Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead

XIN ZHOU, School of Computing and Information Systems, Singapore Management University, Singapore, Singapore
SICONG CAO and XIAOBING SUN, Yangzhou University, Yangzhou, China
DAVID LO, School of Computing and Information Systems, Singapore Management University, Singapore, Singapore

The significant advancements in Large Language Models (LLMs) have resulted in their widespread adoption across various tasks within Software Engineering (SE), including vulnerability detection and repair. Numerous studies have investigated the application of LLMs to enhance vulnerability detection and repair tasks. Despite the increasing research interest, there is currently no existing survey that focuses on the utilization of LLMs for vulnerability detection and repair. In this paper, we aim to bridge this gap by offering a systematic literature review of approaches aimed at improving vulnerability detection and repair through the utilization of LLMs. The review encompasses research work from leading SE, AI, and Security conferences and journals, encompassing 43 papers published across 25 distinct venues, along with 15 high-quality preprint papers, bringing the total to 58 papers. By answering three key research questions, we aim to (1) summarize the LLMs employed in the relevant literature, (2) categorize various LLM adaptation techniques in vulnerability detection, and (3) classify various LLM adaptation techniques in vulnerability repair. Based on our findings, we have identified a series of limitations of existing studies. Additionally, we have outlined a roadmap highlighting potential opportunities that we believe are pertinent and crucial for future research endeavors.

CCS Concepts: • **Security and privacy** → **Software and application security**;

Additional Key Words and Phrases: Literature review, vulnerability detection, vulnerability repair, large language models

ACM Reference format:
Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2025. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 145 (May 2025), 31 pages.
https://doi.org/10.1145/3708522

This research/project was supported by the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.
Authors' Contact Information: Xin Zhou, School of Computing and Information Systems, Singapore Management University, Singapore, Singapore; e-mail: xinzhou.2020@phdcs.smu.edu.sg; Sicong Cao (corresponding author), Yangzhou University, Yangzhou, China; e-mail: DX120210088@yzu.edu.cn; Xiaobing Sun, Yangzhou University, Yangzhou, China; e-mail: xbsun@yzu.edu.cn; David Lo, School of Computing and Information Systems, Singapore Management University, Singapore, Singapore; e-mail: davidlo@smu.edu.sg.

ACM 1557-7392/2025/5-ART145
https://doi.org/10.1145/3708522

1 Introduction

A software vulnerability refers to a flaw or weakness in a software system that can be exploited by attackers. Recently, the number of software vulnerabilities has increased significantly [92], affecting numerous software systems. To mitigate these issues, researchers have proposed methods for automatic detection and repair of identified vulnerabilities. However, traditional techniques, such as rule-based detectors or program analysis-based repair tools, encounter challenges due to high false positive rates [85] and their inability to work for diverse types of vulnerabilities [127], respectively.

Recently, **Large Language Models (LLMs)** pre-trained on large corpus have demonstrated remarkable effectiveness across various natural language and software engineering tasks [37]. Given their recent success, researchers have proposed various LLM-based approaches to improve automated vulnerability detection and repair, demonstrating promising outcomes for both detection and repair tasks [88, 127]. LLM-based approaches for vulnerability detection and repair are increasingly attracting attention due to their potential to automatically learn features from known vulnerabilities and find/fix unseen ones. Furthermore, LLMs have the potential to utilize rich knowledge acquired from large-scale pre-training to enhance vulnerability detection and repair.

Despite the increasing research interest in utilizing LLMs for vulnerability detection and repair, to the best of our knowledge, no comprehensive literature review has yet summarized the state-of-the-art approaches, identified the limitations of current studies, and proposed future research directions in this field. To effectively chart the most promising path forward for research on the utilization of LLM techniques in vulnerability detection and repair, we conducted a **Systematic Literature Review (SLR)** to bridge this gap, providing valuable insights to the community. In this paper, we collected *58* primary studies over the last 6 years (2018–2024). We then summarized the LLMs used in these studies, classified various techniques for adapting LLMs, and discussed the limitations of existing research. Finally, we proposed a roadmap outlining future research directions and opportunities in this area.

Related Literature Reviews. Researchers have undertaken a series of research endeavors concerning **Machine Learning (ML)** for source code vulnerability detection or repair [30, 37, 53, 108, 121]. Hou et al. [37] conducted an SLR on LLMs for SE. Similarly, Zhang et al. [125] systematically reviewed the recent advancements in LLMs for SE. However, due to the extensive range of SE tasks to summarize, their review did not categorize the detailed utilization of LLMs in vulnerability detection and repair. In contrast, our literature review is more focused on vulnerability detection and repair. Ghaffarian et al. [30] studied vulnerability analysis and discovery approaches published till 2016. Lin et al. [53] reviewed vulnerability detection approaches utilizing deep learning till 2020. Wu et al. [108] investigated vulnerability detection approaches published before May 2022. However, their focus was not on LLMs and primarily covered general ML models such as RNNs. In contrast, our literature review includes papers published until March 2024 and focuses on LLMs, encompassing 37 distinct LLMs utilized in 58 relevant studies. Recently, Zhang et al. [122] reviewed learning-based automated program repair, which covered vulnerability repair as part of its study scope. In Different from Zhang et al., our literature review is more focused on LLMs and vulnerability repair: we included more recent studies, offered a detailed categorization of LLM usages in vulnerability repair, and discussed specialized future directions for vulnerability repair with LLMs.

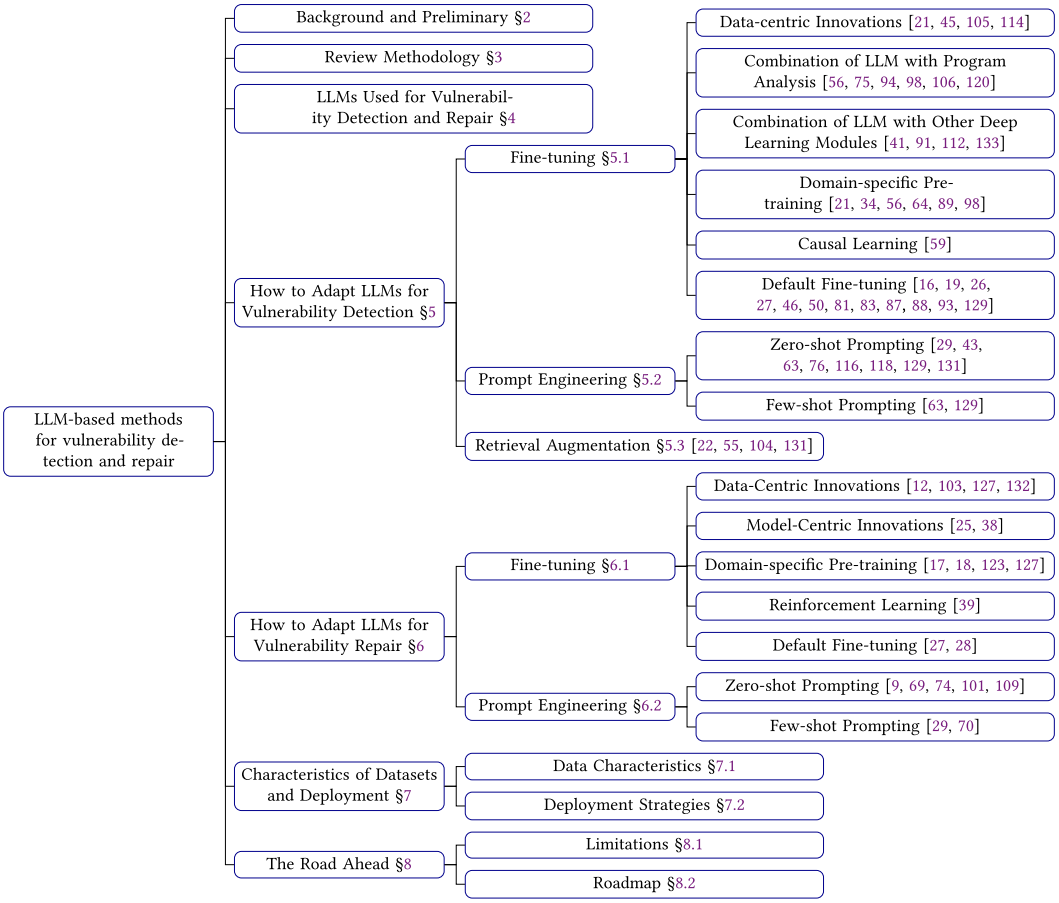

Fig. 1. Structure of this survey.

In general, this study makes the following *contributions*:

—We present a systematic review of recent *58* primary studies focusing on the utilization of LLMs for vulnerability detection and repair.
—We offer a comprehensive summary of the LLMs utilized in the relevant literature and categorize various techniques used to adapt LLMs to do the two tasks (vulnerability detection and repair).
—We discuss the limitations of existing studies on using LLMs for vulnerability detection and repair and propose a roadmap outlining future research directions and opportunities.

Survey Structure. Figure 1 outlines the structure of this paper. Section 2 provides the background information, while Section 3 details the methodology. Section 4 discusses the LLMs employed in vulnerability detection and repair. Sections 5 and 6 cover the relevant work on adapting and enhancing LLMs for vulnerability detection and repair, respectively. Section 7 discusses the characteristics of datasets and deployment in LLM-based vulnerability detection and repair studies. Finally, Section 8 discusses the limitations of current studies and presents a roadmap for future research and opportunities.

2 Background and Preliminaries

2.1 Vulnerability Detection/Repair Formulation

In this study, we focus on source code vulnerability detection and repair. Here, we present the task formulations of these two tasks.

Vulnerability Detection. Vulnerability detection is typically framed as a binary classification task: $X_i \rightarrow Y_i$. Specifically, given an input source code function X_i, a model predicts whether the input function is vulnerable ($Y_i = 1$) or non-vulnerable ($Y_i = 0$).

Vulnerability Repair. Vulnerability repair studies using LLMs frame the task as a sequence-to-sequence problem: $X_i \rightarrow Y_i$. Specifically, given a vulnerable code snippet X_i, an LLM model generates the corresponding repaired code Y_i.

2.2 LLMs

The term Large Language Model was introduced to distinguish language models based on their parameter size, specifically referring to large-sized pre-trained language models [126]. However, the literature lacks a formal consensus on the minimum parameter scale for LLMs [99]. In this paper, we adopt the LLM scope division and taxonomy introduced by Pan et al. [73] and categorize the mainstream LLMs into three groups according to their architectures: (1) encoder-only, (2) encoder-decoder, and 3) decoder-only LLMs. We will provide a brief introduction to some representative LLMs for each category due to the space limit.

Encoder-Only LLMs. Encoder-only LLMs are a type of neural network architecture that utilizes only the encoder component of the Transformer model [20]. In the SE domain, examples of encoder-only LLMs include CodeBERT [23], GraphCodeBERT [33], CuBERT [42], VulBERTa [34], CCBERT [130], SOBERT [36], and BERTOverflow [90].

Encoder-Decoder LLMs. Encoder-decoder LLMs integrate both the encoder and decoder modules of the Transformer model [96]. The encoder processes the input sentence, while the decoder generates the target output text/code. Prominent examples of encoder-decoder LLMs include PLBART [10], T5 [79], CodeT5 [102], UniXcoder [32], and NatGen [14].

Decoder-Only LLMs. Decoder-only LLMs exclusively utilize the decoder module of the Transformer model to generate the target output text/code. The GPT series, including GPT-2 [78], GPT-3 [13], GPT-3.5 [71], and GPT-4 [72], stand as prominent implementations of this model series. Additionally, in the SE domain, there are numerous decoder-only LLMs specialized for code as well. Examples include CodeGPT [58], Codex [15], Polycoder [110], Incoder [24], CodeGen series [66, 68], Copilot [31], Code Llama [60], and StarCoder [49].

