Scilight

Transactions on Artificial Intelligence
https://www.sciltp.com/journals/tai

Review

A Contemporary Survey of Large Language Model Assisted Program Analysis

Jiayimei Wang [1], Tao Ni [1], Wei-Bin Lee [2,3] and Qingchuan Zhao [1,*]

[1] Department of Computer Science, City University of Hong Kong, Hong Kong
[2] Information Security Research Center, Hon Hai Research Institute, Taipei City 114699, Taiwan
[3] Department of Information Engineering and Computer Science, Feng Chia University, Taichung 407, Taiwan
* Correspondence: cs.qczhao@cityu.edu.hk

How To Cite: Wang, J.; Ni, T.; Lee, W.-B.; et al. A Contemporary Survey of Large Language Model Assisted Program Analysis. *Transactions on Artificial Intelligence* **2025**, *1*(1), 6. https://doi.org/10.53941/tai.2025.100006.

Received: 6 February 2025
Revised: 20 April 2025
Accepted: 28 April 2025
Published: 26 May 2025

Abstract: The increasing complexity of software systems has driven significant advancements in program analysis, as traditional methods are unable to meet the demands of modern software development. To address these limitations, deep learning techniques, particularly Large Language Models (LLMs), have gained attention due to their context-aware capabilities in code comprehension. Recognizing the potential of LLMs, researchers have extensively explored their application in program analysis since their introduction. Despite existing surveys on LLM applications in cybersecurity, comprehensive reviews specifically addressing their role in program analysis remain scarce. This survey reviews the application of LLMs in program analysis, categorizing existing work into static, dynamic, and hybrid approaches. We also identify current research hotspots, such as LLM integration in automated vulnerability detection and code analysis, common challenges like model interpretability and training data limitations, and future directions, including using LLMs to convert dynamic analysis tasks into static ones. This survey aims to demonstrate the potential of LLMs in advancing program analysis practices and offer actionable insights for security researchers seeking to enhance detection frameworks or develop domain-specific models.

Keywords: large language model; program analysis; vulnerability detection

1. Introduction

With the continuous advancement of information technology, software plays an increasingly significant role in daily life, making its quality and reliability a critical concern for both academia and industry [1]. This is because software vulnerabilities in domains such as finance, healthcare, critical infrastructure, aerospace, and cybersecurity [2] can lead to considerable financial losses or even societal harm [3]. Examples include data breaches in financial systems [4], malfunctioning medical devices [5], disruptions to power grids [6], failures in aviation control systems [7], and exploitation of security loopholes in sensitive government networks [8]. Accordingly, many techniques have been proposed to detect such vulnerabilities that compromise software quality and reliability, and program analysis has been proven effective in such tasks. It aims to examine computer programs to identify or verify their properties to detect vulnerabilities through abstract interpretation, constraint solving, and automated reasoning [9].

However, as software complexity and scale increase, traditional program analysis methods encounter challenges in meeting the demands of contemporary development. Specifically, these traditional methods face substantial challenges in handling dynamic behaviors, cross-language interactions, and large-scale codebases [10,11]. Fortunately, recent advancements in machine learning have initiated a shift in program analysis [12] and shed light on a promising research direction to address the limitations of traditional program analysis methods.

In particular, the literature has attempted to combine deep learning with program analysis, applying it to strengthen the detection of vulnerabilities and achieve automated code fixes, thereby minimizing human intervention and increasing precision [13]. However, deep learning models lack the ability to effectively integrate contextual

information over long sequences, limiting their performance in tasks requiring deep reasoning or multi-turn understanding [14, 15]. Consequently, these models struggle to handle complex software and large codebases and lack the capability for cross-project analysis.

Fortunately, the most recent advancement, i.e., large language model (LLM), has been found promising in addressing the limitations of early deep learning models, such as constrained contextual understanding and generalization, enabling them to handle tasks across multiple domains with greater versatility [16]. Particularly, as for program analysis, LLMs surpass traditional deep learning methods and have been applied to various tasks [17], including automated vulnerability and malware detection, code generation and repair, and providing scalable solutions that integrate static and dynamic analysis methods. Moreover, it also shows a great potential to cope with the growing difficulty of analyzing modern software systems.

Though promising, the literature lacks a comprehensive and systematica view of LLM-assisted program analysis given the presence of numerous related attempts and applications. Therefore, this work aims to systematically review the state-of-the-art of LLM-assisted program analysis applications and specify its role in the development of program analysis. To collect as many relevant studies as possible, we perform an automated search for papers published between January 2019 and December 2024 in various academic databases, including the ACM Digital Library, IEEE Xplore Digital Library, arXiv, and DBLP, followed by a manual screening for content relevance. In the first round, we use keywords related to program analysis, such as: static analysis, dynamic analysis, software analysis, fuzz, verification, etc. Next, we apply additional keywords related to LLMs, such as: LLM, large language model, GPT, CodeX, LLaMA, etc., to filter out irrelevant studies. Finally, we manually screen the remaining papers, applying the following inclusion criteria: (i) the paper proposes or improves a method, research, or tool/framework aimed at using LLMs to assist program analysis; (ii) the paper designs a testing framework and compares the performance differences of various LLMs applied to program analysis-related fields, providing an optimal solution; (iii) the paper involves specific testing techniques (e.g., fuzzing). If the paper meets any of these three criteria, it is included. Ultimately, we identify 82 relevant papers and organize them into a structured taxonomy. Figure 1 illustrates the classification framework, where the relevant research is categorized into LLMs for static analysis, LLMs for dynamic analysis, and hybrid approach.

Although program analysis is a relatively narrow field, exploring the application of LLMs in this domain allows us to uncover their potential in other fields. For instance, LLMs' success in program analysis could impact fields like such as automated code repair and software optimization, thereby driving the broader development of LLM technology in cybersecurity tasks. Additionally, focusing on LLMs in program analysis provides insights into their contextual understanding and cross-domain adaptation, facilitating their use in more complex, wide-ranging tasks. Therefore, as shown in Table 1, unlike previous surveys that broadly examined the applications of LLMs in cybersecurity, our work narrows its focus to program analysis, delivering a more detailed and domain-specific exploration.

Table 1. Summary and comparison with prior surveys/reviews on program analysis (PA) and LLMs in cybersecurity.

Reference	Year	LLM	PA	Topic
[18]	2005	✗	✓	Static analysis
[19]	2014	✗	✓	Dynamic analysis
[20]	2018	✗	✓	Symbolic execution
[21]	2019	✗	✓	Machine learning-based
[22]	2024	✓	✗	Violence detection
[23]	2024	✓	✗	Cyber threat detection
[24]	2024	✓	✗	Software testing
[25]	2024	✓	✗	Malware detection
[26]	2024	✓	✗	Cyber security
[27]	2024	✓	✗	Cyber defense
[28]	2025	✓	✗	Software security
[29]	2025	✓	✗	Code security

Compared to other similar works, this survey has some unique contributions, as shown below:

- This survey conducts an in-depth study of 82 relevant papers on the application of LLMs in program analysis, providing a detailed overview of how LLMs are applied to program analysis tasks, categorizing existing methods into static, dynamic, and hybrid approaches, and extracting common themes across these categories.

- This survey analyzes commonly used LLMs in program analysis, exploring prompt engineering strategies, input configurations, and how these models are integrated into the program analysis process.
- This survey highlights the key challenges present in current research and proposes potential future directions to advance the development of the field.

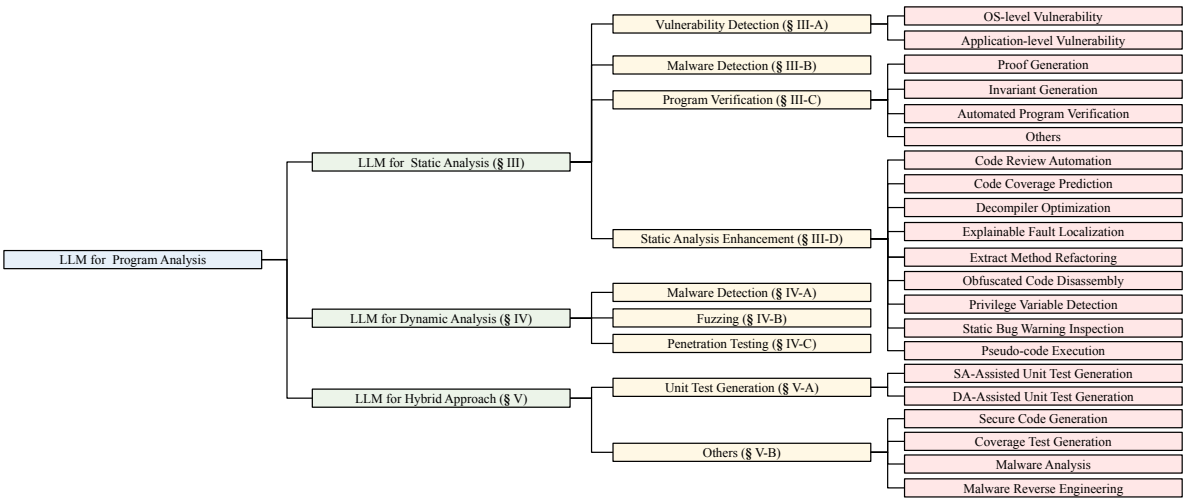

Figure 1. Taxonomy of the survey.

The survey is organized as follows. We first introduce the background of program analysis and large language model in Section 2. We then examine the application of LLMs in static analysis in Section 3 and discuss the use of LLMs in dynamic analysis in Section 4. We next explore how LLMs assist hybrid approaches that combine static and dynamic analysis in Section 5. We finally address the challenges of applying LLMs to program analysis and outline potential future research directions in Section 6 and conclude the survey in Section 7.

2. Background

In this section, we first introduce prior knowledge about program analysis (Section 2.1), including static analysis and dynamic analysis and the limitations in existing approaches, and then present the concepts of LLMs as well as the necessity of leveraging LLMs for advancing program analysis (Section 2.2).

2.1. Program Analysis

Program analysis is the process of analyzing the behavior of computer programs to learn about their properties [30]. Program analysis can find bugs or security vulnerabilities, such as null pointer dereferences or array index out-of-bounds errors. It is also used to generate software test cases, automate software patching and improve program execution speed through compiler optimization. Specifically, program analysis can be categorized into two main types: static analysis and dynamic analysis [31]. Static analysis examines a program's code without execution, dynamic analysis collects runtime information through execution, and hybrid analysis combines both approaches for comprehensive results.

