency and exhaustive coverage, thereby minimizing the likelihood of overlooking significant works.

3.1.1 Search keywords

- **Keywords related to software defect detection:**
  Vulnerability detection | Software defect detection | Bug report | Software testing | Bug detection | Bug localization | Testing techniques | Test generation | Program analysis | Bug classification | Defect prediction | Code review | Code analysis | Fuzzing | Black-Box Testing | Jailbreak | Static analysis.
- **Keywords related to LLM:**
  LLM | Large Language Model | Few-shot learning | Zero-shot learning | Name of the LLM (e.g., GPT, T5) | Prompt engineering | In-context learning | Fine-tuning.

3.2 Literature selection

We systematically applied the designed literature inclusion criteria and conducted a literature quality assessment to sequentially select pertinent publications from the collection outlined in Section 3.1, thereby excluding those that did not satisfy the established criteria.

3.2.1 Literature inclusion criteria

To ensure the relevance and quality of the collected papers, we established specific literature inclusion criteria for screening purposes. Papers that did not adhere to these guidelines were excluded from further consideration.

- Paper pages are greater than 4.
- Paper is not a duplicate study by the same author team.
- Paper is not a literature review or survey.
- Paper describes its use of LLM rather than other deep learning techniques such as graph neural networks.
- The paper deals with tasks related to software defect detection.
- The paper does not cite LLM only in future work or discussions.
- The paper has gone through a full peer-review process, has been published in a conference or journal, and is not part of a book, presentation, etc.
- The paper uses the language model of the encoder-decoder or decoder-only network architecture, not the language model of the encoder-only architecture.

Due to the rapid advancement of LLM technology, numerous recent pre-printed papers have been posted on arXiv. We decided to include these papers given that this field is emerging and many works are in the submission process. Although these papers are not peer-reviewed, we subsequently implemented a quality assessment process to eliminate low-quality papers and ensure the overall quality of this systematic literature review. Papers that do not utilize LLMs for defect detection, such as those relying solely on deep learning methods or using LLMs exclusively for constructing software defect datasets, were excluded from consideration.

3.2.2 Literature quality assessment

We have also established comprehensive quality assessment criteria to mitigate biases associated with subpar research and to ensure the derivation of accurate conclusions for readers. Adhering to these principles, we meticulously screened the literature, retaining only those studies that demonstrated high quality through our rigorous scoring system.

- Was this study published in a reputable location?
- Does the study provide a clear description of the techniques used and is it reproducible?
- What is the experimental setup, including information on the experimental environment and dataset?
- Does this study explicitly validate the techniques evaluated and are the evaluation metrics relevant to the study objectives?
- Does this research contribute to academia or industry in the area of software defect detection?

3.3 Snowballing

Through a comprehensive literature review, we initially selected 56 papers. To identify additional relevant preliminary studies, we performed a snowball search, which yielded 1365 papers. Subsequently, a rigorous research selection process was conducted on these 1365 papers, resulting in the identification of 72 papers.

3.4 Data analysis

We ultimately collected a total of 72 high-quality papers focusing on LLM-based software defect detection. A statistical analysis of the publication timeline and sources of these papers is presented in Fig. 2. The primary sources include arXiv, ICSE, ISSTA, ASE, CCS,

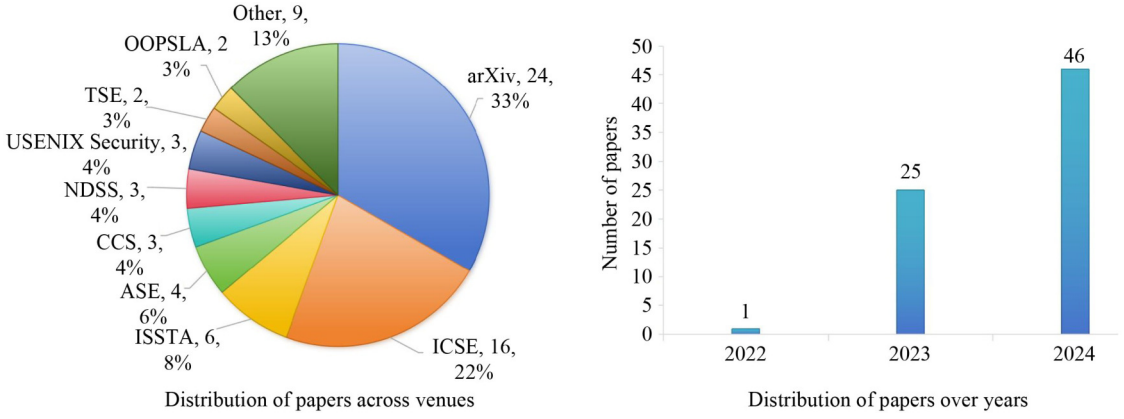

Fig. 2 Analysis of the results of the literature collection

NDSS, USENIX Security, TSE, and OOPSLA. Among these, ICSE, ISSTA, and ASE contributed the most papers, with 16, 6, and 4 publications, respectively. Notably, the earliest papers in the collection were published in 2022, coinciding with the release of ChatGPT. The volume of relevant literature has increased significantly, with 25 papers published in 2023 and 46 papers published so far in 2024. This upward trend highlights the growing research interest in LLMs as an emerging technology, suggesting their potential to further accelerate advancements in software defect detection.

4 Dynamic detection of software defects based on LLMs

In the field of software defect detection, dynamic detection serves as a pivotal technology for pinpointing these defects. The procedure for dynamically detecting software defects is delineated in Fig. 3. First, we need to generate test cases for the software under test. Subsequently, the generated test cases are executed on the software under test, and its operational behavior is monitored to capture execution results. The evaluated test oracle and result reasoning are then utilized to determine whether software defects have occurred and whether new program paths have been triggered. Finally, valuable runtime information and results are feedback to the test case generation module to guide further testing. According to the collected literature, LLMs predominantly feature in the phases of test case generation, feedback guidance, and output assessment during dynamic detection.

This section delves deeply into the role of LLM in facilitating test case generation, feedback guidance, and output assessment, while also elucidating the merits and drawbacks of various techniques. Table 2 shows the classification based on the dynamic detection phase and LLM optimization technology.

4.1 Test case generation

As a generative pre-training model, LLM possesses robust generation capabilities. Numerous researchers have employed LLMs for test case generation in the dynamic detection of software defects. Utilizing the generative capabilities of LLM, effective test cases are derived for the software under test. This section presents the techniques for generating test cases, which are based on two optimization methods for LLM: prompt engineering and fine-tuning.

4.1.1 Prompt engineering

Numerous studies employ the prompt engineering method to generate test cases for specific test scenarios. The content of these prompts varies, making them suitable for different scenarios. This section categorizes prompt words based on their content types from related works, aiming to inspire future research in generating test cases for diverse scenarios.

(1) Prompting with API name

By inputting the API name of the software under test into the LLM, the LLM can generate test cases based on its knowledge base. For example, TitanFuzz [7] prompts Codex [86] with the API name of a deep learning library to obtain seed test cases for the API. Similarly, RESTSpecIT [8] inputs the API name of a RESTful API to GPT-3.5, generating HTTP requests for RESTful API testing and subsequently mutating these requests for black-box testing. FuzzingBusyBox_LLM [9] provides GPT-4 with information about the BusyBox awk applet for defect detection. Consequently, GPT-4 generates test cases related to the BusyBox awk applet, which serve as seed test cases for the fuzz testing process.

(2) Prompting with context information

Providing the LLM with contextual information pertinent to the software under test can potentially enhance the efficacy of the generated test cases. Such contextual information encompasses the look of the graphical user interface, test case examples, and software structured input to be filled.

Look of graphical user interface. Providing the information from the look of graphical user interface (GUI) to LLM can enhance the effectiveness of generated test cases. QTypist [10] automatically creates text inputs for the GUI undergoing testing by utilizing context awareness. It derives contextual data from the GUI page that receives the text input, along with its associated view hierarchy file, and subsequently feeds this data to the LLM to produce valid text inputs for thorough testing. Li et al. [11] have evaluated the capability of LLMs in the automatic generation of Web application form tests. They found that more effective Web form tests are generated when the prompt contains complete and clear contextual information about the Web form. Cui et al. [12] assessed the impact of 9 LLMs on text input generation within the Android environment. Their research revealed that, after extracting contextual data from 114 user interface (UI) pages across 62 open-source Android applications and formulating prompts for text input generation, certain LLMs were capable of producing high-quality text inputs. These inputs had page pass rates between 50.58% and 66.67%, and they could expose actual defects in some open-source applications. Furthermore, they

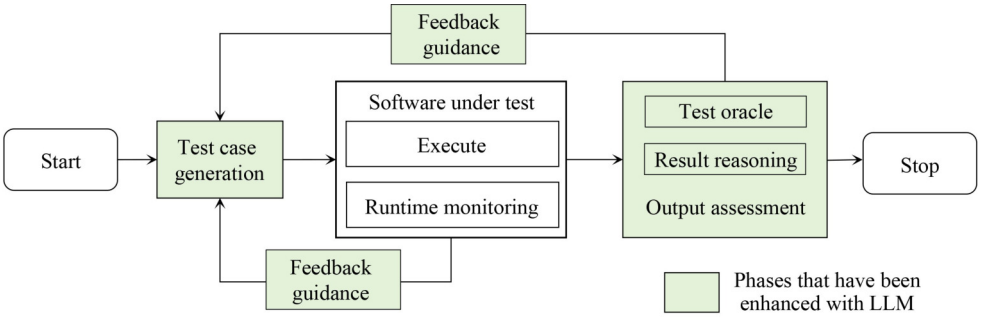

Fig. 3 Flowchart for the dynamic detection of software defects

Table 2 Classification based on dynamic detection phase and LLM optimization strategies

ID	Name	Time	Phases					Prompt engineering						Fine-tuning
			Test case generation	Feedback guidance	Output assessment	API names	Context information	Specification	Defect reports	Source code	Execution responses	Interesting test cases	Unexecuted code	
1	TitanFuzz [7]	2023-07	✓			✓	✓							
2	RESTSpecIT [8]	2024-02	✓			✓	✓							
3	FuzzingBusyBox_ LLM [9]	2024-03	✓			✓								
4	QTypist [10]	2023-05	✓				✓							✓
5	LLM4WebForm [11]	2024-05	✓				✓							
6	LLM4GuiInput [12]	2024-04	✓				✓							
7	ChatAFL [14]	2023-11	✓	✓			✓				✓			
8	CHATFUZZ [15]	2023-06	✓				✓							
9	Fuzz4All [16]	2023-08	✓				✓	✓						
10	TestPilot [17]	2023-02	✓	✓			✓			✓	✓			
11	Parafuzz [18]	2023-08	✓				✓							
12	CEDAR [19]	2023-07	✓				✓							
13	CHEMFUZZ [20]	2023-08	✓		✓		✓				✓			
14	CovRL-Fuzz [21]	2024-02	✓	✓			✓							✓
15	RESTGPT [22]	2023-06	✓					✓						
16	LLMIF [25]	2024-05	✓		✓			✓			✓			
17	Fuzzing Parsers [26]	2023-07	✓					✓						
18	AdbGPT [27]	2023-06	✓						✓					
19	LIBRO [28]	2023-05	✓						✓					
20	CrashTranslator [29]	2023-10	✓						✓					
21	LLM generate security [30]	2023-10	✓						✓					
22	FuzzGPT [31]	2023-04	✓						✓					✓
23	InputBlaster [32]	2023-10	✓	✓					✓		✓			
24	GPTFUZZER [33]	2023-09	✓						✓					
25	WhiteFox [35]	2023-10	✓	✓						✓		✓		
26	ProtocolGPT [36]	2024-05	✓							✓				
27	PromptFuzz [37]	2023-12	✓							✓				
28	Masterkey [43]	2023-07	✓											✓
29	mGPTFuzz [23]	2024-08	✓					✓						
30	Kande et al. [38]	2024-03	✓							✓				
31	Magneto [39]	2024-10	✓					✓						
32	HITS [40]	2024-10	✓					✓						
33	DistillSeq [42]	2024-09	✓											✓
34	Zhang et al. [41]	2024-09	✓	✓				✓		✓	✓			
35	YanHui [34]	2024-09	✓						✓					
36	Guardian [13]	2024-09	✓				✓							

Table 2 (Continued)

ID	Name	Time	Phases			Prompt engineering								Fine-tuning
			Test case generation	Feedback guidance	Output assessment	API names	Context information	Specification	Defect reports	Source code	Execution responses	Interesting test cases	Unexecuted code	
37	ProphetFuzz [24]	2024-10	✓					✓						
38	GPTDroid [44]	2023-05		✓							✓			
39	IRIS [45]	2024-05		✓	✓						✓			
40	CODEMOSA [46]	2023-07		✓									✓	
41	KARTAL [47]	2023-08			✓									✓

demonstrated that employing more comprehensive UI contextual information significantly enhanced the quality and efficiency of text input generation by LLMs. Guardian [13] automates UI testing with ChatGPT by optimizing action space and enabling dynamic replanning. It refines the action space to ensure LLM compliance with instructions and restores the state with replanning after invalid actions.

