frontiers | Frontiers in Computer Science

TYPE Review
PUBLISHED 24 November 2025
DOI 10.3389/fcomp.2025.1655469

OPEN ACCESS

EDITED BY
Novarun Deb,
University of Calgary, Canada

REVIEWED BY
Livinus Obiora Nweke,
Noroff Education AS, Norway
T. Gururaj,
Visvesvaraya Technological University, India

*CORRESPONDENCE
Eleonora Iotti
✉ eleonora.iotti@unipr.it

RECEIVED 27 June 2025
ACCEPTED 24 October 2025
PUBLISHED 24 November 2025

CITATION
Dolcetti G and Iotti E (2025) A dual perspective review on large language models and code verification. *Front. Comput. Sci.* 7:1655469. doi: 10.3389/fcomp.2025.1655469

A dual perspective review on large language models and code verification

Greta Dolcetti[1] and Eleonora Iotti[2]*

[1]Department of Environmental Sciences, Informatics and Statistics, Ca' Foscari University of Venice, Venice, Italy, [2]Department of Mathematical, Physical and Computer Sciences, University of Parma, Parma, Italy

Recent advances in Large Language Models (LLMs) have sparked significant interest in their application to code verification and the assessment of LLM-generated code safety. This review examines current research on the intersection of LLMs with software verification, focusing on two main aspects: the use of LLMs as verification tools and the verification of code produced by LLMs. We analyze the emerging approaches for integrating LLMs with traditional static analyzers and formal verification tools, including prompt engineering techniques and combinations with established verification frameworks. The review explores various verification methodologies, from standalone LLM applications to hybrid approaches incorporating traditional verification methods. We examine research addressing the safety assessment of LLM-generated code and investigate frameworks developed for vulnerability detection and repair. Through this analysis, we aim to provide insights into the current state of LLM applications in code verification, identify key challenges in the field, and outline important directions for future research in this rapidly evolving domain.

KEYWORDS

review, large language models, code verification, code generation, code intelligence

1 Introduction

Large Language Models (LLMs) have substantially transformed Artificial Intelligence, particularly influencing the domain of automatic code generation (e.g., Tihanyi et al., 2023). In recent years, these models have expanded remarkably in scale and capability, reshaping the ways in which code is generated and utilized (Jiang et al., 2025; Jain et al., 2024). Moreover, recent studies, including Chapman et al. (2024) and Li Z. et al. (2024), have introduced innovative approaches that leverage LLMs for static analysis, more precisely, positioning LLMs as static analyzers themselves. However, the increased productivity enabled by LLMs comes with inherent risks. As developers increasingly incorporate these models into automated coding workflows, concerns about the integrity and security of the produced code have become increasingly prominent, as highlighted in recent studies like Pearce et al. (2022) and Ullah et al. (2024): although automated code generation can streamline development processes, it can also introduce subtle errors or vulnerabilities that traditional verification methods or fast deployment might not immediately detect. This inherent trade-off between increased productivity and potential compromises in code security forms the core motivation behind this review. This tension encapsulates a multifaceted challenge. On one side, the promise of LLMs is tied to their ability to process voluminous datasets and generate code quickly, which can significantly enhance developer productivity, for example allowing fast prototyping (Kolthoff et al., 2025). On the other side, the opacity of these models, inherently due to their black

box and non-explainable architectures, often makes debugging and security verification a challenging task (Liu et al., 2024b).

In response to the challenges accompanying LLM-driven code generation, our primary objective is to map the intersection between automated code production and code security. We observed that review and survey studies available in literature focus on specific aspects of these problems: for example, strategies for program repair (Huang et al., 2025; Zhang et al., 2024), security issues in code generation (Basic and Giaretta, 2024; Negri-Ribalta et al., 2024), vulnerability detection capabilities (Zhou et al., 2024a), and so on. These aspects probe very different skills of LLMs, thereby positioning them along only a single axis of evaluation. However, we also noticed that cutting-edge proposals in the experimental literature on LLMs increasingly attempt to address multiple tasks in an all-in-one fashion (Alrashedy et al., 2023; Pearce et al., 2023; Gong et al., 2024), often relying on a partial understanding of the true capabilities of these models. By examining this body of work more closely, we concluded that a major subdivision should be made by identifying two distinct points of view. Therefore, in this review, we propose a dual perspective: (i) exploring the potential of LLMs as tools for code verification (LLM4Verification) and (ii) assessing techniques for verifying the integrity of code produced by these models (Verification4LLM). This mapping exercise involves analyzing the literature to pinpoint the strengths and weaknesses of current approaches, illustrating them and their limitations. To achieve this, we conducted a comprehensive review of current methodologies and empirical studies that address both the use of LLMs for verification and the development of techniques aimed at enhancing the security of LLM-generated code.

A taxonomy of existing LLM-based techniques for secure code generation and verification is provided to classify the diverse range of strategies proposed to improve both the productivity and security of the LLM-generated code and outputs. This taxonomy includes methods such as prompt engineering (Marvin et al., 2023)—where the quality of input prompts is optimized to achieve better and more secure outputs—as well as techniques like Retrieval-Augmented Generation (RAG) (Lewis et al., 2020), which leverages external databases to enhance the generated content. Additionally, we examine tool integrations that combine LLM outputs with traditional security verification tools, offering a layered defense mechanism against vulnerabilities. This taxonomy serves as a structured framework that simplifies the complex landscape of current research.

This paper presents a comprehensive review of the intersection between LLMs and code verification, offering a dual-perspective framework that advances understanding and research in this emerging field. Our main contributions are summarized as follows:

- Dual perspective framework: We introduce a novel dual perspective by analyzing both the use of LLMs as tools for code verification (LLM4Verification) and the verification of code generated by LLMs themselves (Verification4LLM). This bifocal approach enables a holistic understanding of the challenges and opportunities in leveraging LLMs for software security.
- Taxonomy of strategies: We develop a detailed taxonomy categorizing the diverse strategies employed to enhance the productivity and security of LLM-generated code. This taxonomy encompasses prompt engineering techniques, fine-tuning, integration with traditional static analyzers and formal verification tools, RAG, and emerging agentic frameworks, providing a structured framework to navigate the complex landscape of current research.
- Overview of framework and datasets: We review and analyze a broad spectrum of LLM-based framework and datasets for program verification addressed in the literature. This overview highlights prevalent programming languages, common security weaknesses introduced by LLM-generated code, and the state-of-the-art models and benchmarks utilized in secure code generation, vulnerability detection and repair.
- Identification of future directions: Through critical analysis, we identify key limitations and open challenges in current methodologies, such as high false positive rates in vulnerability detection, limitations in handling complex vulnerabilities, and scalability issues.

Together, these contributions aim to provide a foundational reference for researchers and practitioners seeking to harness LLMs for secure and reliable software development.

The structure of the paper is organized as follows. Section 2 details the review methodology, including research questions, inclusion and exclusion criteria, and quality assessment. Section 3 surveys related work on LLMs in code verification and security. Section 4 examines the use of LLMs as verification tools, presenting strategies, empirical findings, and limitations. Section 5 focuses on the verification of code generated by LLMs, discussing methodologies, effectiveness, and common vulnerabilities. Section 6 addresses cross-cutting aspects such as dataset availability, open-source practices, and ethical considerations. The paper is then concluded by Section 7.

2 Methodology

2.1 Relevant research questions

We identified eight Research Questions (RQs) and two Discussion (D) questions, which are categorized and shown in Table 1. Three categories are defined to subdivide the RQs and D questions conceptually, namely the use of LLMs as code verification tools (LLM4Verification), the verification of LLMs' generated code (Verification4LLM), and broader contextual and ethical considerations (Cross-cutting Aspects). In this section, as in the rest of the article, we will use the following color scheme: blue for LLM4Verification, green for Verification4LLM, and fuchsia for the remaining cross-cutting aspects. The first question, RQ1, aims to search for studies that compare baseline LLMs with traditional static- or dynamic-analysis approaches, as well as composite frameworks proposed in the literature. RQ2 and RQ3 investigate the effectiveness and limitations of common strategies by investigating their use in current literature, both for verifying existing code by means of LLMs powered approaches, and for verifying code directly produced by LLMs. RQ4 and RQ5 consider those studies that

TABLE 1 Research questions and discussion points.

Name and topic	Research question
RQ1: Effectiveness of LLMs and LLM-based frameworks	How effective are LLMs in detecting vulnerabilities in code, compared to traditional approaches and custom methods based on LLM?
RQ2: Strategies, effectiveness, and limitations of LLMs in code verification tasks	What are the key strategies (e.g., prompt engineering, tool integration, agents, RAG, etc.) that enhance the performance of LLMs in code verification tasks, how effective are they and what are their limitations?
RQ3: Strategies, effectiveness, and limitations of LLMs' secure code generation	What are the key strategies (e.g., prompt engineering, tool integration, agents, RAG, etc.) that enhance the safety and security of the code generated by LLMs, how effective are they and what are their limitations?
RQ4: Strategies for tool integration	How can LLMs be integrated with existing static analyzers and formal verification tools to create more robust verification frameworks?
RQ5: Code generation and tool integration	How can LLMs be integrated with existing static analyzers and formal verification tools to create more robust code generations?
RQ6: Common vulnerabilities introduced by LLMs	What types of vulnerabilities are most commonly introduced by LLMs? Are they the same introduced by human generated code?
RQ7: Effectiveness and limitations of LLMs in repairing vulnerabilities	How effective are LLMs in repairing vulnerabilities in code, and what are the limitations of LLM-based program repair techniques (eventually w.r.t. traditional techniques)?
RQ8: Open-source reliance in LLM-based software verification	To what extent is open-source code used in the selected paper on these topics?
D1: Code security awareness with LLMs	How can LLMs be used to educate developers about secure coding practices and improve overall code security awareness?
D2: Ethical considerations and potential risks	What are the ethical considerations and potential risks associated with using LLMs in code verification and software security?

integrate formal verification tools with LLMs. With respect to code generation, another question arises when comparing LLMs introduced vulnerabilities with those present in human-produced code, namely RQ6. As another topic of research, many studies focus on program repair in addition to, or in parallel with, vulnerability detection. Even in this case, we performed a survey on repair techniques, highlighting effectiveness and limitations in RQ7. We checked the availability of open-source code, data, and models, for each investigated source. Finally, we open a discussion about secure coding practices and the education of developers that could emerge from recent literature about LLMs in D1. Then we discuss the ethical considerations and potential risks that could derive from the use of those tools in D2.

TABLE 2 Numbered inclusion and exclusion criteria for paper selection.

Inclusion criteria	Exclusion criteria
I1. The paper employs LLMs for code-related tasks.	**E1.** The paper is less than 5 pages in length.
I2. The paper concerns the verification of code generated by LLMs or utilizes LLMs to verify code.	**E2.** The paper is gray literature (e.g., theses, technical reports, reviews, editorials, demos, etc.).
I3. The paper is available in full text, written in English, and published in conferences, workshops, journals, or on arXiv[a].	**E3.** The paper is not written in English.
I4. The paper proposes a novel approach, study (including empirical or experimental work), or tool.	**E4.** The paper uses LLMs for code but does not concern software verification.
	E5. The paper focuses on hardware bugs or vulnerabilities.
	E6. The paper is a duplicate.
	E7. The paper mentions LLMs only in the context of future work.

[a]https://arxiv.org/.

2.2 Research strategy and inclusion-exclusion criteria

In order to ensure the relevance, quality, and consistency of the papers included in this study, a set of well-defined inclusion and exclusion criteria was established. These criteria have been used as guidelines for creating the initial dataset of papers to analyze in the review. They have been selected to include papers that directly address the research questions, while filtering out those that do not meet the necessary standards or scope. Due to the vast and rapidly growing body of LLM-related studies, our inclusion criteria were designed to capture papers that employ LLMs for code-related tasks, particularly in the context of software verification, and that present novel approaches, empirical studies, or tools. Only papers available in full text, written in English, and published in reputable venues such as conferences, workshops, journals, or on arXiv were considered. Conversely, the exclusion criteria were formulated to omit papers that lack sufficient length, are not original research (e.g., gray literature), are not written in English, do not focus on software verification, address hardware bugs, are duplicates, or mention LLMs solely as future work. The careful application of these criteria is critical for maintaining the integrity and validity of the review, as it ensures that the selected dataset of 154 papers is both comprehensive and directly relevant to the investigation of LLMs in software verification. Table 2 summarizes the inclusion and exclusion criteria.

The number of papers relevant to the general topic of LLMs in software verification, and compliant with the inclusion and exclusion criteria, is **154**. These papers constitute the initial dataset.

2.3 Quality assessment

Of the 154 articles in the initial database, we assessed the overall quality of each single paper to select the portion of more impactful

TABLE 3 Quality assessment criteria.

Criteria	Description
Publisher/venue classification	Journals are evaluated based on their impact factor and quantile, referring to Scimago Journal & Country Rank[a]: Q1 journals are valued 1 point, Q2 journals are valued 0.5 points, Q3 and Q4 are valued 0 points; venues are evaluated using ICORE[b] classification: A* and A conferences are valued 1 point, B conferences are valued 0.5 points, others are valued 0 points.
Contributions	Contributions of the analyzed paper are scored based on their clarity and presentation in the text, possibly in the introductory part of the paper. Points are assigned as 0, 0.5, or 1.
Experimental setting	Experimental setup sections are evaluated based on the clarity, and on the presence of sufficient details to, at least in principle, reproduce the experiments. Points are assigned as 0, 0.5, or 1.
Results	Results are evaluated based on their presentation and clarity, taking into account appropriate comparison with other LLM-based SOTA methods, possible ablation studies, and understandable visualization. Points are assigned as 0, 0.5, or 1.
Limitations	Limitations are measured based on how clearly and in-depth they are discussed in the analyzed paper. Points are assigned as 0, 0.5, or 1.
Comparisons	The presence of comparisons with traditional methods and not-LLM-based approaches is indicated with a flag 0 or 1.
LLMs and strategies	If more than one LLM is used, and more than one strategy (i.e., prompt-engineering, RAG, fine-tuning, and so on) is leveraged, a score of 1 is assigned. 0.5 points are scored if only one of these two conditions holds, and 0 in other cases.
Open-source	The availability of an open-source reproducibility package for the analyzed paper is evaluated with a score of 1, otherwise 0 points are assigned.