Static Analysis. Static analysis (a.k.a. compile-time analysis) is a program analysis approach that identifies program properties by examining its source code without executing the program. The pipeline for static analysis consists of key stages illustrated in Figure 2. The process begins with parsing the code to extract essential structures and relationships, which are transformed into intermediate representations (IRs) such as symbol tables, abstract syntax trees (ASTs), control flow graphs (CFGs), and data flow graphs (DFGs). These IRs are then analyzed to detect issues such as unreachable code, data dependencies, and syntactic errors. These series of processes ultimately enhance code quality and reliability.

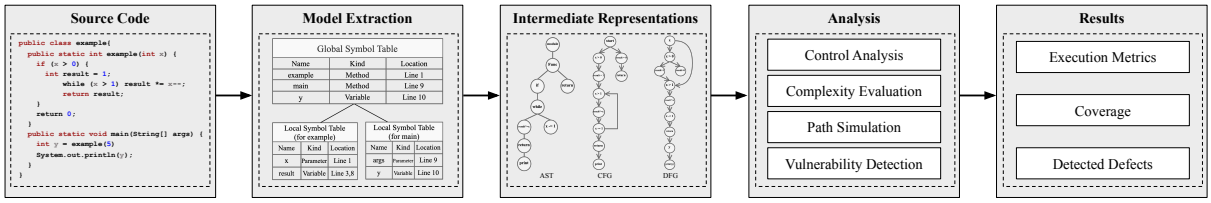

Figure 2. Static analysis workflow.

Dynamic Analysis. Dynamic analysis (a.k.a. run-time analysis) is a program analysis approach that uncovers program properties by repetitively executing programs in one or more runs [32]. The stages involved in dynamic analysis are depicted in Figure 3. These stages include instrumenting the source code to enable runtime tracking, compiling the instrumented code into a binary, and executing it with test suites. After completing the above steps, program traces such as function calls, memory accesses and system calls are captured.

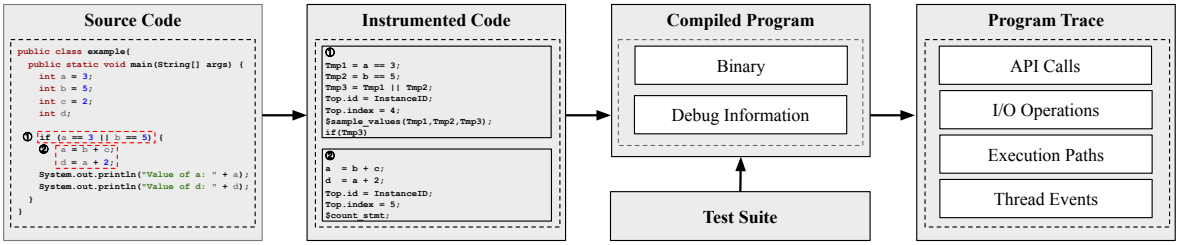

Figure 3. Dynamic analysis workflow.

2.2. Large Language Models

Large Language Models (LLMs) are large-scale neural networks built on deep learning techniques, primarily utilizing the Transformer architecture [33]. Transformer models utilize self-attention mechanism to identify relationships between elements within a sequence, which enables them to outperform other machine learning models in understanding contextual relationships. Trained on vast datasets, LLMs learn syntax, semantics, context, and relationships within language, enabling them to generate and comprehend natural language [34]. Furthermore, LLMs possess knowledge reasoning capabilities, allowing them to retrieve and synthesize information from large datasets to answer questions involving common sense and factual knowledge.

The architecture and configuration features of LLMs (e.g., model families, parameter size, and context window length) collectively determine their capabilities, performance and applicability. The studies selected in this survey involve LLM model families such as LLaMA [35], CodeLLaMA [36] and GPT [37,38]. The parameter size of a large model typically refers to the number of variables used for learning and storing knowledge. The parameter size represents a model's learning capacity, indicating its ability to capture complexity and detail from data. Generally, larger parameter sizes enhance the model's expressive power, enabling it to learn more intricate patterns and finer details. The context window refers to the range of text fragments a model uses when generating each output. It determines the amount of contextual information the model can reference during generation. Selecting appropriate architectures and configurations for LLMs in different scenarios is crucial for optimizing their performance.

3. LLM for Static Analysis

Static analysis examines various objects, such as analyzing vulnerabilities and detecting malware in source code binary executables. Analyzing vulnerabilities in source code requires techniques like dependency analysis and taint tracking to trace the flow of sensitive data. On the other hand, detecting malware focuses on control flow examination and behavior modeling to identify malicious patterns. Consequently, LLM assistance differs by program type and analysis purpose, which will be discussed in this section across four directions: (i) vulnerability detection (Section 3.1), (ii) malware detection (Section 3.2), (iii) program verification (Section 3.3), and (iv) static analysis enhancement (Section 3.4).

3.1. LLMs for Vulnerability Detection

Vulnerability detection focuses on identifying potential security risks or weaknesses in software through automated tools and techniques, which demand precise code analysis and a deep understanding of program behavior. Leveraging their advanced contextual comprehension, LLMs can analyze both semantic and syntactic patterns in source code, providing actionable suggestions and remediation strategies for addressing vulnerabilities. As a result, integrating LLMs into vulnerability detection has become a prominent application in program analysis.

To provide a clearer understanding of LLM applications in vulnerability detection, Table 2 summarizes the intermediate representations (IRs) utilized and the specific vulnerability types addressed in selected studies. Figure 4 offers a visual overview of LLM integration at various stages, highlighting their roles in contextual understanding, feature extraction, enhanced detection accuracy, and remediation strategies. These capabilities enable efficient and precise identification of OS-level and application-level vulnerabilities. Additionally, a detailed comparison of the best-performing LLMs in the reviewed studies reveals key factors influencing their effectiveness and adoption.

Table 3 presents a comprehensive summary of these models, including their model family, parameter sizes, context window sizes, and open-source availability.

Table 2. Overview of the intermediate representations (AST, CFG, DFG) employed, their application domains (OS-level or application-level vulnerabilities), their application to specific vulnerability types, and the assistance provided by LLMs across selected studies

Reference	AST	CFG	DFG	OS	App	Vulnerability Type	LLM's Assistance
LLift [39]	✗	✓	✗	✓	✗	Use-before-initialization (UBI).	Path analysis.
SLFHunter [40]	✓	✓	✓	✓	✗	Command injection vulnerabilities.	Taint sinks.
LATTE [41]	✗	✓	✓	✓	✗	Binary taint analysis for data flows.	Binary taint analysis and code slicing.
IMMI [42]	✗	✓	✓	✓	✗	Kernel memory bugs.	Memory allocation and deallocation intentions
DefectHunter [43]	✓	✓	✓	✗	✓	General vulnerability.	Code sequence embeddings.
IRIS [44]	✗	✓	✓	✗	✓	Taint analysis in smart contracts.	Taint sources and sinks.
VERACATION [45]	✓	✗	✓	✗	✓	Syntactic-based vulnerability.	Filters non-vulnerability-related statements.
Mao et al. [46]	✓	✗	✓	✗	✓	Vulnerabilities in Code review processes.	Simulates multi-role discussions.
MSIVD [47]	✗	✓	✓	✗	✓	General vulnerability.	Fine-tuned with multitask self-instructed learning.
GPTScan [48]	✓	✗	✓	✗	✓	Smart contract logic vulnerabilities.	Analyzes smart contract semantics.
Yang et al. [49]	✓	✗	✗	✗	✓	IoT software vulnerability.	Explains vulnerabilities in code.
LLbezpeky [50]	✗	✗	✗	✗	✓	Android security vulnerability.	Android application security.
SkipAnalyzer [51]	✓	✗	✓	✗	✓	Bug detection.	Identifies bugs and generates patches.
HYPERION [52]	✗	✓	✓	✗	✓	DApp Inconsistencies.	Extracts attributes of smart contract bytecode.
Zhang et al. [53]	✓	✗	✓	✗	✓	General vulnerability.	Detects vulnerabilities and fixes.
GPTLENS [54]	✓	✗	✓	✗	✓	Smart contract vulnerability.	Generates diverse vulnerability hypotheses.
LuaTaint [55]	✓	✓	✓	✗	✓	IoT vulnerability	Prunes false alarms.

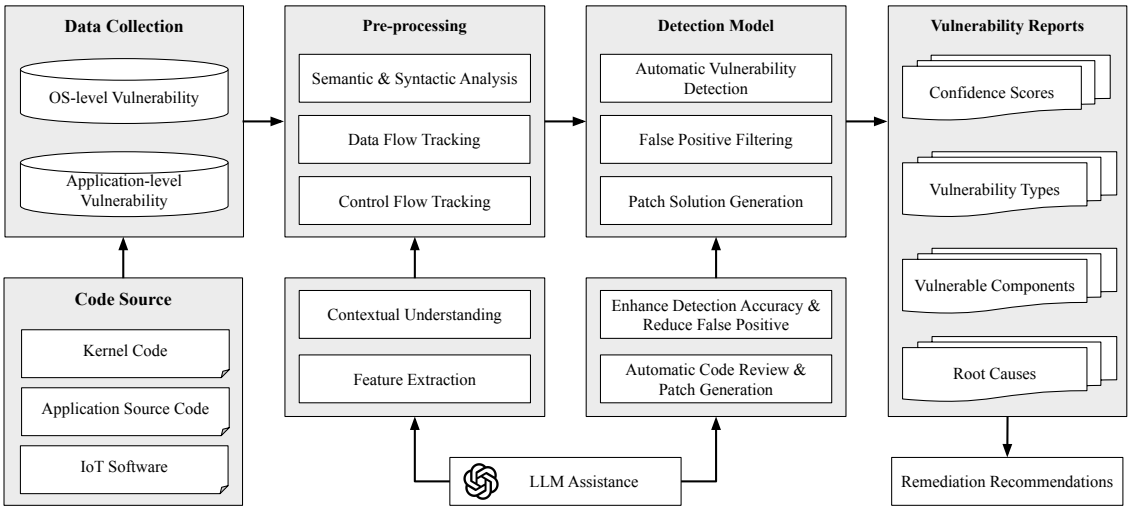

Figure 4. A diagram of LLMs' application in vulnerability detection.