3 Review Methodology

3.1 Research Question

In this paper, we focus on investigating four **Research Questions (RQs)**. The first three RQs mainly focus on identifying which LLMs are used and how they are adapted for vulnerability detection and repair:

—*RQ1: What LLMs have been utilized to solve vulnerability detection and repair tasks?*
—*RQ2: How are LLMs adapted for vulnerability detection?*
—*RQ3: How are LLMs adapted for vulnerability repair?*

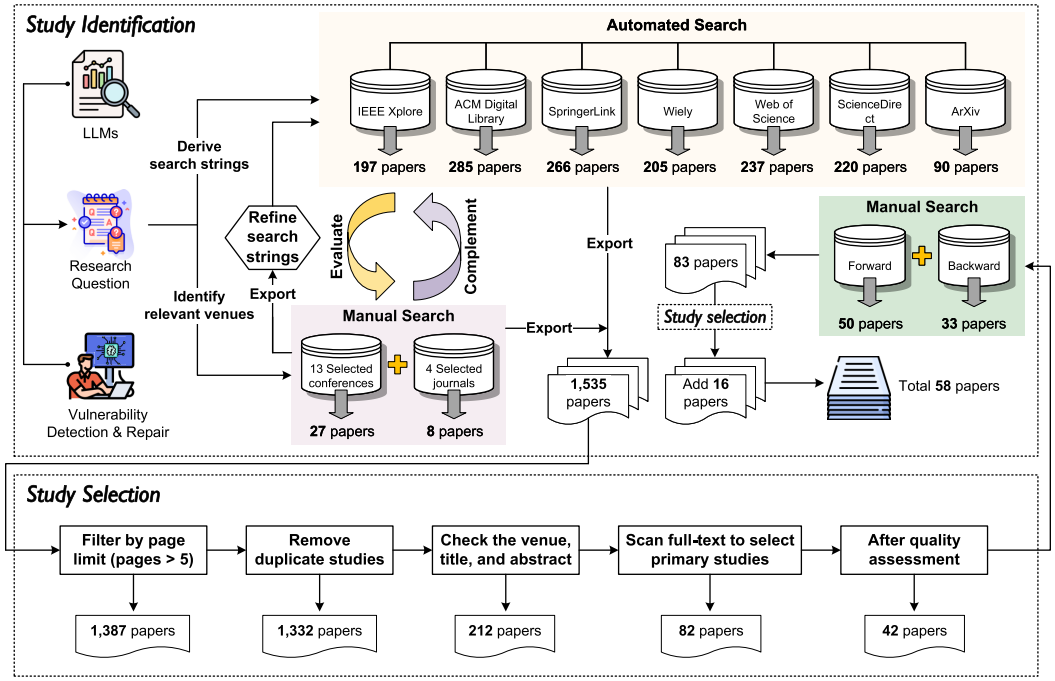

Fig. 2. Study identification and selection process.

The first three RQs pertain primarily to the model construction phase. Generally, the pipeline of a learning-based method can be divided into three phases: (1) data preparation, (2) model construction, and (3) deployment. We also investigate the characteristics of the datasets and deployment designs in LLM-based vulnerability detection and repair studies:

— *RQ4: What are the characteristics of the datasets and deployment strategies in LLM-based vulnerability detection and repair studies?*

3.2 Search Strategy

As shown in Figure 2, following the guide by Zhang et al. [119], our initial step is to identify primary studies to answer the RQs above. Because the first LLM (i.e., BERT [20]) was introduced in 2018, our search focused on papers published from 2018 onwards (i.e., from January 2018 to March 2024). Next, we identified the top peer-reviewed and influential conference and journal venues in the domains of SE, AI, and Security. We included 13 conferences (ICSE, ESEC/FSE, ASE, ISSTA, CCS, S&P, USENIX Security, NDSS, AAAI, IJCAI, ICML, NIPS, ICLR) and 4 journals (TOSEM, TSE, TDSC, TIFS). After the manual searching, we identified 35 papers that were relevant to our research objectives.

In addition to manually searching primary studies from top-tier venues, we also conducted an automated search across 7 popular databases, including IEEE Xplore [3], ACM Digital Library [1], SpringerLink [5], Wiely [7], ScienceDirect [4], Web of Science [6],] and arXiv [2]. The search string used in the automated search is crafted from the relevant papers identified in the manual search. Please kindly check the complete set of search keywords in our online appendix [8] due to the limited space. After conducting the automatic search, we collected 1,500 relevant studies with the automatic search from these 7 popular databases.

3.3 Study Selection

Inclusion and Exclusion Criteria. After the paper collection, we conducted a relevance assessment according to the following inclusion and exclusion criteria:

- ✓ *The paper must be written in English.*
- ✓ *The paper must have an accessible full text.*
- ✓ *The paper must be a peer-reviewed full research paper published either in a conference proceeding or a journal.*[1]
- ✓ *The paper must adopt LLM techniques to solve source code vulnerability detection or repair.*
- ✗ *The paper has less than 5 pages.*
- ✗ *Books, keynote records, panel summaries, technical reports, theses, tool demos papers, editorials, or venues not subject to a full peer-review process.*
- ✗ *The paper is a literature review or survey.*
- ✗ *Duplicate papers or similar studies authored by the same authors.*
- ✗ *The paper does not utilize LLMs, e.g., using* **graph neural networks (GNNs)**.
- ✗ *The paper mentions LLMs only in future work or discussions rather than using LLMs in the approach.*
- ✗ *The paper does not involve source code vulnerability detection or repair tasks.*

In the first phase, by filtering out short papers (exclusion criteria 1) and deduplication (exclusion criteria 4), the total number of included papers was reduced to 1,332. In the second phase, we manually examined the venue, title, and abstracts of the papers, and the total number of included papers declined to 212. Please kindly note that, in this step, we retain preprint papers released on arXiv in 2023 or 2024 that are not yet published due to their recent release. Books, keynote records, panel summaries, technical reports, theses, tool demo papers, editorials, literature reviews, or survey papers were also discarded in this phase (exclusion criteria 2–3). In the third phase, we manually read the full text of the paper to remove irrelevant papers. Specifically, the vulnerability detection or repair papers, which do not utilize LLMs but other methods, e.g., GNNs or **recurrent neural networks (RNNs)**, were dropped (exclusion criteria 5). We also excluded studies that do not focus on source code vulnerability, such as those on binary code, protocols, or network communication. Furthermore, we removed studies that just discussed LLM as an idea or future work (exclusion criteria 6). We also removed the papers focusing on other tasks rather than vulnerability detection or repair, such as vulnerable data generation, vulnerability assessment, etc. (exclusion criteria 7). After the third phase, we identify 82 primary studies directly relevant to our research topic.

Quality Assessment. To prevent biases introduced by low-quality studies, we formulated five **Quality Assessment Criteria (QAC)**:

- QAC_1: Was the study published in a prestigious venue?
- QAC_2: Does the study make a contribution to the academic or industrial community?
- QAC_3: Does the study provide a clear description of the workflow and implementation of the proposed approach?
- QAC_4: Are the experiment details, including datasets, baselines, and evaluation metrics, clearly outlined?

[1]For preprint papers released on arXiv in 2023 or 2024 that are not yet published due to their recent release, we will retain them when checking the venues and will perform a manual quality check to decide whether to include them in our study.

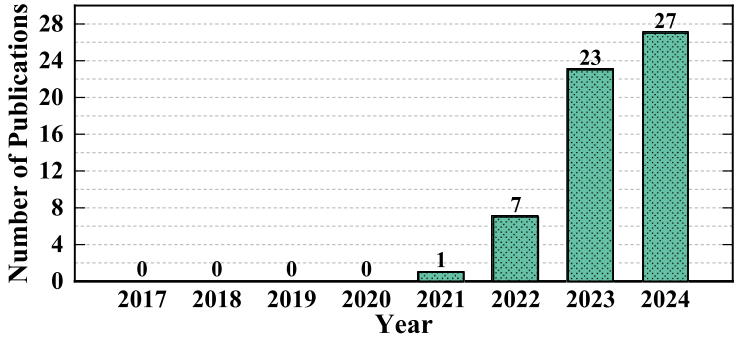

Fig. 3. Distribution of publications per year.

— QAC_5: Do the findings from the experiments strongly support the main arguments presented in the study?

We employed a scoring system ranging from 0 to 3 (poor, fair, good, excellent) for each quality assessment criterion. Following the manual assignment of scores, we selected papers with total scores reaching 12 (80% of the maximum possible score). Please note that for preprint papers released on arXiv in 2023 or 2024 that are not yet published, their score for the "venue" criterion is 0. However, if their total score reaches 12, we still consider them in our study. After this quality assessment, we obtained 42 papers. Among them, 37 papers were published, and 5 high-quality preprint papers were included after passing the quality assessment.

Forward and Backward Snowballing. To avoid omitting any possibly relevant work during our manual and automated search process, we also performed lightweight backward and forward snowballing [107]. This involved reviewing both the references cited in our selected 42 primary studies and the publications that cited these studies. As a supplement, we gathered 83 more papers and repeated the entire study selection process, including filtering, deduplication, and quality assessment, which resulted in the identification of 16 additional papers. *Thus, we obtained a final set of 58 papers to study.*

3.4 Data Extraction and Analysis

Figure 3 illustrates the distribution of selected primary studies across each year. The earliest relevant study we identified was published in 2021 [133]. Subsequently, interest in exploring LLMs for vulnerability detection and repair has steadily increased, peaking in 2024, with 46.6% of the total studied papers. The growing number of papers on these topics indicates increasing research interest in leveraging LLMs for vulnerability detection and repair. We also analyzed the publication venues of the included studies. ICSE emerges as the predominant conference venue for LLM studies on vulnerability detection and repair, contributing 20.7% of the total number of studies. Other notable venues are TSE (5.2%), FSE (3.4%), EMSE (3.4%), ISSTA (3.4%), MSR (3.4%), and TOSEM (1.7%).

4 RQ1: What LLMs Have Been Utilized?

In general, out of the 58 included studies, we identified 37 distinct LLMs that have been utilized.

LLMs Used for Vulnerability Detection. Figure 4 illustrates the distribution of the LLMs utilized for vulnerability detection. Please note that we count the occurrence of LLMs used in each study, and a single study may utilize multiple LLMs. CodeBERT [23] emerges as the predominant LLM in addressing vulnerability detection to date, representing 26.1% (24/92) of all the use of LLMs

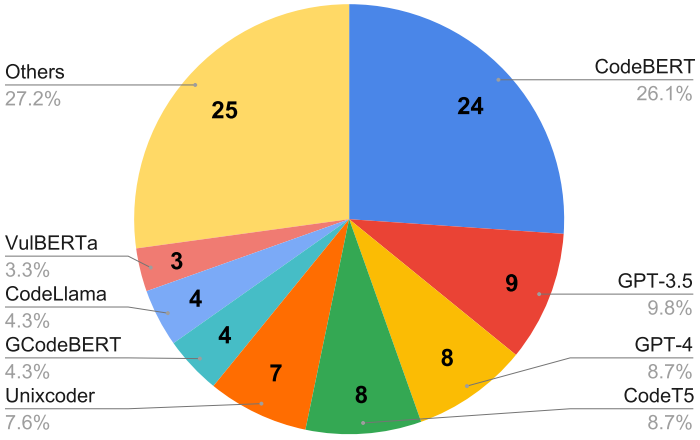

Fig. 4. Distribution of LLMs for vulnerability detection.

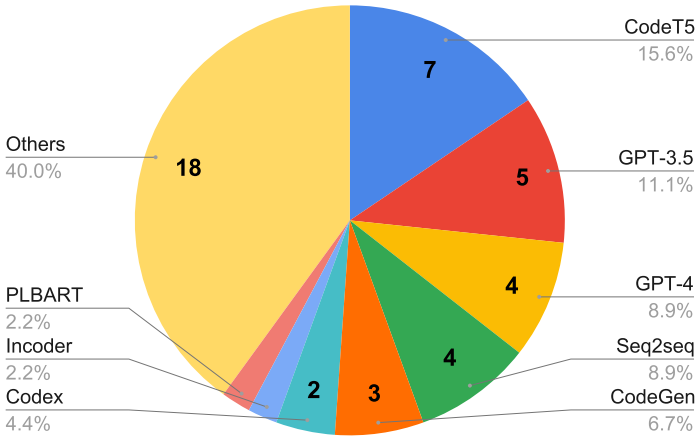

Fig. 5. Distribution of LLMs for vulnerability repair.

in the included studies. GPT-3.5 becomes the second most frequently studied model, accounting for 9.8% (9/92). Regarding the categories of LLMs, encoder-only LLMs comprise 47.8% (44/92) of the use of LLMs in the included studies. Following that, decoder-only LLMs account for 23.9% (22/92), and encoder-decoder LLMs constitute 9.8% (9/92). Lastly, commercial LLMs with undisclosed architectures, such as GPT-3.5 and GPT-4, account for 18.5% (17/92) of the LLM usages.