[a]https://www.scimagojr.com/. [b]https://portal.core.edu.au/conf-ranks/.

and promising research. Each article was independently reviewed by the authors, who assigned numerical scores according to the quality assessment criteria described in Table 3. The assigned points are then discussed by the authors of this paper, reaching a common agreement on each score for each criteria, for each article analyzed. The final score of each paper ranges from 0 points to 8 points, and the articles with an average score greater than 6 were selected to contribute to the next investigation phase.

The number of papers that passed the quality assessment phase is **50**. These articles form the evidence base examined to address the research questions presented in Section 2.1.

2.4 Data extraction and categorization

The majority of papers included in our review were published between 2023 and 2024, with 16 and 24 papers selected from those years, respectively, with an average publication year of 2,023.56. Fewer papers were chosen from 2021 (1 paper), 2022 (4 papers), reflecting the initial period of publication in this topic, and 2025 (5 papers) due to the year not being finished yet. This reflects the temporal focus of our selection process and overall reflects the growing publication trends in the field. The distribution of selected papers by year is shown in Appendix Figure 3.

Most of the papers included in our review were published in A* (16 papers) and Q1 (13 papers) venues. Additionally, 11 papers were selected from arXiv, while fewer papers were chosen from A (8 papers), B (1 paper), and Q2 (1 paper) venues. This reflects our selection's emphasis on highly ranked publication sources. The distribution of selected papers by ranking is presented in Appendix Figure 4.

The distribution of adopted LLMs across studies shows a strong preference for CodeBERT, GPT-3.5 Turbo, and GPT-4, each of which appears in the highest number of selected papers. CodeT5 and GraphCodeBERT also feature prominently, indicating their widespread adoption in recent research. Other models such as GPT-3.5, RoBERTa, T5, and PLBART are frequently utilized, reflecting a diverse landscape of LLM usage. Notably, both open-source models (e.g., CodeBERT, CodeT5, CodeLlama) and proprietary models (e.g., GPT-3.5 Turbo, GPT-4, Bard) are well represented, suggesting that researchers are leveraging a range of resources depending on task requirements and accessibility. The presence of models like CodeGen and Bard, albeit less frequently, highlights ongoing experimentation with both established and emerging LLMs in the field. Overall, this distribution underscores the central role of code-oriented and general-purpose LLMs in current research practices. Appendix Figure 5 displays the top 15 most-used LLMs among the analyzed papers.

The most frequently targeted languages are C/C++ (24 papers), Java (12 papers), Python (9 papers), and C (8 papers), with fewer papers focusing on JavaScript (4), Solidity (3), Verilog (1), Dafny (1), and Alloy (1). This distribution reflects a strong emphasis on languages that are widely used in software development and are known for their relevance to software security and vulnerability research. In particular, C and C++ are historically associated with critical security issues such as buffer overflows and memory management errors, making them a central focus in vulnerability detection studies. Java and Python, as popular high-level languages, also present unique security challenges. The inclusion of languages like Solidity highlights growing interest in the security of smart contracts and blockchain applications. Overall, the selection of target languages aligns with the broader landscape of software security research, prioritizing those environments where vulnerabilities are both prevalent and impactful. Appendix Figure 6 summarizes the distribution of target programming languages across the selected papers.

In Appendix Table 6, a comprehensive list of the analyzed articles is presented, along with a description of the downstream task addressed in each paper. Each task is conceptually decomposed into its inputs, the approach adopted, and the resulting outputs. To highlight the nature of each task, a color-coding scheme is used: tasks related to LLM4Verification are marked in blue, those related to Verification4LLM are shown in green, and tasks that involve both verifying LLM-generated code and using LLMs as code verification tools are indicated in cyan.

For classification purposes, the selected papers were grouped into two categories: those employing Transformer-based models (prior works, representing only 9 out of 50) and those utilizing LLMs (the others). The thematic areas identified include Vulnerability Detection (VulDet), Vulnerability Explanation

(VulExpl), Software Verification (SWVerif), Static Analysis (SA), Repair—Correctness (RepCorr), Repair—Verification (RepVerif), Code Generation (CodeGen), Smart Contracts (Smart), Frameworks (FW), Datasets (Dataset), and Benchmarks (Bench). For brevity and clarity, the concise topic labels shown in parentheses are used in Appendix Table 6 to present the classification results in a compact form. As an example, 17 papers focus only on LLMs for Vulnerability Detection, while 6 papers use Transformer approaches for Vulnerability Detection. 12 papers are focused on LLMs for Software Verification, 9 papers on LLMs for Repair—Verification, and, notably, 7 papers intertwine LLMs with Static Analysis.

Of the 50 analyzed papers, **41** are focused on LLM4Verification, **6** on Verification4LLM, and **3** on both.

3 Related work

As related work, we examined surveys, reviews, and systematic reviews (Systematic Literature Review - SLR) previously published in the literature, summarizing their key contributions and findings.

LLMs have demonstrated significant potential in transforming software security practices by reducing manual effort while improving the effectiveness and accuracy of security applications, as demonstrated in Zhu et al. (2025) (survey study), and Zubair et al. (2025) (SLR). These studies reveal that LLMs excel particularly in generating short, localized bug fixes and test assertions, with performance enhanced through sophisticated prompt engineering strategies. Differently from our work, these investigations focus solely on the techniques of fuzzing, unit test, program repair, bug reproduction, data-driven bug detection, and bug triage, using LLMs as companions. In our dual perspective, we recognize that such tasks could be accomplished using both LLMs as verifiers, and LLMs as code-generator to be verified, and discussed them accordingly. Similarly, Huang et al. (2025) (review) emphasizes the critical role of deep learning in Automated Program Repair (APR), particularly highlighting how learning-based techniques outperform traditional search-based and constraint-based approaches. The perspective guiding (Huang et al., 2025) is organized around four patch generation schemes: search-based, constraint-based, template-based, and learning-based. In contrast, our work is less concerned with these specific categories and instead examines the broader landscape of LLM-based approaches. Zhang et al. (2024) (SLR) provides a comprehensive systematic literature review revealing a growing trend in LLM-based APR and emphasizes that prompt input represents the predominant application form for LLM-based program repair, highlighting the critical importance of effective prompt design for leveraging natural language processing capabilities. This comprehensive SLR concentrates on three specific strategies for deploying LLMs in APR, a focus that is considerably narrower compared to the scope of our study.

In the context of code security, studies like Basic and Giaretta (2024) (SLR) and Dolcetti et al. (2024) (experimental study) reveal a critical paradox where LLMs can both introduce, detect and repair security vulnerabilities, with common vulnerabilities including injection flaws, memory management issues, file management problems, and sensitive data exposure stemming from unsafe coding practices and potentially compromised training data. While LLMs demonstrate potential in vulnerability detection and repair, with performance varying based on model size and training data quality, they frequently produce high false positive rates, particularly in complex codebases, and struggle with correcting complex vulnerabilities requiring deep contextual understanding. Basic and Giaretta (2024) emphasizes that prompting techniques significantly influence LLM effectiveness in vulnerability detection and correction, with Chain-of-Thought (CoT) prompting showing particular promise, while data poisoning poses substantial risks to LLMs' ability to generate secure code and accurately identify vulnerabilities. Likewise, Negri-Ribalta et al. (2024) (SLR) reveals critical security concerns regarding AI-generated code, demonstrating that such code is not inherently secure and contains documented security vulnerabilities, with particular attention to MITRE CWE Top-25 vulnerabilities including CWE-787 and CWE-89, while security performance varies significantly across programming languages with Python appearing more secure than C-family languages and Verilog experiencing similar issues to C. These works concentrate exclusively on the types of vulnerabilities that LLMs may introduce when generating code, as well as on their ability to detect and repair vulnerabilities in existing code, and on how the chosen prompting strategy influences performance in these two tasks. In our view, this focus pertains solely to the Verification4LLM perspective.

Zhou et al. (2024a) (SLR) reveals that, in the context of vulnerability detection, fine-tuning represents the dominant adaptation technique at approximately 73%, followed by prompt engineering at 17% and retrieval augmentation generation at 10%, with datasets primarily focusing on function or line-level analysis while showing limited exploration at class or repository levels. This work focuses exclusively on surveying studies related to vulnerability detection, and partially on vulnerability repair; within our framework, this falls solely under the LLM4Verification perspective.

Wang and Chen (2023) (review) identifies a significant research gap between LLM application for code generation and the evaluation of generated code quality, with evaluation receiving considerably less attention despite LLMs' capability to handle various software engineering tasks and enhance developer productivity through applications like GitHub Copilot. The study reveals that while LLMs excel in bug correction and can outperform traditional APR tools, the generated code presents security concerns as vulnerabilities and malicious code from training data can leak into LLM outputs, with current evaluation focusing primarily on functional correctness and security while neglecting other critical quality aspects such as compatibility, maintainability, and portability. While this study presents a comprehensive view of LLMs in the context of code generation, it offers comparatively little on their use as verifiers. Our work complements this by analyzing the verification perspective as well.

In summary, while LLMs have shown substantial promise in software security and automated program repair, persistent challenges include limited understanding of complex code, insufficient generalization, data quality issues, and high computational costs (Zhu et al., 2025; Zhang et al., 2024; Wang and Chen, 2023). Security concerns remain due to LLMs' tendency to introduce vulnerabilities and produce high false positive rates, especially in complex or real-world scenarios (Basic and Giaretta, 2024; Negri-Ribalta et al., 2024). Evaluation frameworks

often neglect important code quality aspects beyond functional correctness, such as maintainability and portability (Wang and Chen, 2023). Future directions emphasize developing high-quality, leakage-free benchmarks, hybrid approaches combining LLMs with traditional tools, and comprehensive evaluation methods. Addressing these challenges is crucial for realizing the full potential of LLMs in secure, reliable software engineering.

Compared to our work, existing analyses in the field tend to concentrate on narrow dimensions of the broader challenges at hand, such as methods for repairing faulty programs, addressing security flaws in automatically generated code, or detecting vulnerabilities in software. Each of these directions examines a distinct capability of LLMs, effectively placing them along a single line of assessment. In contrast, more recent experimental efforts increasingly aim to tackle a wide range of tasks within a single framework. A closer examination of this body of work indicates that it can be meaningfully structured by distinguishing between two fundamentally different perspectives: LLM4Verification and Verification4LLM. Our study seeks to bridge this gap.

4 LLM for verification

LLM4Verification addresses RQ1, RQ2, RQ4, and RQ7. Figure 1 illustrates the roadmap of this section, which is described in the following. To answer RQ1, we surveyed approaches that leverage LLMs for various verification tasks, with vulnerability detection emerging as the most prominent. Both standalone LLMs and custom frameworks were considered. In response to RQ2, we identified five key strategies: prompt engineering, fine-tuning, integration with tools and feedback loops, retrieval-augmented generation (RAG), and agentic approaches. For each strategy, we assessed both effectiveness and limitations. RQ4 narrows the focus to tool integration, which is examined in greater detail. Finally, RQ7 explores the applicability of LLMs in program repair, with particular attention to custom frameworks designed for this purpose.

4.1 RQ1: Effectiveness of LLMs in vulnerability detection compared to traditional approaches and custom LLM-based methods

Of the 50 analyzed papers, **30** show insights on this research question, i.e., **60%** of the database.

According to Thapa et al. (2022), LLMs generally outperformed traditional RNN-based models (e.g., BiLSTM) in vulnerability detection, showing significant improvements in F1-score, False Positive Rate (FPR), and False Negative Rate (FNR). Nevertheless, significant limitations and inconsistencies remain, particularly in realistic or complex scenarios. Using the DiverseVul dataset (Chen et al., 2023), LLMs significantly outperformed Graph Neural Networks (GNNs) (47.15% vs. 29.76% F1-score); however, overall F1-scores remained low, suggesting the technology is not yet viable for large-scale deployment because of high FPR. Liu et al. (2024a) observed LLMs perform well in simple binary classification tasks (e.g., vulnerability existence, >85% accuracy), but their performance significantly drops (accuracy below 30%) in more complex tasks like localizing root causes or trigger points.

Ullah et al. (2024) concluded that LLMs are currently unreliable for vulnerability detection, showing high FPR, non-deterministic outputs, incorrect reasoning even when vulnerabilities are correctly identified, and fragility to simple code augmentations. Some sources (Yin et al., 2024; Wen X.-C. et al., 2024; Yang et al., 2024) imply that improvements often depend heavily on fine-tuning on domain-specific data, and without it, general-purpose LLMs struggle with the intricate semantics required for vulnerability detection. Moreover, potential data leakage in training datasets can lead to artificially inflated performance. Yin et al. (2024) reported that although fine-tuned LLMs show improvements, they remain inferior to transformer-based methods, with baseline techniques often outperforming LLMs, particularly in few-shot scenarios. Another paper, by Ding et al. (2024), directly stated that LLMs, including state-of-the-art models like GPT-4, are largely ineffective and "useless in practice" in realistic vulnerability detection settings.

OpenAI models are worth a separate discussion. According to Liu P. et al. (2024), ChatGPT demonstrated capabilities comparable or superior to some state-of-the-art methods in security bug report identification, but its F1-score remained low compared to other top-performing approaches. For vulnerability severity evaluation, its performance was slightly inferior to some baselines. Studies like Gong et al. (2024) indicated that GPT-3.5 had limited practical value for vulnerability detection (43.6% accuracy, comparable to random guessing), and while GPT-4 showed superior code understanding (74.6% accuracy), it still exhibited unacceptably high FPR compared to static analyzers like Bandit and CodeQL. For smart contracts, Chen et al. (2025), ChatGPT showed high recall but low precision (e.g., 88.2% recall but 22.6% precision for GPT-4), leading to many false positives. It performed better for specific vulnerability types like Denial of Service and Front Running, but underperformed for most others compared to traditional tools.