Test case examples. The OpenAI report [84] illustrates that by supplying concise, context-rich examples to the LLM, few-shot prompts ensure the generated output's relevance and accuracy [91]. Test case examples encapsulate comprehensive contextual information pertinent to the software under test. By incorporating seed test case demonstrations in LLM prompts, there is a notable enhancement in defect detection efficiency. ChatAFL [14] integrates the initial seed from ProFuzzBench [92] into its prompt words during the augmentation phase of initial test case seeds for protocol defect detection leveraging LLM. This initial seed from ProFuzzBench is derived by monitoring the network traffic between the client and the server under examination, thereby aiding LLM in generating varied and potent test message sequences. CHATFUZZ [15] augments grey-box fuzz testing through generative artificial intelligence. It procures seed prompts from the test case seed pool and submits them to ChatGPT. The resultant test cases from ChatGPT potentially align more closely with format syntax prerequisites. CHATFUZZ systematically reviews the seed test cases produced by ChatGPT and retains those new-coverage seeds in the seed pool. Fuzz4All [16] proposed three test case generation strategies based on LLM: generating new test cases, mutating existing test cases, and creating semantically equivalent test cases. According to the test case examples and the test case generation strategy, StarCoder is used to generate the corresponding mutated test cases. This process is repeated until the set time threshold is reached to stop the test. TestPilot [17] extracts sample code and comment fragments of the function under test from the function description document of the npm package under test, and inputs them into LLM as part of the prompt words to generate test cases. Parafuzz [18] employs LLM to identify poisoned samples in natural language processing (NLP) models. It uses ChatGPT to rewrite the input while preserving its semantics and eliminating potential triggers. The detection of poisoned samples depends on whether the model predicts their influence by the triggers. CEDAR [19] applies LLM to generate test case assertions for Java projects automatically. First, the authors collected a large number of correct test assertions and built a demonstration pool. Then, based on similarity, several demonstrations similar to the function under test are retrieved from the demonstration pool. The retrieved similar demonstrations, the function under test, and the natural language instructions are combined and provided to Codex for few-shot learning, and the test assertions corresponding to the function under test are output. Through evaluation and comparison, CEDAR outperforms the most advanced task-specific models [93] and fine-tuned models [94,95].

Software structured input to be filled. Several studies have utilized the filling capability of LLM to provide LLM with structured input of the software to be filled, thereby obtaining diverse and rich effective test cases. During the grammar-based test case generation stage for protocol defect detection using LLM, ChatAFL [14] employs masked protocol message grammar templates to prompt LLM for certain message fields, leveraging LLM's generation capabilities to populate the masked areas with pertinent information, thereby expanding the state space exploration. RESTSpecIT [8], on the other hand, masks parts of the RESTful API seed test case, such as routes and parameter values, and prompts LLM with these masked test cases to complete the masked portions of the corresponding requests, thereby enriching the generated test cases. TitanFuzz [7] uses InCoder [96] to mutate masked seed test cases, enriching the test case set and ultimately detecting defects in the deep learning library through differential testing. CHEMFUZZ [20] addresses the issue where generated test cases often fail to satisfy grammatical criteria, utilizing contextual information from quantum chemistry software. By presenting masked input files of quantum chemistry software to LLM, it leverages its chemical domain knowledge to populate the masked fields, aiding in the detection of quantum chemistry software defects. CovRL-Fuzz [21] generates test cases for JavaScript engines leveraging LLM's inherent understanding of code language context. It randomly employs insertion, rewriting, and splicing strategies to masked seed test cases and inputs them to LLM to produce grammatically accurate and contextually relevant test cases.

Utilizing contextual information from the software under test can produce more effective test cases, which in turn enhances testing efficiency and addresses the intricate constraints associated with text input and execution sequence. This approach facilitates comprehensive defect detection within the software. However, the contextual data acquired during dynamic detection is limited, perpetuating a bottleneck in the coverage of software defect

detection. Therefore, some studies have proposed providing software specification documents to LLM for more comprehensive testing.

(3) Prompting with specification documents

Most software comes with specification documents that guide users on how to use the software. These documents often contain a significant amount of content that is not machine-readable, yet this content holds valuable semantic information. Traditional methods [97] for software defect detection struggle to extract this valuable information to aid in defect detection. However, with advancements in LLMs technology, such machine-unreadable information can also be processed by LLMs to generate high-value test cases, effectively supporting software defect detection. Given that the content of the document often exceeds the token limit of the LLM context window, some studies restrict their analysis to documents within this limit. Others pre-process the documents to ensure they are within the limit, allowing for efficient prompting of the LLM with relevant content and facilitating knowledge acquisition from these standard documents.

Prompting with short specification documents directly. Fuzz4All [16] is a tool that implements defect detection for various systems, such as compilers, runtime engines, and constraint solvers, using programming languages as input. The defect detection process in Fuzz4All is divided into two stages: automatic prompting and loop fuzz testing. In the automatic prompting stage, GPT-4 is used to distill the system specification documents and obtain streamlined candidate prompts. These generated candidate prompts are then scored using heuristic rules. The optimal candidate prompts are then input into StarCoder during the loop fuzz testing stage to generate test cases that can detect software defects in the system under test. However, Fuzz4all's capability is limited to processing specification documents that are shorter than the LLM window limit [98]. The OpenAPI specification for RESTful APIs is designed to be both machine-readable, facilitating automated processes, and human-readable, ensuring clear communication of information. Each section within the specification comprises both machine-readable and human-readable segments, which are concise and can be entered within the LLM length limit window. RESTGPT [22] utilizes these specification documents as input and harnesses the robust, context-aware capabilities of LLM to augment RESTful API testing. Specifically, RESTGPT extracts machine-interpretable specifications and generates example parameter values based on the natural language descriptions provided in the specification. Subsequently, it employs these rules and values to refine the original specification, enhancing its efficacy for detecting defects in RESTful APIs. mGPTFuzz [23] leverages GPT-4 to extract key information from Matter IoT device specifications, such as data types and value ranges for commands or attributes. Using this extracted information, it generates finite-state machines (FSMs) via GPT, which are then used to construct test messages for security testing of Matter IoT devices. ProphetFuzz [24] employs LLMs to extract constraints between options from the documentation. Using program and option descriptions, it infers potential conflicts and dependencies. Through CoT prompts, ProphetFuzz guides the LLM to understand program functionality, analyze options and their impacts, avoid invalid combinations, and identify potential vulnerability-prone combinations.

Preprocessing long specification documents. The protocol specification documents adhere to a standardized format and incorporate semantically rich metadata, facilitating efficient knowledge indexing and retrieval. Notably, the document's outline features numerous informative titles. LLMIF [25] initially processes the protocol specification documents, establishing a structured hierarchy of its outline. Utilizing regular expressions, it identifies and matches description documents pertinent to the test message across all section headers. Employing background-augmented prompting [99] technology, LLMIF synthesizes prompts by integrating the extracted message description with instructions for downstream tasks, guiding the construction of the message format in accordance with the specification. LLMIF employs the LLM to augment the four distinct phases of seed generation, mutation, test case evaluation, and enrichment within the dynamic detection process, effectively addressing the issue of LLM's limited knowledge in the protocol domain. Fuzzing Parsers [26] employs a retrieval-enhanced prompting [100] approach for parser defect detection. This method involves segmenting the format specification documents of the parser into chunks that do not exceed the size of the LLM's context window. Each chunk is then embedded, and parser defects are dynamically detected using the LLM. The defect detection process comprises four phases: understand, expand, form, and deform. During each stage, the text block related to the corresponding stage query is retrieved from the document chunks using cosine similarity. This block is merged with the query as a prompt to query LLM for parser defect detection.

Most of the test cases generated by LLMs based on specification documents are standard test cases. However, the test cases that identify software defects often involve abnormal and deformed inputs. The current prompting method exhibits limitations in formulating these types of test cases.

(4) Prompting with historical defect reports

Utilizing LLMs and providing them with historical defect reports from the software under test enhances their capability to create test cases that can potentially trigger software defects. This enhanced capability is achieved through the application of a few-shot in-context learning approach.

Reproduce known defects. AdbGPT [27] automatically produces test cases from mobile application defect reports to replicate software defects. It employs LLM to extract defect reproduction step entities from the defect reports and reproduces mobile application defects using few-shot learning and chain-of-thought reasoning. Through chain-of-thought reasoning, AdbGPT can replicate mobile application defects in a manner similar to developers. LIBRO [28] prompts LLM with information like descriptions and defect stack traces in software defect reports to generate test cases that trigger corresponding software defects. It then adds dependencies to the generated test cases and executes them. LIBRO uses heuristic rules to sort and score test cases based on execution results, and selects those that successfully replicate software defects. Similarly, CrashTranslator [29] automatically reproduces mobile application

defects based on stack trace context, prompting LLM stack trace to predict exploration steps that trigger mobile application defects. Zhang et al. [30] utilizes LLM to address software supply chain security issues introduced by the use of insecure third-party software libraries. It generates test cases by prompting LLM with the security defect test codes of third-party software libraries and performs defect detection on applications that use insecure third-party libraries.

Detect unknown defects. FuzzGPT [31] is a tool predicated on the hypothesis that code snippets that trigger historical software defects may contain code components that are valuable for defect detection. This tool utilizes the historical defect data of deep learning libraries, which includes the defect-related API, the defect description, and the code snippet that triggers the defect. This data prompts LLM to automatically generate test cases that activate edge code and fuzz test the deep learning library in a contextual learning manner. Experimental evaluation indicates that FuzzGPT significantly outperforms TitanFuzz [7]. InputBlaster [32], on the other hand, employs an LLM to auto-generate abnormal text input that could potentially cause mobile application crashes. By using the in-context learning model, it provides the LLM with examples of abnormal input samples that lead to mobile application crashes. The LLM then learns to create test cases similar to these examples, which can also trigger crashes, thereby enhancing the performance of defect detection. GPTFUZZER [33] applies heuristic rules to select jailbreak templates from a collection of successfully jailbroken LLMs and prompts the LLM to generate new jailbreak templates. Successful jailbreak templates are retained for future iterations, ensuring a dynamic and evolving method to examine the robustness of LLMs. YanHui [34] detects model optimization bugs (MOBs) by using an LLM to generate test cases. It extracts code structures of historical defects (e.g., model definitions, optimization API calls, input data) and error details (e.g., exception types, locations, reasons), applies six variant strategies, and uses CoT prompting to guide the model in generating high-quality test cases for MOB testing.

(5) Prompting with software source code

Some input constraints are exclusive to the software source code and do not exist in the context of the software under test, specification documents, or other information. It is challenging for LLMs to generate test cases that adhere to these input constraints based solely on the context and specification documents. This difficulty leads to a plateau in dynamic test coverage. To address this issue, some studies have proposed prompting LLMs with relevant software source code.