LLMs Used for Vulnerability Repair. Figure 5 illustrates the distribution of LLMs utilized for vulnerability repair. Unlike the detection task, CodeT5 [102] emerges as the predominant LLM in addressing vulnerability repair to date, representing 15.6% (7/45) of the use of LLMs in the included studies. GPT-3.5 becomes the second most frequently studied model, accounting for 11.1% (5/45). Following closely, GPT-4 and domain-specific pre-trained Seq2Seq Transformers become the third most frequently studied model, accounting for 8.9% (4/45) of the use of LLMs. Regarding the categories of LLMs, decoder LLMs comprise 31.1% (14/45) of LLMs used. Following that, encoder-decoder LLMs and encoder-only LLMs constitute 26.7% (12/45) and 6.7% (3/45) of the use of LLMs, respectively. Lastly, commercial LLMs with undisclosed architectures, such as GPT-3.5 and GPT-4, account for 35.6% (16/45) of the LLM usages.

Table 1. Top 10 LLMs Used in Vulnerability Detection and Repair

Usages ranking	Vulnerability detection			Vulnerability repair		
	LLM	**Structure**	**Size**	**LLM**	**Structure**	**Size**
1	CodeBERT	Encoder-only	125 M	CodeT5	Encoder-Decoder	220 M
2	GPT-3.5	Unknown	Unknown	GPT-3.5	Unknown	Unknown
3	GPT-4	Unknown	Unknown	GPT-4	Unknown	Unknown
4	CodeT5	Encoder-Decoder	220 M	Seq2seq	Encoder-Decoder	60 M
5	UnixCoder	Encoder-only	126 M	CodeGen	Decoder-only	350 M, 2.7 B, 6.1 B, 16.1B
6	GraphCodeBERT	Encoder-only	125 M	Codex	Unknown	12.0 B
7	CodeLlama	Decoder-only	7 B, 13 B, 34 B, 70 B	Incoder	Decoder-only	1.3 B, 6.7 B
8	VulBERTa	Encoder-only	125 M	PLBART	Encoder-Decoder	406 M
9	RoBERTa	Encoder-only	125 M	UnixCoder	Encoder-only	126 M
10	BERT	Encoder-only	109 M	CodeGPT	Decoder-only	124 M

In addition, Table 1 presents the architecture and model sizes of the top 10 most commonly used LLMs in vulnerability detection and repair. The architectures and model sizes of GPT-3.5 and GPT-4, along with the architecture of Codex, remain undisclosed due to their commercial status. As shown in Table 1, the LLMs widely utilized for vulnerability detection are predominantly lightweight models, with many having sizes under or equal to 126 million parameters. Most of these models are encoder-only architectures. In contrast, the models commonly used for vulnerability repair tend to be larger, with several exceeding 1 billion parameters, such as CodeGen [68] and Incoder [24]. The majority of these models have more than 126 million parameters and are primarily decoder-only and encoder-decoder architectures.

Answer to RQ1: Our analysis indicates that, to date, encoder-only LLMs have dominated vulnerability detection, while commercial LLMs and decoder-only LLMs have been prominent in vulnerability repair.

5 RQ2: How Are LLMs Adapted for Vulnerability Detection?

In RQ2, we shift our focus to examining the specific adaptation techniques of LLMs in vulnerability detection. Regarding the usage of LLMs, we summarize *three major categories* from the included studies: (1) *fine-tuning* [20], which updates the parameters of LLMs using a labeled dataset, (2) *prompt engineering* [13], which designs prompts to guide LLMs in generating relevant responses without updating parameters, and (3) *retrieval augmentation (RAG)* [47], which integrates knowledge from retrieval systems into the LLMs' context to improve their performance, also without changing LLMs' parameters. As depicted in Figure 6, 73% of the studies utilize fine-tuning, while 17% and 10% employ prompt engineering and RAG, respectively.

In the following subsection, we will introduce detailed categories of LLM adaptation techniques for each of the following: (1) *fine-tuning*, (2) *prompt engineering*, and (3) *RAG*. Table 2 summarizes the existing LLM-based methods for vulnerability detection.

5.1 Fine-Tuning

Fine-tuning is a commonly used technique for adapting LLMs to vulnerability detection. In this process, labeled code samples, indicating whether they are vulnerable or not, are provided as training data. The model is then fine-tuned through supervised learning, where its parameters

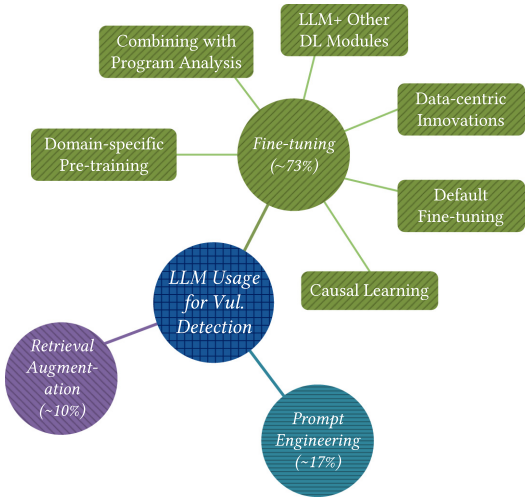

Fig. 6. Adaptation techniques of LLMs for vul. detection.

are adjusted based on these labeled examples. Several vulnerability detection studies [16, 19, 26, 27, 46, 50, 81, 83, 87, 88, 93, 129] have directly applied this default fine-tuning approach without introducing additional designs. In this subsection, we focus on studies that propose advanced fine-tuning techniques that go beyond the default approach for improved effectiveness.

The fine-tuning process usually involves several steps/stages, such as data preparation, model design, model training, and model evaluation. Regarding fine-tuning, we classify adaptation techniques into five groups based on the stages they mainly target: *Data-centric innovations* (data preparation), *Combination with program analysis* (data preparation), *LLM+ other deep learning modules* (model design), *Domain-specific pre-training* (model training), and *Causal learning* (training optimization).

Data-Centric Innovations. Data-centric innovations focus on optimizing the vulnerability detection data used for fine-tuning LLMs. Prior studies [16, 19, 45, 65, 114] have found that existing vulnerability detection data can suffer from imbalanced label distribution, noisy or incorrect labels, and scarcity of labeled data. Researchers [21, 45, 105, 114] have explored how to address the data issues.

- —*Imbalanced Learning*: To address label imbalance, i.e., having more non-vulnerable code samples than vulnerable ones in the dataset, Yang et al. [114] applied various data sampling techniques. In addition, Ding et al. [21] up-weighted the loss value for the rare class (i.e., the vulnerable samples) to pay comparable attention to both vulnerable and clean classes. They found that random oversampling on raw code data enhances the ability of the LLM-based vulnerability detection approach to learn real vulnerable patterns.
- —*Positive and Unlabeled Learning*: To address label quality issues such as noisy or incorrect labels, Wen et al. [105] proposed PILOT, which learns solely from positive (vulnerable) and unlabeled data for vulnerability detection. Specifically, PILOT generates pseudo-labels for selected unlabeled data and mitigates the data noise by using a mixed-supervision loss.
- —*Counterfactual Training*: To enhance the diversity of labeled data, Kuang et al. [45] proposed perturbing user-defined identifiers in the source code while preserving the syntactic and semantic structure. This approach generates diverse counterfactual training data, which refers to hypothetical data (e.g., data after perturbing identifiers) differing from actual data (i.e., data

Table 2. Existing LLM-Based Methods for Vulnerability Detection

	LLMs	LLM adaptation techniques								Data		Deployment	
	Size	Data-centric	LLM+ PA	LLM+ DL	Domain training	Causal learning	Default tuning	Prompt eng.	RAG	Input granularity	Data labeling	Interact with users	Integrated to workflow
Ziems et al. [133]	<1 B			✓						Function	synthetic	✗	✗
Fu et al. [26]	<1 B						✓			Line	heuristic	✗	✗
Hanif et al. [34]	<1 B				✓					Function	synthetic, heuristic, partially manually labeled	✗	✗
Thapa et al. [93]	<10 B						✓			Function	heuristic	✗	✗
Fu et al. [29]	>10 B							✓		Function	heuristic	✗	✗
Wen et al. [105]	<1 B	✓								Function	heuristic, partially manually labeled	✗	✗
Purba et al. [76]	>1 B							✓		Function	heuristic	✗	✗
Liu et al. [55]	>10 B								✓	Function	partially manually labeled	✗	✗
Peng et al. [75]	<1 B		✓							Line	heuristic	✗	✗
Kuang et al. [45]	<1 B	✓								Function	heuristic, partially manually labeled	✗	✗
Zhang et al. [120]	<1 B		✓							Function	heuristic, partially manually labeled	✗	✗
Tang et al. [91]	<1 B			✓						Function	partially manually labeled	✗	✗
Ni et al. [64]	<1 B				✓					Function	heuristic, crowd-sourced	✗	✗
Zhou et al. [131]	>10 B							✓	✓	Function	heuristic	✗	✗
Fu et al. [27]	<1 B						✓			Function	heuristic	✗	✗
Sejfia et al. [83]	<1 B						✓			Function	heuristic, partially manually labeled	✗	✗
Wang et al. [98]	<1 B		✓		✓					Function	synthetic, heuristic, manually labeled	✗	✗
Liu et al. [56]	<1 B		✓		✓					Function	heuristic, partially manually labeled	✗	✗
Rahman et al. [59]	<1 B					✓				Function	heuristic, partially manually labeled	✗	✗
Ding et al. [21]	<1 B, 1-10 B, >10 B	✓			✓					Function	heuristic, partially manually labeled	✗	✗
Steenhoek et al. [89]	<1 B				✓					Function, Line	synthetic, heuristic, partially manually labeled	✗	✗
Khare et al. [43]	>1 B							✓		Function	heuristic, partially manually labeled	✗	✗
Zhang et al. [118]	>10 B							✓		Function	synthetic, heuristic	✗	✗
Risse et al. [81]	<1 B						✓			Function	partially manually labeled, manually labeled	✗	✗
Steenhoek et al. [88]	<1 B						✓			Function, Line	heuristic, partially manually labeled	✗	✗
Yang et al. [114]	<1 B	✓								Function	heuristic, partially manually labeled	✗	✗
Chen et al. [16]	<1 B						✓			Function	heuristic	✗	✗
Li et al. [50]	<1 B						✓			Function	heuristic	✗	✗
Croft et al. [19]	<1 B						✓			Function	heuristic, manually labeled	✗	✗
Le et al. [46]	<1 B						✓			Function, Line	heuristic, partially manually labeled	✗	✗
Zhou et al. [129]	<10 B						✓	✓		Repo	heuristic, manually labeled	✗	✗
Tran et al. [94]	<1 B		✓							Line	heuristic	✗	✗
Jiang et al [41]	<1 B			✓						Function	heuristic, partially manually labeled	✗	✗
Shestov et al. [87]	>10 B						✓			Function	heuristic, manually labeled	✗	✗
Ni et al [63]	>10 B							✓		Function	heuristic	✗	✗
Weng et al. [106]	<1 B		✓							Line	heuristic	✗	✗
Du et al. [22]	>10 B								✓	Function	heuristic	✗	✗
Wen et al. [104]	<1 B, 1-10 B, >10 B								✓	Function	heuristic	✗	✗
Xin Yin [116]	>10 B							✓		Function	heuristic	✗	✗
Yang et al. [112]	<1 B, >10 B			✓						Function	heuristic	✓	✗

without perturbation), useful for analyzing the effect of certain factors. Incorporating these counterfactual data enriches the training data for LLMs.

Combination of LLM with Program Analysis. Many LLMs undergo pre-training on extensive datasets through unsupervised objectives like masked language modeling [20] or next token prediction [78]. For those LLMs, they may prioritize capturing sequential features and could potentially overlook certain structural aspects crucial for understanding code. To address this limitation, several studies have proposed integrating program analysis techniques with LLMs. The idea involves utilizing program analysis to extract structural features/relations within code, which are then incorporated into LLMs to enhance their understanding. Specifically, Liu et al. [56] leveraged Joern [111] to build the AST and PDG of the function, leveraging this data to pre-train their LLM to predict statement-level control dependencies and token-level data dependencies within the function. Peng et al. [75] utilized program slicing to extract control and data dependency information, aiding LLMs in vulnerability detection. Wang et al. [98] proposed to learn program presentations by feeding static source code information and dynamic program execution traces with LLMs. Additionally, Zhang et al. [120] proposed decomposing syntax-based Control Flow Graphs into multiple execution paths and feeding these paths to LLMs for vulnerability detection. Tran et al. [94] utilize a program analysis tool to normalize all user-defined names, such as variables and functions, into symbolic names (e.g., VAR_1, VAR_2, and FUNC_1) to enhance the model's robustness against adversarial attacks. Weng et al. [106] incorporated inter-statement data and control dependency information into an LLM by masking irrelevant attention scores based on the program dependency graph, which resulted in improved code representation.