Generally, LLM-based approaches show promising, often superior, performance in vulnerability detection *when adequately trained or integrated with other tools*. We conclude this section with an analysis of the performance of the examined tools and frameworks, with numerical details when available:

- AIBugHunter (Fu et al., 2024b) demonstrated greater accuracy than traditional program analysis tools like Cppcheck for line-level vulnerability prediction. Its Multi-Objective Optimization (MOO) approach achieved 65% multiclass accuracy for vulnerability identification, surpassing baselines like BERT-base and CodeBERT by 10%–141%. However, it struggled with underrepresented vulnerabilities and severity identification, and accuracy as a metric was noted as problematic due to dataset imbalance.
- DLAP (Yang et al., 2025) consistently outperformed traditional deep learning models and other LLM-based prompting frameworks, showing improvements like a 10% higher F1-score and 20% higher Matthews Correlation Coefficient (MCC) over baselines. The overall F1-score for DLAP was noted as still low, indicating it might not be suitable for real-world production systems despite improvements over baselines.

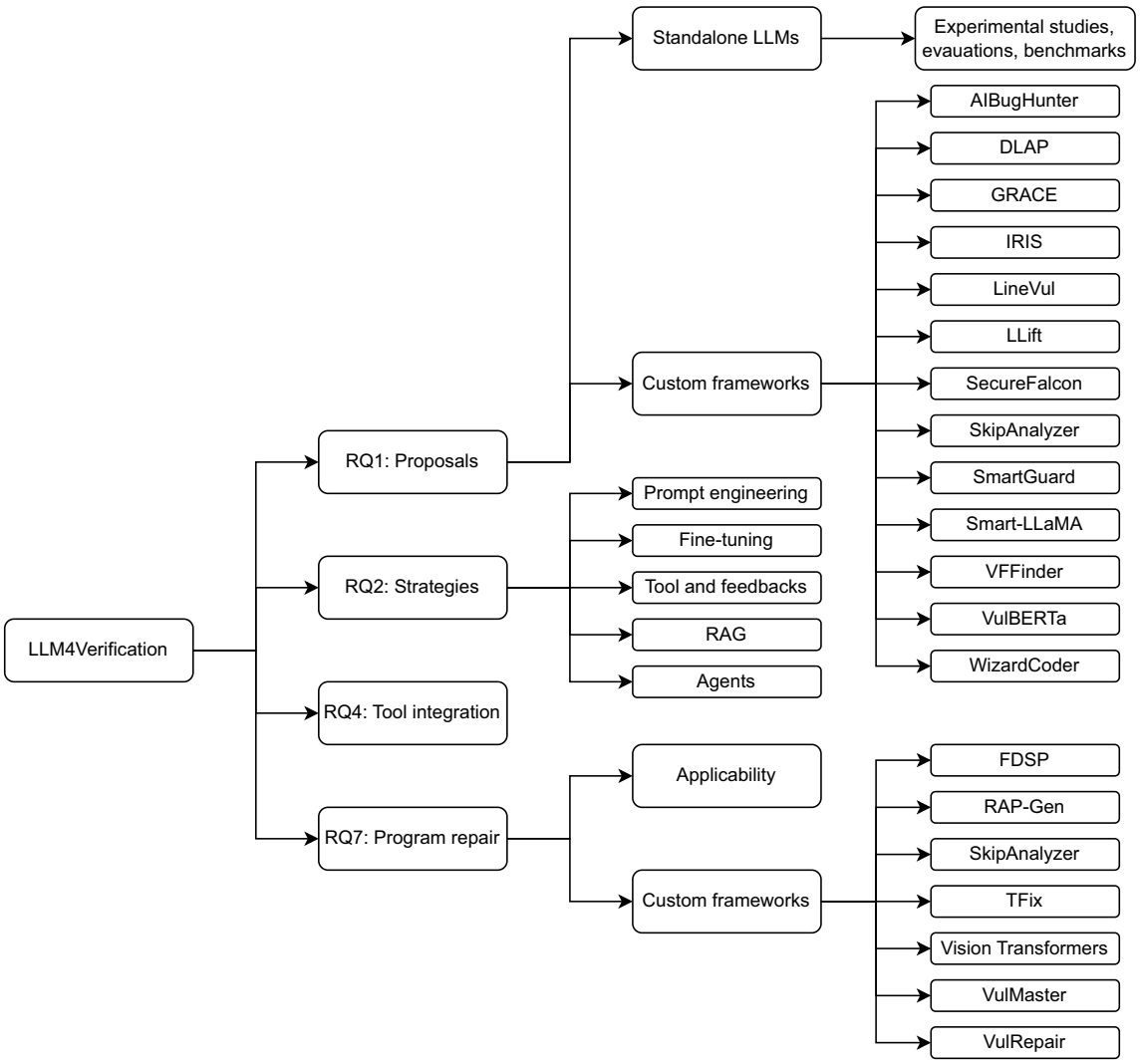

FIGURE 1
Roadmap of Section 4, including the list of names of LLM-based custom frameworks for verification, both for vulnerability detection (RQ1) and program repair (RQ2).

- GRACE (Lu et al., 2024) was found to be more effective than six state-of-the-art graph-based and sequence-based methods across various datasets.
- IRIS (Li et al., 2025) was significantly more effective than traditional static analysis tools alone, detecting 103.7% more vulnerabilities and achieving a lower false discovery rate compared to CodeQL.
- LineVul (Fu and Tantithamthavorn, 2022) achieved an F-measure of 91% for function-level vulnerability prediction, a 160%-379% improvement over state-of-the-art methods, and outperformed baselines in precision (97%) and recall (86%).
- LLift (Li et al., 2023) framework propose to use ChatGPT alongside with UBITech to produce bug reports. They observed GPT-4 significantly outperformed GPT-3.5 in terms of soundness (94% vs. 61%) and completeness (94% vs. 44%). It also successfully identified 16 out of 20 false positives and one missed bug that UBITect failed to detect due to symbolic execution limitations.
- SecureFalcon (Ferrag et al., 2025) achieved 94% binary and 92% multiclass accuracy, outperforming traditional ML algorithms by up to 11% and other LLMs by 4%. It also showed higher detection rates than static/formal tools, could handle non-compilable code, and was faster than Bounded Model Checking (BMC).
- SkipAnalyzer (Mohajer et al., 2023) significantly improved precision over the state-of-the-art static bug detector Infer, with increases of 12.86% for Null Dereference bugs and 43.13% for Resource Leak bugs.
- SmartGuard (Ding et al., 2025) significantly outperformed traditional static analysis tools like Slither (by 41.16% recall) and previous LLM-based approaches, achieving 95.06% recall and 94.95% F1-score on a benchmark dataset, and uniquely detecting multiple vulnerabilities simultaneously.
- Smart-LLaMA (Yu et al., 2024) consistently outperformed 16 state-of-the-art baselines, including traditional rule-based tools, neural network-based techniques, and other LLM-based

approaches, achieving average improvements of 6.49% in F1-score and 3.78% in accuracy.
- VFFinder (Wu et al., 2024) showed promising performance (Top-1 accuracy of 27.27% on vulnerable function identification, which is 2.37 times higher than the best baseline methods), but overall results were still generally low.
- VulBERTa (Hanif and Maffeis, 2022) proved highly effective, consistently outperforming traditional deep learning baselines and existing vulnerability detection approaches, often achieving state-of-the-art performance with high precision and F1-scores.
- Finetuned WizardCoder (Shestov et al., 2025) demonstrated superior effectiveness over CodeBERT-based models in vulnerability detection, on both balanced and imbalanced datasets.

4.1.1 Discussion

The use of LLMs for vulnerability detection is a rapidly emerging **trend**, though the scientific literature presents **conflicting perspectives** about their effectiveness. Experimental evaluations such as those in Thapa et al. (2022), Chen et al. (2023), Liu P. et al. (2024), and Liu et al. (2024a) highlight promising results when comparing LLMs to traditional ML and DL approaches. In contrast, studies like Ullah et al. (2024), Yin et al. (2024), Ding et al. (2024), Gong et al. (2024), and Chen et al. (2025) emphasize the limitations of LLMs, finding them unreliable and ineffective when used out-of-the-box for vulnerability detection. Despite these discrepancies, there has been a proliferation of custom frameworks that integrate static analysis tools, specialized training strategies, and tailored model architectures into novel pipelines that claim superior performance. However, meaningful comparison across these approaches remains difficult due to the heterogeneity of datasets, benchmarks, and evaluation methodologies: this marks a **gap** in current literature. A promising **future direction** lies in the development of unified and comprehensive frameworks that go beyond the standalone use of LLMs.

4.2 RQ2: Strategies, effectiveness, and limitations of LLMs in code verification tasks

Of the 50 analyzed papers, **42** show insights on this research question, i.e., **84%** of the database.

4.2.1 Fine-tuning

The first notable strategy is fine-tuning. This is a pervasive strategy, often involving pre-trained models on domain-specific data. A specific approach is the one used in Fu et al. (2024b), Multi-Objective Optimization (MOO), with customized loss functions. In work like Liu et al. (2024a) and Li D. et al. (2024), Parameter-Efficient Fine-Tuning (PEFT) techniques like LoRA (Low-Rank Adaptation) are leveraged for specific tasks. Yang et al. (2024) proposed Multitask Self-Instructed Fine-Tuning (MSIVD), which involves integration with GNNs and the use of PEFT. For smart contracts, Yu et al. (2024) proposed Smart Contract-Specific Continual Pre-Training (SCPT) and Explanation-Guided Fine-Tuning (EGFT). Another technique is to perform fine-tuning across diverse error types simultaneously, pre-training the model on bug-fix data (Fu et al., 2024a).

4.2.1.1 Effectiveness

Fine-tuned LLMs generally show great potential for vulnerability repair and strong generalization, outperforming various baselines (Wu et al., 2023; Zhang et al., 2023; Shestov et al., 2025; Huang et al., 2023). They can achieve high accuracy and instant inference times (SecureFalcon, Ferrag et al., 2025).

4.2.1.2 Limitations

Fu et al. (2024b) noted that their tool (AIBugHunter), being in an early stage, it may not generalize well to languages other than C/C++ (the one was tuned on). Another tool, PDBERT (Liu et al., 2024d) suffers from the same problem, struggling to generalize outside the C/C++ language on which it was trained on. This is further confirmed by Huang et al. (2023) and Yang et al. (2025), which observed that fine-tuning a pre-trained model could lead to *catastrophic forgetting* of the previous knowledge and linguistic capabilities of the LLM, even producing repetitive outputs (Yu et al., 2024). Moreover, Thapa et al. (2022) remarked that the high computational costs of fine-tuning make it impractical for the majority of tasks, while Fu et al. (2023) noticed that knowledge distillation depends on the quality of teacher models and may struggle with noisy or incorrect labels. On DiverseVul, by Chen et al. (2023), fine-tuning could lead to poor generalization performance of the LLMs on unseen projects. Contrary to many other studies, Yang et al. (2024) assessed the performance of fine-tuning as only an incremental improvement versus LLMs-based approaches such as LineVul (Fu and Tantithamthavorn, 2022). Berabi et al. (2021) and Hanif and Maffeis (2022) also observed that a fine-tuned model's effectiveness is often tied to the quality of datasets, which usually require extensive cleaning and filtering to remove noise.

4.2.2 Prompt engineering

The second notable strategy is prompt engineering. This involves crafting prompts to elicit the desired responses from LLMs, enhancing their performance on specific tasks. The most popular strategies to perform prompt engineering are listed in the following.

- **Chain-of-Thought (CoT)**: guiding models through intermediate reasoning steps. Papers that use this approach are Fu et al. (2022), Ding et al. (2024), Li et al. (2023), Wen et al. (2024a), Lu et al. (2024), Wang et al. (2024), Ullah et al. (2024), Misu et al. (2024), Yang et al. (2025), Li H. et al. (2024), and Ding et al. (2025).
- **Few-shot** and **Zero-shot** prompting: providing a few examples or no examples in the prompt to demonstrate the desired output format or task. Papers that use this approach are Yin et al. (2024), Liu et al. (2024a), Wen et al. (2024a), Liu P. et al. (2024), Ullah et al. (2024), Mohajer et al. (2023), and Li et al. (2025).

- **In-Context Learning (ICL)**: providing a few examples in the prompt to add context to the request(s). This method could be seen in DLAP (Yang et al., 2025), in VFFinder (Wu et al., 2024), in GRACE (Lu et al., 2024), and in SmartGuard (Ding et al., 2025) frameworks. It slightly differs from few-shot prompting in the scope and type of examples.
- **Task decomposition**: breaking down complex problems into smaller, more manageable sub-problems for the LLM, as in Li et al. (2023), Wang et al. (2024), and Li H. et al. (2024).
- **Progressive prompting**: allowing LLMs to request additional information (e.g., function definitions) as needed, as in Li et al. (2023) and Li H. et al. (2024).
- **Role-oriented**: assigning the LLM a specific persona, e.g., "security expert" as in Ullah et al. (2024).
- **Contextual information**: including comments or external feedback within the prompt. Papers that use this approach are Lu et al. (2024), Pearce et al. (2023), Liu P. et al. (2024), and Li et al. (2025).

4.2.2.1 Effectiveness

Prompt engineering strategies (CoT, few-shot, task decomposition, progressive prompts) substantially improve LLM performance. Notably, GPT-4, with these strategies, achieved 94% soundness and completeness for UBI bug analysis, a significant improvement over GPT-3.5 (Li H. et al., 2024). ICL with CoT can lead to significant performance leaps. In particular, the GRACE framework (Lu et al., 2024) first demonstrated that graph structure integration, such as Abstract Syntax Trees (ASTs), Program Dependencies Graphs (PDGs), and Control Flow Graphs (CFGs), significantly improves vulnerability detection.