OS-level Vulnerability. OS-level vulnerabilities refer to security flaws within critical components of an operating system, such as the kernel, system libraries, or device drivers. These vulnerabilities can compromise the stability and security of the entire system, allowing attackers to gain unauthorized access, disrupt operations, or cause system-wide failures affecting all running applications. Common examples include memory management errors, privilege escalation, and resource misuse. Leveraging LLMs, tools like the LLift framework [39] address challenges such as path sensitivity and scalability in detecting OS-level vulnerabilities. By combining constraint-guided path analysis with task decomposition, LLift improves the detection of issues like use-before-initialization (UBI) in large-scale codebases. Ye et al. [40] developed SLFHunter, which integrates static taint analysis with LLMs to identify command injection vulnerabilities in Linux-based embedded firmware. The LLMs are utilized to analyze custom dynamically linked library functions and enhance the capabilities of traditional analysis tools. Furthermore, Liu et al. [41] proposed a system called LATTE, which combines LLMs with binary taint analysis. The code slicing and prompt construction modules serve as the core of LATTE, where dangerous data flows are isolated for analysis. These modules reduce the complexity for LLMs by providing context-specific input, allowing improved efficiency and precision in vulnerability detection through tailored prompt sequences that guide the LLM in the analysis process. In addition, Liu et al. [42] proposed a system for detecting kernel memory bugs using a novel heuristic called Inconsistent Memory Management Intentions (IMMI). The system detects kernel memory bugs by summarizing memory operations and slicing code related to memory objects. It uses static analysis to infer inconsistencies in memory management

responsibilities between caller and callee functions. LLMs assist in interpreting complex memory management mechanisms and enable the identification of bugs such as memory leaks and use-after-free errors with improved precision.

Table 3. Overview of the best-performing LLMs used in referenced papers, their model families (MF), parameter sizes (Param), context window sizes (CW), and open-source availability.

Reference	LLM	MF	Param	CW	Open-Source
LLift [39]	GPT-4-0613	GPT-4	-	32,768	✗
SLFHunter [40]	GPT-4.0	GPT-4	-	32,768	✗
LATTE [41]	GPT-4.0	GPT-4	-	32,768	✗
IMMI [42]	ChatGPT-4-1106	GPT-4	-	32,768	✗
DefectHunter [43]	UniXcoder	-	250 M	768	✓
IRIS [44]	GPT-4.0	GPT-4	-	32,768	✗
VERACATION [45]	GPT-4.0	GPT-4	-	1024	✗
Mao et al. [46]	GPT-3.5-turbo	GPT-3.5	175 B	4096	✗
MSIVD [47]	CodeLlama-13B	CodeLlama	13 B	2048	✓
GPTScan [48]	GPT-3.5-turbo	GPT-3.5	175 B	4096	✗
Yang [49]	ChatGPT-4.0	GPT-4	-	32,768	✗
LLbezpeky [50]	GPT-4.0	GPT-4	-	32,768	✗
SkipAnalyzer [51]	ChatGPT-4.0	GPT-4	-	8192	✗
HYPERION [52]	LLaMA2	LLaMA	-	4096	✓
Zhang et al. [53]	ChatGPT-4.0	GPT-4	-	8192	✗
GPTLENS [54]	GPT-4.0	GPT-4	-	32,768	✗
LuaTaint [55]	GPT-4.0	GPT-4	-	1920	✗

Application-level Vulnerability. Application-level vulnerabilities are security weaknesses found within individual software programs. These vulnerabilities can compromise the application's performance, data integrity, or user privacy. However, they typically do not affect the overall stability of the operating system. Common examples include input validation issues, logic errors, and misconfigurations. These vulnerabilities can result in unauthorized access or data breaches, as well as application-specific security incidents.

To address the challenges in application-level vulnerability detection, Wang et al. [43] introduced the Conformer mechanism, which integrates self-attention and convolutional networks to capture both local and global feature patterns. To further refine the detection process, they optimize the attention mechanism to reduce noise in multi-head attention and improve model stability. By combining structural information processing, pre-trained models, and the Conformer mechanism in a multi-layered framework, the approach improves detection accuracy and efficiency. Building on these advancements, IRIS [44] proposes a neuro-symbolic approach that combines LLMs with static analysis to support reasoning across entire projects. The static analysis is responsible for extracting candidate sources and sinks, while the LLM infers taint specifications for specific CWE categories. Similarly, Cheng et al. [45] combined semantic-level code clone detection with LLM-based vulnerability feature extraction. By integrating program slicing techniques with the LLM's semantic understanding, they refined vulnerability feature detection. This approach addresses the limitations of traditional syntactic-based analysis.

Mao et al. [46] implemented a multi-role approach where LLMs act as different roles, such as testers and developers, simulating interactions in a real-life code review process. This strategy fosters discussions between these roles, enabling each LLM to provide distinct insights on potential vulnerabilities. MSIVD [47] introduces a multi-task self-instructed fine-tuning technique that combines vulnerability detection, explanation, and repair, improving the LLM's ability to understand and reason about code through multi-turn dialogues. Additionally, the system integrates LLMs with a data flow analysis-based GNN, which models the program's control flow graph to capture variable definitions and data propagation paths. This enables the model to rely not only on the literal information in the code but also on the program's graph structure for more precise detection. Similarly, GPTScan [48] demonstrates how GPT can be applied to code understanding and matching scenarios, reducing false positives and uncovering new vulnerabilities previously missed by human auditors.

In the domain of IoT software, Yang et al. [49] explored the application of LLMs combined with static code analysis for detecting vulnerabilities. By leveraging prompt engineering, LLMs enhance the efficiency of vulnerability detection and reduce costs, ultimately improving scalability and feasibility in large IoT systems. Meanwhile, Xiang et al. [55] proposed LuaTaint, a static analysis framework designed to detect vulnerabilities in the web configuration interfaces of IoT devices. LuaTaint integrates flow-, context-, and field-sensitive static

taint analysis with key features such as framework-specific adaptations for the LuCI web interface and pruning capabilities powered by GPT-4. By converting Lua code into ASTs and CFGs, the framework performs precise taint analysis to identify vulnerabilities like command injection and path traversal. The system uses dispatching rules and LLM-powered alarm pruning to improve detection precision, reduce false positives, and efficiently analyze firmware across large-scale datasets.

Mohajer et al. [51] presented SkipAnalyzer, a tool that employs LLMs for bug detection, false positive filtering, and patch generation. By improving the precision of existing bug detectors and automating patching, this approach significantly reduces false positives and ensures accurate bug repair.

Additionally, Zhang et al. [53] introduced tailored prompt engineering techniques with GPT-4 [38], leveraging auxiliary information such as API call sequences and data flow graphs to provide structural and sequential context. This approach also employs chain-of-thought prompting to enhance reasoning capabilities, demonstrating improved accuracy in detecting vulnerabilities across Java and C/C++ datasets.

Extending the application of LLMs in decentralized applications and smart contract analysis, Yang et al. [52] developed HYPERION, which combines LLM-based natural language analysis with symbolic execution to address inconsistencies between DApp descriptions and smart contracts. The system integrates a fine-tuned LLM to analyze front-end descriptions, while symbolic execution processes contract bytecode to recover program states, effectively identifying discrepancies that may undermine user trust.

For smart contract vulnerability detection, Hu et al. [54] introduced GPTLENS, a two-stage adversarial framework leveraging LLMs. GPTLENS assigns two synergistic roles to LLMs: an auditor generates a diverse set of vulnerabilities with associated reasoning, while a critic evaluates and ranks these vulnerabilities based on correctness, severity, and profitability. This open-ended prompting approach facilitates the identification of a broader range of vulnerabilities, including those that are uncategorized or previously unknown. Experimental results on real-world smart contracts show that GPTLENS outperforms traditional one-stage detection methods while maintaining low false positive rates. Focusing on Android security and software bug detection, Mathews et al. [50] introduced LLbezpeky, an AI-driven workflow that assists developers in detecting and rectifying vulnerabilities. Their approach analyzed Android applications, achieving over 90% success in identifying vulnerabilities in the Ghera benchmark.

Conclusion 1. Researchers use static analysis with various intermediate representations (IRs) and LLMs to detect vulnerabilities. ASTs enhance syntax-related vulnerability detection but struggle with semantic issues. CFGs handle control flow vulnerabilities but may miss certain execution paths. DFGs focus on data-flow vulnerabilities but can be computationally expensive in large systems. GPT-4 is widely adopted for its versatility but may still produce hallucinations and inconsistent results. UniXcoder performs well in specific tasks but lacks generalizability, while CodeLlama offers flexibility and reproducibility but may not match the performance of proprietary models. These approaches, despite their strengths, face challenges related to scalability, consistency, and complexity.

3.2. LLMs for Malware Detection

Malware detection determines whether a program has malicious intent and is an essential aspect of program analysis research. Initially, signature-based detection methods were predominantly used. As malware evolved, new detection techniques emerged, including behavior-based detection, heuristic detection, and model checking approaches. Data mining and machine learning algorithms soon followed, further enhancing detection capabilities [56].

Traditional malware detection methods struggle with challenges like obfuscation and polymorphic malware. LLMs offer a new approach to enhance detection accuracy and adapt to evolving threats by analyzing code semantics and patterns. Fujii et al. [57] utilized decompiled and disassembled outputs of the Babuk ransomware as inputs to the LLM to generate function descriptions through carefully designed prompts. The generated descriptions were evaluated using BLEU [58] and ROUGE [59] metrics to measure functional coverage and agreement with analysis articles.

Additionally, Simion et al.[60] evaluated the feasibility of using out-of-the-box open-source LLMs for malware detection by analyzing API call sequences extracted from binary files. The study benchmarked four open-source LLMs (Llama2-13B, Mistral[61], Mixtral, and Mixtral-FP16 [62]) using API call sequences extracted from 20,000 malware and benign files. The results showed that the models, without fine-tuning, achieved low accuracy and were unsuitable for real-time detection. These findings highlight the need for fine-tuning and integration with traditional security tools.

Analyzing malicious behaviors to detect malware is another approach. Zahan et al. [63] employed a static analysis tool named CodeQL [64] to pre-screen npm packages. This step filtered out benign files, thereby reducing the number of packages requiring further investigation. Following this step, they utilized GPT-3 and GPT-4 models to analyze the remaining JavaScript code for detecting complex or subtle malicious behaviors. The outputs from the

LLMs were refined iteratively. Accuracy improved through continuous adjustments to the model's focus based on feedback and re-evaluation.