Combination of LLM with Other Deep Learning Modules. LLMs have their own inherent limitations. Firstly, most LLMs are based on the Transformer architecture, which primarily models sequential relations and features. Secondly, some LLMs (e.g., CodeBERT) impose restrictions on the length of input code snippets. For instance, the most frequently used LLM in vulnerability detection, CodeBERT, can only process 512 tokens. To address these limitations, researchers have attempted to combine LLMs with other DL modules:

- *LLM + GNN*: To leverage the structural features of code more effectively, Tang et al. [91] introduced CSGVD, which employs GNN to extract the graph features of code and combine them with the features extracted by CodeBERT. Jiang et al. [41] parsed **data flow graphs (DFGs)** from the input code and used the data types of each variable as the node characteristics of the DFG. They then embedded the DFG information into the graph representation using a GNN and integrated this graph representation with an LLM. In addition, Yang et al. [112] combined an LLM with a GNN model. Specifically, they first used data flow analysis embeddings to establish a GNN model. They then concatenated the learned embeddings from the GNN with the hidden states of the LLM. By concatenating the embeddings during each forward pass, they enabled simultaneous training of both the LLM and the GNN.
- *LLM* + **Bidirectional Long Short-Term Memory (Bi-LSTM)**: To address the length constraints of LLMs, Ziems et al. [133] first segmented the input code into multiple fixed-size segments. They then utilized BERT to encode each segment and incorporated a Bi-LSTM module to process the output of BERT on each segment. Finally, a softmax classifier was applied to the last hidden state of the Bi-LSTM to produce the final classification scores.

Domain-Specific Pre-Training. Domain-specific pre-training involves pre-training an LLM on data specific to a particular domain, such as vulnerability data, before fine-tuning it for a specific task within that domain. This process enables the LLM to better understand the data relevant to the

domain. Several studies in the field of vulnerability detection have adopted this technique [21, 34, 56, 64, 89, 98]. These studies typically use one of three pre-training objectives:

— *Masked Language Modeling*: This pre-training objective involves training the LLM to predict masked tokens in a corrupted code snippet. Hanif and Mahmood [34] introduced VulBERTa, which pre-trained a RoBERTa [54] model on open-source C/C++ projects using the Masked Language Modeling objective. Since C/C++ is the programming language of the vulnerability detection data used to evaluate VulBERTa, this pre-training enhances VulBERTa's ability to understand C/C++ code.
— *Contrastive Learning*: This pre-training objective is to minimize the distance between similar functions while maximizing the distance between dissimilar functions. Through contrastive learning, the LLM can learn to capture the distinctive features of code. In particular, Ni et al. [64] and Wang et al. [98] utilized different hidden dropout masks to transform the same input function into positive samples (i.e., those similar to the input function) while regarding other distinct functions as dissimilar samples. Then they pre-trained an LLM following the contrastive learning objective. Additionally, Ding et al. [21] performed Class-aware Contrastive Learning to maximize the representation similarity between each sample and the perturbed version of itself and minimize the representation similarity between two randomly chosen samples.
— *Predicting Program Dependencies*: This pre-training objective aims to enhance the LLM by guiding the model to learn the knowledge necessary for analyzing dependencies in programs. Specifically, Liu et al. [56] pre-trained an encoder-only LLM on code from open-source C/C++ projects to predict the statement-level control dependency and token-level data dependency.
— *Annotating Vulnerable Statements*: This pre-training objective assumes that improving alignment to **potentially vulnerable statements (PVS)** would improve the performance of the model. For example, Steenhoek et al. [89] developed two annotation approaches (i.e., Mark and Prepend) to give guidance or extra information to the model. Mark inserted special "marker" tokens before and after each token inside the PVS, while Prepend added the tokens inside PVS at the beginning of the code. This can be considered as inserting domain knowledge into the model's input.

After the pre-training, those domain-specific pre-trained LLMs undergo fine-tuning on the vulnerability detection dataset to perform vulnerability detection.

Causal Learning. Although promising, LLMs have been found to lack robustness under perturbation or when encountering out-of-distribution data [59]. Rahman et al. [59] suggested that this weak robustness may be attributed to LLMs learning non-robust features, such as variable names, that have spurious correlations with labels. To address this issue, Rahman et al. proposed CausalVul, which first designs perturbations to identify spurious features and then applies causal learning algorithms, specifically do-calculus, on top of LLMs to promote causal-based prediction, enhancing the robustness of LLMs in vulnerability detection.

5.2 Prompt Engineering

Zero-Shot Prompting. Researchers have attempted to devise effective prompts to guide LLMs in conducting vulnerability detection in the zero-shot prompting manner. Their prompt engineering designs generally consist of one or more components outlined below:

— *Task Description.* This part of the prompt aims to equip LLMs with valuable task-specific information related to vulnerability detection, clarifying the objectives the model should achieve (e.g., identifying vulnerabilities in the provided code). Different studies employ varying

task descriptions, and there is no consensus on which approach is optimal. Specifically, Zhou et al. [131] used a task description: *"If has any potential vulnerability, output: 'this code is vulnerable'. Otherwise, output: 'this code is non-vulnerable'. The code is [code]. Let's start:"*. Fu et al. [29] proposed a task description: *"Predict Whether the C/C++ function below is vulnerable. Strictly return 1 for a vulnerable function and 0 for a non-vulnerable function without further explanation."* Purba et al. [76] devised a task description: *"[code] Is this code vulnerable? Answer in only Yes or No". "[code]"* refers to the input code in these task descriptions. Xin Yin [116] used the task description: *"I will provide you a C code snippet and want you to tell whether it has a vulnerability. You need to output 'yes' or 'no' first (output no if uncertain), and then explain."* Zhou et al. [129] used *"If the following code snippet has any vulnerabilities, output Yes; otherwise, output No"*.

— *Role Description.* This aims to help LLMs shift their operational mode from a general language model to that of a vulnerability detector. Specifically, Zhou et al. [131] defined a role description: *"You are an experienced developer who knows the security vulnerability very well"*. Fu et al. [29] used a role description: *"I want you to act as a vulnerability detection system"*. Khare et al. [43] utilized a role description: *You are a security researcher"*. Xin Yin [116] used the role description as: *"I want you to act as a vulnerability scanner"*.

— *Vulnerability-Related Auxiliary Information.* Some studies have integrated vulnerability-related auxiliary information into their prompt designs to enhance LLM performance. Specifically, Zhou et al. [131] suggested including vulnerable code examples representing the top 25 most dangerous CWEs of 2022 as part of the prompts to help LLMs better understand the characteristics of serious vulnerabilities. Additionally, Khare et al. [43] specified the types of vulnerabilities when querying LLMs about potential vulnerabilities in the provided code.

— *Program-Analysis Auxiliary Information.* Some studies utilize program-analysis tools to extract code dependencies from the input code, aiming to enhance LLMs' understanding of the input code. Specifically, Khare et al. [43] designed a dataflow analysis-based prompt that guides the model to simulate a source-sink-sanitizer-based dataflow analysis on the target code snippet before predicting if it is vulnerable. Zhang et al. [118] designed prompts that incorporate auxiliary information on data flow or API calls.

— *Chain-of-thought Prompting.* It is proposed by Kojima et al. [44] for improved reasoning, which involves adding "Let's think step by step" to the original prompt. Specifically, Zhou et al. [129] and Ni et al. [63] added the "Let's think step by step" into the prompt.

Few-Shot Prompting. In this approach, a few examples of input and ground truth label pairs are provided to the LLMs as additional guidance. Specifically, Ni et al. [63] utilize between 1 and 6 few-shot learning examples to fill the fixed context window (i.e., 4,096 tokens). Zhou et al. [129] use two few-shot examples.

5.3 RAG

RAG is a technique used to enhance the few-shot prompting. It involves retrieving similar labeled data samples from the training set when given a test data sample and using these retrieved data as examples to guide the prediction of LLMs on the test sample. Liu et al. [55] proposed using efficient retrieval tools such as BM-25 and TF-IDF for retrieval. Zhou et al. [131] suggested using CodeBERT as a retrieval tool. This method first transforms code snippets into semantic vectors and then quantifies the similarity between two code snippets by calculating the Cosine similarity of their respective semantic vectors. It finally returned the top similar code based on the similarity scores. Du et al. [22] leverage a knowledge-level retrieval-augmentation framework consisting of three phases to identify vulnerabilities in a given code. First, they constructed a vulnerability

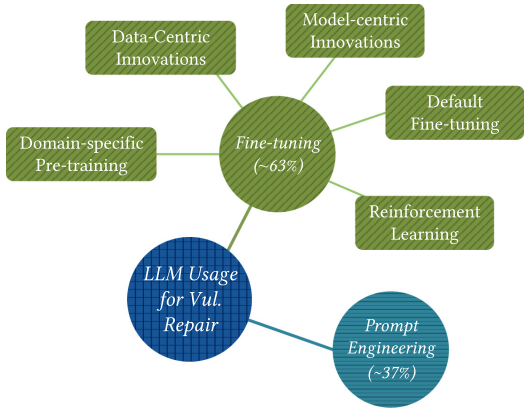

Fig. 7. Adaptation techniques of LLMs for vul. repair.

knowledge base by extracting multi-dimensional knowledge from existing CVE instances using LLMs. Second, for a given code snippet, they retrieved relevant vulnerability knowledge from the constructed knowledge base based on functional semantics. Finally, they utilized LLMs to assess the vulnerability of the given code snippet by reasoning about the presence of vulnerability causes and proposing potential fixing solutions derived from the retrieved knowledge. Wen et al. [104] first retrieved the most relevant dependencies from call graphs based on a given input code, and then integrated these dependencies with the input code into LLMs to detect vulnerabilities.

Answer to RQ2: Our analysis revealed three commonly used techniques to adapt LLMs for vulnerability detection: including *fine-tuning* (≈73%), *prompt engineering* (≈17%), and *retrieval augmentation* (≈10%).

6 RQ3: How Are LLMs Adapted for Vulnerability Repair?

Regarding the usage of LLMs, we summarize *three major categories* from the included studies: (1) *fine-tuning* (approximately 63%), which updates the parameters of LLMs using a labeled dataset, and (2) *prompt engineering* (approximately 37%), which freezes the parameters of LLMs and designs prompts to guide LLMs in generating relevant responses.

In the following subsection, we will introduce detailed categories of LLM adaptation techniques for each of the following: (1) *fine-tuning* and (2) *prompt engineering*.

6.1 Fine-Tuning

Some studies [27, 28] employ default fine-tuning methods to adapt LLMs for vulnerability repair. In this subsection, we focus on research that introduces advanced fine-tuning techniques that extend beyond the default approach to achieve improved effectiveness.

Regarding fine-tuning, we categorize advanced adaptation techniques into four groups based on the stages they mainly target: *Data-centric innovations* (data preparation), *Model-centric innovations* (model design), *Domain-specific pre-training* (model training), and *Reinforcement learning* (training optimization). Table 3 summarizes the existing LLM-based methods for vulnerability repair.