4.2.2.2 Limitations

Li et al. (2023) noted inconsistency in CoT prompt engineering: ChatGPT showed inconsistency between the reasoning steps obtained and the final verdict, occasionally mis-categorizing a variable despite intermediate steps suggesting otherwise. This is a known limitation of CoT prompting (Ullah et al., 2024): Multi-round conversations sometimes show limited enhancement or even decline in detection performance (Chen et al., 2025). Few-shot prompting can improve performance for some tasks (e.g., bug report summarization), but may negatively impact others (e.g., localization) due to diversity or misleading examples (Liu et al., 2024a), and may introduce randomness that hinders reproducibility (Yin et al., 2024). Moreover, Ding et al. (2025) noted that performance degrades with too many input examples (>8). Few-shot prompting can be labor-intensive and expensive due to engineering efforts for tuning and post-processing (Wang et al., 2023). According to Wu et al. (2024), ICL depends on the relevance of the examples in the case pool: at the same time, creating the case pool is time-consuming and requires a substantial manual effort. Progressive prompts may use too many tokens (Li H. et al., 2024). Complex prompts do not consistently enhance ChatGPT's performance and can even confuse it (Liu P. et al., 2024). However, they also noted that prompts effectiveness could vary from the used model, making the results hardly generalizable. Overall, the majority of the above cited studies agree in recognizing that long sequences and large functions could be challenging due to token limitations and possibly small context windows of available models, especially when ICL is used or other context information are provided (Lu et al., 2024; Li et al., 2025). Context and generalization challenges persist, as extracted code snippets may lack sufficient context. Generalizability to different programming languages, complex side-effects, or new vulnerability types is still limited, and effectiveness relies heavily on prompt quality (Mohajer et al., 2023).

4.2.3 Tool integration and external feedback

Combining LLMs with traditional Static Analysis (SA) tools, formal verification tools, or other external analyzers is another notable strategy. This includes: feeding SA reports to LLMs for analysis or false positive reduction, e.g., using ESLint (Berabi et al., 2021), Bandit (Alrashedy et al., 2023), CodeQL (Pearce et al., 2023), and other tools (Li et al., 2023; Mohajer et al., 2023; Wen et al., 2024a; Li H. et al., 2024; Ge et al., 2024); LLMs generating inputs (e.g., function summaries, taint specifications) for SA tools (Li et al., 2023, 2025); using formal verifiers, e.g., Alloy Analyzer (Alhanahnah et al., 2024), Frama-C (Wen et al., 2024b), ESBMC (Ferrag et al., 2025), to validate LLM-generated specifications or code; integration with SMT solvers for path feasibility validation (Wang et al., 2024).

4.2.3.1 Effectiveness

External feedback from static analyzers (e.g., Bandit, CodeQL, ESLint, Infer) significantly improves LLM capabilities in vulnerability refinement and bug detection, leading to improved precision, recall, and accuracy. It can reduce false positives and address cases where static analysis times out (Li et al., 2023; Li H. et al., 2024). Integration with formal verification tools (e.g., Alloy Analyzer, Frama-C, ESBMC) ensures validation and correctness of generated specifications and code.

4.2.3.2 Limitations

In Wen et al. (2024b), the integration with SA and theorem prover helps validate the specifications, but there is a need for iterative validation to avoid error accumulation. Li H. et al. (2024) also noted that the analysis could have a limited scope. Differently from traditional tools, LLMs are black boxes, so there is uncertainty even when performance are high, and no explanation for their decisions (Pearce et al., 2023). These approaches may suffer from long prompts problems (see previous subsection), leading to challenges in handling large functions or complex path conditions (Wang et al., 2024, 2023; Mohajer et al., 2023). Using a specialized SA tool, such as ESLint for JavaScript in Berabi et al. (2021), could lead to poor generalization of the proposed method, limiting the applicability to other languages or error types. Finally, computational cost is a significant concern due to frequent LLM queries in iterative processes, leading to high financial and resource expenditures, e.g., integrating formal verification (BMC) in such processes could be slow or resource-intensive (Ferrag et al., 2025).

4.2.4 Retrieval augmented generation

More and more research papers are exploring the method of Retrieval-Augmented Generation (RAG). This involves leveraging external knowledge or semantically similar examples retrieved from

a codebase to augment LLM input. This strategy is used in the following analyzed papers: Shestov et al. (2025), Yang et al. (2025), Lu et al. (2024), Wang et al. (2023), Misu et al. (2024), and Chen et al. (2025).

4.2.4.1 Effectiveness

RAG generally enhances performance by leveraging relevant examples or external knowledge. The particular RAG utilized in DLAP (Yang et al., 2025) is reported to be the reason of it outperforming CoT, role-based prompting, and GRACE techniques. Also in GRACE (Lu et al., 2024) is observed the crucial role of RAG, by means of an ablation study that demonstrated its significant contribution. Even when fine-tuning is the main technique, in Shestov et al. (2025), RAG demonstrated significant improvement in precision, recall and F1-score.

4.2.4.2 Limitations

According to Yang et al. (2025), RAG is an effective technique, but it heavily depends on training data. Moreover, as a result of RAG could be generated overloading prompts, which confuses the models (Chen et al., 2025).

4.2.5 Agentic frameworks

Finally, agentic frameworks are emerging as a strategy to enhance LLM capabilities. These frameworks involve implementing multi-agent systems where LLMs interact to refine outputs or perform tasks. For example, in Alhanahnah et al. (2024), a dual-agent LLM framework involving a Repair Agent and a Prompt Agent has been explored. In Ding et al. (2024), an integration with traditional tools and agents is discussed, but not implemented. As agents and multi-agent systems approaches continue to evolve, experiencing a new flowering and increasing growth in commercial frameworks as well (e.g., Langchain[1], Autogen[2], MetaGPT[3] by Hong et al., 2024, and so on), we expect this strategy to become one of the leading methodologies for AI-based program verification frameworks in the next years.

4.2.5.1 Effectiveness

According to Alhanahnah et al. (2024), auto-feedback generated by LLMs is more effective than human-created generic prompts for automatic program repair.

4.2.5.2 Limitations

In Alhanahnah et al. (2024), the use of a multi-agent framework may result in a marginal increase in token consumption, since auto-feedback usually require a greater number of iterations.

4.2.6 Discussion

Among the various strategies employed in code verification tasks using LLMs, prompt engineering remains by far the **most widely adopted**. Its popularity continues to grow, largely due to its low cost and immediate applicability, often yielding surprisingly strong results with minimal overhead. However, the scientific literature presents **mixed views** on its effectiveness, highlighting issues such as inconsistent behavior, randomness in outputs, and limitations when handling long or complex input sequences. An alternative and more robust approach involves fine-tuning LLMs to transfer learned patterns into a specific target domain. While this method can significantly improve performance, it is computationally expensive and carries risks such as catastrophic forgetting. A notable **gap** in current research is the limited exploration of agentic frameworks, which are beginning to gain traction in commercial applications but remain under-analyzed in the context of security and code verification. **Future research directions** should prioritize the development and evaluation of agentic approaches, as they hold potential for more autonomous and adaptive verification pipelines.

4.3 RQ4: Strategies for tool integration

Of the 50 analyzed papers, **19** show insight into this research question, i.e., **38%** of the database.

Integration strategies between LLMs and security tools are diverse and sophisticated. A core method involves feedback loops, i.e. intertwining static/dynamic analyzer with LLMs to check the safety and security of analyzed code (as in Yang et al., 2025), and iterative refinements, i.e. asking the LLMs to fix, patch, or repair the detected insecure code in one or more steps (as in Alrashedy et al., 2023).

LLM-generated context and inputs for tools are another key strategy. LLMs can generate precise function summaries (Li et al., 2023) or taint specifications (Li et al., 2025) for SA tools, enhancing their accuracy by reducing false positives and negatives. Conversely, LLMs can act as post-processors for tool outputs. Static analyzers often produce numerous warnings. LLMs can process and classify these reports, helping to filter out false positives and identify genuine bugs, thus streamlining the manual triage process (Wen et al., 2024a; Wu et al., 2024).

Automated validation of fixes (e.g., by ESLint in Pearce et al., 2023 and Wang et al., 2023) guarantees error removal without introducing new ones (Berabi et al., 2021). These integrations are crucial in overcoming tools and LLMs' limitations. LLMs can handle cases where SA tools time out (e.g., symbolic execution issues, as in Li et al., 2023 and Li H. et al., 2024), while contextual code snippet extraction mitigates LLM token limitations (Wen et al., 2024a).

In more advanced setups, direct tool invocation by LLM agents allows LLM-based agents to directly call external validation tools (e.g., Alloy Analyzer, as in Alhanahnah et al., 2024, SMT solver as in Wang et al., 2024) to check the correctness of proposed specifications or dataflow paths in real-time.

Finally, hybrid frameworks and data integration combine LLMs with various security tools and data sources. Program dependency analysis tools (e.g., Joern, as in Liu et al., 2024d) can extract structural data [ASTs, PDGs, as in the GRACE framework by Lu et al. (2024)] to train LLMs or validate their analysis. GNNs modeling CFGs can provide embeddings that, when combined with LLM outputs, improve vulnerability classification (Yang et al., 2024). LLMs can also integrate with Software

1 https://github.com/langchain-ai/langchain

2 https://microsoft.github.io/autogen/stable//index.html

3 https://github.com/FoundationAgents/MetaGPT

Composition Analysis (SCA) tools to reduce false positives in vulnerability detection (Wu et al., 2024; Ge et al., 2024), or combine with BMC and traditional static analyzers for comprehensive vulnerability detection (Ferrag et al., 2025). For example, the framework LLMDFA (Wang et al., 2024) leverages LLMs' semantic understanding and formal tools' logical reasoning for a robust hybrid approach.

The integration significantly enhances security capabilities, with studies showing vulnerability refinement improvements of up to 30% with GPT-4 using Bandit feedback (Alrashedy et al., 2023). Precision of static analyzers is also enhanced by filtering false positives (e.g., SkipAnalyzer, by Mohajer et al., 2023 improved Infer's precision by 12.86% to 43.13%). Furthermore, integrated approaches increase the number of detected vulnerabilities and reduce false discovery rates (e.g., IRIS by Li et al., 2025 with GPT-4 detected 55 vulnerabilities vs. 27 by CodeQL). Enhanced code/specification quality is another benefit, as integration with formal verification tools ensures generated specifications are satisfiable and adequate (Wen et al., 2024b).

4.3.1 Discussion

Integrating LLMs with static analysis tools is a strategy **increasingly adopted** by researchers, with the goal of correcting, specializing, and enhancing both the outputs of LLMs and the feedback provided by static analyzers. This synergy has shown generally promising results, suggesting that the combination can help mitigate the limitations inherent in each approach when used independently. However, these results remain difficult to compare meaningfully due to the lack of standardized benchmarks and ground truth datasets, which creates a **gap** in the field. To move forward, a **promising direction** lies in the development of robust and reliable frameworks that systematically combine LLMs and static analysis, enabling consistent evaluation and maximizing the strengths of both methodologies.

4.4 RQ7: Effectiveness and limitations of LLMs in repairing vulnerabilities

Of the 50 analyzed papers, **17** show insight into this research question, i.e., **34%** of the database.

LLMs exhibit promising capabilities in the realm of vulnerability repair, marking a significant advancement in Automated Program Repair (APR). Certain LLM-based approaches, such as FDSP (Alrashedy et al., 2023), VulMaster (Zhou et al., 2024b), RAP-Gen (Wang et al., 2023), SkipAnalyzer (Mohajer et al., 2023), TFix (Berabi et al., 2021), Vision Transformers based on vulnerability queries and masks (Fu et al., 2024a), and VulRepair (Fu et al., 2022, 2024b), have demonstrated a notable outperformance of traditional or baseline learning-based APR methods. GPT-4 with expertise prompts has achieved an impressive 92.8% valid repair rate, showcasing its potential in real-world scenarios (Liu P. et al., 2024). Moreover, fine-tuning generally enhances LLMs' vulnerability-fixing capabilities, allowing them to specialize in security-related code modifications (Zhang et al., 2023; Wu et al., 2023). Among pre-trained models, VulRepair (Fu et al., 2022) stands out by achieving a perfect prediction accuracy of 44%, which is 13%-21% higher than two baseline approaches (VRepair and CodeBERT). This is further remarked by Zhang et al. (2023), that reported prediction accuracy ranging from 32.94% to 44.96% in APR using pre-trained models, consistently outperforming existing techniques.

As described in the previous section, iterative feedback from static analyzers substantially improves LLMs' vulnerability refinement capabilities (Gong et al., 2024; Alrashedy et al., 2023). This closed-loop approach allows LLMs to learn from detected errors and iteratively correct their generated patches. Concrete examples underscore the high performance achieved by LLM-based repair techniques:

- According to Gong et al. (2024), GPT-4 has demonstrated promising abilities in self-correction and cross-model repair, achieving an 85.5% success rate when repairing its own generated vulnerable code and a 77.4% success rate when repairing code generated by other LLMs, particularly when provided with Common Weakness Enumeration (CWE) descriptions to guide the repair process.
- Using ChatGPT, multi-round fixing processes have demonstrated high success rates, capable of successfully addressing over 89% of vulnerabilities (Liu et al., 2024c). However, they also note that ChatGPT's ability to directly fix erroneous code is relatively weak without iterative interaction.
- SkipAnalyzer has demonstrated high logic correctness (up to 97.30%) and syntax correctness (up to 99.55%) for specific bug types, indicating its precision in targeted repairs (Mohajer et al., 2023).
- Alhanahnah et al. (2024) used GPT 4 with auto-feedback to repair Alloy specifications, surpassing traditional state-of-the-art repair tools.

4.4.1 Discussion

Despite their promising capabilities, LLM-based program repair techniques are subject to several significant **limitations** that hinder their widespread adoption and reliability. A primary concern is that LLMs are not yet sufficiently reliable for APR in zero-shot scenarios (Pearce et al., 2023). Generated fixes can often be implausible, unreasonable, or, critically, introduce new vulnerabilities or bugs, potentially failing to preserve the original functional correctness of the code (Alrashedy et al., 2023; Pearce et al., 2023). LLMs may also struggle to effectively incorporate valid feedback, especially when dealing with highly complex vulnerabilities that require deep contextual understanding (Alrashedy et al., 2023). Furthermore, some generated patches can be uncompileable or lack necessary contextual information, rendering them impractical (Wu et al., 2023).