TestPilot [17] conducts unit testing on npm packages by inputting the source code of the function under test into the LLM. This action enables the LLM to learn the constraints specified in the function under test from the source code, thereby generating more effective test cases. WhiteFox [35] is a pioneering white-box compiler fuzz tester that utilizes source code to interact with LLM. Initially, it employs GPT-4 to analyze the source code of the compiler optimization implementation and summarizes the necessary test case patterns. Subsequently, it prompts StarCoder to use these summarized patterns to generate meaningful test cases efficiently and continuously for the corresponding optimization. ProtocolGPT [36] infers the protocol state machine based on LLM. It introduces code filtering and splitting technology to extract source code related to the protocol state machine. It then employs text embedding technology to represent the complex protocol implementation code as a vector input for LLM analysis and parsing. Through targeted prompt engineering, the protocol implementation state machine is systematically identified and inferred, leading to the generation of effective test sequences based on the inferred state machine. PromptFuzz [37] begins by extracting information such as the signatures and type definitions of the API functions under test from the C/C++ code base. It then combines these API functions to construct prompts and guides LLM to generate test cases that call these API functions. [38] leverages Codex to prompt hardware design source code and generate accurate hardware test assertions within specific contexts, enabling effective hardware security verification. Magneto [39] employs step-wise LLM-empowered directed fuzzing to detect vulnerabilities in Java dependency libraries. It uses an LLM for step-by-step data flow reasoning along the call chain to generate test seeds, followed by hybrid feedback-directed fuzzing. HITS [40] applies LLMs to Java project unit test case generation. It uses an LLM to slice the source code of methods under test, generates test cases for each slice, executes the test cases, and leverages the LLM to repair any non-executable test cases. [41] evaluates the application of LLMs for fuzz-driven generation, revealing that enhancing queries with API documentation and sample code snippets significantly improves performance, albeit with increased token overhead.

4.1.2 Fine-tuning

In the context of defect detection for a specific scenario, LLMs encounter challenges in generating effective test cases due to their lack of domain-specific knowledge. To address this limitation, some studies have proposed fine-tuning the LLM and updating its internal parameters to facilitate the acquisition of domain-specific knowledge by the LLM.

QTypist [10] and FuzzGPT [31] employ a prompt-based fine-tuning methodology utilizing question-answer pairs to refine LLMs before the test case generation phase. QTypist [10] incorporates prompts for text input on each GUI page of the mobile application, including those found in drop-down boxes. In addition, these training data also come from the correct text inputs within the Rico [101] dataset, subsequently fine-tuning GPT-3 to enhance its knowledge pertaining to text input on mobile GUI pages, thereby improving the model's performance in this domain. FuzzGPT [31] utilizes code examples that have historically triggered defects in deep learning library code as prompt answers. This process fine-tunes CodeGen [102], broadening its capabilities in generating test cases capable of eliciting defects in deep learning libraries. DistillSeq [42] fine-tunes Vicuna-13B using the RealToxicPrompts [103] dataset to enhance its understanding and handling of malicious queries. This enables the model to generate high-quality malicious queries for dialogue extensions, supporting rigorous testing of its capacity to produce harmful content.

Masterkey [43] initially employed a set of prompt words capable of successfully executing a jailbreak on ChatGPT as fine-tuning data. Subsequently, they utilized the reward ranked fine-tuning [104] method to fine-tune the Vicuna [105] model, thereby enhancing its capability to generate innovative jailbreak prompt words.

Fine-tuning LLMs typically demands more computational resources than prompt engineering, resulting in fewer research endeavors utilizing fine-tuning methods compared to those employing prompt engineering.

4.2 Feedback guidance

Randomly generated test cases often fail to adhere to software constraints, leading to a plateau in coverage during dynamic detection. This challenge has been addressed by implementing a dynamic detection feedback guidance strategy rooted in LLMs. By supplying valuable feedback from test case execution results to LLMs, this approach significantly augments the software defect detection capability.

4.2.1 Prompt engineering

(1) Prompting with execution responses

Contrary to merely providing the LLMs with look of graphical user interface information for test case generation, iteratively adding execution responses to LLM prompts can continuously guide the optimization of test inputs. GPTDroid [44] converts the problem of automated testing of mobile application GUI into a question-answering task, provides feedback from mobile applications to LLM, and iterates the entire process to perform automated testing of mobile application GUI. InputBlaster [32] uses the execution feedback related to input of mobile applications as prompts to help LLM understand valid input specifications. ChatAFL [14] inputs the protocol communication history messages between the client and the server to LLM, learns the state transitions in the protocol implementation, and guides the generation of client request sequences that may affect the server state changes to solve the problem of coverage stagnation during dynamic detection. TestPilot [17] appends error information from executed test cases to the prompt input for LLM. This process guides the LLM to learn to generate legal test cases based on the provided error data. IRIS [45] iteratively prompts LLM with its own response output to explain its own behavior and guides the modification of the initial prompt until LLM responds with a non-rejection message, indicating that the LLM may have been successfully jailbroken. Zhang et al. [41] found that incorporating error feedback to refine queries resolved 91% of issues in fuzz-driven generation with LLMs.

(2) Prompting with interesting test cases

During each test iteration, if WhiteFox [35] identifies newly generated tests that are optimized, these tests are utilized as a candidate set for few-shot learning, and prompts are generated for future tests. By incorporating successful triggering tests into the prompts, WhiteFox can enhance the targeted guidance of the LLM, enabling it to generate more inputs to trigger the target optimization.

(3) Prompting with unexecuted source code

CODEMOSA [46] identifies the source code in Python module libraries where test coverage is stagnant and forwards it to LLM. Initially, it pinpoints the specific function call that is stagnant. Subsequently, it prompts Codex with the source code of this identified function to generate a corresponding test case. This generated test case is then employed for fuzz testing. Experimental results indicate that the proposed method effectively enhances coverage and addresses the issue of coverage stagnation, which arises when test case parameters fail to meet expected values during dynamic defect detection in Python module libraries.

4.2.2 Fine-tuning

In contrast to the studies discussed in Section 4.1.2, which involves fine-tuning a model before initiating dynamic software defect detection, the method delineated in this section synchronizes the model fine-tuning and test case generation process. This real-time fine-tuning leverages the execution results of newly generated test cases, using the feedback from these results to guide subsequent model adjustments and enhance the efficacy of the generated test cases.

CovRL-Fuzz [21] improves upon previous fine-tuning with reinforcement learning methods by utilizing reward scores derived from test case code coverage to refine CodeT5+ [106]. This approach prioritizes test cases that cover unique code areas, resulting in a more comprehensive LLM capable of exploring a broader range of code regions. Subsequently, this refined LLM generates test cases, and the process is iteratively repeated to detect defects in the javascript engine.

4.3 Output assessment

In the dynamic detection of software defects, valuable test cases and states that expose software defects can be identified by analyzing the execution output of the program under test. The specific methodologies for this process can be categorized into Result Reasoning and Test Oracle. Result Reasoning involves determining whether a new execution path or state has been explored during testing, while the Test Oracle verifies whether a software defect has been detected. However, the diversity and complexity of the execution results for the software under test present significant challenges in accurately evaluating these results. To address this issue, researchers have introduced the application of LLMs to enhance the evaluation of results in the software defect detection process.

4.3.1 Prompt engineering

By providing complex and diverse program execution result responses in prompts to the LLM, the status of the program execution can be monitored. The dynamic detection process of quantum chemistry software necessitates the assistance of domain-specific knowledge in quantum chemistry to monitor its status. This task proves challenging for both software engineers and chemical scientists. CHEMFUZZ [20], leveraging the vast training data and instruction-following capabilities of LLM, can mitigate the challenges arising from the lack of domain knowledge and monitor the status of the dynamic detection process for defects in quantum chemistry software. Furthermore, when dynamically detecting

protocol implementation software, it becomes difficult to effectively monitor and evaluate the diverse response status codes and descriptions of numerous test cases. LLMIF [25] addresses this challenge by inputting the response message descriptions, detailed information, and response status codes to the LLM. It then inquires whether a state transition has occurred, assesses if the transition is normal or abnormal in accordance with the specification, and records the corresponding test cases for future research. IRIS [45] Input the response of testing LLM jailbreak to LLM, score the jailbreak prompt, and rate the degree of harm on a scale of 1–5.

4.3.2 Fine-tuning

KARTAL [47] utilizes OWASP [107] and MITRE [108] resources to gather refined data on HTTP request and response messages containing actual security vulnerabilities. They further fine-tuning the LLM using prompt and answer pairs, allowing the LLM to monitor the status of HTTP requests and response sequences to detect potential authentication bypass vulnerabilities.

4.4 Summary

We conducted a comprehensive survey of research papers focused on dynamic software defect detection leveraging LLMs. The distribution of these papers is depicted in Fig. 4.

From the data, it is evident that:

1) Research on test case generation surpasses both feedback guidance and state monitoring in terms of volume.

2) Studies employing prompt engineering methods significantly outnumber those relying on fine-tuning techniques.

These observations suggest that LLM's superior generation capabilities are particularly favored for test case generation. Moreover, the scarcity of studies utilizing fine-tuning methods can be attributed to the high computational resource requirements associated with fine-tuning LLMs, making prompt engineering a more popular choice for optimization.

4.4.1 From the perspective of the content in the prompt

For papers utilizing the prompt engineering method, we performed a statistical analysis based on the content of the prompts. This analysis was categorized into three parts: test case generation, feedback guidance, and output assessment. The prompt content encompasses various elements such as the software API name under test, context information, specification documents, historical defect reports, software source code, execution responses, interesting test cases, and unexecuted source code. The statistical results are displayed in Fig. 5. From this, it is evident that prompt context information, historical defect reports, and execution responses constitute the majority.

Furthermore, in the papers that incorporated prompt engineering during the test case generation phase, several contained prompts with multiple content types. This is illustrated in Fig. 6, where four papers featured prompts with more than two distinct content types.

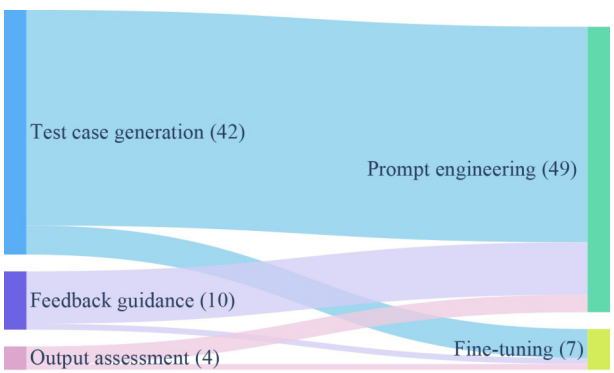

Fig. 4 Distribution map for LLM-based dynamic detection of software defects

5 Static detection of software defects based on LLMs

The application of artificial intelligence technologies, such as deep learning, for static detection of software defects has reached a level of maturity. With the rapid advancement of LLMs, researchers have extended their use to the static detection of software defects. This review categorizes existing research into defect detection for software source code and binary, based on the form of software under test, offering insights for readers. Table 3 shows the statistics for the form of software under test based on static defect detection, the LLM optimization strategy, and whether LLM is combined with a static analysis tool or not.

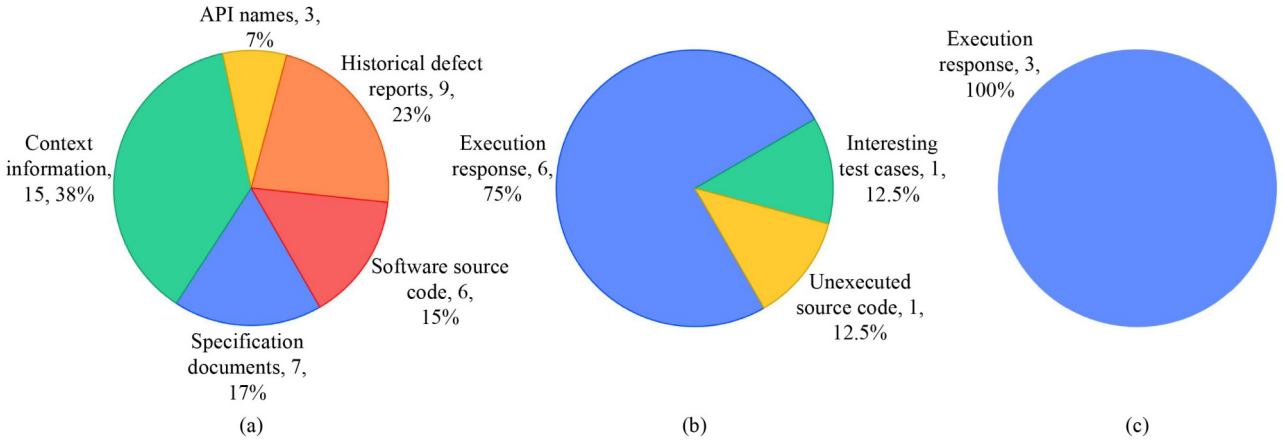

Fig. 5 Distribution of prompt content of papers applying the prompt engineering approach. (a) Test case generation; (b) feedback guidance; (c) output assessment

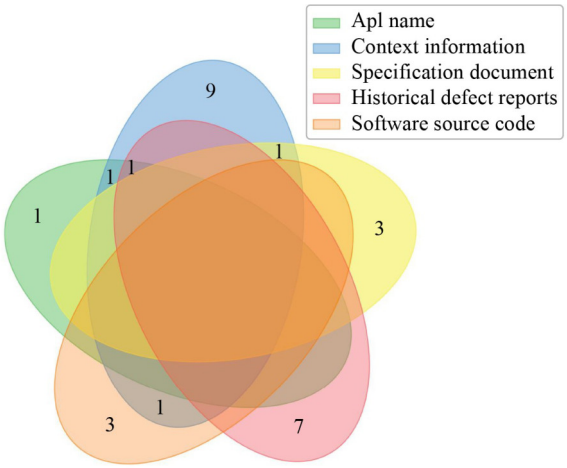

Fig. 6 Distribution of prompts in the test case generation phase applying the prompt engineering approach

5.1 Defect detection for software source code

This section primarily presents research pertaining to the detection of software defects within source code.