Other studies focus on applying LLMs specifically to Android malware detection. Khan et al. [65] extracted Android APKs to obtain source code and opcode sequences, constructing call graphs to represent the structural relationships between functions. Models such as CodeBERT [66] and GPT were employed to generate semantic feature representations, which were used to annotate the nodes in the call graphs. The graphs were enriched with structural and semantic information. These enriched graphs were then processed through a graph-based neural network to detect malware in Android applications. Zhao et al. [67] first extracted features from Android APK files using static analysis, categorizing them into permission view, API view, and URL & uses-feature view. A multi-view prompt engineering approach was applied to guide the LLM in generating textual descriptions and summaries for each feature category. The generated descriptions were transformed into vector representations, which served as inputs for a deep neural network (DNN)-based classifier to determine whether the APK was malicious or benign. Finally, the LLM produced a diagnostic report summarizing the potential risks and detection results.

Conclusion 2. The integration of LLMs with static analysis techniques enables the analysis of structured input sources, including decompiled functions, API call sequences, JavaScript code files, and APK attributes. A key commonality across approaches is the reliance on LLMs to process static features and generate semantic representations, textual descriptions, or embeddings, which are subsequently used for classification or detection tasks. Additionally, we notice that both open-source LLMs (e.g., Llama2-13B and Mistral) and proprietary models (e.g., GPT-4) are widely utilized in this task.

3.3. LLMs for Program Verification

Automated program verification employs tools and algorithms to ensure that a program's behavior aligns with predefined specifications, enhancing both software reliability and security. Traditional verification methods often require substantial manual effort, particularly for writing specifications and selecting strategies. These processes are often complex and prone to errors, especially in large-scale systems. In contrast, automated verification generates key elements such as invariants, preconditions, and postconditions, using techniques like static analysis and model checking to ensure correctness. The integration of LLMs further enhances this process by enabling the automatic analysis of code features and the efficient selection of verification strategies. This reduces manual intervention and significantly accelerates verification. Consequently, automated program verification has evolved into a more efficient and reliable method for ensuring software quality. This subsection introduces diverse applications of LLMs in program verification, highlighting their role in automating and enhancing critical tasks.

Table 4 provides an overview of various studies utilizing LLMs for program verification. It summarizes their targets, methodologies, and outcomes to highlight the diverse applications of these models in automating verification tasks. The inputs in these studies can be categorized into four types: (i) Code, which includes program implementations or snippets used for analysis or synthesis. (ii) Specifications, referring to formal descriptions of program behavior, such as preconditions, postconditions, or logical formulas. (iii) Formal methods, encompassing mathematical constructs like theorems, proofs, and loop invariants for ensuring correctness. (iv) Error and debugging information, such as counterexamples, type hints, or failed code generation cases that aid in resolving programming issues.

Proof Generation. Proof generation in program verification automates the creation of formal proofs to ensure program correctness, logical consistency, and compliance with specifications. This process reduces the need for manual effort and enhances verification efficiency by streamlining complex proof tasks. Kozyrev et al. [68] developed CoqPilot, a VSCode plugin that integrates LLMs such as GPT-4, GPT-3.5, LLaMA-2 [35], and Anthropic Claude [82] with Coq-specific tools like CoqHammer [83] and Tactician [84] to automate proof generation in the Coq theorem prover. The authors implemented premise selection for better LLM prompting and created an LLM-guided mechanism that attempted fixing failing proofs with the help of the Coq's error messages. Additionally, Zhang et al. [69] developed the Selene framework to automate proof generation in software verification using LLMs. The framework is built on the industrial-level operating system microkernel [85], seL4 [86], and introduces the technique of lemma isolation to reduce verification time. Its key contributions include efficient proof validation, dependency augmentation, and showcasing the potential of LLMs in automating complex verification tasks.

Invariant Generation. Invariant generation identifies properties that remain true during program execution, providing a logical foundation for verifying correctness and analyzing complex iterative structures like loops and recursion.

Table 4. Overview referenced studies, detailing their targets, LLMs employed, parameter sizes (Param), open-source availability (OS), input types, and resulting outputs. The best-performing models are marked with *

Reference	Target	LLM	Param	OS	Input	Output
CoqPilot [68]	Proof generation	Claude	-	✗	Formal methods	Coq proofs
		LLaMA-2-13B	13 B	✓		
		GPT-3.5	-	✗		
		GPT-4*	-	✗		
Selene [69]	Proof generation	GPT-3.5-turbo	175 B	✗	Specifications	Formal proofs
		GPT-4*	-	✗		
iRank [70]	Loop invariant ranking	GPT-3.5-turbo	175 B	✗	Formal methods	Reranked LLM-generated invariants
		GPT-4*	-	✗		
Janßen et al. [71]	Loop invariant generation	GPT-3.5	175 B	✗	Specifications	Valid loop invariants
Pirzada et al. [72]	Loop invariant generation	GPT-3.5-Turbo-Instruct	175 B	✗	Formal methods	Loop invariants
LaM4Inv [73]	Loop invariant generation	LLaMA-3-8B	8 B	✓	Code	Loop invariants
		GPT-3.5-Turbo	175 B	✗		
		GPT-4-Turbo*	-	✗		
Pei et al. [74]	Invariant prediction	GPT-4	-	✗	Code	Static invariants
AutoSpec [75]	Specification synthesis	GPT-3.5-turbo-0613*	175 B	✗	Code	Specifications
		Llama-2-70B	70 B	✓		
LEMUR [76]	Automated verification	GPT-3.5-turbo	175 B	✗	Specifications	Loop invariants
		GPT-4*	-	✗		
SynVer [77]	Automated verification	GPT-4	-	✗	Specifications	Candidate C programs
PropertyGPT [78]	Smart contract verification	GPT-4-0125-preview	-	✗	Code and specifications	Formal verification properties
LLM-Sym [79]	Python symbolic execution	GPT-4o-mini	-	✗	Error and debugging	Initial Z3Py code
		GPT-4o	-	✗	Error and debugging	Refined Z3Py code
CFStra [80]	Verification strategy selection	GPT-3.5-turbo	175 B	✗	Code and specifications	Identified code features
Chapman et al. [81]	Error specification inference	GPT-4	-	✗	Formal methods	Error specifications

Some studies have explored various ways to leverage LLMs for generating and ranking loop invariants. Janßen et al. [71] investigated the utility of ChatGPT in generating loop invariants. The authors used ChatGPT to annotate 106 C programs from the SV-COMP Loops category [87] with loop invariants written in ACSL [88], evaluating the validity and usefulness of these invariants. They integrated ChatGPT with the Frama-C [89] interactive verifier and the CPAchecker [90] automatic verifier to assess how well the generated invariants enable these tools to solve verification tasks. Results showed that ChatGPT can produce valid and useful invariants for many cases, facilitating software verification by augmenting traditional methods with insights provided by LLMs. Additionally, Chakrabor et al. [70] observed that employing LLMs in a zero-shot setting to generate loop invariants often led to numerous attempts before producing correct invariants, resulting in a high number of calls to the program verifier. To mitigate this issue, they introduced iRank, a re-ranking mechanism based on contrastive learning, which effectively distinguishes correct from incorrect invariants. This method significantly reduces the verification calls required, improving efficiency in invariant generation.

Besides, Pei et al. [74] explored using LLMs to predict program invariants that were traditionally generated through dynamic analysis. By fine-tuning LLMs on a dataset of Java programs annotated with invariants from the Daikon [91] dynamic analyzer, they developed a static analysis-based method using a scratchpad approach. This technique incrementally generates invariants and achieves performance comparable to Daikon without requiring code execution. It also provides a static and cost-effective alternative to dynamic analysis.

Bounded Model Checking (BMC) is a verification technique that checks the correctness of a system within a finite number of steps. Integrating LLMs with BMC has shown potential in enhancing loop invariant generation. Pirzada et al. [72] proposed a modification to the classical BMC procedure that avoids the computationally expensive process of loop unrolling by transforming the CFG. Instead of unrolling loops, the framework replaces loop segments in the CFG with nodes that assert the invariants of the loop. These invariants are generated using LLMs and validated for correctness using a first-order theorem prover. This transformation produces loop-free program variants in a sound manner, enabling efficient verification of programs with unbounded loops. Their experimental results demonstrate that the resulting tool, ESBMC ibmc, significantly improves the capability of the industrial-strength software verifier ESBMC [92], verifying more programs compared to state-of-the-art tools such as SeaHorn [93] and VeriAbs [94], including cases these tools could not handle. Wu et al. [73] proposed LaM4Inv, a framework that integrates LLMs with BMC to improve this process. The framework employs a 'query-filter-reassemble' pipeline. LLMs generate candidate invariants, BMC filters out incorrect predicates, and valid predicates are iteratively refined and reassembled into invariants.

Automated Program Verification. Automating program specification presents challenges such as handling

programs with complex data types and code structures. To address these issues, Wen et al. [75] introduced an approach called AutoSpec. Driven by static analysis and program verification, AutoSpec uses LLMs to generate candidate specifications. Programs are decomposed into smaller components to help LLMs focus on specific sections. The generated specifications are iteratively validated to minimize error accumulation. This process enables AutoSpec to handle complex code structures, such as nested loops and pointers, making it more versatile than traditional specification synthesis techniques. Wu et al. [76] introduced the LEMUR framework. In this hybrid system, LLMs generate program properties like invariants as sub-goals, which are then verified and refined by reasoners such as CBMC [95], ESBMC [92] or UAUTOMIZER [96]. The framework is based on a sound proof system, thus ensuring correctness when LLMs propose incorrect properties. An oracle-based refinement mechanism improves these properties, enabling LEMUR to enhance efficiency in verification and handle complex programs more effectively than traditional tools. Additionally, Mukherjee et al. [77] introduced SynVer, a framework that integrates LLMs with formal verification tools for automating the synthesis and verification of C programs. SynVer takes specifications in Separation Logic, function signatures, and input-output examples as input. It leverages LLMs to generate candidate programs and uses SepAuto, a verification backend, to validate these programs against the specifications. The framework prioritizes recursive program generation, reducing the dependency on manual loop invariants and improving verification success rates.