In particular, Alrashedy et al. (2023) noted that LLMs face considerable difficulty in handling complex vulnerabilities, such as SQL injection or OS command injection, which require intricate semantic understanding and context-aware modifications. They also struggle with certain specific CWE types (e.g., CWE-325,

CWE-444, CWE-119, and other rare CWE types) and multi-hunk code changes (Wu et al., 2023; Zhang et al., 2023; Fu et al., 2022).

Their applicability is often limited to a few dominant programming languages, with very few Java vulnerabilities reported as fixed by existing LLMs according to Wu et al. (2023). Token limitations (e.g., typically around 500 tokens) pose a challenge when dealing with long vulnerable functions or complex path conditions, as the model may not be able to process all relevant information (Zhou et al., 2024b; Zhang et al., 2023; Pearce et al., 2023; Fu et al., 2022; Huang et al., 2023; Wang et al., 2023). Consequently, the effectiveness of LLMs tends to decrease with longer or more complex fixes.

The performance of LLM-based repair techniques heavily relies on the availability of high-quality, balanced training data (Berabi et al., 2021; Fu et al., 2024a; Huang et al., 2023). Performance can significantly suffer from small sample sizes, class imbalance, or noisy data.

As for program verification, a **critical issue** is the potential for catastrophic forgetting of pre-trained knowledge during fine-tuning, which can degrade the model's general capabilities while specializing (Huang et al., 2023). Yang et al. (2025) propose a potential solution by using DLAP to guide the repair process, rather than directly fine-tuning LLMs, since fine-tuning may yield better binary classifiers, but it could come at the cost of linguistic capabilities that are valuable for effective code repair.

Moreover, frequent LLM queries within iterative repair processes can lead to high computational and financial costs, making continuous deployment challenging for some organizations (Wang et al., 2023; Li et al., 2025). When compared to traditional APR techniques, LLM-based approaches sometimes fall short. For instance, in Alhanahnah et al. (2024) for specification repair, GPT-3.5 Turbo has been noted to evidently underperform w.r.t. other Alloy repairers, which are not based on LLM, and only GPT-4 with auto-feedback proved to be effective. Berabi et al. (2021) further observed that traditional techniques, such as symbolic analysis or test-suite-based repair, often offer stronger correctness guarantees, primarily because they are founded on rigorous logical reasoning or exhaustive test coverage. In contrast, LLMs often focus on syntactic fixes and may introduce subtle bugs due to their statistical nature and lack of formal guarantees.

In summary, LLMs represent a promising **future direction** for ML-based APR. However, their effective adoption depends on the development of comprehensive frameworks that integrate complementary techniques, such as static analysis, prompt engineering, and symbolic reasoning, to fully leverage their potential.

5 Verification of LLMs' generated code

RQ3, RQ5, and RQ6 pertain to the verification of code generated by LLMs. Figure 2 traces the roadmap of this section, as detailed in the following. In addressing RQ3, we identified four main strategies: prompt engineering, multi-round generation processes, integration with external tools and feedback loops, and model-based approaches including fine-tuning. Among these, the custom framework CoSec is discussed in detail. RQ5 narrows the focus to tool integration, examining its role in enhancing verification accuracy, and describing PromSec. Finally, RQ6 provides an overview of the types of vulnerabilities commonly found in LLM-generated code, as reported in the literature.

5.1 RQ3: Strategies, effectiveness, and limitations of LLMs' secure code generation

Of the 50 analyzed papers, only **8** show insight into this research question, i.e., **16%** of the database.

5.1.1 Prompt engineering

As for program verification using LLMs, also in LLMs' generated program verification (Verification4LLM), prompt engineering is a key strategy. Prompt engineering represents a fundamental approach to enhancing LLM security capabilities through sophisticated input design. Neutral zero-shot prompting was tested by Tihanyi et al. (2025) and Gong et al. (2024) to explicitly assess the baseline capabilities of LLMs in generating secure code. To improve this baseline, one key technique involves using property-specific continuous vectors, known as prefixes, to guide LLMs toward generating secure code, as demonstrated in the SVEN framework by He and Vechev (2023). This approach is complemented by the employment of optimized and structured prompts that incorporate detailed construction principles, significantly improving both the security and functionality of generated code (Liu et al., 2024c; Nazzal et al., 2024). The effectiveness of prompt engineering is further enhanced by leveraging variations of prompts and incorporating contextual information, few-shot examples, CoT reasoning for complex tasks such as generating formal method postconditions or verified methods, as in Misu et al. (2024). Moreover, iterative prompt optimization based on feedback from SA tools, as implemented in PromSec (Nazzal et al., 2024), creates a dynamic improvement cycle.

5.1.1.1 Effectiveness

Prompting's impact reveals the profound influence that sophisticated prompt design can have on security outcomes in LLM-generated code. Optimized prompts and property-specific vectors, as implemented in the SVEN framework (He and Vechev, 2023), significantly improve both the security and functionality of generated code, with secure code generation rates increasing dramatically from 59.1% to 92.3% in documented cases (He and Vechev, 2023). The effectiveness of advanced prompting techniques is further demonstrated through CoT prompts combined with RAG, which can achieve remarkably high success rates in specification synthesis, including 100% specification synthesis and 58% strong specification verification when using GPT-4 (Misu et al., 2024). In PromSec (Nazzal et al., 2024), optimized prompts ensure the quality and security of generated code, while combined with gGAN (graph Generative Adversarial Networks), observing that the gGAN itself without LLMs contribution does not achieve adequate results since it does not capture the semantics.

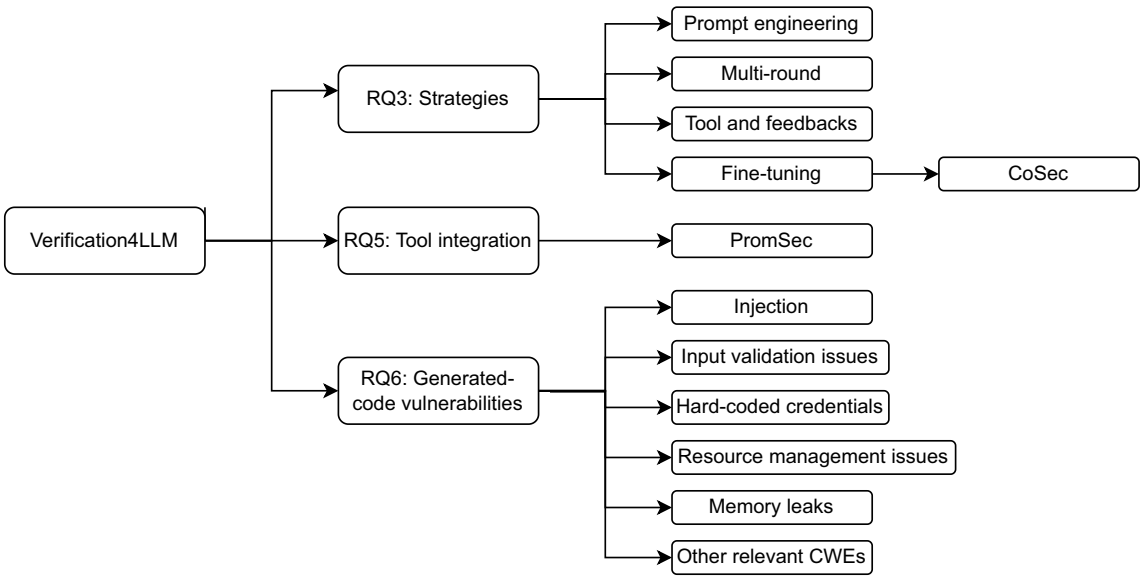

FIGURE 2
Roadmap of Section 5. Details about the findings for each RQ are provided. CoSec and PromSec are the names of custom LLM-based frameworks for secure code generation.

5.1.1.2 Limitations

Many studies reveal significant limitations in the ability of prompt engineering techniques to ensure secure code generation. Evidence demonstrates that prompt engineering alone is insufficient to guarantee security, regardless of sophistication in design (Liu et al., 2024c; Tihanyi et al., 2025). According to Tihanyi et al. (2025), all the examined LLMs employing neutral zero-shot prompting still produce vulnerabilities in generated code, at an unacceptable rate. Gong et al. (2024) further confirms this insight, testing four GPT models and observing that all performed poorly, with an average 76.2% of the generated code being insecure. Technical constraints further complicate prompt design, as token limits often restrict the amount of contextual information or examples that can be provided in prompts, while overloading prompts can confuse the model and degrade performance (Misu et al., 2024). Furthermore, designing effective prompts and examples requires significant manual effort, creating scalability challenges for widespread deployment (Misu et al., 2024). Furthermore, code security generated by LLMs is highly scenario-dependent, creating inconsistent reliability across different contexts (Gong et al., 2024). These challenges culminate in significant risks, including a false sense of security and the systematic replication of insecure coding patterns (Tihanyi et al., 2025; Nazzal et al., 2024).

5.1.2 Tool integration and feedback loops

Tool integration and feedback loops establish sophisticated mechanisms for validating and improving LLM-generated code through external verification systems. A primary strategy involves integrating LLMs with SA tools such as Bandit, CodeQL, and ESLint to provide external feedback, identify vulnerabilities, and validate generated fixes. This integration forms an iterative refinement loop where static analysis outputs can drive prompt optimization or directly inform subsequent generation steps (Liu et al., 2024c; Nazzal et al., 2024; Alrashedy et al., 2023; Pearce et al., 2023; Gong et al., 2024). Complementing this approach, researchers have demonstrated the value of combining LLM-generated code with formal verification tools, including ESBMC (Tihanyi et al., 2025) and Dafny (Misu et al., 2024), to systematically identify issues, ensure correctness, and validate generated specifications such as preconditions, post-conditions, and loop invariants. This methodology often incorporates a verification feedback loop to refine solutions iteratively.

5.1.2.1 Effectiveness

Research has shown that providing external feedback from static analyzers, such as Bandit, significantly improves vulnerability refinement capabilities, with improvements of up to 30% observed when using GPT-4 (Alrashedy et al., 2023). Notably, verbalizing this feedback can offer slight additional gains beyond the base improvement (Alrashedy et al., 2023). The integration with formal verification tools, such as ESBMC and Dafny, further enhances robustness by ensuring generated specifications are satisfiable and adequate, while helping refine solutions to ensure correctness. This integration approach leads to a substantial increase in successful program handling across various benchmarks (Tihanyi et al., 2025; Misu et al., 2024). Additionally, combining LLMs with vulnerability detection tools such as CodeQL has proven effective in mitigating security issues in generated code and improving overall security robustness (Liu et al., 2024c).

5.1.2.2 Limitations

LLMs sometimes fail to effectively incorporate valid feedback, particularly when dealing with complex vulnerabilities such as SQL injection and high-risk library calls (Alrashedy et al., 2023). Paradoxically, excessive iteration with SA tools may not prove

effective for repair tasks, suggesting diminishing returns from over-reliance on iterative approaches (Gong et al., 2024). The coverage limitations of static security tools used for feedback can create blind spots in vulnerability detection and remediation (Nazzal et al., 2024). Certain specialized tools like gGAN may not function effectively without LLMs, as they lack the semantic understanding necessary for independent operation (Nazzal et al., 2024).

5.1.3 Iterative refinement and multi-round processes

Iterative Refinement and multi-round fixing have emerged as critical strategies for improving the quality and security of LLM-generated code through repeated enhancement cycles. Research has demonstrated that employing iterative refinement with multiple solution attempts for vulnerability patching shows significantly greater impact than single iterations (Alrashedy et al., 2023). This principle extends to implementing multi-round fixing processes specifically designed for vulnerable code snippets detected in LLM-generated code (Liu et al., 2024c).

5.1.3.1 Effectiveness

Iterative processes yield strong results by leveraging multiple refinement cycles to achieve superior security outcomes compared to single-pass approaches. Multi-round fixing processes for vulnerable code snippets have demonstrated exceptionally high success rates, successfully addressing between 89.4% to 100% of identified vulnerabilities in previously generated code (Liu et al., 2024c). This success is reinforced by evidence that iterative refinement with multiple solutions consistently improves performance beyond approaches that rely solely on single feedback loops, establishing the superiority of multi-iteration strategies (Alrashedy et al., 2023).

5.1.3.2 Limitations

As in feedback loops, iterative refinements are subject to a decline in performance when too many iterations are produced. According to Alrashedy et al. (2023), potential correctness issues could be introduced during automatic repairs due to multiple interactions with external systems or databases that cannot be easily verified through unit tests.

5.1.4 Model-specific approaches and fine-tuning

Model-specific approaches and fine-tuning encompass advanced techniques that modify or enhance the underlying LLM architecture to improve security outcomes. One prominent approach involves applying Security Model Fine-Tuning techniques such as LoRA for more secure token generation, as implemented in the CoSec framework (Li D. et al., 2024). This is complemented by Supervised Co-Decoding, where a specialized security model co-decodes with the base LLM, adjusting token probabilities to favor secure outputs (Li D. et al., 2024). Additionally, innovative approaches include implementing gGAN in conjunction with LLMs to reduce vulnerabilities while preserving semantic and functional integrity, guided by security analyzer feedback as demonstrated in PromSec (Nazzal et al., 2024).

5.1.4.1 Effectiveness

Model-specific hardening techniques have shown measurable improvements in LLM security capabilities through targeted architectural modifications and specialized training approaches. The CoSec framework significantly improves the average relative security ratio of LLMs by up to 37.14%, demonstrating that security enhancements need not come at the expense of code functionality (Li D. et al., 2024).

5.1.4.2 Limitations

Generalization to new programming languages or security behaviors outside the training scope remains a persistent challenge, limiting the adaptability of current approaches (He and Vechev, 2023). Security hardening approaches such as CoSec can lead to slight reductions in functional correctness or increased inference speed due to dual-model operation requirements (Li D. et al., 2024).