5.1.1 Prompt engineering

Within this field of study, scholars have suggested two primary methodologies: the direct application of LLMs for software defect detection, and the use of LLMs to augment existing static analysis tools. The subsequent discussion will delve into these two areas of research.

(1) Direct defect detection using only LLM

There are two primary methods for directly detecting software defects using prompt engineering. The first approach involves querying LLMs to check defects in the software under test by providing the software code and additional relevant content as prompts. The second method entails initially prompting the LLM with a portion of the code from the software under test, allowing the LLM to complete the code. Subsequently, the code completed by the LLM is compared with the original code of the software under test to identify any defects.

ChatGPTasSAST [48] utilizes ChatGPT for detecting security vulnerabilities in Python source code. By supplying ChatGPT with pertinent prompts and the code for analysis, its performance on two datasets is benchmarked against three prevalent static application security testing tools: Bandit [109], Semgrep [110], and SonarQube [111]. The findings indicate that ChatGPT diminishes both false positive and false negative rates. In contrast, LLbezpeky [49] employs LLM to pinpoint defects in Android applications, furnishing it with a succinct vulnerability description. Given the vast quantity of real software code, which is cumbersome to input into LLM en masse, LLbezpeky initiates a structured retrieval augmentation generation (RAG) [90] procedure. This involves generating a summary of all files in the target application, followed by prompting LLM with the content of specific files based on this summary, enhancing both efficiency and accuracy in defect detection. Ullah et al. [50] discovered through experimentation that mimicking human step-by-step reasoning aids LLM in efficiently scrutinizing code and identifying defects. Their experiments, grounded in GPT3.5 and incorporating few-shot context learning for prompt construction, guided LLM using a human-like sequential reasoning approach. This method generates reasoning outcomes based on a CoT process, subsequently making decisions rooted in these reasoning outcomes for defect identification. Providing explanations for detected defects, this approach aids users in comprehending the issues and tracing their origins. GPTLens [51] employs an adversarial framework leveraging LLM to address the challenges of low completeness and elevated false positive rates in smart contract defect detection. It bifurcates conventional smart contract defect detection into two interdependent phases: audit and critique. Within this framework, LLM undertakes dual roles as auditor and critic. The auditor aims to encompass a broad spectrum of vulnerabilities, ensuring comprehensive correct answers, while the critic assesses the validity of identified defects, striving to minimize false positives.

FLAG [52] diverges from the aforementioned techniques by employing a distinct approach. In the context of software source code testing, FLAG utilizes LLM to regenerate each line, facilitating self-comparison. Through this comparison between the original code and the LLM-generated alternatives, notable discrepancies are flagged as anomalies for subsequent examination. This process significantly narrows the search scope for defect detection.

Numerous studies have empirically evaluated the static detection capabilities of software defects using LLMs with prompt engineering, examining various prompt methods across multiple LLMs and datasets.

Studies [53–56] utilized datasets [79,112–114] from previous research to assess the static detection capabilities of LLMs for software defects. The findings indicate that LLMs possess certain defect detection capabilities and outperform traditional static analysis tools. Notably, GPT-4 can accurately identify analysis-related sources and sinks that CodeQL [115] strict queries overlook. This evaluation highlights the significant impact of prompt engineering strategies on defect detection performance. Incorporating information related to the Common Weakness Enumeration (CWE) [116] list, similar defect code examples, etc., in the prompts, or employing a step-by-step analysis prompt strategy, can substantially enhance LLM's defect detection performance. The role-oriented prompt engineering method proves superior to the task-oriented one. However, the studies also revealed several limitations in LLM-based software defect detection. Firstly, LLM's responses are often lengthy and ambiguous. Secondly, LLM's capability to handle long contexts is limited, resulting in suboptimal defect detection for software with large code volumes.

Research [57–61] has proposed new datasets to assess the static detection capability of LLMs for software defects, addressing issues such as poor data quality, low label accuracy, high duplication, and limited diversity in prior datasets. It highlights that current studies overstate LLM's defect detection capabilities, which fall short of expectations and face significant challenges. LLM often causes false positives, exhibits poor response robustness, provides inconsistent responses to multiple duplicate queries, and lacks interpretability. Even when defect judgments are accurate, the identified cause and

Table 3 Classification based on the form of the software under test for static detection, the LLM optimization strategy and whether LLM is combined with a static analysis tool

ID	Name	Time (Year-Month)	Form of software under test		Optimization strategies		Direct defect detection using only LLM	LLM combined with static analysis tools
			Source code	Binary	Prompt Engineering	Finetuning		
1	ChatGPTasSAST [48]	2023-08	✓		✓		✓	
2	LLbezpeky [49]	2024-01	✓		✓		✓	
3	Ullah et al. [50]	2023-08	✓		✓		✓	
4	GPTLens [51]	2023-10	✓		✓		✓	
5	FLAG [52]	2023-06	✓		✓		✓	
6	Yu et al. [53]	2024-01	✓		✓		✓	
7	Wu et al. [54]	2023-12	✓		✓		✓	
8	Zhou et al. [55]	2024-01	✓		✓		✓	
9	Khare et al. [56]	2023-11	✓		✓		✓	
10	Ullah et al. [57]	2023-12	✓		✓		✓	
11	Ding et al. [58]	2024-03	✓		✓		✓	
12	Diversevul [59]	2023-04	✓		✓		✓	
13	Gao et al. [60]	2023-11	✓		✓		✓	
14	Cheshkov et al. [61]	2023-04	✓		✓		✓	
15	Fang et al. [62]	2024-08	✓		✓		✓	
16	LLift [63]	2023-08	✓		✓			✓
17	GPTScan [64]	2023-08	✓		✓			✓
18	Kharkar et al. [65]	2022-03	✓		✓			✓
19	WitheredLeaf [66]	2024-05	✓		✓			✓
20	LLM4Vuln [67]	2024-01	✓		✓			✓
21	AdvSCanner [68]	2024-10	✓		✓			✓
22	Shestov et al. [69]	2024-01	✓			✓	✓	
23	Chen et al. [70]	2023-08	✓			✓	✓	
24	RealVul [71]	2024-10	✓		✓			✓
25	LATTE [72]	2023-10		✓	✓			✓
26	DeGPT [73]	2024-01		✓	✓			✓
27	BinaryAI [74]	2024-01		✓		✓		✓
28	Nova [75]	2023-11		✓		✓		✓
29	WaDec [76]	2024-10		✓		✓		✓
30	ReSym [77]	2024-10		✓		✓		✓
31	LLM4Decompile [78]	2024-03		✓		✓		✓

reasoning process may be incorrect. Furthermore, due to the restricted input window length of LLMs, their defect detection performance in complex, multi-functional, and multi-variable scenarios is suboptimal. Fang et al. [62] evaluate popular LLMs on code analysis tasks, revealing strong performance by larger models (e.g., GPT family) compared to weaker results from smaller models. While limited in handling obfuscated code, LLMs show significant potential. Future research should optimize LLM performance, build obfuscated code datasets, and improve similarity measures.

(2) LLM combined with static analysis tools

Existing static detection tools [111,115] frequently yield a high rate of false positives, necessitating substantial manual verification efforts

that are both time-consuming and costly. Utilizing LLMs exclusively for the static detection of software defects exists challenges, particularly in managing intricate and extensive code sequences. Consequently, some studies have integrated LLMs with static program analysis tools to enhance the accuracy of software defect detection.

These studies utilize LLM to filter potential defects identified by program static analysis tools, thereby eliminating false positives. LLift [63] employs LLM in conjunction with the UBITect [117] to detect and analyze use-before-initialization (UBI) errors within the Linux kernel. For each potential UBI defect identified by UBITect, LLift primarily uses LLM to perform three tasks: identifying initializer class functions, extracting post-constraint constraints, and leveraging these constraints to analyze the behavior of the initializer for detecting UBI software defects. GPTScan [64] applies LLM to evaluate potential logical defects flagged by smart contract static analysis tools. It describes various types of smart contract vulnerabilities in natural language, considering both scenario and property, and constructs prompts for GPTScan to determine if the given code aligns with the described vulnerabilities, thereby confirming their presence. Kharkar et al. [65] opted to employ LLM to augment the efficacy of existing static analysis tools, specifically to filter the defect detection outcomes of the Infer [118]. They designed an automatic code completion feature based on GPT-C [119] to aid in defect detection. This is achieved by generating completion code at specified locations and assessing whether it includes operations such as checks and resource release, which can indicate the presence of a defect. WitheredLeaf [66] harnesses LLM to identify entity inconsistency bugs (EIBs). Specifically, WitheredLeaf automatically submits code related to suspected EIB defects, detected by static analysis, to GPT-4 for investigation and verification. Subsequently, it compiles the findings into an error report for further manual inspection.

In scenarios where LLM are integrated with program analysis tools, the latter can preprocess intricate and extensive codes, extracting pertinent contextual data to assist LLM in identifying defects. LLM4Vuln [67] separates the vulnerability reasoning capabilities of LLM from its other functions and uses LLM's API call function to invoke static analysis tools. This allows for the acquisition of definitions for functions, classes, and objects, as well as data flow and control flow information. AdvSCanner [68] uses static analysis tools to extract attack streams related to smart contract reentry vulnerabilities, guiding LLMs to generate adversarial smart contracts (ASCs) for exploitation. It incorporates a self-reflective component to refine ASC generation based on compilation and execution feedback.

LLM can also be used to reduce the false negatives of program analysis tools. A comparison of the defect detection capabilities of LLM and CodeQL was conducted in research [56], revealing that GPT-4 can often accurately identify sources and sinks related to data flow analysis that are overlooked by CodeQL's strict queries. This research suggests that the capability of LLM to infer data flow analysis-related sources and sinks in the code, combined with its context understanding capabilities, should be used to assist CodeQL-type static analysis tools in achieving optimal results.

5.1.2 Fine-tuning

Evaluation studies [57–61] indicate that the basic prompt engineering technique of LLMs does not yield results significantly superior to random predictions for defect detection tasks. This finding underscores the potential of fine-tuning LLMs for this task as a promising research direction. Work [69] employed the LORA [120] fine-tuning method to adapt WizardCoder [121] to unbalanced datasets, enhancing classification performance through various techniques. The refined WizardCoder model outperformed CodeBERT-like [122] models on both balanced and unbalanced vulnerability datasets, particularly in terms of ROC AUC and F1 scores. This study highlights the significance of optimizing the training process and addressing data imbalance issues to improve model performance. It also affirms the transfer learning potential of LLMs for specific source code analysis tasks and underscores the importance of dataset quality during training. Chen et al. [70] introduced VulLibGen to identify third-party libraries with software defects. VulLibGen adopts a generation approach rather than retrieval, directly generating the version range of defective software packages based on vulnerability descriptions. It utilizes supervised fine-tuning, contextual learning, and retrieval mechanisms to enhance generation accuracy. RealVul [71] demonstrates outstanding performance in PHP vulnerability detection by leveraging comprehensive data preprocessing and efficient model fine-tuning, offering innovative insights and practical approaches for LLM-based code security research.

5.2 Defect detection for binary

The application of LLMs in detecting binary defects is mainly used in scenarios such as binary taint analysis, software component analysis, binary similarity detection and reverse engineering.