Others. Other applications of LLMs in program verification include smart contract verification, symbolic execution, strategy selection and error specification inference. For instance, Liu et al. [78] developed a novel framework named PropertyGPT, leveraging GPT-4 to automate the generation of formal properties such as invariants, pre-/post-conditions, and rules for smart contract verification. The framework embeds human-written properties into a vector database and retrieves reference properties for customized property generation, ensuring their compilation, appropriateness, and runtime verifiability through iterative feedback and ranking. Similarly, Wang et al. [79] introduced an iterative framework named LLM-Sym. This tool leverages LLMs to bridge the gap between program constraints and SMT solvers. The process begins by extracting control flow paths, performing type inference, and iteratively generating Z3 [97] code to solve path constraints. A notable feature of LLM-Sym is its self-refinement mechanism, which utilizes error messages to debug and enhance the generated Z3 code. If the code generation process fails, the system directly employs LLMs to solve the constraints. Once constraints are resolved, Python test cases are automatically generated from Z3's outputs.

Another approach [80] automates the selection of verification strategies to overcome limitations of traditional tools like CPAchecker [90]. These tools often require users to manually select strategies, making the process more complex and time-consuming. LLMs analyze code features to identify suitable strategies, streamlining the verification process and minimizing user input. This automation not only improves efficiency but also minimizes reliance on expert knowledge. Additionally, Chapman et al. [81] proposed a method that combines static analysis with LLM prompting to infer error specifications in C programs. Their system queries the LLM when static analysis encounters incomplete information, enhancing the accuracy of error specification inference. This approach is effective for third-party functions and complex error-handling paths.

Conclusion 3. The applications of LLMs in program verification span various tasks, including proof generation, specification synthesis, loop invariant generation, and strategy selection. These methods streamline the verification process by automating the generation of properties, invariants, and other critical components essential for program analysis. Despite their diverse applications, these methods share a common goal: reducing reliance on expert knowledge and improving verification efficiency. A key aspect of achieving this goal is the iterative refinement of LLM-generated outputs. This refinement process often incorporates static analysis or hybrid frameworks that integrate formal verification tools, further enhancing reliability.

3.4. LLMs for Static Analysis Enhancement

Beyond the previously mentioned applications of LLMs, other studies focus on leveraging LLMs to assist in certain processes of static analysis.

Code Review Automation. Lu et al. [98] proposed LLaMA-Reviewer, a model that leverages LLMs to automate code review. It incorporates instruction-tuning of a pre-trained model and employs Parameter-Efficient Fine-Tuning techniques to minimize resource requirements. The system automates essential code review tasks, including predicting review necessity, generating comments, and refining code.

Code Coverage Prediction. Dhulipala et al. [99] introduced CodePilot, a system that integrates planning strategies and LLMs to predict code coverage by analyzing program control flow. CodePilot first generates a plan by analyzing program semantics, dividing the code into steps derived from control flow structures, such as loops and branches. Subsequently, CodePilot adopts either a single-prompt approach (Plan+Predict in one step) or a

two-prompt approach (planning first, followed by coverage prediction). These approaches guide LLMs to predict which parts of the code are likely to be executed based on the formulated plan.

Decompiler Optimization. Hu et al. [100] proposed DeGPT, a framework designed to enhance the clarity and usability of decompiler outputs for reverse engineering tasks. DeGPT begins by analyzing the raw output of decompilers, identifying issues such as ambiguous variable names, missing comments, and poorly structured code. The framework leverages LLMs in three distinct roles:Referee, Advisor, and Operator to propose and implement optimizations while preserving semantic correctness.

Explainable Fault Localization. Yan et al. [101] proposed CrashTracker, a hybrid framework that combines static analysis with LLMs. This approach improves the accuracy and explainability of crashing fault localization in framework-based applications. CrashTracker introduces Exception-Thrown Summaries (ETS) to represent fault-inducing elements in the framework. It also uses Candidate Information Summaries (CIS) to extract relevant contextual information for identifying buggy methods. ETS models are employed to identify potential buggy methods. LLMs then generate natural language fault reports based on CIS data, enhancing the clarity of fault explanations. CrashTracker demonstrates state-of-the-art performance in precision and explainability when applied to Android applications.

Extract Method Refactoring. Pomian et al. [102] introduced EM-Assist, a tool that combines LLMs and static analysis to enhance Extract Method (EM) refactoring in Java and Kotlin projects. EM-Assist uses LLMs to generate EM refactoring suggestions and applies static analysis to discard irrelevant or impractical options. To improve the quality of suggestions, the tool employs program slicing and ranking mechanisms to prioritize refactorings aligned with developer preferences. EM-Assist automates the entire refactoring process by leveraging the IntelliJ IDEA platform to safely implement changes.

Obfuscated Code Disassembly. Rong et al. [103] introduced DISASLLM, a framework that combines traditional disassembly techniques with LLMs. The LLM component validates disassembly results and repairs errors in obfuscated binaries, enhancing the quality of the output. Through batch processing and GPU parallelization, DISASLLM achieves substantial improvements in both the accuracy and speed of decoding obfuscated code, outperforming state-of-the-art methods

Privilege Variable Detection. Wang et al. [104] presented a hybrid workflow that combines LLMs with static analysis to detect user privilege-related variables in programs. The program is first analyzed to identify relevant variables and their data flows, which provides an initial set of potential user privilege-related variables. The LLM is used to evaluate these variables by understanding their context and scoring them based on their relationship to user privileges.

Static Bug Warning Inspection. Wen et al. [105] proposed LLM4SA, a framework that integrates LLMs with static analysis tools to automatically inspect large volumes of static bug warnings. LLM4SA first extracts bug-relevant code snippets using program dependence traversal. It then formulates customized prompts with techniques such as Chain-of-Thought reasoning and few-shot learning. To ensure precision, the framework applies pre- and post-processing steps to validate the results. This approach tackles challenges like token limitations by optimizing input size, reduces inconsistencies in LLM responses through structured prompt engineering, and mitigates false positives via comprehensive validation.

Static Analysis Alert Adjudication. Flynn et al. [106] proposed using LLMs to automatically adjudicate static analysis alerts. The system generates prompts with relevant code and alert details, enabling the LLM to classify alerts as true or false positives. To address context window limitations, the system summarizes relevant code and provides mechanisms for the LLM to request additional details or verify its classifications.

Static Analysis Enhancement by Pseudo-code Execution. Hao et al. [107] presented E&V, a system designed to enhance static analysis using LLMs by simulating the execution of pseudo-code and verifying the results without needing external validation. It validates the results of the analysis through an automatic verification process that checks for errors and inconsistencies in the pseudo-code execution. This system is particularly useful for tasks like crash triaging and backward taint analysis in large codebases like the Linux kernel.

Conclusion 4., These methods illustrate how LLMs improve static analysis in areas like debugging, fault localization, refactoring, and privilege detection, but they also face several challenges. For example, LLMs are heavily reliant on prompt engineering, and poorly designed prompts can result in false positives or missed vulnerabilities. They also struggle with scalability, especially when processing large or complex codebases, as shown by CodePilot, where computational overhead and token limits can hinder performance. Additionally, LLMs have limited context understanding, particularly in analyzing complex logic vulnerabilities, as evidenced by DISASLLM, which, while improving disassembly accuracy, still struggles with intricate control flows.

4. LLMs for Dynamic Analysis

Static analysis primarily focuses on detecting structural and logical flaws in code by analyzing the source code to identify potential issues. In contrast, dynamic analysis emphasizes the program's behavior during execution, including performance profiling and testing. Profiling focuses on understanding program performance by analyzing execution, such as counting statement or procedure executions through instrumentation. Testing aims to make sure the test suites can cover a program. Statement coverage verifies that every statement in the code is executed at least once during testing. Branch, condition, and path coverage evaluate how thoroughly all branches, conditions, and execution paths are tested [32]. This section examines how LLMs enhance dynamic analysis, focusing on (i) malware detection (Section 4.1) under profiling, (ii) fuzzing (Section 4.2) and (iii) penetration testing (Section 4.3) under testing.

4.1. LLMs for Malware Detection

As discussed in Section 3.2, the definition of malware detection is provided. This subsection focuses on using LLMs to analyze runtime data for malware detection. The distinction between static and dynamic analysis depends primarily on the input source. For instance, if API call sequences are captured during program runtime, such as through sandboxes, debuggers, or runtime analysis frameworks, they are classified as dynamic analysis. Conversely, API call sequences extracted through methods like decompilation or disassembly from static files are classified as static analysis. Table 5 provides an overview of LLMs in both static and dynamic approaches and their testing accuracy.

Yan et al. [108] proposed a dynamic malware detection method that utilizes GPT-4 to generate text representations for API calls, which are an essential feature in dynamic malware analysis. Their method incorporates the innovative use of prompt engineering, allowing GPT-4 to generate highly detailed, context-rich descriptions for each API call in a sequence. These descriptions go beyond simple API names and delve into the specifics of how each API call behaves within the context of the malware's execution. This provides a much deeper understanding of the malware's actions, as opposed to traditional approaches that primarily rely on raw, unprocessed sequences of API calls. After generating these descriptions, the next step in the pipeline involves using BERT to convert the textual descriptions into embeddings. These embeddings encapsulate the semantic information of the API calls and their interactions, thereby forming a high-quality representation of the entire API sequence. These representations are then passed through a CNN, which performs feature extraction and classification. This comprehensive approach addresses several major challenges faced by traditional API-based models.

Table 5. Overview of the LLMs used in referenced papers, their target malware, input sources, type of analysis, parameter sizes (Param), context window sizes (CW), open-source availability (OS), and testing accuracy.

Reference	Target Malware	Input Source	Type	LLM	Param	CW	OS	Accuracy
Fujii et al. [57]	Babuk ransomware	Decompiled/disassembled functions	Static	ChatGPT-4.0	-	8192	✗	90.90%
Simion et al. [60]	General malicious files	API call sequences	Static	Llama2-13B	13 B	4096	✓	50%
				Mistral	7.3 B	8192	✓	51%
				Mixtral	7∼13 B	4096	✓	67%
				Mixtral-FP16	7∼13 B	4096	✓	72%
Zahan et al. [63]	Malicious packages	JavaScript code files	Static	GPT-3.5-turbo-1106	175 B	4096	✗	91%
				GPT-4-1106-preview	-	8192	✗	99%
Khan et al. [65]	Android malware	APK files	Static	CodeBERT	125 M	512	✓	95.29%
				GPT-2	1.5 B	1024	✓	94.89%
				RoBERTa	125 M	512	✓	94.94%
Zhao et al. [67]	Android malware	APK files	Static	GPT-4-1106-preview	-	8192	✗	97.15%
Yan et al. [108]	General malware	API call sequences	Dynamic	BERT	110 M	512	✓	95.61%
				GPT-4	-	8192	✗	
Sun et al. [109]	Linux-based malware	System call traces	Dynamic	ChatGPT-3.5	175 B	4096	✗	-
Sanchez et al. [110]	IoT malware	System call traces	Dynamic	BERT	110 M	512	✓	67.72%
				DistilBERT	66 M	512	✓	63%
				GPT-2	1.5 B	1024	✓	69%
				BigBird	110 M	4096	✓	87%
				Longformer	150 M	4096	✓	86%
				Mistral	7.3 B	8192	✓	58%
Li et al. [111]	Android malware	Code features and system calls	Hybrid	ChatGPT	-	-	✗	-

Similarly, Sun et al. [109] developed a framework that uses dynamic analysis and LLMs to generate detailed cyber threat intelligence (CTI) reports. The framework captures syscall execution traces of malware and converts them into natural language descriptions using a Linux syscall transformer. These descriptions are organized into an Attack Scenario Graph (ASG) to preserve essential details and reduce redundancy. Sanchez et al. [110] applied pre-trained LLMs with transfer learning for malware detection. They fine-tuned the models with a classification

layer on a dataset of benign and malicious system calls. This approach allows the model to distinguish between normal and malicious behavior while avoiding the need for training from scratch by leveraging pre-trained LLMs.