5.1.5 Discussion

Prompt engineering is currently the **most widely** used strategy for code generation with LLMs. However, this area is **underrepresented** in our database, primarily due to the scarcity of scientific literature focused on evaluating the quality of AI-generated code. This represents a **significant gap** in our understanding of LLM capabilities, especially considering their widespread real-world adoption for source code generation. A promising **future direction** would be the development of robust benchmarks to systematically assess the quality and safety of generated code. Such benchmarks should ensure that LLMs do not produce unacceptable or insecure outputs, ideally by evaluating them alongside complementary tools like static analyzers.

5.2 RQ5: Code generation and tool integration

Of the 50 analyzed papers, only **6** show insight into this research question, i.e., **12%** of the database.

A strategy for improving the quality and security of LLM-generated outputs involves establishing robust iterative feedback loops. In this approach, LLMs generate initial versions of code, proposed fixes for vulnerabilities, or formal specifications. These outputs are then systematically evaluated by a range of automated security and verification tools. The crucial step involves feeding the analysis results back to the LLM. Then, the LLM utilizes this comprehensive feedback to iteratively refine its generated code, fixes, or specifications. In Alrashedy et al. (2023), initially generated code is analyzed by Bandit; identified vulnerabilities trigger solution generation by the LLM; these solutions iteratively refine vulnerable code until no further issues are detected by Bandit or maximum iterations are reached. In Liu et al. (2024c), the multi-round process with CodeQL proved effectiveness to address vulnerabilities in generated code snippets using ChatGPT, and leveraging CWE information for automatic fixes. This approach demonstrates promising results in improving security robustness.

PromSec (Nazzal et al., 2024) framework integrates an LLM with static security analysis tools (Bandit and SpotBugs) in an iterative loop into its pipeline to identify vulnerabilities (CWEs)

in generated code. The output of the security analyzers (number of CWEs) drives prompt optimization via a gGAN, which in turn influences the LLM to generate more secure code. This iterative approach, where the output of the verification tools informs the LLM through prompt optimization, creates a more robust verification framework for generating secure code.

Automated validation is another critical strategy to ensure the high quality and security of LLM-generated artifacts. SA tools, such as CodeQL, are employed to automatically validate LLM-generated code and proposed fixes. Their role is to ensure functional correctness by identifying common programming errors, enforce coding standards, and, crucially, prevent the introduction of new security vulnerabilities or regressions in the codebase. Pearce et al. (2023) uses CodeQL to evaluate generated code for functional correctness and security vulnerabilities. This integration allows for automated validation of generated fixes, demonstrating how SA tools can improve robustness in code generation workflows. Formal verification tools, including ESBMC and Dafny, are utilized to rigorously check LLM-generated code and formal specifications. The aim here is to ensure that these outputs are satisfiable (i.e., not inherently contradictory), adequate (i.e., sufficiently capture the intended properties), and free from logical inconsistencies. Tihanyi et al. (2025) discusses integrating LLMs with formal verification tools like ESBMC for vulnerability detection. The approach does not explicitly integrates LLMs with SA tools, but ESBMC was used to classify vulnerabilities in generated code, proving that combining LLM-generated code with formal verification tools can enhance robustness by identifying potential issues systematically. Misu et al. (2024) demonstrates that LLMs can complement formal verification tools like Dafny by generating both method implementations and specifications (pre-conditions, post-conditions, and loop invariants). When LLMs generate code, formal verification tools can provide feedback on whether the code meets the specifications. This iterative process helps refine the generated solutions and ensures correctness.

5.2.1 Discussion

Recent work reveals a clear **trend** toward integrating LLMs with automated security analysis and formal verification tools in iterative feedback loops, aiming to progressively enhance code quality and security. These approaches, ranging from vulnerability-driven refinement with Bandit or CodeQL to prompt optimization pipelines like PromSec, demonstrate promising gains in robustness. However, there is a notable **divergence** in focus: some studies emphasize static analysis for vulnerability detection and repair, while others explore formal verification for logical soundness, with limited methodological integration between the two. This separation highlights a **gap** in unified frameworks that combine both functional correctness and security guarantees in a single iterative process. Moreover, most current methods rely on tool-specific feedback, raising questions about generalizability across domains and programming languages. **Future research** should explore hybrid pipelines that integrate static analysis, formal verification, and adaptive prompting in a cohesive loop, supported by standardized benchmarks to evaluate both security resilience and specification compliance.

5.3 RQ6: Common vulnerabilities introduced by LLMs

Of the 50 analyzed papers, only **8** show insight into this research question, i.e., **16%** of the database.

LLMs, while powerful code generators, frequently introduce a range of vulnerabilities into the code they produce. These weaknesses often mirror those found in human-written code, stemming from the patterns and practices learned from their training data. LLMs commonly generate code containing the following types of vulnerabilities:

- Injection-related issues: these are pervasive and include critical flaws like OS command injection (CWE-78), as noted by Gong et al. (2024); and frequently reported SQL Injection (CWE-89), e.g., by Alrashedy et al. (2023), Pearce et al. (2023), and Nazzal et al. (2024). These vulnerabilities arise when user-supplied data is directly incorporated into commands or queries without proper sanitization, allowing attackers to execute arbitrary code or manipulate databases.
- Input validation issues: as shown by Alrashedy et al. (2023), Gong et al. (2024), and Liu et al. (2024c) LLMs often neglect robust input validation, leading to vulnerabilities such as improper input validation (CWE-20), path traversal (CWE-22), and unrestricted upload of file with dangerous type (CWE-434). These allow attackers to bypass security controls by manipulating input to access unauthorized files or resources.
- Hard-coded credentials: a recurring security flaw (Alrashedy et al., 2023; Nazzal et al., 2024) is the use of hard-coded password (CWE-259), where sensitive authentication information is embedded directly into the code, making it easily discoverable and exploitable.
- Resource management issues: in addition, Alrashedy et al. (2023) and Liu et al. (2024c) observed that LLMs can generate code that leads to uncontrolled resource consumption (CWE-400), potentially enabling denial-of-service attacks by exhausting system resources.
- Memory and pointer errors: particularly prevalent in languages like C/C++, LLMs can introduce dangerous memory and pointer errors. These include dereference failures (e.g., NULL pointer issues/MissingNullTest), buffer overflows (e.g., PotentialBufferOverflow), arithmetic overflows, array bounds violations, and other unsafe memory operations. As a matter of fact, Tihanyi et al. (2025) and Liu et al. (2024c) found a prevalence of CWE-416, and CWE-476. Liu et al. (2024c) also spotted CWE-787, CWE-125 and CWE-119. Such errors can lead to crashes, data corruption, or arbitrary code execution.
- Other common weaknesses: in Gong et al. (2024), it was pointed out that the generated code may also exhibit vulnerabilities like improper restriction of XML external entity reference (CWE-611), open redirect (CWE-601), and cross-site scripting (XSS) (CWE-725), which can compromise user data and application integrity.

Beyond these categories, LLMs also commonly introduce vulnerabilities related to the use of unsafe library functions and a

general lack of robust credentials protection mechanisms within the generated code (Tihanyi et al., 2025; Gong et al., 2024).

Moreover, Misu et al. (2024) identify two main types of issues in unverified methods generated by LLMs, namely compilation errors and verification failures. The former refers to syntax errors or references to unknown types or functions not defined in Dafny's standard library. Verification failures refers to incorrect or weak specifications (e.g., post-conditions) or incorrect implementations that fail to meet specifications.

Interestingly, He and Vechev (2023) observed that the types of vulnerabilities introduced by LLMs are largely similar to those frequently found in human-generated code. This commonality suggests that LLMs, by learning from vast code corpora, internalize not only secure patterns but also insecure ones. For instance, LLM-generated runtime errors, such as overflows in static languages, align perfectly with common mistakes made by human developers. This similarity is often attributed to the replication of insecure coding practices prevalent in the massive open-source datasets on which LLMs are trained. Essentially, if a common vulnerability pattern exists in the training data, the LLM is likely to reproduce it. In fact, Liu et al. (2024c) reports that runtime errors in code generated in static languages (C, C++, Java) are mainly overflows, while in dynamic languages (Python3, JavaScript), type errors predominate, similar to human errors.

5.3.1 Discussion

The literature **consistently shows** that LLM-generated code is prone to a wide spectrum of vulnerabilities, ranging from injection flaws and weak input validation to hard-coded credentials, unsafe memory operations, and resource mismanagement, mirroring many of the same weaknesses found in human-written code. This parallel suggests a clear trend: LLMs inherit both secure and insecure coding patterns from their training data. **Future work** should focus on developing integrated frameworks that combine vulnerability detection, formal verification, and secure-by-design prompting, alongside curated training datasets that actively filter insecure patterns to reduce the replication of systemic weaknesses.

6 Cross-cutting aspects

6.1 RQ8: Open-source reliance in LLM-based software verification

Of the 50 analyzed papers, **49** released reproducibility packages, with the only exception of Wen et al. (2024b). All **50** use open-source datasets and benchmarks, but **10** use proprietary models.

Overall, there is a high reliance on open-source components, including datasets, benchmarks, tools, and some LLMs, as also shown in Appendix Figure 5, even though a recurring theme is the use of proprietary LLMs like GPT-3.5 and GPT-4.

To further analyze this aspect, we studied the available datasets that are used or mentioned in the papers we analyzed, and we show the results in Table 4, which highlights how open source benchmarks are a key component in this field.

6.1.1 Discussion

Open-source code, datasets, and benchmarks are **widely used** in LLM research for software verification, with 49 out of 50 reviewed papers releasing reproducibility packages and all using open-source datasets or benchmarks. While some studies rely on proprietary models like GPT-3.5 or GPT-4, open-source LLMs (e.g., CodeBERT, CodeT5, LLaMA) are frequently adopted as baselines, supporting transparency and reproducibility. Public datasets are central to this field, though specialized tasks may still require more tailored datasets. However, while the wide variety of datasets and benchmarks can be advantageous, the proliferation of custom datasets **risks** undermining the ability to evaluate and compare approaches within a unified framework. In particular, existing open datasets and benchmarks vary considerably in their vulnerability taxonomies (whether based on CWEs, alternative schemes, or undisclosed classifications), level of granularity, and programming language coverage. **Future research directions** should address the generalization and standardization of these benchmarks to guarantee comprehensive reproducibility.

6.2 D1: Code security awareness with LLMs

Of the 50 analyzed papers, only **5** show insight into this research question, i.e., **10%** of the database.

The most relevant proposal for this question is AIBugHunter by Fu et al. (2024b), which shows that LLMs can classify vulnerability types (CWE-ID and CWE-Type) and estimate their severity, providing developers with a clearer understanding of security breaches. Moreover, it is able to integrate real-time security alerts during coding, helping developers immediately identify potential vulnerabilities as they write code. Another useful feature is to explain suggested repairs and safe alternatives: VulRepair (Fu et al., 2022), integrated into AIBugHunter, suggests repair patches. Similarly, DLAP (Yang et al., 2025) and MSIVD, proposed in Yang et al. (2024), can educate developers on how to repair code, providing the underlying causes for its detection results, thus making vulnerabilities and their implications more comprehensible to developers. Ding et al. (2024) suggest exploring ways to teach code LLMs about security concepts inspired by how human developers learn secure coding practices. For example, pre-training methods focused on security awareness or hybrid systems combining LLMs with traditional program analysis tools could be promising directions.

6.3 D2: Ethical considerations and potential risks

Of the 50 analyzed papers, **19** show insight into this research question, i.e., **38%** of the database.

The integration of LLMs into security workflows introduces a critical set of ethical considerations and potential risks that warrant careful attention. However, these topics are not broadly discussed in all the examined works.

The most relevant discussions about potential risks could be found in Tihanyi et al. (2025) and Fu et al. (2024a), where

TABLE 4 Vulnerability datasets overview.

Name	Year	Vulnerabilities	Granularity	Language
Big-Vul	2020	91 CWE types	Function level	C/C++
CodeXGLUE (Devign)	2021	Memory leaks, buffer overflows, memory corruption, crashes	Function level	C/C++
CPATMiner	2019	/	Function level	Java
Cross-project Vulnerability Patch Dataset (CVPD)	2023	More than 100 different CWE	Function level	Java
CrossVul	2021	168 CWEs	File level	More than 40 languages
CVEFixes	2021	272 CWEs	File and function level	27 programming languages
CWE-Bench-Java	2024	120 CVEs spanning 4 CWEs	Repository	Java
D2A	2021	Infer warnings (buffer overflow, integer overflow, memory leak, etc.)	Function level	C/C++
Defects4J	2014	854 real bugs	Method level	Java
Devign	2019	~23k vulnerabilities, mostly memory-related	Function level	C/C++
DiverseVul	2023	150 CWE	Function level	C/C++
Draper	2018	149 CWE (Buffer Overflow, NULL Ptr Deref.)	Function level	C/C++
EvalPlus	2023	Not disclosed	Function level	Python
ExploitDB	2022	Not disclosed	Unspecified	Multi-language
FormAI-v2 dataset	2023	41 unique CWEs	Single file	C/C++
GitHub Advisory Database	2022	CVE/CWE-mapped advisories	Repository	Multi-language
Juliet C/C++ Test Suite	2017	118 CWEs	Single file	C/C++
LLMSecEval	2023	18 of Top 25 CWE scenarios	Single file	Multi-language
MBPP	2021	Not disclosed	Function level	Python
MBPP-san-DFY	2024	Not disclosed	Function level	Python
National Vulnerability Database (NVD)	2005	>250k CWE vulnerabilities	Different level	Multi-language
OOPSLA-13	2013	Not disclosed	Function level	C
PreciseBugs	2023	Many CWEs, CVEs, OOB read/write	Project level	Multi-language
PRIMEVUL	2024	140 CWEs	Function level	C/C++
RealVul	2024	Many CWEs (e.g., CWE-495, 256, 94, 20)	Function level	C++
Recoder	2023	Not disclosed	Function level	Java
Reveal	2020	Not disclosed	Function level	C/C++
SARD	2018	Over 150 classes of weaknesses	File level	Multi-language
SATE IV Juliet	2013	Many CVEs, not exactly disclosed	File level	Multi-language
SecBench.js	2023	12 CVEs	File level	JavaScript
SecurityEval	2022	75 CWEs	Function level	Python
Semantics-based Vulnerability Candidate (SeVC) dataset	2021	126 CWEs	Function level	C/C++
SequenceR	2021	Not disclosed	Line level	Java
SmartBugs	2021	208 vulnerabilities	Contract level	Solidity
SolidiFI benchmark	2020	7 different bug types	Smart contract	Solidity
SV-COMP	2021	Not disclosed	File level	C
SVEN	2023	9 CWEs	Function level	Python, C/C++
SVulD	2023	Not disclosed	Function level	C/C++