5.2.1 Prompt engineering

Existing binary taint analysis tools necessitate substantial expertise for the manual customization of taint propagation rules and vulnerability checking protocols. LATTE [72] first obtains a slice from external sensitive input to potential vulnerability trigger points. It then generates a prompt sequence based on this slice. Finally, it uses the prompt sequence to interact with the LLM, guiding it step-by-step to perform various analysis tasks and check for potential vulnerabilities in binary files. LLMs can be employed in reverse engineering tasks of binary, aiming to retrieve the source code for defect detection. DeGPT [73] optimizes the output of decompilers using LLMs by reconstructing semantic information and simplifying the structure to enhance readability and simplicity. This assists reverse engineers in comprehending binary files more effectively. To ensure that the LLM output retains the original functional semantics, a three-role mechanism (referee, consultant, operator) and micro-segment semantic computing (MSSC) technology are introduced. These mechanisms check for changes in symbol values. Since fine-tuning processes are not involved, different LLMs can be adaptively used as components. However, performance is contingent on the specific LLM deployed.

5.2.2 Fine-tuning

LLMs can be utilized in software component analysis and binary code similarity detection for software defect detection. Jiang et al. [74] uses the binary-source code matching function of LLM to perform software component analysis. This study employs the BAI [123] code-matching model, which is based on an autoregressive large language model. The model generates function vectors for source code and binary functions, facilitating similarity calculations. On the other hand, Nova [75] pre-trains the standard language modeling task on a binary corpus and uses two new pre-training tasks: optimization generation and optimization level prediction. These tasks are designed to learn binary optimization and align equivalent binary files. Nova demonstrates superior results in the binary code similarity detection task. WaDec [76] constructs a dataset of wat and C code snippets, applies preprocessing with code slicing, temporal-spatial information supplementation, and variable renaming, and fine-tunes CodeLlama-7b-hf for decompiling Wasm binary code. ReSym [77] combines LLMs with program analysis to enhance disassembled code readability. Two fine-tuned LLMs recover names and types of local variables and user-defined data structures. A Prolog-based algorithm aggregates and cross-validates multiple LLM queries to reduce uncertainty and errors.

LLM4Decompile [78] serves a similar function as DeGPT, yet it forgoes the decompilation tool and directly facilitates the mapping from binary to source code. This process is achieved by utilizing constructed training data and two designated training tasks to fine-tune the DeepSeek-Coder [124] model. As the model undergoes fine-tuning using this data and pre-training tasks, there is an enhancement in the model's understanding capability and domain knowledge content. Nonetheless, there is a significant cost associated with upgrading the model parameters in subsequent stages.

5.3 Summary

We conducted a statistical analysis of the collected papers related to static detection of software defects based on LLMs, and the results are shown in Fig 7.

In scenarios where the software source code is available, the results are similar to those of dynamic defect detection based on LLMs, with a significantly larger number of papers utilizing the prompt engineering method compared to the model fine-tuning method. However, in scenarios where only the software binary files are available, the number of studies using the fine-tuning method is slightly larger than the number of studies using the prompt engineering method. This suggests that LLMs may have a limited understanding of binary files and require fine-tuning to enhance their knowledge in the binary field for binary defect detection.

5.3.1 Whether combined with static analysis tools

We classified the collected papers into two categories based on whether they utilized static analysis tools for software defect detection. The results are displayed in Fig. 7, which shows that seven papers combined static analysis tools in their detection process. Furthermore, we categorized these papers according to the roles that the static analysis tools played in the detection process, including reducing false positive rates, handling long complex code, decompiling, and reducing false negatives. These results are shown in Fig. 8.

6 Discussion

We summarize the statistics of collected papers regarding various LLMs used, targeted software types, the datasets employed, and the technical performance outcomes.

6.1 Type of LLM

We analyzed the types of LLMs utilized in the papers addressing both dynamic and static detection of software defects, with the detailed data presented in Table 4. For dynamic defect detection using LLMs, 15 papers employed GPT-4, followed by 11 that used GPT-3.5, 7 that utilized Codex, and 4 that relied on StarCoder. A smaller number of studies adopted models such as LLaMA, GLM, Bard, CodeT5+, CodeGen, InCoder, and CodeLlama. Similarly, in static defect detection based on LLMs, GPT-4 was the most frequently used model, appearing in 16 studies, followed by GPT-3.5 in 13 studies, CodeLlama in 8 studies, and StarCoder in 4 studies.

These findings highlight that GPT-4 and GPT-3.5 dominate in both dynamic and static detection tasks, attributed to the robust

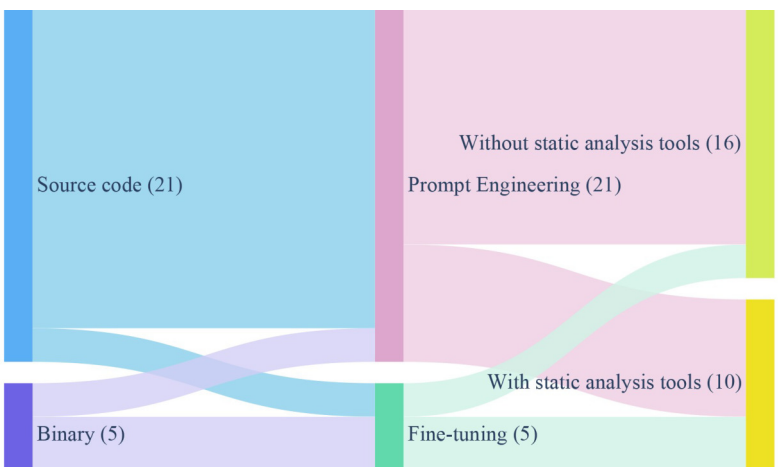

Fig. 7 Distribution map for LLM-based static detection of software defects

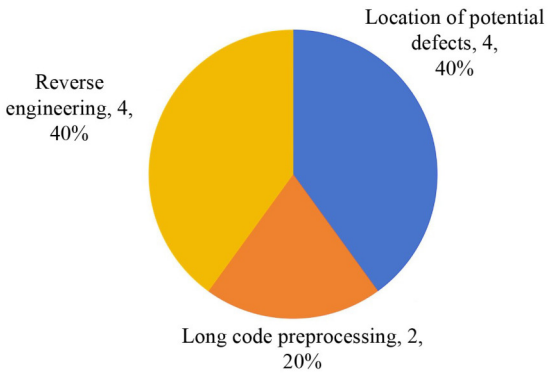

Fig. 8 Role played by static analysis tools in combination with LLMs to detect defective scenarios

capabilities of the GPT family and the cost-efficiency of utilizing commercial LLM APIs compared to locally deployed LLMs. Among open-source models, CodeLlama and StarCoder stand out as the most frequently used, reflecting their advantages in usability and performance for software defect detection.

6.2 Type of target software

We have analyzed the target software addressed in the collected papers on LLM-based dynamic and static detection of software defects, as illustrated in Fig. 9. In the literature concerning LLM-based dynamic detection, the test software primarily targets mobile applications, Java projects, intelligent models, protocol implementations, Python projects, Web applications, RESTful APIs, and compilers. Notably, mobile application GUI testing, intelligent

Table 4 Different types of LLMs used in software defect detection

Classification of detection methods	LLM	Network architecture	Size	Related papers	Open source	Release time	Field
Dynamic detection of software defects based on LLMs	GPT-4	Decoder-only	–	[9], [11], [12], [16], [26], [30], [35], [36], [45], [23], [39], [41], [34], [24]	×	2023.03	General
	GPT-3.5	Decoder-only	–	[8], [11], [12], [14], [15], [18], [20], [22], [25], [27], [32], [33], [37], [44], [40], [13]	×	2022.11	General
	GLM-3	Encoder-decoder	6B	[11], [12]	✓	2023.03	General
	Baichuan2	Decoder-only	13B	[11], [12]	✓	2023.09	General
	LLaMa2	Decoder-only	7–70B	[11], [12]	✓	2023.07	General
	Vicuna	Decoder-only	7–13B	[43], [42]	✓	2023.03	General
	Bard	Decoder-only	–	[20]	×	2023.02	General
	Codex	Decoder-only	–	[7], [17], [19], [28], [31], [46], [38]	×	2021.08	Code Domain
	StarCoder	Decoder-only	15.5B	[16]. [17], [35], [34]	×	2023.05	Code Domain
	CodeT5+	Encoder-decoder	16B	[21]	✓	2023.05	Code Domain
	CodeGen	Decoder-only	16.1B	[31]	✓	2023.01	Code Domain
	InCoder	Decoder-only	6.7B	[7]	✓	2022.04	Code Domain
	CodeLlama	Decoder-only	7–34B	[34]	✓	2023.08	Code Domain
Static detection of software defects based on LLMs	GPT-4	Decoder-only	–	[49], [51], [53], [54], [55], [56], [125], [58], [60], [63], [66], [67], [72], [68], [57], [62]	×	2023.03	General
	GPT-3.5	Decoder-only	–	[48], [50], [52], [55], [56], [125], [58], [60], [61], [64], [73], [57], [62]	×	2022.11	General
	GLM-3	Encoder-decoder	6B	[60]	✓	2023.03	General
	CodeLlama	Decoder-only	7–34B	[56], [125], [60], [67], [76], [57], [62], [71]	✓	2023.08	Code Domain
	StarCoder	Decoder-only	15.5B	[58], [77], [57], [71]	×	2023.05	Code Domain
	DeepSeek-Coder	Decoder-only	16B	[75], [78]	✓	2023.11	Code Domain
	CodeT5+	Encoder-decoder	16B	[59], [71]	✓	2023.05	Code Domain
	Codex	Decoder-only	–	[52]	×	2021.08	Code Domain
	WizardCode	Decoder-only	–	[69]	✓	2023.08	Code Domain

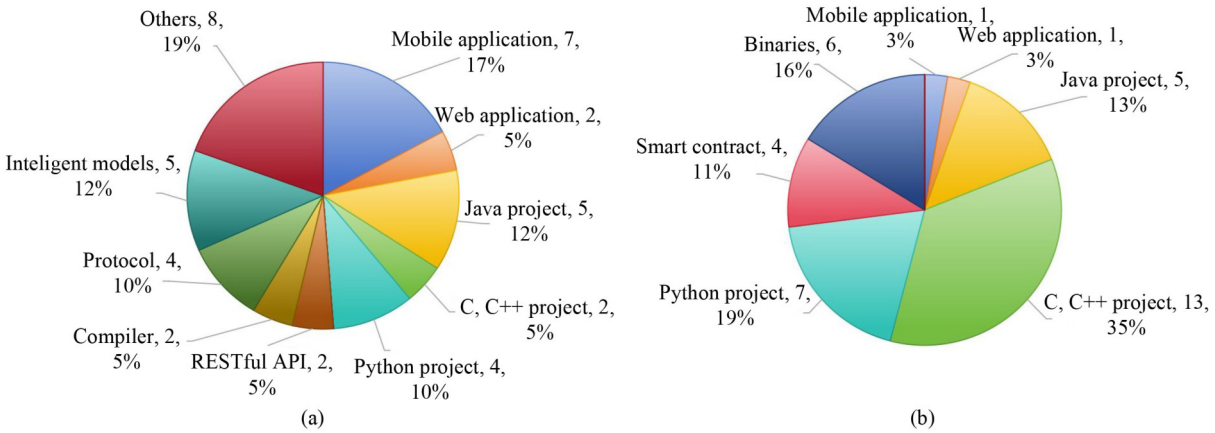

Fig. 9 Distribution of target software using LLMs to detect software defects. (a) Dynamic detection of software defects; (b) Static detection of software defects

model jailbreak testing, and protocol implementation software testing are predominantly covered. Conversely, in the literature focused on static detection of software defects using LLM, the testing software mainly addresses C/C++, Python, Java projects, binary software, smart contracts, mobile applications, and Web applications, with the majority of defects detected in C/C++, Python, and Java projects. Overall, LLMs in software defect detection cover a wide range of software scenarios. As LLMs' comprehension and reasoning abilities continue to advance, they will empower most software defect detection scenarios, significantly enhancing the efficiency of this process.

6.3 Datasets used in the collected literature

We conducted an analysis of the datasets utilized in the collected studies, with a focus on the frequently used open-source datasets. The detailed information is presented in Table 5, which outlines the programming languages included in each dataset, the dataset size, the number of defective and non-defective data entries, and the types of software defects represented. Additionally, the final column specifies whether each dataset is derived from synthetic vulnerabilities or real-world scenarios.