Data-Centric Innovations. Several studies have shown that in addition to the input vulnerable code, incorporating other diverse relevant inputs can boost the effectiveness of LLMs. Specifically, the AST of the vulnerable code input [132], vulnerability descriptions [103, 132], and vulnerable

Table 3. Existing LLM-Based Methods for Vulnerability Repair

	LLMs	LLM adaptation techniques						Data		Deployment	
	Size	Data-centric	Model-centric	Domain pre-training	Reinforce. learning	Default tuning	Prompt eng.	Input granularity	Real-world & tests	Interact with users	Integrated to workflow
Chen et al. [17]	<1 B			✓				Function	✗	✗	✗
Chi et al. [18]	<1 B			✓				Function	✗	✗	✗
Fu et al. [28]	<1 B					✓		Function	✗	✗	✗
Zhou et al. [132]	<1 B	✓						Function	✗	✗	✗
Wu et al. [109]	<1 B, 1-10 B, >10 B						✓	Function	✓	✗	✗
Fu et al. [29]	>10 B						✓	Function	✗	✗	✗
Zhang et al. [123]	<1 B			✓				Function	✗	✗	✗
Pearce et al. [74]	<1 B, 1-10 B, >10 B						✓	Function	✓	✗	✗
Wei et al. [103]	<1 B	✓						Function	✗	✗	✗
Fu et al. [25]	<1 B		✓					Function	✗	✗	✗
Islam et al. [39]	1-10 B				✓			Function	✗	✗	✗
Fu et al. [27]	<1 B					✓		Function	✗	✗	✗
Zhou et al. [127]	<1 B	✓		✓				Function	✗	✗	✗
Nong et al. [69]	>1 B						✓	Function	✗	✗	✗
Berabi et al. [12]	>1 B	✓						Function	✗	✗	✗
Ahmad et al. [9]	>10 B						✓	Function	✗	✗	✗
Islam et al. [38]	<10 B		✓					Function	✗	✓	✗
Nong et al. [70]	>10 B						✓	Function	✗	✗	✗
Wang et al. [101]	>1 B						✓	Function	✗	✗	✗

code examples shared on CWE websites [127] can enhance the LLMs in repairing vulnerability. In addition, Wei et al. [103] found that vulnerability-inducing commits and vulnerability-fixing commits can also improve the effectiveness of the model. Lastly, Zhou et al. [127] found that certain vulnerable functions exceed the length limit of Transformer-based LLMs like CodeT5 (512 subtokens) [127]. To address this limitation, they applied the Fusion-in-Decoder framework to partition a long code function into multiple segments and feed those segments into LLMs one by one. Their findings demonstrated that those segments can boost model effectiveness. Additionally, a study shows that removing less important parts of the code can help the model focus more effectively on the vulnerable lines, thereby improving performance. Berabi et al. [12] presented a method for generating reduced code by leveraging static analysis outcomes, creating a compact representation of the program. This smaller version fits within the attention window of an LLM while preserving the crucial information required to learn a correct fix. The technique ensures that the reduced code snippet still produces the same static analysis report as the original program, maintaining the core problem for effective analysis.

Model-Centric Innovations. Model-centric innovations encompass methodologies that prioritize revising the model architecture of LLMs (i.e., the Transformer [96]). Specifically, Fu et al. [25] drew inspiration from Vision Transformer-based object detection techniques in computer vision. They proposed vulnerability queries within the pre-trained Transformer model to identify vulnerable code blocks within the source code. Additionally, they trained a model to learn a vulnerability mask, enhancing the attention of the vulnerability queries on the vulnerable code areas. Islam et al. [38] proposed fine-tuning an LLM (CodeGen2 [66]) to perform two tasks simultaneously. First, the model generates a fix for the vulnerable code, and second, it produces a developer-friendly description of the code as an explanation.

Domain-Specific Pre-Training. Domain-specific pre-training involves pre-training an LLM on data specific to a particular domain before fine-tuning it for the target task. Considering the similarity between bug-fixing and vulnerability-fixing tasks, several studies [17, 18, 123, 127] considered the task of bug-fixing as a pre-training task to enhance LLMs. Specifically, they first pre-trained LLMs on a bug fix corpus to fix bugs, and then fine-tuned LLMs on a vulnerability fix dataset to repair vulnerabilities. This kind of pre-training technique can be considered as *Transfer Learning*.

Reinforcement Learning. Islam et al. [39] introduced SecureCode, an LLM tuned using a reinforcement learning framework. This approach integrated syntactic and semantic rewards to generate fixes for vulnerable code. Specifically, they leveraged the CodeBLEU [80] score as the syntactic reward and BERTScore [124] as the semantic reward. After combining these rewards, they applied the Proximal Policy Optimization algorithm [82] to fine-tune the CodeGen2-7B [67] model.

6.2 Prompt Engineering

Zero-Shot Prompting. For zero-shot prompting, researchers have sought to create effective prompts that guide LLMs in performing vulnerability repair. Prompt engineering typically includes components outlined below:

- *Vulnerability Descriptions.* This component provides specific details about the types of vulnerabilities to be addressed, helping the LLM understand the nature of the issues within the code. Specifically, Pearce et al. [74] propose to use a format like "// BUG: [vulnerability-description]" to provide the descriptions on the vulnerability. Ahmad et al. [9] provided the bug description with explicit instructions for repair with LLMs, as illustrated in the prompt: "// BUG: Access Control Check Implemented After the Asset is Accessed. // Ensure that access is granted before data is accessed".
- *Vulnerability Location.* This element indicates the specific sections of the code where vulnerabilities may exist. Wu et al. [109] proposed a prompt that begins by commenting on the buggy code lines and adds "BUG:" to highlight them, which provides the location of the vulnerability to LLMs.
- *Vulnerability-Related Auxiliary Information.* This component encompasses auxiliary information about vulnerabilities. Nong et al. [69] incorporate vulnerability semantics—specifically, the behaviors of a vulnerable program that contribute to its susceptibility—into the prompt to enhance the performance of LLMs.
- *Program-Analysis Auxiliary Information.* Wang et al. [101] proposed a program-analysis-based approach. Given a patch, their method first utilizes both the vulnerable code and its fixed version as input. The process begins by localizing the **Minimum Edit Node (MEN)**, which is the common ancestor node of all modified nodes. Once the MEN is identified, customized rules are applied based on the MEN type to extract vulnerable code patterns. These patterns are then incorporated into the prompt to assist LLMs in generating more effective vulnerability fixes.

Few-Shot Prompting. Furthermore, for few-shot prompting, Fu et al. [29] investigated a straightforward prompt strategy involving the provision of three repair examples within each prompt. This approach aimed to aid GPT-3.5 and GPT-4 in performing the repair task more effectively. Nong et al. [70] proposed a novel few-shot prompting method. Given a vulnerable program with a known vulnerability location, they first narrowed the analysis scope to only the relevant subset of the program. Next, they elicited the LLM to identify the vulnerability's root cause within this reduced scope, dynamically selecting exemplars from a pre-mined database that best fit the program. With the chosen exemplars, they formed the patching prompt to generate patches with LLMs.

Answer to RQ3: Our analysis revealed three commonly used techniques to adapt LLMs for vulnerability repair: including *fine-tuning* (≈63%) and *prompt engineering* (≈37%).

7 RQ4: What Are the Characteristics of Datasets and Deployment?

A typical learning-based pipeline can be divided into three phases: (1) data preparation, (2) model construction, and (3) deployment. The previous RQs primarily focus on the model construction phase. In this RQ, we focus on examining the characteristics of the datasets used, as well as the deployment strategies employed in LLM-based vulnerability detection and repair studies.

7.1 Data Characteristics

In our investigation of the data utilized in LLM-based vulnerability detection and repair studies, we primarily focus on two aspects: (1) input code size and (2) data quality.

7.1.1 Input Granularity. Input granularity refers to the size or scope of the code provided to the model for vulnerability detection and repair. For example, in a vulnerability detection task with repo-level input granularity, the entire repository is given to the model, and it is tasked with identifying vulnerabilities within that repository. Input granularity can vary from the line-level, function-level, and class-level to repo-level.

For vulnerability detection, as shown in Table 2, the majority of studies concentrate on function-level vulnerability detection, with only 7 addressing line-level detection. The function-level approach analyzes entire functions for potential vulnerabilities, whereas line-level detection examines specific lines of code, which can be advantageous for pinpointing fine-grained locations of vulnerabilities. Regarding vulnerability repair, Table 3 reveals that all studies target function-level repairs.

However, there is a notable gap in research exploring larger input granularity for both vulnerability detection and repairs, such as class-level and repo-level approaches. Larger input granularity is essential for identifying vulnerabilities that span multiple functions. Specifically, there are no studies focused on class-level vulnerability detection and repair. For repo-level detection, one recent work [129] initially explored the use of LLMs (e.g., Code Llama) to detect vulnerabilities across entire repositories, comparing their performance with SAST tools (e.g., CodeQL). Notably, none of the studied papers investigate repo-level vulnerability repair. Overall, research on vulnerability detection and repair for larger input granularity remains limited.

7.1.2 Data Quality. We separately investigate the data quality issues in vulnerability detection and repair. Specifically, we find that vulnerability detection studies primarily utilize datasets with labels generated by heuristic-based methods. Additionally, vulnerability repair studies often rely on datasets that lack test cases, making it difficult to fully evaluate the correctness of the generated repairs since there can be multiple correct fixes.

Vulnerability Detection Studies Mainly Rely on Datasets Using Heuristic-Based Labeling Methods. Table 2 shows that many studies have employed heuristic-based labeling functions to generate data for vulnerability detection tasks. Typically, they collect vulnerability-fixing commits from databases like NVD and label the pre-commit versions of the functions modified by these commits as vulnerable. Additionally, all unchanged functions in the same files are labeled as non-vulnerable. In this way, the vulnerable functions are those in the files affected by vulnerability-fixing commits. However, recent studies [16, 19, 65] have highlighted quality issues with vulnerability data generated by the heuristic-based method, including noisy or incorrect labels (e.g., labeling clean code as vulnerable).

Vulnerability Repair Studies Mainly Rely on Datasets without Test Cases. As present in Table 3, obtaining real-world vulnerability data along with corresponding test cases is challenging. Most studies either utilize synthetic vulnerable code and the corresponding fixes (e.g., [39]) or use real-world vulnerable code and fixes without test cases (e.g., [17, 18, 28]). Two recent studies [74, 109] utilized real-world vulnerability-fixing datasets with test cases, however, the number of samples is very limited, with only 42 and 12 samples being evaluated, respectively. This underscores the urgent need for more extensive real-world datasets and accompanying test cases.

7.2 Deployment Strategies

In our investigation of the deployment strategies utilized in LLM-based vulnerability detection and repair studies, we focus on two key aspects: (1) the communication and collaboration between developers and LLM-based methods, and (2) the integration of these methods into developers' workflows. We focus on the communication and collaboration between users and LLM-based methods because they are essential in deploying LLMs in practice and are pivotal in maximizing the real-world value of LLMs [113]. Moreover, a recent survey [52] shows that developers desire opportunities for natural language interaction with LLM-based solutions, also highlighting the significance of this capability. In addition, seamless integration of these solutions into developers' workflows is crucial for successful deployment. A recent study [117] found that even for the popular LLM-based tool GitHub Copilot, the top 1 limitation reported is the difficulty of integrating Copilot with IDEs or other plugins. This emphasizes the need for better support in integrating LLM-based tools with other frequently used tools in developers' workflows such as IDEs.

Firstly, as shown in Table 2, limited studies on vulnerability detection in our review support communication or collaboration between developers and LLM-based methods. One exception is a study [112] where the method generates explanations for vulnerabilities during the detection process. However, interaction and communication with developers are essential; for instance, unexplained suggestions may be disregarded, as developers need to understand the rationale behind the recommendations to trust and implement them effectively. In vulnerability repair studies (as shown in Table 3), most also do not support communication between developers. An exception is a recent study by Islam et al. [38], which provides explanations for the fixes when addressing vulnerabilities.

Secondly, as illustrated in Tables 2 and 3, all studies in our review are not integrated into developers' workflows and tools but are evaluated only offline using static and historical data. For these solutions to be effectively adopted, they should seamlessly integrate into existing development environments and tools.

Answer to RQ4: The datasets used in LLM-based vulnerability detection and repair studies primarily focus on function-level or line-level inputs, with limited exploration of class or repository level. Many vulnerability detection datasets employ heuristic-based labeling methods, and repair datasets frequently lack associated test cases. Furthermore, current approaches do not emphasize integration with developer workflows or interaction and collaboration between developers and LLM-based models.

8 The Road Ahead

In this section, we first discuss the limitations of current studies and then introduce a roadmap, illustrating the research trajectory shaped by prior work and highlighting future directions for exploration.