Conclusion 5. Dynamic malware detection with LLMs analyzes runtime behaviors such as API and system call traces to improve accuracy and interpretability. Larger models like GPT-4 enhance adaptability to unseen patterns, providing in-depth context for each API call, but they come with high computational costs and slower processing speeds, which may limit their applicability in real-time detection. Smaller models like BERT are efficient for real-time tasks, but they may struggle with complex patterns and lack the depth of understanding provided by larger models.

4.2. LLMs for Fuzzing

Fuzzing is a technique for automated software testing that inputs randomized data into a program to detect vulnerabilities like crashes, assertion failures, or undefined behaviors. The classifications of fuzzing is shown in Figure 5. Fuzzing approaches are categorized by three dimensions: test case generation, input structure, and program structure. Test case generation can be mutation-based which alters existing inputs, or generation-based which creates new inputs from scratch. Input structure distinguishes smart fuzzing which utilizes input format knowledge, from dumb fuzzing which generates inputs blindly. Program structure analysis classifies fuzzing as black-box, grey-box, or white-box, based on the tester's level of program insight.

The use of LLMs for fuzzing is summarized in Table 6, which highlights the strategies, program structures, LLMs employed, and applications in the studies. Most research utilizing LLMs for fuzzing focuses on greybox fuzzing.

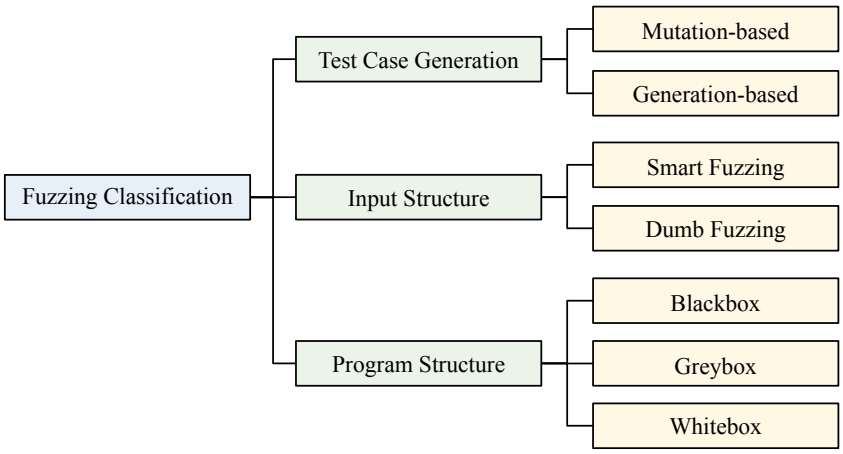

Figure 5. Fuzzing classifications.

Table 6. Overview of the LLM-based fuzzers used in referenced papers, including their target software, test case generation (TCG), program structure (PS), model parameters, open-source availability (OS), and usage details. The best-performing model is marked with *

Reference	Target	TCG	PS	LLM	Param	OS	LLMs Usage
CHEMFuzz [112]	Quantum chemistry software	Mutation	Greybox	GPT-3.5 Claude-2* [113] Bard [114]	175 B - -	✗ ✗ ✗	Input file mutation and output analysis
CovRL [115]	JavaScript Engines	Mutation	Greybox	CodeT5+	220 M	✓	Generates valid test cases
LLAMAFUZZ [116]	Real-world programs	Mutation	Greybox	llama-2-7b-chat-hf	7 B	✓	Mutate structured data inputs and generate new seeds
FuzzGPT [117]	Deep Learning Libraries	Mutation	Greybox	Codex (code-davinci-002) CodeGen (350M/2B/6B-mono)	- 350 M/2 B/6 B	✗ ✓	Mutatie and refine test cases Generates initial test cases
CHATFUZZ [118]	General programs	Mutation	Greybox	GPT-3.5-turbo	175 B	✗	Generates format-conforming variations of existing seeds
CODAMOSA [119]	Python modules	Mutation	Greybox	Codex	-	✗	Generates tailored inputs and extends callable sets
CHATAFL [120]	Network protocol implementations	Mutation	Greybox	GPT-3.5-turbo	175 B	✗	Extracts grammars and enrich seed corpora
Asmita et al. [121]	BusyBox	Mutation	Greybox, blackbox	GPT-4-0613	-	✗	Generate seeds
Fuzz4All [122]	Compilers, SMT solvers, quantum frameworks and programming toolchains.	Mutation, generation	Greybox	GPT-4.0	-	✗	Generates fuzzing inputs

Qiu et al. [112] introduced CHEMFuzz, an LLM-assisted fuzzing framework designed for quantum chemistry software. CHEMFuzz uses an evolutionary fuzzing approach with LLM-based input mutation and output analysis to address the syntactic and semantic complexities of quantum chemistry software. The two-module system combining syntactic mutation operators with anomaly detection detected 40 bugs and 81 potential warnings in Siesta 4.1.5 [123]. Eom et al. [115] introduced CovRL, a framework that integrates coverage-guided reinforcement learning with LLMs to enhance fuzzing for JavaScript engines. The approach combines Term Frequency-Inverse Document Frequency (TF-IDF) [124] weighted coverage maps with reinforcement learning to guide the LLM-based mutator. TF-IDF is a statistical measure used to evaluate the importance of a word in a document relative to a collection of documents, where the term frequency (TF) reflects how often a term appears in a document, and the inverse document frequency (IDF) measures how unique or rare the term is across the entire dataset. This enables the generation of more effective test cases, discovering new coverage areas and improving the efficiency of JavaScript engine fuzzing.

In addition, Deng et al. [117] introduced FuzzGPT, a framework for fuzzing deep learning libraries. By mining historical bug-triggering programs and leveraging LLMs such as Codex[125] and CodeGen [126], FuzzGPT generates edge-case inputs using strategies like few-shot, zero-shot, and fine-tuned learning. This targeted approach exploits API-specific vulnerabilities, illustrating the effectiveness of LLMs in managing complex software ecosystems. Meng et al. [120] introduced CHATAFL, an LLM-guided mutation-based framework for protocol fuzzing. The framework extracts protocol grammars, enhances seed diversity, and transitions to unexplored protocol states. This approach overcomes challenges like reliance on initial seeds and restricted state-space exploration.

Beyond domain-specific applications, frameworks like LLAMAFUZZ [116] and CHATFUZZ [118] showcase the adaptability of LLMs for general program fuzzing. Zhang et al. proposed LLAMAFUZZ, which combines greybox fuzzing with LLM-based mutation to enhance branch coverage and bug detection. Its focus on structured data inputs makes it an effective tool for augmenting traditional fuzzing methods, demonstrating improvements over AFL++ [127]. Similarly, Hu et al. introduced CHATFUZZ [118], leveraging ChatGPT to generate format-conforming test cases for highly structured inputs, addressing the efficiency limitations of traditional mutation- and grammar-based fuzzers. These frameworks demonstrate the ability of LLMs to adapt to structured program requirements while advancing fuzzing efficiency.

Lemieux et al. [119] introduced CODAMOSA, an approach that integrates LLMs into testing workflows. Search-Based Software Testing (SBST) [128] is a technique that uses search algorithms to automatically generate test cases for software applications. CODAMOSA combines SBST with Codex to generate test cases and address coverage stagnation. It integrates LLM-generated Python code into SBST workflows, highlighting the collaboration between traditional testing and LLM-driven techniques. Asmita et al. [121] explored LLM-based fuzzing in BusyBox [129], a widely used Linux utility suite. Their approach combines LLM-assisted seed generation with crash reuse to enhance efficiency in black-box fuzzing workflows. Using GPT-4, they demonstrated how LLMs handle complex inputs and reuse crashes for cross-variant testing, improving vulnerability detection.

Additionally, Xia et al. [122] proposed Fuzz4All, a universal fuzzing framework that extends fuzzing beyond language- or system-specific constraints. Autoprompting is a technique that automatically generates and refines prompts to effectively guide a model in producing the desired outputs or test cases. In this context, Fuzz4All uses autoprompting and an iterative fuzzing loop to transform user-provided inputs into prompts for generating diverse test cases.

Conclusion 6. LLM-based fuzzing frameworks have advanced automated testing by combining mutation-based and generation-based strategies with models like GPT-3.5, Codex, and CodeGen. As shown in Table 6, these tools share common goals, such as improving test coverage, addressing domain-specific challenges, and automating seed generation and refinement.

4.3. LLMs for Penetration Testing

Penetration testing is a controlled security assessment that simulates real-world attacks to identify, evaluate, and mitigate vulnerabilities in systems and networks[130].

Deng et al. [131] explored the capabilities of LLMs in penetration testing, revealing that while these models excel at sub-tasks, they face challenges in maintaining context across multi-step workflows. To address this limitation, the authors proposed PentestGPT, a framework integrating reasoning, generation, and parsing modules. This framework significantly improved task completion rates by 228.6% compared to GPT-3.5 and demonstrated effective performance in real-world scenarios. Huang et al. [132] developed PenHeal, an LLM-based framework combining penetration testing and remediation. PenHeal includes a Pentest Module that uses techniques like counterfactual prompting to autonomously detect vulnerabilities. Its remediation module offers tailored strategies based on severity and cost efficiency. Compared to PentestGPT, PenHeal increased detection coverage by 31%,

improved remediation effectiveness by 32%, and reduced costs by 46%. Additionally, Goyal et al. [133] proposed Pentest Copilot, a framework that uses GPT-4-turbo to enhance penetration testing workflows. Pentest Copilot incorporates chain-of-thought reasoning and retrieval-augmented generation to automate tool orchestration and exploit exploration. It ensures adaptability with a web-based interface. This approach combines automation with expert oversight, enhancing the accessibility of penetration testing while preserving technical depth.