6.4 Effectiveness of the compared techniques

We compared and analyzed the performance of defect detection using LLMs on the same target software as reported in the collected studies. Given the larger volume of research focusing on mobile applications and deep learning library scenarios, our analysis primarily examines defect detection performance in these two contexts.

Table 5 Datasets used in the collected literature

Dataset	Language	Size	Defects/no defects	Number of software defect types (CWE)	Note
[79]	C/C++/Java	23,502	19684/13812	199	Real-World vulnerability
[126]	C/C++/Java	117,220	58610/58610	230	Synthetic
[80]	Python	130	130/0	75	Synthetic
[81]	Java	60	60/0	8	Android App Real-World vulnerability
[52]	C/C++/Python/Verilog	121	121/0	13	Security-related and functional defects
[53]	C/C++/Python	534	534/0	61	Synthetic
[54]	C/C++/Java/Python	134	134/0	6	100 Synthetic, 34 Real-World
[114]	Java	2740	1415/1325	11	Synthetic
[57]	C/C++/Python	228	228/0	8	48 Synthetic, 30 Real-World, and 150 with code augmentations
[58]	C/C++	235,768	6968/228800	140	Real-World vulnerability
[59]	C/C++	349,437	18,945/330,492	150	Real-World vulnerability
[60]	C/C++	455	455/0	9	CTF and Real World vulnerability
[61]	Java	154	154/0	40	Real-World vulnerability
[62]	Over 40 languages	27,476	13738/13738	168	Real-World vulnerability

6.4.1 Mobile application software defect detection

We categorized mobile application software defect detection into the following scenarios: GUI text input for mobile applications, GUI unusual inputs for mobile applications, mobile application defect reproduction, and mobile application GUI exploration. The performance of various methods in each scenario was analyzed, with the results presented in Table 6. The findings indicate that leveraging more comprehensive contextual information to prompt the LLM significantly improves the pass rate for GUI text input. Furthermore, the integration of In-Context Learning (ICL) and CoT techniques enhances the efficiency of detecting abnormal input defects and reproducing known defects in mobile applications. Last, mobile application GUI exploration can be effectively improved by prompting LLMs with execution responses from the application and utilizing the replanning capabilities of LLMs.

6.4.2 Deep learning library defect detection

For deep learning library defect detection, as shown in Table 7, compared to zero-shot prompting, leveraging in-context learning and fine-tuning enables LLMs to generate more diverse test

Table 6 Performance of mobile application software defect detection

Scenario	Paper	Technique	LLM	Dataset	Performance
GUI text input for mobile applications	[10]	Prompting LLM with UI contextual information	GPT-3	[10]	Achievement of 87% page through rates, 93% higher than TextExerciser [127].
	[12]	Prompting LLM with UI contextual information	GPT-3.5 GPT-4 Baichuan2 Spark LLaMa2-7B LLaMa2-13B GLM-3 GLM-4 GLM-4V	[12]	Achievement of 50.58% to 66.67% page through rates, found that using more complete UI contextual information can increase the page-pass-through rates.
GUI unusual inputs for mobile applications	[32]	Prompting LLM with sample unusual inputs that trigger defects using ICL	GPT-3.5	[32]	Achievement of 78% defect detection rate, 136% higher than TextExerciser [127].
Mobile application software defects reproduce	[27]	Prompting LLM with bug reports using CoT	GPT-3.5	[27]	Achievement of 81.3% of bug reports reproduce rate in 253.6 s.
	[29]	Combining LLM and Reinforcement learning provides test guidance	GPT-3	[29]	Achievement of 61.3% of bug reports reproduce rate.
Mobile application GUI exploration	[44]	GUI page information and execution feedback	GPT-3.5	[44]	Achievement of 71% activity coverage, with 32% higher than the best baseline, and can detect 36% more bugs with faster speed than the best baseline.
	[13]	Optimization of the action space and dynamic re-planning with LLM	GPT-3.5	[13]	Achievement of 48.3% success rate and 64.0% average completion proportion, outperforming state-of-the-art approaches Reflexion with 154%.

Table 7 Performance of deep learning library defect detection

Paper	Technique	LLM	Code coverage		API coverage		Bug found	Performance
			TensorFlow	PyTorch	TensorFlow	PyTorch		
[7]	Using zero-shot prompting with LLMs to generate test cases	InCoder Codex	107685 (39.97%)	23823 (20.98%)	2215	1329	53	Achievement of a 24.09% / 91.11% improvement in API coverage and a 50.84% / 30.38% enhancement in code coverage over the SOTA fuzzing tools for PyTorch and TensorFlow.
[31]	Using in-context learning prompts and fine-tuning with LLMs to generate test cases	Codex CodeGen	146487 (54.37%)	38284 (33.72%)	2309	1377	49	Achievement of 60.70% and 36.03% higher code coverage on PyTorch and TensorFlow, respectively, compared to TitanFuzz [7].
[34]	Using CoT prompting with domain knowledge to generate test cases	GPT-4 CodeLlama StarCoder	–	–	–	–	12	Achievement of up to 13.0% improvement in valid outputs and 23.7% increase in relevant outputs, identifying more real model optimization bugs compared to FuzzGPT [31].

cases, thereby exploring different areas of the program and achieving higher API coverage and code line coverage. Note that [34] does not discuss API coverage and code coverage, so we have not reported its coverage data. The focus of YanHui [34] differs from that of TitanFuzz and FuzzGPT, which primarily target bugs such as crashes and inconsistencies. In contrast, YanHui focuses on model optimization bugs. As a result, YanHui may discover fewer bugs, but its research shows that combining knowledge-aware prompts can effectively guide LLMs in generating efficient test cases, and it discovers more model optimization bugs and generates more valid test cases than FuzzGPT on the same baseline.

7 Future prospects

Upon examining the progress in LLM-based dynamic (Section 4) and static (Section 5) detection methods for software defects, it is clear that the advancement of LLM-based defect detection techniques has been rapid and has influenced multiple phases of software defect detection, covering a wide range of software scenarios. However, there is still considerable potential for further investigation, prompting us to suggest the following future research directions.

7.1 Improving code understanding capability of LLMs

Based on the literature reviewed, most studies assume that LLMs have a certain degree of code understanding and can analyze various programming languages used in software development. Consequently, LLMs are employed for defect detection in diverse software applications. However, some of the evaluation researches suggest that existing studies may have overstated the code understanding capabilities of LLMs. Despite their potential, LLMs face challenges such as hallucinations and outdated knowledge, which significantly hinder their code understanding abilities. For example, LLMs might not recognize newly introduced programming constructs and libraries. As a result, their performance on some recent and large-scale software projects may not meet the required standards for effective defect detection.

Enhancing the code understanding capability of LLMs is crucial for significantly advancing defect detection. Therefore, a viable solution is to create higher quality and more targeted training datasets specifically designed for detecting defects in target software. Moreover, the development languages, code size, and project architecture of different types of software vary greatly. These factors should be fully considered when constructing specific datasets for fine-tuning LLMs.

7.2 Exploring defect detection in more types of software with LLMs

LLMs have shown great potential in the field of dynamic detection of software defects. Prior studies have successfully applied them to diverse software scenarios, including mobile application GUI testing, protocol implementation software, and deep learning libraries. Given that LLMs are trained on extensive, publicly available datasets encompassing a variety of programming languages used across different software development contexts, they possess the capability to comprehend software source code, generate structured inputs for the software, and interpret its output responses effectively. By capitalizing on this broad knowledge base, LLMs can analyze various software behaviors and characteristics, positioning them as a valuable tool for defect detection in certain software or as an indirectly supportive instrument.

Existing studies cover only a small part of this area. The application of LLMs for dynamic detection of software defects is still in its early stages. A wide range of software types, such as operating system kernels, embedded IoT devices, and cyber-physical systems, remains to be explored and tested. Defect detection for these large and complex software systems presents unique challenges and opportunities.

7.3 Mitigating context length limitations of LLMs for defect detection

The constrained input window context length of LLMs presents a significant challenge, impeding their efficacy in software defect detection. Consequently, prevailing research primarily employs LLMs for defect detection in smaller codebases, such as synthetic programs. In numerous real-world software systems, defects often transcend multiple files and functions. Providing the complete source code of the target software to LLMs for processing augments their comprehension of the overarching software code logic, thereby facilitating the detection of defects that span across files and functions. However, for real-world software with extensive code sizes, encapsulating the entire source code within LLMs for processing becomes arduous, yielding suboptimal defect detection outcomes.

For large-scale real-world software systems with extensive codebases, there is a pressing need for appropriate techniques to represent the code and effectively input it into LLMs. Techniques such as program analysis and slicing can be highly effective in addressing this challenge. By utilizing these methods, key segments of source code can be extracted and fed into LLMs for defect detection. Furthermore, integrating approaches based on RAG or leveraging LLMs with extended context lengths can help mitigate the complexity of processing vast and intricate software code. However, an effective method still needs to be developed to adequately represent the complexities of software code so that LLMs can process it efficiently. This research direction demands urgent attention and exploration.

7.4 Implementing automated software defect repair using LLMs

LLMs are effective in identifying defects within specific software. However, the primary objective of defect identification is to correct these defects, thus rendering software defect repair using LLMs a significant research focus. Many existing studies separate the defect detection and repair processes. This segregation may impact the efficiency and precision of the subsequent repair phase and could result in the loss of contextual information essential for LLM-based defect detection. The interrelation between defect detection and repair has been largely ignored in current research. Investigating the use of LLMs for simultaneous defect detection and repair could enhance the efficiency and quality of both procedures, making it a valuable research direction.

8 Conclusions

In this work, we conducted a comprehensive survey of recent advances in applications of LLMs to software defect detection. First, we introduce concepts related to software defect detection and LLMs, discussing the connections between them. Subsequently, existing research is divided into two components: dynamic and static detection of software defects based on LLMs. For dynamic detection, we categorize them according to the stages in which LLMs can be applied, detailing methods for enhancing each stage. For static detection, we introduce methods for applying LLMs in source code and binary defect detection. Finally, we summarize the overall development of existing research and provide prospects for potential research directions.

Competing interests

The authors declare that they have no competing interests or financial conflicts to disclose.

Open Access

References

[1] Hou X, Zhao Y, Liu Y, Yang Z, Wang K, Li L, Luo X, Lo D, Grundy J, Wang H. Large language models for software engineering: a systematic literature review. ACM Transactions on Software Engineering and Methodology, 2024, 33(8): 220

[2] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: survey, landscape, and vision. IEEE Transactions on Software Engineering, 2024, 50(4): 911−936

[3] Zhang J, Bu H, Wen H, Liu Y, Fei H, Xi R, Li L, Yang Y, Zhu H, Meng D. When LLMs meet cybersecurity: a systematic literature review. Cybersecurity, 2025, 8(1): 55

[4] Zhou X, Cao S, Sun X, Lo D. Large language model for vulnerability detection and repair: literature review and the road ahead. 2024, arXiv preprint arXiv: 2404.02525

[5] Zhao W X, Zhou K, Li J, Tang T, Wang X, Hou Y, Min Y, Zhang B, Zhang J, Dong Z, Du Y, Yang C, Chen Y, Chen Z, Jiang J, Ren R, Li Y, Tang X, Liu Z, Liu Y, Nie J-Y, Wen J-R. A survey of large language models. 2023, arXiv preprint arXiv: 2303.18223

[6] Lewis M, Liu Y, Goyal N, Ghazvininejad M, Mohamed A, Levy O, Stoyanov V, Zettlemoyer L. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. 2020, 7871−7880

[7] Deng Y, Xia C S, Peng H, Yang C, Zhang L. Large language models are zero-shot fuzzers: fuzzing deep-learning libraries via large language models. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2023, 423−435

[8] Decrop A, Perrouin G, Papadakis M, Devroey X, Schobbens P-Y. You can rest now: automated specification inference and black-box testing of RESTful APIs with large language models. 2024, arXiv preprint arXiv: 2402.05102

[9] Asmita, Oliinyk Y, Scott M, Tsang R, Fang C, Homayoun H. Fuzzing BusyBox: leveraging LLM and crash reuse for embedded bug unearthing. In: Proceedings of the 33rd USENIX Security Symposium. 2024

[10] Liu Z, Chen C, Wang J, Che X, Huang Y, Hu J, Wang Q. Fill in the blank: context-aware automated text input generation for mobile GUI testing. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering. 2023, 1355−1367