8.1 Limitations

Limitation 1: Small Input Granularity. Currently, LLM-based vulnerability detection and repair solutions primarily target the function level (or more fine-grained, at the line level). However, this small input granularity indicates that these approaches may not perform optimally when presented with a wider range of programs, such as classes or a whole repository. Function-level vulnerability detection approaches could overlook vulnerabilities that span multiple functions or classes [84]. Similarly, function-level vulnerability repair approaches fall short when tasked with modifications across multiple functions within the repository [127]. *In addition to functions, future research could propose LLM-based detection/repair approaches capable of handling a broader range of programs.*

Limitation 2: Lack of High-Quality Vulnerability Dataset. One major challenge is the lack of high-quality vulnerability datasets. For vulnerability repair, previous studies [16, 19, 65] have highlighted issues with existing vulnerability data, including noisy or incorrect labels (e.g., labeling clean code as vulnerable). This data quality issue is primarily attributed to the use of automatic vulnerability collection [65], which can gather large enough data for training DL-based models including LLMs but cannot ensure the complete correctness of the labels. While manually checking each data sample can ensure high quality, it is a very tedious and expensive process, especially when aiming for a large dataset. *Constructing a high-quality vulnerability detection benchmark remains an open challenge to date.*

For vulnerability repair, most studies either utilize synthetic data (e.g., [39]) or use real-world data without test cases (e.g., [17, 18, 28]). The reliance on synthetic data can limit the generalizability of the findings, as it may not accurately reflect the complexities and nuances of actual vulnerabilities encountered in production environments. On the other hand, real-world data without test cases makes it difficult to evaluate the effectiveness of proposed solutions comprehensively, as the absence of test cases hampers the ability to verify the correctness of the vulnerability repairs. *There is an urgent need for large real-world vulnerability-fixing datasets that include test cases.*

Moreover, there is a growing concern that the capabilities of LLMs may derive from the inclusion of evaluation datasets in the pre-training corpus of LLMs, a phenomenon known as data contamination [40]. To mitigate this concern, high-quality vulnerability detection/fixing benchmarks are preferred to have no overlap with the pre-training corpus of LLMs.

Limitation 3: Suboptimal Performance Caused by Complexity in Vulnerability Data. Vulnerabilities can be inherently complicated, which brings challenges for their detection and repair with LLMs. For instance, inter-procedural vulnerabilities are prevalent in vulnerability data, and Li et al. [51] discovered that detecting inter-procedural vulnerabilities poses greater challenges than intra-procedural ones. Moreover, vulnerabilities encompass a wide array of **Common Weakness Enumeration (CWE)** types [128], but LLMs may struggle with less frequent CWE types compared to frequent types [127, 128]. In addition, vulnerabilities are typically represented in terms of code units, such as code lines, functions, or program slices within which the vulnerabilities occur. Sejfia et al. [84] observed a significant accuracy drop when detecting vulnerabilities that span multiple code units, such as spanning multiple functions. *Future research should consider the complex nature of vulnerabilities when designing LLM-based solutions.*

Limitation 4: Reliance on Lightweight LLMs. As highlighted in Tables 2 and 3, most studies adapted lightweight LLMs (#parameter $<$ 1 B) for vulnerability detection and repair. For lightweight LLMs like CodeBERT, researchers have explored various strategies to boost their performance, including data-centric enhancements, model-centric innovations, integration with program analysis, combining LLMs with other deep learning methods, domain-specific pre-training, causal learning, and reinforcement learning. In some cases, such as Liu et al. [56], multiple techniques have

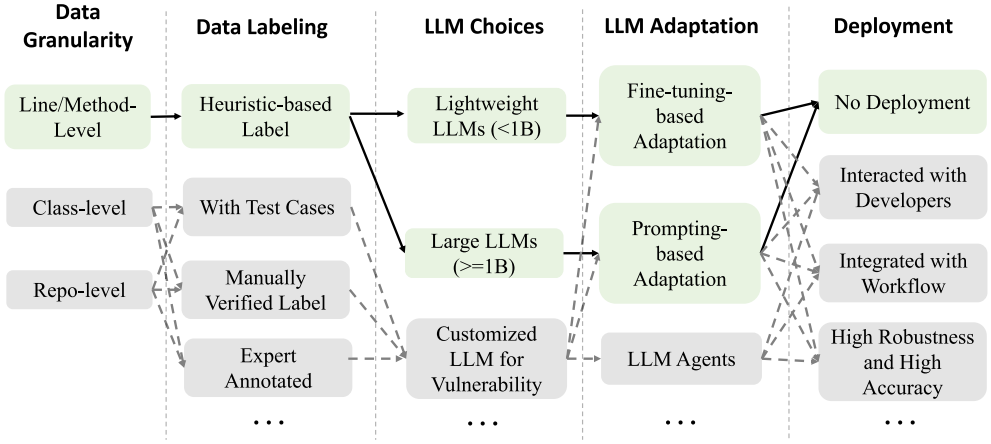

Fig. 8. Roadmap and future directions.

been combined to achieve even greater performance, highlighting the versatility and potential of lightweight models. In contrast, the use of large LLMs (with over 1 billion parameters) remains relatively limited compared to lightweight LLMs, leaving a large room to explore.

Limitation 5: Lack of Deployment Consideration. We examine two crucial aspects of deployment: (1) interaction with developers and (2) integration into developers' current workflows. Currently, none of the studies in our review have been incorporated into developers' workflows; instead, they primarily rely on static and historical data to evaluate their effectiveness. Moreover, limited studies in our review incorporate interaction with developers, such as engaging with developers through feedback or explaining the rationale behind vulnerability detection. Limited interaction between developers and LLM-based solutions may impede the establishment of trust and collaboration during practical applications. This lack of interactive features restricts the practical applicability of these models in real-world scenarios, where effective vulnerability management relies on collaboration between automated tools and developers.

To address these gaps, future research should explore more effective strategies for fostering collaboration and trust between developers and LLM-based solutions [57]. Additionally, seamlessly integrating LLM-based solutions can enable them to evolve into intelligent partners, providing enhanced support to developers.

Limitation 6: Lack of High Accuracy and Robustness. A vulnerability detection or repair solution with high accuracy is generally preferred, as it boosts developers' confidence in the reliability of detections and fixes. However, current state-of-the-art approaches [56, 127] have not yet achieved satisfactory accuracy, with 67.6% and 20% accuracy scores for vulnerability detection and repair, respectively. Moreover, the solution should maintain robustness against data perturbations or adversarial attacks to ensure its resilience. However, Yang et al. [115] and Rahman et al. [59] discovered that LLMs are not robust against data perturbations. *Future research should seek ways to improve the accuracy and robustness of LLM-based solutions.*

8.2 Roadmap

Figure 8 presents the roadmap. The green modules highlight the aspects that existing studies have primarily focused on, while the gray modules represent areas that have not been extensively explored but warrant further investigation. The solid black lines represent the trajectories followed

by most existing LLM-based methods for vulnerability detection and repair, while the gray dotted lines highlight relatively under-explored paths that could offer promising directions for future research in vulnerability detection and repair.

8.2.1 Established Pathways in Current Studies. As shown in both Tables 2 and 3 and the solid black lines in Figure 8, the existing methods typically leverage either lightweight LLMs or large LLMs adapted using techniques such as fine-tuning or prompting. The existing methods often focus on method-level or line-level analysis. The datasets used are usually collected from heuristic approaches for vulnerability detection or real-world data without test cases for vulnerability repair. Finally, these methods usually neglect deployment considerations, such as ensuring effective interaction with developers.

8.2.2 Path Forward. As shown in Figure 8, there are many promising yet under-explored modules (highlighted in green) and paths (represented by the gray dotted lines.) Looking ahead, we can outline three progressively advanced stages for exploring new directions. In the first stage, researchers can study these under-explored modules individually. In the second stage, they can investigate the under-explored paths, which may include combinations of under-explored modules and well-established ones or consist solely of multiple under-explored modules. Finally, in the third stage, by transforming all under-explored modules into well-understood components and studying various under-explored paths and combinations, researchers can summarize their findings and analyze strategies for better developing LLM-based solutions, ultimately leading to the new generation of powerful LLM-based vulnerability detection and repair solutions. Below, we describe the three stages in detail.

Stage 1: Exploring Under-Explored Modules Individually. Figure 8 highlights several under-explored modules in green, each offering exciting potential for discovery. We outline below some promising opportunities for deeper investigation into these areas:

— *Opportunity 1: Curating High-Quality Test Set for Vulnerability Detection.* The absence of a high-quality vulnerability dataset poses a significant obstacle to vulnerability detection. While obtaining fully correct labels for a large dataset is expensive, a viable solution is to curate a high-quality test set (which is much smaller than the whole dataset) that can accurately assess progress in vulnerability detection. An easy approach for future work is to combine the manually checked vulnerability data samples scattered across several separate research works (e.g., [19, 35, 48]) to form a high-quality test set. Future works can then do an empirical study to recognize the real progress in vulnerability detection with the high-quality test set. Furthermore, the community can maintain a living high-quality test set by adding new manually verified data to it. This curated test set can serve as a reliable benchmark for vulnerability detection.

— *Opportunity 2: Repo-Level Vulnerability Detection/Repair.* Current vulnerability detection and repair techniques primarily focus on the function or line level. One key reason is the input length limitations (512 subtokens) of the small LLMs like CodeBERT and CodeT5, which are predominantly used in existing studies. The 512 subtokens limitation aligns well with the function level data but faces difficulty in scaling up to classes or repositories. However, the emergence of recent larger LLMs with significantly higher input length capacities, such as GPT-4 (which can handle 128k subtokens), facilitates more effective processing of repo-level data. This presents an opportunity for future research to explore repo-level vulnerability detection/repair by leveraging these larger LLMs. For this direction, one recent study [129] has initially explored the repo-level vulnerability detection task. Specifically, it compared SAST tools (e.g., CodeQL) with popular or state-of-the-art open-source LLMs (e.g., Code Llama) for

detecting software vulnerabilities in software repositories. The experimental results indicated that SAST tools achieve low vulnerability detection rates with relatively low false positives, whereas LLMs can detect more vulnerabilities but tend to suffer from high false positive rates. Alongside this initial effort, there is considerable potential and opportunities for further research in this direction.

— *Opportunity 3: Customized LLMs for Vulnerability.* Currently, widely used LLMs in vulnerability detection/repair are general-purpose LLMs (e.g., CodeBERT, CodeT5, and GPT-3.5) that do not fully exploit the wealth of open-source vulnerability data. A promising avenue is the development of customized LLMs tailored for vulnerability data. Some initial attempts in this direction include vulnGPT [97] and Microsoft Security Copilot [61]. However, as these solutions are proprietary, their customized LLM details may not be fully disclosed. We advocate for collaborative efforts to develop open-sourced and effective customized LLMs for vulnerability.

— *Opportunity 4: Advanced LLM Usage and Adaptation.* Regarding LLMs usages, beyond the techniques observed in the included studies—such as *fine-tuning*, *prompt engineering*, and RAG—there exist two more advanced usages of LLMs [62] that have not been explored yet in vulnerability detection and repair: (1) *LLM Agent*: LLMs can serve as agents to decompose complex tasks into smaller components and employ multiple LLMs to address them [100]; (2) *Usage of External Tools*: LLMs can utilize external tools such as search engines, external databases, and other resources to enhance them [77]. Regarding detailed LLMs adaptation techniques, as illustrated in Figures 6 and 7, the majority of proposed adaptations in this field are designed for *fine-tuning* and *prompt engineering*. However, a plethora of advanced adaptation techniques for RAG remains under-explored. These techniques include iterative RAG [86] and recursive RAG [95], and adaptive RAG [11]. Researchers can consider using those unexplored advanced LLM usages/adaptations in future works.

— *Opportunity 5: Support Deployment-Ready Features.* Efforts could focus on enhancing LLM-based solutions with deployment-ready features, such as user interaction capabilities and seamless integration into existing developer workflows. This includes developing intuitive interfaces that allow developers to provide real-time feedback on suggestions made by the LLMs, thereby creating a more collaborative environment. Additionally, implementing functionalities that explain the reasoning behind the model's recommendations can help demystify the decision-making process, fostering trust and facilitating better adoption among developers. Moreover, integrating these solutions into popular IDEs or version control systems can streamline the workflow, making it easier for developers to utilize LLMs as part of their daily practices. By prioritizing these enhancements, LLM-based tools can significantly improve their usability and effectiveness in real-world scenarios, ultimately contributing to more robust vulnerability detection and repair processes.

Stage 2: Exploring Under-Explored Paths. The under-explored paths (mainly represented by gray dotted lines from data to deployment) may include combinations of under-explored modules with well-established ones or consist solely of multiple under-explored modules. This stage presents numerous opportunities. Below, we first highlight some examples of studying these combinations of under-explored modules alongside well-established ones:

— Once the community develops customized LLMs, researchers can apply well-studied fine-tuning or prompting-based adaptation techniques to improve function-level or line-level vulnerability detection or repair. These enhanced models can further be extended to support developer interaction, creating more practical and integrated solutions.

—Once new datasets are established—such as those containing manually verified vulnerabilities and associated test cases—researchers can re-evaluate existing LLM methods to assess their effectiveness in vulnerability detection and repair. Exploring various combinations of under-explored aspects and well-studied approaches can yield valuable insights and advancements.
—Researchers can leverage general-purpose LLMs (e.g., CodeBERT, CodeT5, and GPT-3.5) to develop repo-level vulnerability detection and repair tools, also facilitating interaction with developers.