Additionally, some frameworks are designed as agent-based systems. Bianou et al. [134] presented PENTEST-AI, a framework guided by the MITRE ATT&CK [135] framework for multi-agent penetration testing. The framework automates reconnaissance, exploitation, and reporting tasks using specialized LLM agents. PENTEST-AI reduces human intervention while aligning with established cybersecurity methodologies, illustrating the synergy between LLMs and structured security frameworks in addressing real-world challenges. Muzsai et al. [136] proposed HackSynth, an LLM-driven penetration testing agent with two modules: a Planner for generating commands and a Summarizer for processing feedback. Tested on newly developed CTF-based benchmarks, HackSynth demonstrated its capability to autonomously exploit vulnerabilities and achieve optimal performance with GPT-4. Gioacchini et al. [137] developed AutoPenBench, a framework with 33 tasks covering experimental and real-world penetration testing scenarios. AutoPenBench compares autonomous and semi-autonomous agents, tackling reproducibility challenges in penetration testing research. Fully autonomous agents achieved a 21% success rate, significantly lower than the 64% success rate of semi-autonomous setups. Shen et al. [138] introduced PentestAgent, leveraging LLMs and Retrieval-Augmented Generation (RAG) to automate intelligence gathering, vulnerability analysis, and exploitation. PentestAgent dynamically integrates tools and adapts to diverse environments, improving task completion and operational efficiency. It outperforms existing LLM-based penetration testing systems.

As illustrated in Figure 6, penetration testing involves six stages: pre-engagement interactions, reconnaissance, vulnerability identification, exploitation, post-exploitation, and reporting. Pre-engagement interactions establish objectives, define scope, and set rules of engagement. Reconnaissance gathers target information through passive and active methods to identify attack vectors. In the reconnaissance stage, tools like PentestGPT [131] interpret tool outputs and generate actionable steps. Vulnerability identification employs both automated tools and manual techniques to detect weaknesses. For example, PenHeal [132] enhances vulnerability detection through Counterfactual Prompting, while HackSynth [136] identifies vulnerabilities via iterative commands. Exploitation uses these vulnerabilities to demonstrate potential risks, with tools like HackSynth [136] automating exploitation processes and Pentest Copilot [133] generating and optimizing exploitation scripts. Post-exploitation assesses the breach's impact and ensures persistence if needed. In post-exploitation, PentestGPT [131] assists with lateral movement and multi-step tasks, and PentestAgent [138] supports attack path analysis and persistence strategies. Finally, reporting consolidates findings into structured documentation, where PenHeal [132] provides remediation strategies, and Pentest Copilot [133] and PentestAgent [138] automate and optimize the generation of comprehensive reports.

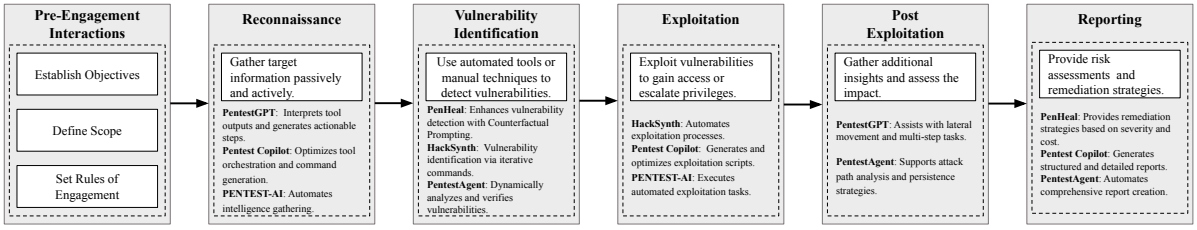

Figure 6. Integration of LLMs across the six steps of penetration testing.

Conclusion 7. LLMs can be applied across multiple stages of penetration testing. For example, LLM-driven frameworks simplify reconnaissance by automating tool output interpretation and intelligence gathering. They improve vulnerability identification through dynamic analysis methods, including counterfactual prompting. Additionally, LLMs assist in post-exploitation by facilitating multi-step attack strategies.

5. LLMs for Hybrid Approach

A hybrid approach employs both static and dynamic analysis techniques at different stages. For example, combining static features like code structure or permissions with dynamic behaviors such as system calls or memory usage represents a hybrid approach. This section discusses the role of LLMs in hybrid approaches, focusing on two aspects: (i) LLMs for unit test generation (Section 5.1) and (ii) other hybrid methods (Section 5.2).

5.1. LLMs for Unit Test Generation

Unit testing is a fundamental practice in software development that focuses on verifying the functionality of individual components or ”units” of a program. By isolating and testing each unit, developers can ensure code correctness, detect errors early, and improve overall code quality. Traditional unit test generation methods are written manually by developers and generally involve search-based, constraint-based, or random techniques to maximize code coverage [139]. Automated unit test generation leverages tools and techniques to generate tests automatically, reducing developer workload and improving coverage. Figure 7 illustrates an end-to-end workflow for LLM-powered unit test generation, beginning with three key inputs (source code, existing tests, and coverage reports) that feed into an automated generation module where prompt-engineered LLMs produce new test suites. These generated tests undergo execution and validation, with passing cases contributing to improved coverage before being incorporated back into the test files through post-processing, creating a closed-loop system that continuously enhances test quality through iterative refinement. The streamlined process effectively bridges traditional testing artifacts with modern LLM capabilities in a single integrated pipeline. In this LLM-powered test generation workflow, static analysis is essential in guiding test generation by examining the program's structure, dependencies, and control flow. Dynamic analysis complements this by evaluating the generated tests through runtime execution, identifying errors, and refining test quality. Together, these hybrid approaches enhance the efficiency and effectiveness of unit test generation.

Performance Comparison Between LLMs and Traditional Test Generation Tools. A study evaluated the performance of ChatGPT and Pynguin [140] in generating unit tests for Python programs, focusing on three types of code structures: procedural scripts, function-based modular code, and class-based modular code. Bhatia et al. [139] compared the tools in terms of coverage, correctness, and iterative improvement through prompt engineering. They found that ChatGPT and Pynguin achieved comparable statement and branch coverage. Iterative prompting improved ChatGPT's coverage for function- and class-based code, saturating after four iterations, but showed no improvement for procedural scripts. The study also revealed minimal overlap in missed statements, suggesting combining the tools could enhance coverage. However, ChatGPT often generated incorrect assertions, especially for less structured code, due to its focus on natural language over code semantics. The authors concluded that while LLMs like ChatGPT are promising for unit test generation, integrating semantic understanding and combining them with traditional tools could address current limitations and improve performance.

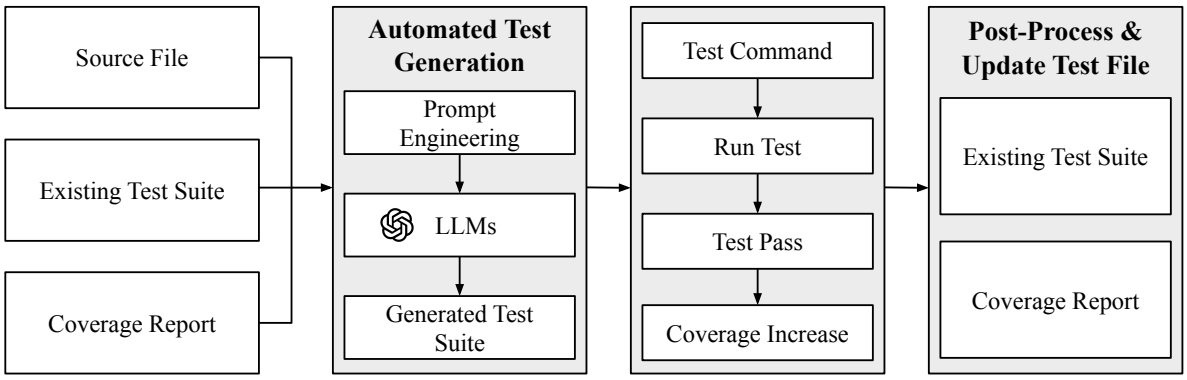

Figure 7. Workflow of unit test generation with LLMs.

Static Analysis-Assited Unit Test Generation. One improvement is the ability of LLMs to generate focused and meaningful test cases by using static analysis to extract and structure relevant context. For instance, aster [141] and ChatUniTest [142] integrate techniques such as dependency extraction, program slicing, and adaptive focal context. These methods ensure that prompts sent to LLMs are concise and focused, enabling the generation of tests that better align with the target methods. Similarly, APT [143] employs a property-based approach to guide LLMs in generating tests using the ”Given-When-Then” paradigm, which improves logical structure in generated tests. These static analysis techniques address the limitations of traditional methods, which often struggle to extract relevant dependencies or fail to focus on critical components of the code.

Dynamic Analysis-Assited Unit Test Generation. Dynamic analysis complements static techniques by validating and refining test cases through iterative processes, improving coverage and correctness. For example, TestART [144] uses a co-evolutionary framework to iteratively generate and repair tests based on runtime feedback, addressing flaky or invalid tests often produced by traditional methods. In ChatUniTest [142], dynamic validation integrates runtime error detection with rule-based and LLM-driven repair, ensuring that generated tests are compil-

able and logically sound. Furthermore, ChatTester [145] demonstrates how iterative prompting based on dynamic feedback can address missed statements and branches, progressively improving line and branch coverage. These dynamic techniques allow LLM-based approaches to adapt and refine tests, addressing limitations of traditional static tools that lack iterative capabilities.

Prompt Engineering. Techniques like adaptive focal context in ChatUniTest [142] and program slicing in HITS [146] streamline prompts by reducing irrelevant information, ensuring the LLM remain focused. Chain-of-thought reasoning, as seen in aster [141], enhanced the LLM's ability to handle complex dependencies and generated logically coherent tests. Additionally, AGONETEST [147] employed structured prompts incorporating mock dependencies and example inputs, guiding the LLM to generate more comprehensive test cases. These techniques address the inflexibility of traditional tools, which often rely on predefined templates and lack the ability to dynamically adapt prompts based on code context.