(Continued)

TABLE 4 (Continued)

Name	Year	Vulnerabilities	Granularity	Language
TaintBench Suite	2022	CWEs, not fully disclosed	File level	Android
TFix	2021	Not disclosed	File level	JavaScript
VJBench	2023	42 vulnerabilities, 23 CWE types	File level	Java
Vul4J	2022	25 CWE types	File level	Java
Vuldeepecker	2018	2 CWE	Snippet of code	C/C++
VulnPatchPairs	2024	Not disclosed	Function level	C
VulRepair	2022	>25 types of CWEs	Function level	8 Programming Languages

authors warn against deploying LLM-generated code directly into production environments without human oversight, due to its high vulnerability rates. In particular, Fu et al. (2024a) discusses potential risks through a user study with software practitioners regarding AI-generated vulnerability repairs. Participants highlighted that AI algorithms, while suggesting fixes, "may not always produce the most effective or secure solutions" and emphasize that "human expertise is still necessary to validate and test the proposed repairs, ensuring that they do not introduce new vulnerabilities or have unintended consequences." The suggestion is to integrate AI repair systems in a broader security strategy that includes human expertise. The paper (Fu et al., 2024a) also identifies threats to external validity, noting that the Vulnerability Repair Through Vulnerability Query and Mask (VQM) approach might not generalize to other vulnerabilities, projects, or programming languages outside of the studied dataset, indicating a risk of limited applicability in diverse real-world scenarios. Moreover, Tihanyi et al. (2025) further observes that, when relying on code generated by LLMs, there is a risk of replicating insecure coding patterns, which can foster a false sense of security while potentially allowing code completion to silently inject security bugs into the software.

The authors of PromSec (Nazzal et al., 2024) warn about risks such as inherent vulnerabilities in LLM-generated code due to insecure training data from open-source repositories. They emphasize the need for caution when relying on these outputs without verification or optimization processes. This perspective is also supported by Alrashedy et al. (2023) and Liu P. et al. (2024), which briefly conclude that LLMs do not produce code that is free from vulnerabilities, and there are also difficulties in ensuring the correctness of generated snippets. Also, Pearce et al. (2023) briefly mentions risks such as alignment failure, where generated outputs may not align with user intentions, potentially introducing new vulnerabilities or incorrect fixes.

Wang et al. (2024) findings indicate that using LLMs for code-reasoning tasks can entail specific risks, including potential source code leakage within private organizations and the possibility of incurring substantial token-related expenses.

In vulnerability detection, Ullah et al. (2024) highlights risks such as high FPR and incorrect reasoning leading to wasted developer time and potential security misjudgments if deployed prematurely. Ding et al. (2024) found that existing benchmarks significantly overestimate the performance of code LMs for vulnerability detection, which can lead to a false sense of security among developers.

In datasets, DiverseVul (Chen et al., 2023) warn about label noise that can introduce inaccuracies into model evaluation metrics and potentially mislead developers or security analysts relying on these models for vulnerability detection tasks. The authors highlight the need for deeper investigation into label accuracy issues to mitigate these risks.

He and Vechev (2023) observes that an attacker could use their proposed SVEN, and the adversarial testing of SVEN_vul, to intentionally degrade an LLM's security performance by guiding it to generate insecure code. On the topic of security attacks, Nazzal et al. (2024), Liu et al. (2024a), and Chen et al. (2025) also noted that LLMs can be used for both defensive (security hardening) and offensive (vulnerability creation) purposes.

Chen et al. (2025) highlight risks such as false positives caused by biases (e.g., Protected Mechanism Bias and Development Intent Bias) and susceptibility to attacks like code poisoning or obfuscation attacks that can exploit ChatGPT's limitations. These issues could lead to overlooking real vulnerabilities or misidentifying secure code as flawed. The problem of biases in data was also discussed in Yu et al. (2024).

Regarding ethical considerations, a common remark is the inherent lack of interpretability of these models, due to their black-box nature (Endres et al., 2024; Pearce et al., 2023; Yang et al., 2025; Fu et al., 2024a). As a consequence, future deprecations of models could undermine the reproducibility of experiments, as discussed by Wen et al. (2024a) about their proposed LLM4SA, and by Endres et al. (2024). Similar warnings come from authors of GRACE (Lu et al., 2024). To this extent, Hanif and Maffeis (2022) emphasizes the need for explainability techniques to address false positives when applied to real-world projects.

7 Conclusions

LLMs have rapidly transformed the landscape of software verification and secure code generation, offering both substantial opportunities and notable challenges. This review mapped the dual role of LLMs: as tools for code verification (LLM4Verification) and as generators of code requiring verification (Verification4LLM), analyzing 50 high-quality, recent studies in the field.

TABLE 5 Summary of answers for each research question.

RQ	Answer
RQ1	Scientific literature is divided on this question, recognizing the high potential of LLMs in software verification, especially compared to previous ML or DL approaches, but also pointing out critical weaknesses (such as high FPR), from which even big models, considered state-of-the-art, like OpenAI models are not exempt. However, careful training, integration with other tools, and system design produced robust and powerful LLM-based frameworks.
RQ2	Main techniques for program verification with LLMs include fine-tuning, prompt engineering, integration with SA tools giving external feedback, RAG, and agentic frameworks. Fine-tuning is overall the most effective, especially in combination with SA tools, but also the most costly, and it may suffer from catastrophic forgetting. External feedback, tool integration, and RAG offer valid alternatives that could be leveraged to make more useful prompts (e.g., ICL), and combined with prompt engineering techniques, proved to be effective. However, prompt engineering remains limited and less controllable compared to fine-tuning. Agentic frameworks are promising approaches, but not yet fully explored.
RQ3	Multiple strategies show promise for enhancing LLM security in code generation, including prompt engineering, tool integration with static analyzers and formal verification, iterative refinement processes, and model-specific hardening techniques. These approaches can achieve significant improvements. However, fundamental limitations persist: LLMs inherently generate insecure code at unacceptable rates, prompt engineering alone is insufficient, and complex vulnerabilities remain challenging. Success requires careful integration of multiple techniques rather than relying on any single approach.
RQ4	LLM-security tool integration employs diverse strategies, including feedback loops with static analyzers, LLM-generated inputs for formal verification, post-processing of tool outputs, direct tool invocation by agents, and hybrid frameworks combining multiple data sources. These integrations demonstrate significant effectiveness: vulnerability refinement improvements. The synergistic approach overcomes individual tool limitations by leveraging LLMs' semantic understanding alongside formal tools' logical reasoning, creating robust hybrid systems enhance both detection capabilities and code quality while addressing scalability and accuracy challenges inherent in standalone approaches.
RQ5	Integrating LLMs with static analyzers and formal verification tools enables more robust code generation by combining LLMs' generative strengths with the precision of automated analysis. Two main approaches are iterative feedback loops—where LLM-generated code is repeatedly checked and refined using tools like Bandit, CodeQL, and formal verifiers—and automated validations. These approaches are reported to improve the validity and security of generated code.
RQ6	LLMs frequently introduce vulnerabilities such as injection flaws (e.g., SQL and OS command injection), improper input validation, hard-coded credentials, resource management issues, and memory errors—mirroring the most common weaknesses in human-written code. These issues arise because LLMs learn from large code corpora that include insecure patterns, so the vulnerabilities they generate largely reflect those found in their training data.
RQ7	LLMs are highly effective at repairing code vulnerabilities, often outperforming traditional and earlier learning-based techniques, especially when combined with iterative feedback from static analyzers or enhanced with retrieval-augmented and knowledge-based methods. However, they still struggle with complex or rare vulnerabilities, may introduce new bugs, and their reliability depends on task complexity, training data quality, and integration with external tools. Overall, LLM-based repair is a major advance but not yet a complete replacement for traditional approaches or human oversight.
RQ8	Overall, there is a strong reliance on open-source resources, which enhances scientific rigor and enables broader community participation in LLM-driven software security research. However, issues regarding generalization and standardization still pose significant challenges, constraining the comparability of findings, limit both the reproducibility and the development of unified benchmarking methodologies.
D1	Some LLM-based frameworks show promising potential to enhance the security development process by combining detailed vulnerability explanations, real-time contextual alerts, comprehensive repair guidance, cause-effect comprehension, and improved code review efficiency. However, these capabilities are yet largely unexplored in literature about LLMs and secure coding.
D2	The integration of LLMs into security workflows presents significant ethical considerations and practical risks that require careful management. Key ethical imperatives include maintaining human oversight and ensuring explainability and transparency. Critical risks encompass the introduction of new vulnerabilities, over-reliance leading to false security confidence, reproducibility challenges, data leakage concerns, performance gaps with high false-positive rates, potential adversarial exploitation, and limited generalizability across diverse contexts. Success requires balancing AI capabilities with human expertise, robust validation processes, and comprehensive risk mitigation strategies rather than blind adoption of LLM technologies.

We formulated eight research questions to guide our analysis. Table 5 presents detailed responses that also serve as a summary of the key insights and findings.

LLMs demonstrate strong potential in vulnerability detection, often surpassing traditional machine learning and static analysis tools when carefully fine-tuned or integrated with external analyzers. However, high false positive rates, limited robustness in complex scenarios, and a dependency on high-quality, balanced datasets remain persistent issues. Prompt engineering, retrieval-augmented generation, and agentic frameworks have emerged as effective strategies to enhance LLM performance, but each brings its own set of trade-offs in terms of scalability, generalizability, and resource requirements. Recent developments show a shift toward hybrid pipelines that combine LLMs with static analysis and domain-specific heuristics, aiming to mitigate these limitations and improve reliability. Yet, the lack of standardized benchmarks and evaluation protocols continues to hinder meaningful comparison across approaches.

Hybrid systems that combine LLMs with static analyzers, formal verification tools, and iterative feedback loops have proven especially effective. These integrations not only improve detection and repair rates but also help mitigate some inherent weaknesses of LLMs, such as their tendency to introduce common vulnerabilities—including injection flaws, improper input validation, and memory errors—mirroring those found in human-written code.

In program repair, LLM-based approaches have achieved impressive results, particularly when guided by vulnerability-specific information and validated through multi-round feedback. Nonetheless, challenges persist in handling complex or rare vulnerabilities, ensuring functional correctness, and preventing the introduction of new bugs. Their effective integration into practical workflows requires the development of comprehensive and systematic frameworks that incorporate complementary methodologies, such as static analysis, prompt engineering, and symbolic reasoning, in order to fully exploit their capabilities.

The field is characterized by a strong reliance on open-source datasets, benchmarks, and, increasingly, open-source LLMs, which have fostered transparency and reproducibility. Yet, the use of proprietary models and the limitations of current

benchmarks highlight the need for continued development of robust, representative evaluation frameworks.

Finally, ethical considerations and practical risks—such as over-reliance on automated tools, lack of explainability, and the propagation of insecure coding patterns—underscore the necessity of maintaining human oversight and comprehensive validation processes. Although the diversity of datasets and benchmarks can foster innovation, the widespread use of custom resources poses a challenge to consistent evaluation across approaches. To advance the field, future research should prioritize the generalization and standardization of vulnerability benchmarks, ensuring reproducibility and enabling meaningful comparison within unified frameworks.

In summary, while LLMs are reshaping software verification and secure code generation, realizing their full potential will require ongoing advances in model robustness, hybrid verification frameworks, dataset quality, and ethical deployment practices. Collaboration between AI researchers, software engineers, and the broader security community remains essential to ensure that these powerful tools contribute to the development of secure, reliable software systems.

Author contributions

GD: Writing – original draft, Conceptualization, Writing – review & editing, Investigation, Methodology, Formal analysis, Visualization, Data curation. EI: Funding acquisition, Writing – original draft, Investigation, Writing – review & editing, Data curation, Formal analysis, Conceptualization, Methodology, Visualization.

Funding

The author(s) declare that financial support was received for the research and/or publication of this article. This work was supported by the Bando di Ateneo 2024 per la Ricerca, funded by University of Parma (FIL 2024 PROGETTI B IOTTI-CUP D93C24001250005).

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Generative AI statement

The author(s) declare that no Gen AI was used in the creation of this manuscript.

Any alternative text (alt text) provided alongside figures in this article has been generated by Frontiers with the support of artificial intelligence and reasonable efforts have been made to ensure accuracy, including review by the authors wherever possible. If you identify any issues, please contact us.

Publisher's note

Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fcomp.2025.1655469/full#supplementary-material

References

Alhanahnah, M., Hasan, M. R., and Bagheri, H. (2024). An empirical evaluation of pre-trained large language models for repairing declarative formal specifications. *arXiv preprint arXiv:2404.11050*.

Alrashedy, K., Aljasser, A., Tambwekar, P., and Gombolay, M. (2023). Can LLMs patch security issues? *arXiv preprint arXiv:2312.00024*.

Basic, E., and Giaretta, A. (2024). Large language models and code security: a systematic literature review. *CoRR, abs/2412.15004*.

Berabi, B., He, J., Raychev, V., and Vechev, M. (2021). TFix: Learning to fix coding errors with a text-to-text transformer," in *International Conference on Machine Learning* (PMLR), 780–791.