There are also many opportunities in paths that consist solely of multiple under-explored modules. For instance, researchers could design a repo-level vulnerability detection or repair method using customized vulnerability-specific LLMs and enhance its effectiveness by incorporating LLM agent techniques. Finally, the tool can be integrated into IDEs, leveraging LLMs to improve interaction capabilities.

Stage 3: Summarizing Findings and Applying Them for Next-Generation LLM-Based Solutions. In this most advanced stage, researchers are expected to transform various under-explored aspects into well-studied areas and investigate diverse under-explored paths. At this stage, they can summarize their findings and analyze strategies for improving the development of LLM-based solutions. By leveraging this experience and knowledge, they may push the boundaries of what LLMs can achieve in vulnerability detection and repair to new heights.

Ideally, by 2030, following the roadmap illustrated in Figure 8, the community can develop a highly effective method capable of performing line-level, method-level, class-level, and repository-level vulnerability detection and repair, utilizing an expert-annotated benchmark to ensure accurate evaluation. Furthermore, the ideal solution should facilitate seamless communication and collaboration with developers, achieving high accuracy and robustness while providing trustworthy, expert-level insights into real-world vulnerabilities.

9 Threats to Validity

The potential threat to validity is the risk of inadvertently excluding relevant studies during the literature search and selection phase. Incomplete summarization of keywords for vulnerability detection/repair or varied terminologies of LLMs may have caused relevant research studies to be missed in our review. To mitigate this risk, we initially performed a manual selection of 17 high-impact venues and extracted a relatively comprehensive set of standard keywords from relevant papers within these venues. In addition, we further augmented our search results by combining automated search with forward-backward snowballing.

10 Conclusion and Future Work

The use of LLMs for vulnerability detection and repair has been garnering increasing attention. This paper presents an SLR of 58 primary studies on LLMs for vulnerability detection and repair. This review begins by analyzing the types of LLMs used in primary studies, shedding light on researchers' preferences for different LLMs. Subsequently, we categorized a variety of techniques for adapting LLMs. Through our analysis, this review also identifies the limitations in this field and proposes a research roadmap outlining promising avenues for future exploration. In the future, we plan to broaden this literature review by incorporating additional vulnerability-related tasks, such as vulnerability localization and vulnerability assessment.

References

[1] ACM Digital Library. Retrieved from https://dl.acm.org
[2] arXiv Database. Retrieved from https://arxiv.org

[3] IEEE Xplore Database. Retrieved from https://ieeexplore.ieee.org
[4] ScienceDirect Database. Retrieved from https://www.sciencedirect.com
[5] SpringerLink Database. Retrieved from https://link.springer.com
[6] Web of Science Database. Retrieved from https://www.webofscience.com
[7] Wiely Database. Retrieved from https://onlinelibrary.wiley.com
[8] Online Appendix for This Review. 2024. Retrieved from https://docs.google.com/document/d/18-UrkfH35CNMGRjjsDYZGK6L1aC9wP3GsKCtrIekcUQ/edit?usp=sharing
[9] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2023. Fixing hardware security bugs with large language models. arXiv:2302.01215. Retrieved from https://arxiv.org/abs/2302.01215
[10] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. arXiv:2103.06333. Retrieved from https://arxiv.org/abs/2103.06333
[11] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection. arXiv:2310.11511. Retrieved from https://arxiv.org/abs/2310.11511
[12] Berkay Berabi, Alexey Gronskiy, Veselin Raychev, Gishor Sivanrupan, Victor Chibotaru, and Martin T. Vechev. 2024. DeepCode AI fix: Fixing security vulnerabilities with large language models. arXiv:2402.13291. Retrieved from https://arxiv.org/abs/2402.13291
[13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Proceedings of the International Conference on Neural Information Processing Systems*, 1877–1901.
[14] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: Generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.), ACM, New York, NY, 18–30. DOI: https://doi.org/10.1145/3540250.3549162
[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374
[16] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David A. Wagner. 2023. 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 (RAID '23)*. ACM, New York, NY, 654–668. DOI: https://doi.org/10.1145/3607199.3607242
[17] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2023. Neural transfer learning for repairing security vulnerabilities in C code. *IEEE Transactions on Software Engineering* 49, 1 (2023), 147–165. DOI: https://doi.org/10.1109/TSE.2022.3147265
[18] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2023. SeqTrans: Automatic vulnerability Fix via sequence to sequence learning. *IEEE Transactions on Software Engineering* 49, 2 (2023), 564–585. DOI: https://doi.org/10.1109/TSE.2022.3156637
[19] Roland Croft, Muhammad Ali Babar, and M. Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE '23)*. IEEE, 121–133. DOI: https://doi.org/10.1109/ICSE48619.2023.00022
[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805. Retrieved from https://arxiv.org/abs/1810.04805
[21] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David A. Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability detection with code language models: How far are we? arXiv:2403.18624. Retrieved from https://arxiv.org/abs/2403.18624
[22] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2024. Vul-RAG: Enhancing LLM-based vulnerability detection via knowledge-level RAG. arXiv:2406.11147. Retrieved from https://arxiv.org/abs/2406.11147
[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. arXiv:2002.08155. Retrieved from https://arxiv.org/abs/2002.08155
[24] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A generative model for code infilling and synthesis. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*. OpenReview.Net. Retrieved from https://openreview.net/pdf?id=hQwb-lbM6EL
[25] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2024. Vision transformer-inspired automated vulnerability repair. *ACM Transactions on Software Engineering and Methodology* 33, 3, Article 78 (2024), 1–29.

[26] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories (MSR '22)*. ACM, New York, NY, 608–620. DOI: https://doi.org/10.1145/3524842.3528452

[27] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Q. Phung, and John C. Grundy. 2024. AIBugHunter: A practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering* 29, 1 (2024), 4. DOI: https://doi.org/10.1007/S10664-023-10346-3

[28] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Q. Phung. 2022. VulRepair: A T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.), ACM, New York, NY, 935–947. DOI: https://doi.org/10.1145/3540250.3549098

[29] Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. 2023. *ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?* APSEC.

[30] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys* 50, 4 (2017), 1–36.

[31] GitHub. 2023. GitHub Copilot. Retrieved from https://copilot.github.com

[32] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, 7212–7225.

[33] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. arXiv:2009.08366. Retrieved from https://arxiv.org/abs/2009.08366

[34] Hazim Hanif and Sergio Maffeis. 2022. VulBERTa: Simplified source code Pre-training for vulnerability detection. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN '22)*. IEEE, 1–8. DOI: https://doi.org/10.1109/IJCNN55064.2022.9892280

[35] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 1865–1879.

[36] Junda He, Xin Zhou, Bowen Xu, Ting Zhang, Kisub Kim, Zhou Yang, Ferdian Thung, Ivana Clairine Irsan, and David Lo. 2023. Representation learning for stack overflow posts: How far are we? *ACM Transactions on Software Engineering and Methodology* 33, 3, Article 69 (2023), 1–14.

[37] Xinying Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. Retrieved from https://api.semanticscholar.org/CorpusID:261048648

[38] Nafis Tanveer Islam, Joseph Khoury, Andrew Seong, Gonzalo De La Torre Parra, Elias Bou-Harb, and Peyman Najafirad. 2024. LLM-powered code vulnerability repair with reinforcement learning and semantic reward. arXiv:2401.03374. Retrieved from https://arxiv.org/abs/2401.03374

[39] Nafis Tanveer Islam and Peyman Najafirad. 2024. Code security vulnerability repair using reinforcement learning with large language models. In *Proceedings of the AAAI Workshop*.

[40] Minhao Jiang, Ken Ziyu Liu, Ming Zhong, Rylan Schaeffer, Siru Ouyang, Jiawei Han, and Sanmi Koyejo. 2024. Investigating data contamination for pre-training language models. arxiv:2401.06059.

[41] Zhonghao Jiang, Weifeng Sun, Xiaoyan Gu, Jiaxin Wu, Tao Wen, Haibo Hu, and Meng Yan. 2024. DFEPT: Data flow embedding for enhancing pre-trained model based vulnerability detection. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware (Internetware '24)*. Hong Mei, Jian Lv, Abdelsalam Helal, Xiaoxing Ma, Shing-Chi Cheung, Jie Zhang, and Tao Zhang (Eds.), ACM, New York, NY. DOI: https://doi.org/10.1145/3671016.3671388

[42] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2019. Learning and evaluating contextual embedding of source code. In *Proceedings of the International Conference on Machine Learning*. Retrieved from https://api.semanticscholar.org/CorpusID:220425306

[43] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the effectiveness of large language models in detecting security vulnerabilities. arXiv: 2311.16169. Retrieved from https://arxiv.org/abs/2311.16169

[44] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS '22)*. Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.), Retrieved from http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[45] Hongyu Kuang, Feng Yang, Long Zhang, Gaigai Tang, and Lin Yang. 2023. Leveraging user-defined identifiers for counterfactual data generation in source code vulnerability detection. In *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '23)*. Leon Moonen, Christian D. Newman, and Alessandra Gorla (Eds.), IEEE, 143–150. DOI: https://doi.org/10.1109/SCAM59687.2023.00024

[46] Triet Huynh Minh Le, Xiaoning Du, and Muhammad Ali Babar. 2024. Are latent vulnerabilities hidden gems for software vulnerability prediction? An empirical study. In *Proceedings of the 21st IEEE/ACM International Conference on Mining Software Repositories (MSR '24)*. Diomidis Spinellis, Alberto Bacchelli, and Eleni Constantinou (Eds.), ACM, New York, NY, 716–727. DOI: https://doi.org/10.1145/3643991.3644919

[47] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS '20)*. Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.), Retrieved from https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[48] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. 2023b. Comparison and evaluation on static application security testing (SAST) tools for Java. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 921–933.

[49] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: May the Source Be with You! Retrieved from https://api.semanticscholar.org/CorpusID:258588247

[50] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the effectiveness of function-level vulnerability detectors for inter-procedural vulnerabilities. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE '24)*. ACM, New York, NY, 157:1–157:12. DOI: https://doi.org/10.1145/3597503.3639218

[51] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the effectiveness of function-level vulnerability detectors for inter-procedural vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*.

[52] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A large-scale survey on the usability of AI programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE '24)*. ACM, New York, NY, 52:1–52:13. DOI: https://doi.org/10.1145/3597503.3608128

[53] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE* 108, 10 (2020), 1825–1848.

[54] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. arXiv:1907.11692. Retrieved from https://arxiv.org/abs/1907.11692

[55] Zhihong Liu, Qing Liao, Wenchao Gu, and Cuiyun Gao. 2023. Software vulnerability detection with GPT and in-context learning. In *Proceedings of the 8th International Conference on Data Science in Cyberspace (DSC '23)*. IEEE, 229–236. DOI: https://doi.org/10.1109/DSC59305.2023.00041

[56] Zhongxin Liu, Zhijie Tang, Junwei Zhang, Xin Xia, and Xiaohu Yang. 2024. Pre-training by predicting program dependencies for vulnerability analysis tasks. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*.

[57] David Lo. 2023. Trustworthy and synergistic artificial intelligence for software engineering: Vision and roadmaps. arXiv:2309.04142. Retrieved from https://arxiv.org/abs/2309.04142

[58] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv:2102.04664. Retrieved from https://arxiv.org/abs/2102.04664

[59] Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2024. Towards causal deep learning for vulnerability detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*.

[60] Meta. 2023. Code llama: Open Foundation Models for Code. Retrieved from https://ai.meta.com/research/publications/code-llama-open-foundation-models-for-code/

[61] Microsoft. 2024. Microsoft Copilot for Security. Retrieved from https://microsoft.github.io/PartnerResources/skilling/microsoft-security-academy/microsoft-security-copilot

[62] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large language models: A survey. arxiv:2402.06196. Retrieved from https://arxiv.org/abs/2402.06196

[63] Chao Ni, Liyu Shen, Xiaodan Xu, Xin Yin, and Shaohua Wang. 2024. Learning-based models for vulnerability detection: An extensive study. arXiv:2408.07526. Retrieved from https://arxiv.org/abs/2408.07526

[64] Chao Ni, Xin Yin, Kaiwen Yang, Dehai Zhao, Zhenchang Xing, and Xin Xia. 2023. Distinguishing look-alike innocent and vulnerable code by subtle semantic representation learning and explanation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1611–1622.