[11] Li T, Cui C, Ma L, Towey D, Xie Y, Huang R. Leveraging large language models for automated web-form-test generation: an empirical study. 2024, arXiv preprint arXiv: 2405.09965

[12] Cui C, Li T, Wang J, Chen C, Towey D, Huang R. Large language models for mobile GUI text input generation: an empirical study. 2024, arXiv preprint arXiv: 2404.08948

[13] Ran D, Wang H, Song Z, Wu M, Cao Y, Zhang Y, Yang W, Xie T. Guardian: a runtime framework for LLM-based UI exploration. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024, 958−970

[14] Meng R, Mirchev M, Böhme M, Roychoudhury A. Large language model guided protocol fuzzing. In: Proceedings of the 31st Annual Network and Distributed System Security Symposium. 2024, 1−17

[15] Hu J, Zhang Q, Yin H. Augmenting greybox fuzzing with generative AI. 2023, arXiv preprint arXiv: 2306.06782

[16] Xia C, Paltenghi M, Tian J L, Pradel M, Zhang L. Fuzz4all: universal fuzzing with large language models. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024, 126

[17] Schaäfer M, Nadi S, Eghbali A, Tip F. An empirical evaluation of using large language models for automated unit test generation. IEEE Transactions on Software Engineering, 2024, 50(1): 85−105

[18] Yan L, Zhang Z, Tao G, Zhang K, Chen X, Shen G, Zhang X. ParaFuzz: an interpretability-driven technique for detecting poisoned samples in NLP. 2023, arXiv preprint arXiv: 2308.02122

[19] Nashid N, Sintaha M, Mesbah A. Retrieval-based prompt selection for code-related few-shot learning. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE). 2023, 2450−2462

[20] Qiu F, Ji P, Hua B, Wang Y. CHEMFUZZ: large language models-assisted fuzzing for quantum chemistry software bug detection. In: Proceedings of the 23rd IEEE International Conference on Software Quality, Reliability, and Security Companion (QRS-C). 2023, 103−112

[21] Eom J, Jeong S, Kwon T. Fuzzing JavaScript interpreters with coverage-guided reinforcement learning for LLM-based mutation. In:

Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024, 1656−1668

[22] Kim M, Stennett T, Shah D, Sinha S, Orso A. Leveraging large language models to improve rest API testing. In: Proceedings of the 44th ACM/IEEE International Conference on Software Engineering: New Ideas and Emerging Results. 2024, 37−41

[23] Ma X, Luo L, Zeng Q. *From one thousand pages of specification to unveiling hidden bugs*: large language model assisted fuzzing of matter IoT devices. In: Proceedings of the 33rd USENIX Conference on Security Symposium. 2024, 268

[24] Wang D, Zhou G, Chen L, Li D, Miao Y. ProphetFuzz: fully automated prediction and fuzzing of high-risk option combinations with only documentation via large language model. In: Proceedings of 2024 on ACM SIGSAC Conference on Computer and Communications Security. 2024, 735−749

[25] Wang J, Yu L, Luo X. LLMIF: augmented large language model for fuzzing IoT devices. In: Proceedings of 2024 IEEE Symposium on Security and Privacy (SP). 2024, 881−896

[26] Ackerman J, Cybenko G. Large language models for fuzzing parsers (registered report). In: Proceedings of the 2nd International Fuzzing Workshop. 2023, 31−38

[27] Feng S, Chen C. Prompting is all you need: automated android bug replay with large language models. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024, 67

[28] Kang S, Yoon J, Yoo S. Large language models are few-shot testers: exploring LLM-based general bug reproduction. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE). 2023, 2312−2323

[29] Huang Y, Wang J, Liu Z, Wang Y, Wang S, Chen C, Hu Y, Wang Q. CrashTranslator: automatically reproducing mobile application crashes directly from stack trace. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024, 18

[30] Zhang Y, Song W, Ji Z, Yao D, Meng N. How well does LLM generate security tests? 2023, arXiv preprint arXiv: 2310.00710

[31] Deng Y, Xia C S, Yang C, Zhang S D, Yang S, Zhang L. Large language models are edge-case generators: crafting unusual programs for fuzzing deep learning libraries. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024, 70

[32] Liu Z, Chen C, Wang J, Chen M, Wu B, Tian Z, Huang Y, Hu J, Wang Q. Testing the limits: unusual text inputs generation for mobile app crash detection with large language model. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024, 137

[33] Yu J, Lin X, Yu Z, Xing X. GPTFUZZER: red teaming large language models with auto-generated jailbreak prompts. 2023, arXiv preprint arXiv: 2309.10253

[34] Guan H, Bai G, Liu Y. Large language models can connect the dots: exploring model optimization bugs with domain knowledge-aware prompts. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024, 1579−1591

[35] Yang C, Deng Y, Lu R, Yao J, Liu J, Jabbarvand R, Zhang L. WhiteFox: white-box compiler fuzzing empowered by large language models. Proceedings of the ACM on Programming Languages, 2024, 8(OOPSLA2): 296

[36] Wei H, Du Z, Huang H, Liu Y, Cheng G, Wang L, Mao B. Inferring state machine from the protocol implementation via large language model. 2024, arXiv preprint arXiv: 2405.00393

[37] Lyu Y, Xie Y, Chen P, Chen H. Prompt fuzzing for fuzz driver generation. In: Proceedings of 2024 on ACM SIGSAC Conference on Computer and Communications Security. 2024, 3793−3807

[38] Kande R, Pearce H, Tan B, Dolan-Gavitt B, Thakur S, Karri R, Rajendran J. (Security) assertions by large language models. IEEE Transactions on Information Forensics and Security, 2024, 19: 4374−4389

[39] Zhou Z, Yang Y, Wu S, Huang Y, Chen B, Peng X. Magneto: a step-wise approach to exploit vulnerabilities in dependent libraries via LLM-empowered directed fuzzing. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. 2024, 1633−1644

[40] Wang Z, Liu K, Li G, Jin Z. HITS: high-coverage LLM-based unit test generation via method slicing. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. 2024, 1258−1268

[41] Zhang C, Zheng Y, Bai M, Li Y, Ma W, Xie X, Li Y, Sun L, Liu Y. How effective are they? exploring large language model based fuzz driver generation. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024, 1223−1235

[42] Yang M, Chen Y, Liu Y, Shi L. DistillSeq: a framework for safety alignment testing in large language models using knowledge distillation. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024, 578−589

[43] Deng G, Liu Y, Li Y, Wang K, Zhang Y, Li Z, Wang H, Zhang T, Liu Y. MASTERKEY: automated jailbreaking of large language model chatbots. In: Proceedings of the 31st Annual Network and Distributed System Security Symposium. 2024, 1−16

[44] Liu Z, Chen C, Wang J, Chen M, Wu B, Che X, Wang D, Wang Q. Make LLM a testing expert: bringing human-like interaction to mobile GUI testing via functionality-aware decisions. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024, 100

[45] Ramesh G, Dou Y, Xu W. GPT-4 jailbreaks itself with near-perfect success using self-explanation. In: Proceedings of 2024 Conference on Empirical Methods in Natural Language Processing. 2024, 22139−22148

[46] Lemieux C, Inala J P, Lahiri S K, Sen S. CodaMosa: escaping coverage plateaus in test generation with pre-trained large language models. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE). 2023, 919−931

[47] Sakaoğlu S. KARTAL: web application vulnerability hunting using large language models. Aalto University, Dissertation, 2023

[48] Bakhshandeh A, Keramatfar A, Norouzi A, Chekidehkhoun M M. Using ChatGPT as a static application security testing tool. 2023, arXiv preprint arXiv: 2308.14434

[49] Mathews N S, Brus Y, Aafer Y, Nagappan M, McIntosh S. LLbezpeky: leveraging large language models for vulnerability detection. 2024, arXiv preprint arXiv: 2401.01269

[50] Ullah S, Coskun A, Morari A, Pujar S, Stringhini G. Step-by-step vulnerability detection using large language models. In: Proceedings of the 32nd USENIX Security Symposium. 2023

[51] Hu S, Huang T, İlhan F, Tekin S F, Liu L. Large language model-powered smart contract vulnerability detection: new perspectives. In: Proceedings of the 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA). 2023, 297−306

[52] Ahmad B, Tan B, Karri R, Pearce H. FLAG: finding line anomalies (in code) with generative AI. 2023, arXiv preprint arXiv: 2306.12643

[53] Yu J, Liang P, Fu Y, Tahir A, Shahin M, Wang C, Cai Y. An insight into security code review with LLMs: capabilities, obstacles and influential factors. 2024, arXiv preprint arXiv: 2401.16310

[54] Wu F, Zhang Q, Bajaj A P, Bao T, Zhang N, Wang R, Xiao C, others. Exploring the limits of ChatGPT in software security applications. 2023, arXiv preprint arXiv: 2312.05275

[55] Zhou X, Zhang T, Lo D. Large language model for vulnerability detection: emerging results and future directions. In: Proceedings of the 44th ACM/IEEE International Conference on Software Engineering: New Ideas and Emerging Results. 2024, 47−51

[56] Khare A, Dutta S, Li Z, Solko-Breslin A, Alur R, Naik M. Understanding the effectiveness of large language models in detecting security vulnerabilities. 2023, arXiv preprint arXiv: 2311.16169

[57] Ullah S, Han M, Pujar S, Pearce H, Coskun A K, Stringhini G. LLMs cannot reliably identify and reason about security vulnerabilities (yet?): a comprehensive evaluation, framework, and benchmarks. In: Proceedings of 2024 IEEE Symposium on Security and Privacy. 2024, 862–880

[58] Ding Y, Fu Y, Ibrahim O, Sitawarin C, Chen X, Alomair B, Wagner D A, Ray B, Chen Y. Vulnerability detection with code language models: how far are we? 2024, arXiv preprint arXiv: 2403.18624

[59] Chen Y, Ding Z, Alowain L, Chen X, Wagner D. DiverseVul: a new vulnerable source code dataset for deep learning based vulnerability detection. In: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses. 2023, 654−668

[60] Gao Z, Wang H, Zhou Y, Zhu W, Zhang C. How far have we gone in vulnerability detection using large language models. 2023, arXiv preprint arXiv: 2311.12420

[61] Cheshkov A, Zadorozhny P, Levichev R. Evaluation of ChatGPT model for vulnerability detection. 2023, arXiv preprint arXiv: 2304.07232

[62] Fang C, Miao N, Srivastav S, Liu J, Zhang R, Fang R, Asmita, Tsang R, Nazari N, Wang H, Homayoun H. Large language models for code analysis: do LLMs really do their job? In: Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24). 2024, 829−846

[63] Li H, Hao Y, Zhai Y, Qian Z. Enhancing static analysis for practical bug detection: an LLM-integrated approach. Proceedings of the ACM on Programming Languages, 2024, 8(OOPSLA1): 111

[64] Sun Y, Wu D, Xue Y, Liu H, Wang H, Xu Z, Xie X, Liu Y. GPTScan: detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024, 166

[65] Kharkar A, Moghaddam R Z, Jin M, Liu X, Shi X, Clement C, Sundaresan N. Learning to reduce false positives in analytic bug detectors. In: Proceedings of the 44th IEEE/ACM International Conference on Software Engineering. 2022, 1307−1316

[66] Chen H, Zhang Y, Han X, Rong H, Zhang Y, Mao T, Zhang H, Wang X, Xing L, Chen X. WitheredLeaf: finding entity-inconsistency bugs with LLMs. 2024, arXiv preprint arXiv: 2405.01668

[67] Sun Y, Wu D, Xue Y, Liu H, Ma W, Zhang L, Shi M, Liu Y. LLM4Vuln: a unified evaluation framework for decoupling and enhancing LLMs' vulnerability reasoning. 2024, arXiv preprint arXiv: 2401.16185

[68] Wu Y, Xie X, Peng C, Liu D, Wu H, Fan M, Liu T, Wang H. AdvSCanner: generating adversarial smart contracts to exploit reentrancy vulnerabilities using LLM and static analysis. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. 2024, 1019−1031

[69] Shestov A, Cheshkov A, Levichev R, Mussabayev R, Zadorozhny P, Maslov E, Vadim C, Bulychev E. Finetuning large language models for vulnerability detection. 2024, arXiv preprint arXiv: 2401.17010

[70] Chen T, Li L, Zhu L, Li Z, Liang G, Li D, Wang Q, Xie T. VulLibGen: generating names of vulnerability-affected packages via a large language model. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2024, 9767−9780