Chapman, P. J., Rubio-González, C., and Thakur, A. V. (2024). "Interleaving static analysis and LLM prompting," in *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2024*, 9–17. doi: 10.1145/3652588.3663317

Chen, C., Su, J., Chen, J., Wang, Y., Bi, T., Yu, J., et al. (2025). When ChatGPT meets smart contract vulnerability detection: how far are we? *ACM Trans. Softw. Eng. Methodol.* 34, 1–30. doi: 10.1145/3702973

Chen, Y., Ding, Z., Alowain, L., Chen, X., and Wagner, D. (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, pages* 654–668. doi: 10.1145/3607199.3607242

Ding, H., Liu, Y., Piao, X., Song, H., and Ji, Z. (2025). SmartGuard: an LLM-enhanced framework for smart contract vulnerability detection. *Expert Syst. Appl.* 269:126479. doi: 10.1016/j.eswa.2025.126479

Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., et al. (2024). Vulnerability detection with code language models: how far are we? *arXiv preprint arXiv:2403.18624*.

Dolcetti, G., Arceri, V., Iotti, E., Maffeis, S., Cortesi, A., and Zaffanella, E. (2024). Helping llms improve code generation using feedback from testing and static analysis. *CoRR, abs/2412.14841*.

Endres, M., Fakhoury, S., Chakraborty, S., and Lahiri, S. K. (2024). "Can large language models transform natural language intent into formal method postconditions?," in *Proceedings of the ACM on Software Engineering*, 1889–1912. doi: 10.1145/3660791

Ferrag, M. A., Battah, A., Tihanyi, N., Jain, R., Maimuţ, D., Alwahedi, F., et al. (2025). SecureFalcon: Are we there yet in automated software vulnerability detection with LLMs? *IEEE Trans. Softw. Eng.* 51, 1248–1265. doi: 10.1109/TSE.2025.3548168

Fu, M., Nguyen, V., Tantithamthavorn, C., Phung, D., and Le, T. (2024a). Vision transformer inspired automated vulnerability repair. *ACM Trans. Softw. Eng. Methodol.* 33, 1–29. doi: 10.1145/3632746

Fu, M., Nguyen, V., Tantithamthavorn, C. K., Le, T., and Phung, D. (2023). Vulexplainer: a transformer-based hierarchical distillation for explaining vulnerability types. *IEEE Trans. Softw. Eng.* 49, 4550–4565. doi: 10.1109/TSE.2023.3305244

Fu, M., and Tantithamthavorn, C. (2022). "LineVul: a transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 608–620. doi: 10.1145/3524842.3528452

Fu, M., Tantithamthavorn, C., Le, T., Kume, Y., Nguyen, V., Phung, D., et al. (2024b). AIBugHunter: a practical tool for predicting, classifying and repairing software vulnerabilities. *Empir. Softw. Eng.* 29:4. doi: 10.1007/s10664-023-10346-3

Fu, M., Tantithamthavorn, C., Le, T., Nguyen, V., and Phung, D. (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*, 935–947. doi: 10.1145/3540250.3549098

Ge, X., Fang, C., Zhang, Q., Wu, D., Yu, B., Zheng, Q., et al. (2024). Pre-trained model-based actionable warning identification: a feasibility study. *arXiv preprint arXiv:2403.02716*.

Gong, J., Duan, N., Tao, Z., Gong, Z., Yuan, Y., and Huang, M. (2024). How well do large language models serve as end-to-end secure code producers? *arXiv preprint arXiv:2408.10495*.

Hanif, H., and Maffeis, S. (2022). "VulBERTa: simplified source code pre-training for vulnerability detection," in *2022 International Joint Conference on Neural Networks (IJCNN)* (IEEE), 1–8. doi: 10.1109/IJCNN55064.2022.9892280

He, J., and Vechev, M. (2023). "Large language models for code: security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 1865–1879. doi: 10.1145/3576915.3623175

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., et al. (2024). "MetaGPT: Meta programming for a multi-agent collaborative framework," in The *Twelfth International Conference on Learning Representations*.

Huang, K., Meng, X., Zhang, J., Liu, Y., Wang, W., Li, S., et al. (2023). "An empirical study on fine-tuning large language models of code for automated program repair," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (IEEE), 1162–1174. doi: 10.1109/ASE56229.2023.00181

Huang, K., Xu, Z., Yang, S., Sun, H., Li, X., Yan, Z., et al. (2025). Evolving paradigms in automated program repair: taxonomy, challenges, and opportunities. *ACM Comput. Surv.* 57:36. doi: 10.1145/3696450

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., et al. (2024). Livecodebench: holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Jiang, J., Wang, F., Shen, J., Kim, S., and Kim, S. (2025). A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Kolthoff, K., Kretzer, F., Bartelt, C., Maedche, A., and Ponzetto, S. P. (2025). "GUIDE: LLM-driven GUI generation decomposition for automated prototyping," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (IEEE), 1–4. doi: 10.1109/ICSE-Companion66252.2025.00010

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, 9459–9474.

Li, D., Yan, M., Zhang, Y., Liu, Z., Liu, C., Zhang, X., et al. (2024). "CoSec: on-the-fly security hardening of code LLMs via supervised co-decoding," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1428–1439. doi: 10.1145/3650212.3680371

Li, H., Hao, Y., Zhai, Y., and Qian, Z. (2023). "Assisting static analysis with large language models: a chatgpt experiment," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2107–2111. doi: 10.1145/3611643.3613078

Li, H., Hao, Y., Zhai, Y., and Qian, Z. (2024). "Enhancing static analysis for practical bug detection: an LLM-integrated approach," in *Proceedings of the ACM on Programming Languages*, 474–499. doi: 10.1145/3649828

Li, Z., Dutta, S., and Naik, M. (2024). LLM-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238*.

Li, Z., Dutta, S., and Naik, M. (2025). "IRIS: LLM-assisted static analysis for detecting security vulnerabilities," in *The Thirteenth International Conference on Learning Representations*.

Liu, P., Liu, J., Fu, L., Lu, K., Xia, Y., Zhang, X., et al. (2024). "Exploring ChatGPT's capabilities on vulnerability management," in *33rd USENIX Security Symposium (USENIX Security 24)*, 811–828.

Liu, Y., Gao, L., Yang, M., Xie, Y., Chen, P., Zhang, X., et al. (2024a). Vuldetectbench: Evaluating the deep capability of vulnerability detection with large language models. *arXiv preprint arXiv:2406.07595*.

Liu, Y., Tantithamthavorn, C., Liu, Y., and Li, L. (2024b). On the reliability and explainability of language models for program generation. *ACM Trans. Softw. Eng. Methodol.* 33, 1–26. doi: 10.1145/3641540

Liu, Z., Tang, Y., Luo, X., Zhou, Y., and Zhang, L. F. (2024c). No need to lift a finger anymore? Assessing the quality of code generation by chatgpt. *IEEE Trans. Softw. Eng.* 50, 1548–1584. doi: 10.1109/TSE.2024.3392499

Liu, Z., Tang, Z., Zhang, J., Xia, X., and Yang, X. (2024d). "Pre-training by predicting program dependencies for vulnerability analysis tasks," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13. doi: 10.1145/3597503.3639142

Lu, G., Ju, X., Chen, X., Pei, W., and Cai, Z. (2024). GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *J. Syst. Softw.* 212:112031. doi: 10.1016/j.jss.2024.112031

Marvin, G., Hellen, N., Jjingo, D., and Nakatumba-Nabende, J. (2023). "Prompt engineering in large language models," in *International Conference on Data Intelligence and Cognitive Informatics* (Springer), 387–402. doi: 10.1007/978-981-99-7962-2_30

Misu, M. R. H., Lopes, C. V., Ma, I., and Noble, J. (2024). "Towards AI-assisted synthesis of verified dafny methods," in *Proceedings of the ACM on Software Engineering*, 812–835. doi: 10.1145/3643763

Mohajer, M. M., Aleithan, R., Harzevili, N. S., Wei, M., Belle, A. B., Pham, H. V., et al. (2023). SkipAnalyzer: an embodied agent for code analysis with large language models. *CoRR*.

Nazzal, M., Khalil, I., Khreishah, A., and Phan, N. (2024). "PromSec: prompt optimization for secure generation of functional source code with large language models (LLMs)," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2266–2280. doi: 10.1145/3658644.3690298

Negri-Ribalta, C., Géraud-Stewart, R., Sergeeva, A., and Lenzini, G. (2024). A systematic literature review on the impact of AI models on the security of code generation. *Front. Big Data* 7:1386720. doi: 10.3389/fdata.2024.1386720

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. (2022). "Asleep at the keyboard? Assessing the security of github copilot's code contributions," in *IEEE Symposium on Security and Privacy, S&P 2022* (IEEE), 754–768. doi: 10.1109/SP46214.2022.9833571

Pearce, H., Tan, B., Ahmad, B., Karri, R., and Dolan-Gavitt, B. (2023). "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)* (IEEE), 2339–2356. doi: 10.1109/SP46215.2023.10179324

Shestov, A., Levichev, R., Mussabayev, R., Maslov, E., Zadorozhny, P., Cheshkov, A., et al. (2025). Finetuning large language models for vulnerability detection. *IEEE Access* 13, 38889–38900. doi: 10.1109/ACCESS.2025.3546700

Thapa, C., Jang, S. I., Ahmed, M. E., Camtepe, S., Pieprzyk, J., and Nepal, S. (2022). "Transformer-based language models for software vulnerability detection," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 481–496. doi: 10.1145/3564625.3567985

Tihanyi, N., Bisztray, T., Ferrag, M. A., Jain, R., and Cordeiro, L. C. (2025). How secure is ai-generated code: a large-scale comparison of large language models. *Empir. Softw. Eng.* 30, 1–42. doi: 10.1007/s10664-024-10590-1

Tihanyi, N., Bisztray, T., Jain, R., Ferrag, M. A., Cordeiro, L. C., and Mavroeidis, V. (2023). "The formai dataset: Generative AI in software security through the lens of formal verification," in *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2023, San Francisco, CA, USA, 8 December 2023*, eds. S. McIntosh, E. Choi, and S. Herbold (ACM), 33–43. doi: 10.1145/3617555.3617874

Ullah, S., Han, M., Pujar, S., Pearce, H., Coskun, A., and Stringhini, G. (2024). "LLMs cannot reliably identify and reason about security vulnerabilities (yet?): a comprehensive evaluation, framework, and benchmarks," in *IEEE Symposium on Security and Privacy, S&P* 2024. doi: 10.1109/SP54263.2024.00210

Wang, C., Zhang, W., Su, Z., Xu, X., Xie, X., and Zhang, X. (2024). "LLMDFA: analyzing dataflow in code with large language models," in *Advances in Neural Information Processing Systems*, 131545–131574.

Wang, J., and Chen, Y. (2023). "A review on code generation with llms: application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, 284–289. doi: 10.1109/MedAI59581.2023.00044

Wang, W., Wang, Y., Joty, S., and Hoi, S. C. (2023). "RAP-gen: retrieval-augmented patch generation with codet5 for automatic program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 146–158. doi: 10.1145/3611643.3616256

Wen, C., Cai, Y., Zhang, B., Su, J., Xu, Z., Liu, D., et al. (2024a). Automatically inspecting thousands of static bug warnings with large language

model: how far are we? *ACM Trans. Knowl. Discov. Data* 18, 1–34. doi: 10.1145/3653718

Wen, C., Cao, J., Su, J., Xu, Z., Qin, S., He, M., et al. (2024b). "Enchanting program specification synthesis by large language models using static analysis and program verification," in *International Conference on Computer Aided Verification* (Springer), 302–328. doi: 10.1007/978-3-031-65630-9_16

Wen, X.-C., Gao, C., Gao, S., Xiao, Y., and Lyu, M. R. (2024). "SCALE: constructing structured natural language comment trees for software vulnerability detection," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 235–247. doi: 10.1145/3650212.3652124

Wu, Y., Jiang, N., Pham, H. V., Lutellier, T., Davis, J., Tan, L., et al. (2023). "How effective are neural networks for fixing security vulnerabilities," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1282–1294. doi: 10.1145/3597926.3598135

Wu, Y., Wen, M., Yu, Z., Guo, X., and Jin, H. (2024). "Effective vulnerable function identification based on cve description empowered by large language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 393–405. doi: 10.1145/3691620.3695013

Yang, A. Z., Tian, H., Ye, H., Martins, R., and Goues, C. L. (2024). Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*.

Yang, Y., Zhou, X., Mao, R., Xu, J., Yang, L., Zhang, Y., et al. (2025). DLAP: a deep learning augmented large language model prompting framework for software vulnerability detection. *J. Syst. Softw.* 219:112234. doi: 10.1016/j.jss.2024.112234

Yin, X., Ni, C., and Wang, S. (2024). Multitask-based evaluation of open-source LLM on software vulnerability. *IEEE Trans. Softw. Eng.* 50, 3071–3087. doi: 10.1109/TSE.2024.3470333

Yu, L., Chen, S., Yuan, H., Wang, P., Huang, Z., Zhang, J., et al. (2024). Smart-LLaMA: two-stage post-training of large language models for smart contract vulnerability detection and explanation. *arXiv preprint arXiv:2411.06221*.

Zhang, Q., Fang, C., Xie, Y., Ma, Y., Sun, W., Yang, Y., et al. (2024). A systematic literature review on large language models for automated program repair. *CoRR, abs/2405.01466*.

Zhang, Q., Fang, C., Yu, B., Sun, W., Zhang, T., and Chen, Z. (2023). Pre-trained model-based automated software vulnerability repair: how far are we? *IEEE Trans. Depend. Sec. Comput.* 21, 2507–2525. doi: 10.1109/TDSC.2023.3308897

Zhou, X., Cao, S., Sun, X., and Lo, D. (2024a). Large language model for vulnerability detection and repair: literature review and the road ahead. *CoRR, abs/2404.02525*.

Zhou, X., Kim, K., Xu, B., Han, D., and Lo, D. (2024b). "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*, 1–13. doi: 10.1145/3597503.3639222

Zhu, X., Zhou, W., Han, Q., Ma, W., Wen, S., and Xiang, Y. (2025). When software security meets large language models: a survey. *IEEE CAA J. Autom. Sinica* 12, 317–334. doi: 10.1109/JAS.2024.124971

Zubair, F., Al-Hitmi, M., and Catal, C. (2025). The use of large language models for program repair. *Comput. Stand. Interfaces* 93:103951. doi: 10.1016/j.csi.2024.103951