[65] Xu Nie, Ningke Li, Kailong Wang, Shangguang Wang, Xiapu Luo, and Haoyu Wang. 2023. Understanding and tackling label errors in deep learning-based vulnerability detection (Experience paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. René Just and Gordon Fraser (Eds.), ACM, New York, NY, 52–63. DOI: https://doi.org/10.1145/3597926.3598037

[66] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for training LLMs on programming and natural languages. arXiv:2305.02309. Retrieved from https://arxiv.org/abs/2305.02309

[67] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. arXiv:2305.02309. Retrieved from https://arxiv.org/abs/2305.02309

[68] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An open large language model for code with multi-turn program synthesis. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*. OpenReview.Net. Retrieved from https://openreview.net/pdf?id=iaYcJKpY2B_

[69] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. 2024. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. arXiv:2402.17230. Retrieved from https://arxiv.org/abs/2402.17230

[70] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2024b. Automated software vulnerability patching using large language models. arXiv:2408.13597. Retrieved from https://arxiv.org/abs/2408.13597

[71] OpenAI. 2022. GPT-3.5. Retrieved from https://platform.openai.com/docs/models/gpt-3-5

[72] OpenAI. 2023. GPT-4 technical report. arXiv:2303.08774. Retrieved from https://arxiv.org/abs/2303.08774

[73] Shirui Pan, Linhao Luo, Yufei Wang, Chen Chen, Jiapu Wang, and Xindong Wu. 2023. Unifying large language models and knowledge graphs: A roadmap. arXiv:2306.08302. Retrieved from https://arxiv.org/abs/2306.08302

[74] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[75] Tao Peng, Shixu Chen, Fei Zhu, Junwei Tang, Junping Liu, and Xinrong Hu. 2023. PTLVD:Program slicing and transformer-based line-level vulnerability detection system. In *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '23)*. Leon Moonen, Christian D. Newman, and Alessandra Gorla (Eds.), IEEE, 162–173. DOI: https://doi.org/10.1109/SCAM59687.2023.00026

[76] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. 2023. Software vulnerability detection using large language models. In *Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering (ISSRE '23)*. IEEE, 112–119. DOI: https://doi.org/10.1109/ISSREW60843.2023.00058

[77] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. arXiv:2307.16789. Retrieved from https://arxiv.org/abs/2307.16789

[78] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019), 9.

[79] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[80] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: A method for automatic evaluation of code synthesis. arXiv:2009.10297. Retrieved from https://arxiv.org/abs/2009.10297

[81] Niklas Risse and Marcel Böhme. 2024. Uncovering the limits of machine learning for automatic vulnerability detection. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security '24)*. Davide Balzarotti and Wenyuan Xu (Eds.), USENIX Association. Retrieved from https://www.usenix.org/conference/usenixsecurity24/presentation/risse

[82] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. arXiv:1707.06347. Retrieved from https://arxiv.org/abs/1707.06347

[83] Adriana Sejfia, Satyaki Das, Saad Shafiq, and Nenad Medvidovic. 2024. Toward improved deep learning-based vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE '24)*. ACM, New York, NY, 62:1–62:12. DOI: https://doi.org/10.1145/3597503.3608141

[84] Adriana Sejfia, Satyaki Das, Saad Shafiq, and Nenad Medvidović. 2024. Toward improved deep learning-based vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 1–12.

[85] Hossain Shahriar and Mohammad Zulkernine. 2012. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys* 44, 3 (2012), 1–46.

[86] Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. arXiv:2305.15294. Retrieved from https://arxiv.org/abs/2305.15294

[87] Alexey Shestov, Anton Cheshkov, Rodion Levichev, Ravil Mussabayev, Pavel Zadorozhny, Evgeny Maslov, Chibirev Vadim, and Egor Bulychev. 2024. Finetuning large language models for vulnerability detection. arXiv:2401.17010. Retrieved from https://arxiv.org/abs/2401.17010

[88] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An empirical study of deep learning models for vulnerability detection. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE '23)*. IEEE, 2237–2248. DOI: https://doi.org/10.1109/ICSE48619.2023.00188

[89] Benjamin Steenhoek, Md Mahbubur Rahman, Shaila Sharmin, and Wei Le. 2023. Do language models Learn semantics of code? A case study in vulnerability detection. arXiv:2311.04109. Retrieved from https://arxiv.org/abs/2311.04109

[90] Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. 2020. Code and named entity recognition in StackOverflow. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*. Retrieved from https://www.aclweb.org/anthology/2020.acl-main.443/

[91] Wei Tang, Mingwei Tang, Minchao Ban, Ziguo Zhao, and Mingjun Feng. 2023. CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software* 199 (2023), 111623. DOI: https://doi.org/10.1016/J.JSS.2023.111623

[92] Edward Targett. 2022. We analysed 90,000+ Software Vulnerabilities: Here's What We Learned. Retrieved from https://www.thestack.technology/analysis-of-cves-in-2022-software-vulnerabilities-cwes-most-dangerous/

[93] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-based language models for software vulnerability detection. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC '22)*. ACM, New York, NY, 481–496. DOI: https://doi.org/10.1145/3564625.3567985

[94] Hoai-Chau Tran, Anh-Duy Tran, and Kim-Hung Le. 2024. DetectVul: A statement-level code vulnerability detection for Python. *Future Generation Computer Systems* 163, 4 (2024), 107504.

[95] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. arXiv:2212.10509. Retrieved from https://arxiv.org/abs/2212.10509

[96] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the International Conference on Neural Information Processing Systems*.

[97] Vicarius. 2023. Vuln_GPT debuts as AI-powered approach to find and remediate software vulnerabilities. Retrieved from https://venturebeat.com/ai/got-vulns-vuln_gpt-debuts-as-ai-powered-approach-to-find-and-remediate-software-vulnerabilities/

[98] Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. 2024. Combining structured static code information and dynamic symbolic traces for software vulnerability prediction. In *Proceedings of the 46th International Conference on Software Engineering*. ACM, New York, NY.

[99] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software testing with large language model: Survey, landscape, and vision. arXiv:2307.07221. Retrieved from https://arxiv.org/abs/2307.07221

[100] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 1–26.

[101] Ruoke Wang, Zongjie Li, Chaozheng Wang, Yang Xiao, and Cuiyun Gao. 2024. NAVRepair: Node-type aware C/C++ code vulnerability repair. arXiv:2405.04994. Retrieved from https://arxiv.org/abs/2405.04994

[102] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859. Retrieved from https://arxiv.org/abs/2109.00859

[103] Ying Wei, Lili Bo, Xiaoxue Wu, Yue Li, Zhenlei Ye, Xiaobing Sun, and Bin Li. 2023. VulRep: Vulnerability repair based on inducing commits and fixing commits. *EURASIP Journal on Wireless Communications and Networking* 2023, 1 (2023), 34.

[104] Xin-Cheng Wen, Xinchen Wang, Yujia Chen, Ruida Hu, David Lo, and Cuiyun Gao. 2024. VulEval: Towards repository-level evaluation of software vulnerability detection. arXiv:2404.15596. Retrieved from https://arxiv.org/abs/2404.15596

[105] Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. 2023. When less is enough: Positive and unlabeled learning model for vulnerability detection. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*. IEEE, 345–357. DOI: https://doi.org/10.1109/ASE56229.2023.00144

[106] Cheng Weng, Yihao Qin, Bo Lin, Pei Liu, and Liqian Chen. 2024. MatsVD: Boosting statement-level vulnerability detection via dependency-based attention. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware (Internetware '24)*. Hong Mei, Jian Lv, Abdelsalam Helal, Xiaoxing Ma, Shing-Chi Cheung, Jie Zhang, and Tao Zhang (Eds.), ACM, New York, NY. DOI: https://doi.org/10.1145/3671016.3674807

[107] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, New York, NY, 38:1–38:10.

[108] Bolun Wu, Futai Zou, et al. 2022. Code vulnerability detection based on deep sequence and graph models: A survey. *Security and Communication Networks* 2022 (2022), 1282–1294.

[109] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. René Just and Gordon Fraser (Eds.), ACM, New York, NY, 1282–1294. DOI: https://doi.org/10.1145/3597926.3598135

[110] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS@PLDI '22)*. Swarat Chaudhuri and Charles Sutton (Eds.), ACM, New York, NY, 1–10. DOI: https://doi.org/10.1145/3520312.3534862

[111] Fabian Yamaguchi. 2023. Joern: A Source Code Analysis Tool. Retrieved from https://github.com/octopus-platform/joern.

[112] Aidan ZH Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. 2024. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. arXiv:2406.05892. Retrieved from https://arxiv.org/abs/2406.05892

[113] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Ben Hu. 2024. Harnessing the power of LLMs in practice: A survey on ChatGPT and beyond. *ACM Transactions on Knowledge Discovery from Data* 18, 6 (2024), 160:1–160:32. DOI: https://doi.org/10.1145/3649506

[114] Xu Yang, Shaowei Wang, Yi Li, and Shaohua Wang. 2023. Does data sampling improve deep learning-based vulnerability detection? Yeas! and Nays!. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE '23)*. IEEE, 2287–2298. DOI: https://doi.org/10.1109/ICSE48619.2023.00192

[115] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*.

[116] Xin Yin. 2024. Pros and cons! Evaluating ChatGPT on software vulnerability. arXiv:2404.03994. Retrieved from https://arxiv.org/abs/2404.03994

[117] Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. 2023. Practices and challenges of using GitHub Copilot: An empirical study. In *Proceedings of the 35th International Conference on Software Engineering and Knowledge Engineering (SEKE '23)*. Shi-Kuo Chang (Ed.), KSI Research Inc., 124–129. DOI: https://doi.org/10.18293/SEKE2023-077

[118] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2023. Prompt-enhanced software vulnerability detection using ChatGPT. arXiv:2308.12697. Retrieved from https://arxiv.org/abs/2308.12697

[119] He Zhang, Muhammad Ali Babar, and Paolo Tell. 2011. Identifying relevant studies in software engineering. *Information and Software Technology* 53, 6 (2011), 625–637.

[120] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2023. Vulnerability detection by learning from syntax-based execution paths of code. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4196–4212. DOI: https://doi.org/10.1109/TSE.2023.3286586

[121] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–69.

[122] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A survey on large language models for software engineering. arXiv:2312.15223. Retrieved from https://arxiv.org/abs/2312.15223

[123] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. 2023d. Pre-trained model-based automated software vulnerability repair: How far are we? *IEEE Transactions on Dependable and Secure Computing* (2023).

[124] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. BERTScore: Evaluating text generation with BERT. In *Proceedings of the 8th International Conference on Learning Representations (ICLR '20)*. OpenReview.net. Retrieved from https://openreview.net/forum?id=SkeHuCVFDr

[125] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the perspectives of NLP and software engineering: A survey on language models for code. arXiv:2311.07989. Retrieved from https://arxiv.org/abs/2311.07989

[126] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. arXiv:2303.18223. Retrieved from https://arxiv.org/abs/2303.18223

[127] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 872–872.

[128] Xin Zhou, Kisub Kim, Bowen Xu, Jiakun Liu, DongGyun Han, and David Lo. 2023. The Devil is in the tails: How long-tailed code distributions impact large language models. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 40–52.

[129] Xin Zhou, Duc-Manh Tran, Thanh Le-Cong, Ting Zhang, Ivana Clairine Irsan, Joshua Sumarlin, Bach Le, and David Lo. 2024. Comparison of static application security testing tools and large language models for repo-level vulnerability detection. arXiv:2407.16235. Retrieved from https://arxiv.org/abs/2407.16235

[130] Xin Zhou, Bowen Xu, DongGyun Han, Zhou Yang, Junda He, and David Lo. 2023. CCBERT: Self-supervised code change representation learning. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 182–193.

[131] Xin Zhou, Ting Zhang, and David Lo. 2024. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE NIER Track)*.

[132] Zhou Zhou, Lili Bo, Xiaoxue Wu, Xiaobing Sun, Tao Zhang, Bin Li, Jiale Zhang, and Sicong Cao. 2022. SPVF: Security property assisted vulnerability fixing via attention-based models. *Empirical Software Engineering* 27, 7 (2022), 171.

[133] Noah Ziems and Shaoen Wu. 2021. Security vulnerability detection using deep learning natural language processing. In *Proceedings of the IEEE Conference on Computer Communications Workshops (INFOCOM Workshops '21)*. IEEE, 1–6. DOI: https://doi.org/10.1109/INFOCOMWKSHPS51825.2021.9484500

Received 3 April 2024; revised 30 September 2024; accepted 19 November 2024