[71] Cao D, Liao Y, Shang X. RealVul: can we detect vulnerabilities in web applications with LLM? In: Proceedings of 2024 Conference on Empirical Methods in Natural Language Processing. 2024, 8268−8282

[72] Liu P, Sun C, Zheng Y, Feng X, Qin C, Wang Y, Xu Z, Li Z, Di P, Jiang Y, Sun L. Harnessing the power of LLM to support binary taint analysis. 2023, arXiv preprint arXiv: 2310.08275

[73] Hu P, Liang R, Chen K. DeGPT: optimizing decompiler output with LLM. In: Proceedings of the 31st Annual Network and Distributed System Security Symposium. 2024, 1−16

[74] Jiang L, An J, Huang H, Tang Q, Nie S, Wu S, Zhang Y. BinaryAI: binary software composition analysis via intelligent binary source code matching. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024, 224

[75] Jiang N, Wang C, Liu K, Xu X, Tan L, Zhang X, Babkin P. Nova: generative language models for assembly code with hierarchical attention and contrastive learning. 2024, arXiv preprint arXiv: 2311.13721

[76] She X, Zhao Y, Wang H. WaDec: decompiling WebAssembly using large language model. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. 2024, 481−492

[77] Xie D, Zhang Z, Jiang N, Xu X, Tan L, Zhang X. ReSym: harnessing LLMs to recover variable and data structure symbols from stripped binaries. In: Proceedings of 2024 ACM SIGSAC Conference on Computer and Communications Security. 2024, 4554−4568

[78] Tan H, Luo Q, Li J, Zhang Y. LLM4Decompile: decompiling binary code with large language models. In: Proceedings of 2024 Conference on Empirical Methods in Natural Language Processing. 3473–3487

[79] Bhandari G, Naseer A, Moonen L. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering. 2021, 30−39

[80] Siddiq M L, Santos J C S. SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In: Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security. 2022, 29−33

[81] Mitra J, Ranganath V P. Ghera: a repository of android app vulnerability benchmarks. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering. 2017, 43−52

[82] Nikitopoulos G, Dritsa K, Louridas P, Mitropoulos D. Crossvul: a cross-language vulnerability dataset with commit data. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021, 1565−1569

[83] Vaswani A, Shazeer N M, Parmar N, Uszkoreit J, Jones L, Gomez A N, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. 2017, 6000−6010

[84] Achiam J, Adler S, Agarwal S, Ahmad L, Akkaya I, et al. GPT-4 technical report. 2023, arXiv preprint arXiv: 2303.08774

[85] Touvron H, Martin L, Stone K, Albert P, Almahairi A, Almahairi. Llama 2: open foundation and fine-tuned chat models. 2023, arXiv preprint arXiv: 2307.09288

[86] Chen M, Tworek J, Jun H, Yuan Q, de Oliveira Pinto H P, et al. Evaluating large language models trained on code. 2021, arXiv preprint arXiv: 2107.03374

[87] Li R, Allal L B, Zi Y, Muennighoff N, Kocetkov D, et al. StarCoder: may the source be with you! 2023, arXiv preprint arXiv: 2305.06161

[88] Touvron H, Lavril T, Izacard G, Martinet X, Lachaux M A, et al. Llama: open and efficient foundation language models. 2023, arXiv preprint arXiv: 2302.13971

[89] Wei J, Wang X, Schuurmans D, Bosma M, Chi h E H, Xia F, Le Q, Zhou D. Chain of thought prompting elicits reasoning in large language models. In: Proceedings of the 36th Conference on Neural Information Processing Systems. 2022, 1−43

[90] Lewis P, Perez E, Piktus A, Petroni F, Karpukhin V, Goyal N, Küttler H, Lewis M, Yih W, Rocktäschel T, Riedel S, Kiela D. Retrieval-augmented generation for knowledge-intensive NLP tasks. 2020, arXiv preprint arXiv: 2005.11401

[91] Liu P, Yuan W, Fu J, Jiang Z, Hayashi H, Neubig G. Pre-train, prompt, and predict: a systematic survey of prompting methods in natural language processing. ACM Computing Surveys, 2023, 55: 1−35

[92] Liu D, Pham V-T, Ernst G, Murray T C, Rubinstein B I P. State selection algorithms and their impact on the performance of stateful network protocol fuzzing. In: Proceedings of 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). 2022, 720−730

[93] Watson C, Tufano M, Moran K, Bavota G, Poshyvanyk D. On learning meaningful assert statements for unit test cases. In: Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering (ICSE). 2020, 1398−1409

[94] Mastropaolo A, Cooper N, Palacio D N, Scalabrino S, Poshyvanyk D, Oliveto R, Bavota G. Using transfer learning for code-related tasks. IEEE Transactions on Software Engineering, 2023, 49(4): 1580−1598

[95] Mastropaolo A, Scalabrino S, Cooper N, Palacio D N, Poshyvanyk D, Oliveto R, Bavota G. Studying the usage of text-to-text transfer transformer to support code-related tasks. In: Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE). 2021, 336−347

[96] Fried D, Aghajanyan A, Lin J, Wang S, Wallace E, Shi F, Zhong R, Yih S, Zettlemoyer L, Lewis M. INCODER: a generative model for code infilling and synthesis. In: Proceedings of the 11th International Conference on Learning Representations. 2023, 1−26

[97] Kim M, Corradini D, Sinha S, Orso A, Pasqua M, Tzoref-Brill R, Ceccato M. Enhancing REST API testing with NLP techniques. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2023, 1232−1243

[98] brutalsavage. Enquiry about canonical document distillation. See github.com/fuzz4all/fuzz4all/issues/4 website, 2024

[99] Luo Z, Xu C, Zhao P, Geng X, Tao C, Ma J, Lin Q, Jiang D. Augmented large language models with parametric knowledge guiding. 2023, arXiv preprint arXiv: 2305.04757

[100] Liu J, Shen D, Zhang Y, Dolan B, Carin L, Chen W. What makes good in-context examples for GPT-3? In: Proceedings of the 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures. 2021, 100−114

[101] Deka B, Huang Z, Franzen C, Hibschman J, Afergan D, Li Y, Nichols J, Kumar R. Rico: a mobile app dataset for building data-driven design applications. In: Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology. 2017, 845−854

[102] Nijkamp E, Pang B, Hayashi H, Tu L, Wang H, Zhou Y, Savarese S, Xiong C. CodeGen: an open large language model for code with multi-turn program synthesis. In: Proceedings of the 11th International Conference on Learning Representations. 2023, 1−25

[103] Gehman S, Gururangan S, Sap M, Choi Y, Smith N A. RealToxicityPrompts: evaluating neural toxic degeneration in language models. In: Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020. 2020, 3356−3369

[104] Dong H, Xiong W, Goyal D, Pan R, Diao S, Zhang J, Shum K, Zhang T. RAFT: reward rAnked finetuning for generative foundation model alignment. 2023, arXiv preprint arXiv: 2304.06767

[105] Chiang W-L, Li Z, Lin Z, Sheng Y, Wu Z, Zhang H, Zheng L, Zhuang S, Zhuang Y, Gonzalez J E, Stoica I, Xing E P. Vicuna: an open-source chatbot impressing GPT-4 with 90%* ChatGPT quality. See vicuna.lmsys.org website, 2023

[106] Wang Y, Le H, Gotmare A, Bui N, Li J, Hoi S. CodeT5+: open code large language models for code understanding and generation. In: Proceedings of 2023 Conference on Empirical Methods in Natural Language Processing. 2023, 1069−1088

[107] OWASP Foundation. OWASP Benchmark Project. See owasp.org/

www-project-benchmark website, 2024
[108] MITRE Corporation. 2024 CWE Top 25 Most Dangerous Software Weaknesses. See cwe.mitre.org/top25/archive/20242024_cwe_top25.html website, 2024
[109] Developers B. Bandit is a tool designed to find common security issues in python code. See bandit.readthedocs.io/en/latest/ website, 2024
[110] Semgrep. Semgrep. See semgrep.dev/ website, 2024
[111] SonarSource SA. Code Quality, Security & Static Analysis Tool with SonarQube | Sonar. See www.sonarsource.com/products/sonarqube website, 2024
[112] Black P E. SARD: thousands of reference programs for software assurance. See nist.gov/publications/sard-thousands-reference-programs-software-assurance website, 2017
[113] Pan S, Bao L, Xia X, Lo D, Li S. Fine-grained commit-level vulnerability type prediction by cwe tree structure. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE). 2023, 957−969
[114] OWASP-Benchmark. BenchmarkJava. See github.com/OWASP-Benchmark/BenchmarkJava website, 2024
[115] GitHub, Inc. CodeQL. See codeql.github.com website, 2024
[116] MITRE Corporation. About CWE. See cwe.mitre.org/about/index.html website, 2024
[117] Zhai Y, Hao Y, Zhang H, Wang D, Song C, Qian Z, Lesani M, Krishnamurthy S V, Yu P. UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020, 221−232
[118] Facebook, Inc. A tool to detect bugs in Java and C/C++/Objective-C code before it ships. See fbinfer.com website, 2024
[119] Svyatkovskiy A, Deng S K, Fu S, Sundaresan N. IntelliCode compose: code generation using transformer. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020, 1433−1443
[120] Hu E J, Shen Y, Wallis P, Allen-Zhu Z, Li Y, Wang S, Chen W. LoRA: low-rank adaptation of large language models. In: Proceedings of the 10th International Conference on Learning Representations. 2022, 1−13
[121] Luo Z, Xu C, Zhao P, Sun Q, Geng X, Hu W, Tao C, Ma J, Lin Q, Jiang D. Wizardcoder: empowering code large language models with EVOL-instruct. In: Proceedings of the 12th International Conference on Learning Representations. 2024, 1−21
[122] Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, others. CodeBERT: a pre-trained model for programming and natural languages. In: Proceedings of the Association for Computational Linguistics: EMNLP 2020. 2020, 1536−1547
[123] Tencent security keen lab. Design and Implementation of BinaryAI Binary Comparison Function | Semantic Matching of Functions in LLMs. See keenlab.tencent.com/zh/2023/07/13/2023-BinaryAI-update230713-release/ website, 2023
[124] Guo D, Zhu Q, Yang D, Xie Z, Dong K, Zhang W, Chen G, Bi X, Wu Y, Li Y K, Luo F, Xiong Y, Liang W. DeepSeek-coder: when the large language model meets programming - the rise of code intelligence. 2024, arXiv preprint arXiv: 2401.14196
[125] Ullah S, Han M, Pujar S, Pearce H, Coskun A , Stringhini G. Can large language models identify and reason about security vulnerabilities? not yet. arXiv preprint arXiv: 2312.12575v1
[126] National Institute of Standards and Technology. Test suites. See samate.nist.gov/SARD/test-suites website, 2024
[127] He Y, Zhang L, Yang Z, Cao Y, Lian K, Li S, Yang W, Zhang Z, Yang M, Zhang Y, Duan H. TextExerciser: feedback-driven text input exercising for android applications. In: Proceedings of 2020 IEEE Symposium on Security and Privacy (SP). 2020, 1071−1087

Yu CHEN received the master degree in 2022 from the College of Electronic Engineering, National University of Defense Technology, China. He is currently pursuing for a PhD degree at the National University of Defense Technology, China. His research interests include software system security.

Yi SHEN received the MSc degree in computer network in 2011 from the National University of Defense Technology, China. He is now an associate professor of College of Electronic Engineering, National University of Defense Technology, China. His areas of interests are web application security and program testing.

Taiyan WANG is currently pursuing for a PhD degree at National University of Defense Technology, China. His research interests include binary code similarity analysis.

Shiwen OU is currently pursuing for a PhD degree at National University of Defense Technology, China. His research interests include fuzz testing and software system security.

Ruipeng WANG is currently pursuing for a PhD degree at National University of Defense Technology, China. His research interests include vulnerability discovery, vulnerability assessment and network security.

Yuwei LI received the PhD degree in Computer Science and Technology from Zhejiang University, China in 2021. She is now an associate professor of the College of Electronic Engineering, National University of Defense Technology, China. Her research interests include system and software security, cyber security, and AI security.

Zulie PAN received the PhD degree from the Electronic Engineering Institute, China. He is currently a professor of the College of Electronic Engineering, National University of Defense Technology, China. His research interests include vulnerability discovery, network security, and computer science.