Conclusion 8. Static and dynamic analysis each play important roles in unit test generation but face challenges. Static analysis, while effective for extracting dependencies, struggles to identify key components in large-scale systems, leading to incomplete tests. Dynamic analysis, although useful for refining tests, faces issues with flaky tests and inconsistent runtime feedback, as seen in TestART and ChatUniTest. Additionally, prompt engineering, which is crucial for generating targeted tests, must adapt to code context, but techniques like Chain-of-thought reasoning in aster and structured prompts in AGONETEST still face difficulties in dynamically adjusting to the evolving code. These challenges highlight the difficulties of integrating static and dynamic approaches effectively.

5.2. Others

In addition to the previously discussed methods for unit test generation, other hybrid approaches integrate static and dynamic analysis through an agent framework. This framework first performs static analysis, such as extracting ASTs and analyzing code structure, and then conducts dynamic testing.

Multi-Agent Framework for Secure Code Generation. Nunez et al. [148] introduced AutoSafeCoder, an innovative multi-agent framework designed to improve the security of automatically generated code. The framework leverages three distinct LLM-driven agents working collaboratively to generate, analyze, and secure code. The Coding Agent is responsible for generating the initial code, while the Static Analyzer Agent identifies potential vulnerabilities through AST analysis. Meanwhile, the Fuzzing Agent detects runtime errors by employing mutation-based fuzzing techniques, ensuring that the generated code performs securely during execution. Interactive feedback loops integrate both static and dynamic testing methods into the code generation process, optimizing the outputs from the LLM at each stage.

Coverage Test Generation. Pizzorno et al. [149] presented CoverUp, a method for generating Python regression tests with high code coverage. CoverUp evaluates existing code coverage, identifies gaps, and uses LLMs to generate new tests informed by static analysis. If tests fail to execute or enhance coverage, CoverUp iteratively refines them using error messages and code context. This process continues until all segments are fully tested and integration issues are resolved.

Malware Analysis. Li et al. [111] used reverse engineering tools to extract static and dynamic features from Android APK files, organizing them into permissions, system calls, and metadata. They used tailored prompts to guide ChatGPT in generating textual analyses and maliciousness scores for each application. These results were compared with three existing Android malware detection models: Drebin [150], MaMaDroid [151], and XMAL [152]. Although traditional models showed strong classification capabilities, the authors noted their limitations in interpretability and dataset dependency. ChatGPT offered comprehensive analyses and explanations but lacked decision-making capabilities.

Malware Reverse Engineering. Williamson et al. [153] integrated LLMs with static and dynamic analysis techniques to enhance malware reverse engineering. In the static phase, tools like IDA Pro examined binaries to extract structural details such as embedded strings and control flow. In the dynamic phase, sandboxes monitored malware behavior, capturing network and system interactions. LLMs synthesized results from both phases, deriving actionable insights and identifying indicators of compromise (IoCs).

Conclusion 9. LLMs enhance hybrid methods by iteratively refining the outputs of static (e.g., AST analysis in AutoSafeCoder, coverage gaps in CoverUp) and dynamic (e.g., fuzzing, runtime feedback) analysis process. They bridge code structures with runtime behaviors, enabling secure code generation, high-coverage tests, and actionable malware analysis.

6. Discussion

The use of LLMs in the field of program analysis has mitigated several previous limitations such as false positives, performance overhead, inherent knowledge barriers, path explosion, the trade-off between speed and accuracy, and the difficulty of achieving automation across diverse systems without heavy manual intervention. Despite these advancements, new limitations and challenges have emerged with the introduction of LLMs. The following subsections provide an overview of these challenges (Section 6.1) and discuss potential future research directions (Section 6.2).

6.1. Challenges

While the reviewed methods demonstrate significant advancements in program analysis, each comes with its own strengths and limitations. On the positive side, many methods, such as LLM-based approaches, significantly enhance automation and detection accuracy by overcoming issues related to false positives and performance overhead. However, they also introduce new challenges, such as inconsistencies in long-context processing and the issue of hallucinations, where LLMs generate fabricated information. Additionally, the dependency on prompt engineering for LLMs limits their scalability, particularly for large programs, and requires expert intervention to ensure reliable results. Furthermore, the lack of high-quality, consistent labeled data and the inherent complexity of integrating LLMs with other analysis techniques remain significant barriers. Many studies have acknowledged these limitations, but they are yet to be fully addressed. The following sections will discuss these issues in detail.

Technical Limitations. LLMs face several unresolved technical challenges in program analysis. One key issue is incorrect data type identification and information loss during decompilation, which significantly reduce the accuracy of the analysis. Additionally, LLMs often oversimplify patches, limiting their ability to address complex vulnerabilities in real-world applications. In some cases, they even produce empty responses, particularly during software verification and patching tasks. Another unresolved issue is variable reuse; LLMs often confuse identically named variables in different scopes, leading to inaccurate analysis. Moreover, LLMs struggle with analyzing logic vulnerabilities, especially in scenarios with intricate control flows, complex nesting, and time-based competition conditions. These technical limitations reduce the effectiveness of LLMs in handling such scenarios and underscore the need for improvements in these areas.

Model Reliability and Consistency Issues. LLMs are inherently non-deterministic, meaning they may produce varying outputs for identical inputs. This inconsistency complicates repeated vulnerability assessments, hindering the reliability and repeatability of results. The hallucination problem, where LLMs generate fabricated or misleading information, is another significant issue, especially in the context of vulnerability detection. These hallucinations can mislead the analysis and compromise the accuracy of results. The combination of these issues—inconsistency, hallucinations, and lack of reproducibility—makes LLMs insufficient for providing reliable and repeatable results in program analysis. Addressing these issues is crucial for improving the reliability of LLMs in security applications.

Prompt Engineering and Scalability Challenges. LLM-based program analysis heavily relies on prompt engineering, which requires significant expertise. Poorly designed prompts can lead to ineffective results, introducing biases that affect the model's ability to detect vulnerabilities accurately. Furthermore, using LLMs can be costly, particularly when analyzing long code segments, as these require a large number of tokens to process. The token limits of LLMs also restrict their ability to handle extensive or highly complex programs, posing a significant scalability challenge in real-world applications. The high computational costs and dependencies on high-quality prompts further limit the feasibility of deploying LLMs for large-scale program analysis tasks.

Data Quality and Annotation Issues. Training large language models in the security domain faces challenges like data scarcity and annotation inconsistency. High-quality labeled data, essential for vulnerability detection and attack recognition, is scarce due to the subtle nature of security issues, requiring expert manual annotation. This leads to small datasets that can't fully support large language model training. Furthermore, annotation is subjective, especially for classifying vulnerabilities or attack patterns, causing inconsistencies that affect model learning. It is advantageous that, program analysis techniques hold promise in addressing this by generating high-quality training data through automated extraction and annotation of vulnerabilities from source code and binary files. Therefore, advancements in program analysis techniques will also feed back into security large language models, providing high-quality data that enhances their performance.

6.2. Future Directions

Deep Integration of LLMs with Analysis Techniques. Most current methods use LLMs independently of program analysis. Integrating LLMs with static analysis into a unified workflow offers opportunities for enhanced

effectiveness. Some studies [39] have acknowledged that their methods lack effective integration of LLMs with other models or techniques. Frameworks combining LLMs with GNNs [47] for program control and data flow have shown significant improvements in detection accuracy. Future work should focus on integrating LLMs with static and dynamic analysis to create more effective solutions for vulnerability detection.

Transforming Dynamic Analysis into Static Analysis. Transforming tasks traditionally requiring dynamic analysis into static analysis with LLMs is an emerging direction. Tasks like runtime vulnerability detection and memory corruption analysis historically depended on dynamic analysis to capture execution-specific behaviors. LLM integration can shift these processes to static analysis, enabling early vulnerability detection without runtime execution. This reduces computational overhead, avoids repeated executions, and improves scalability for analyzing large systems. Pei et al. [74] showed how fine-tuning LLMs eliminates the need for runtime information by predicting program invariants from source code, enabling earlier safety checks during compilation.

Emulating Human Security Researchers for Vulnerability Detection. Advancing code understanding and reasoning capabilities enable LLMs to replicate systematic approaches used by human security researchers. LLMs overcome the rule-based limitations of traditional tools by analyzing complex code contexts and identifying nuanced vulnerabilities. This enables LLMs to mimic hypothesis-driven processes, identifying subtle vulnerabilities missed by automated methods. Glazunov et al. [154] introduced Project Naptime to replicate human security researchers' workflows for vulnerability detection. The framework employs tools such as a code browser, Python interpreter, and debugger, enabling LLMs to perform expert-level code analysis and vulnerability detection. Evaluated on the CyberSecEval 2 [155] benchmark, this approach improves detection and demonstrates the feasibility of automating complex security tasks.

Exploring the Impact of Advanced Models. LLM technology is evolving rapidly, with new models such as GPT-o1, GPT-o3, DeepSeek [156], and others emerging. These advancements hold the potential to significantly impact the field of program analysis. On one hand, more advanced models could improve the effectiveness of LLMs in tasks like vulnerability detection by providing better long-context understanding, enhanced accuracy, and deeper reasoning capabilities. For example, models like DeepSeek may be able to understand and analyze the underlying logic of complex code structures more effectively, enabling more accurate vulnerability identification and prediction. These models also have the potential to offer more sophisticated decision-making processes and problem-solving strategies, which could be particularly useful in handling intricate, nuanced security tasks. On the other hand, these new models might introduce new challenges, such as increased computational costs, more complex prompt engineering, and the need for extensive fine-tuning to handle specific security tasks. Future work should explore how these new models can be integrated into existing workflows, evaluate their impact on performance, and address any new limitations they might introduce, ensuring that the evolution of LLMs continues to push the boundaries of program analysis.

7. Conclusions

Integrating LLMs into program analysis enhances vulnerability detection, code comprehension, and security assessments. LLMs' natural language processing capabilities, combined with static and dynamic analysis techniques have improved automation, scalability, and interpretability in program analysis. These advancements facilitate faster vulnerability detection and provide deeper insights into software behavior. Challenges such as token limitations, path explosion, complex logic vulnerabilities, and LLM's hallucinations remain barriers. The studies reviewed in this survey highlight recent progress, offering insights into its current state and emerging opportunities. Future directions include developing domain-specific models, refining hybrid methods, and enhancing reliability and interpretability to fully utilize LLMs in program analysis. This survey aims to assist in addressing the mentioned challenges and inspire the development of more effective program analysis frameworks.

Author Contributions

J.W.: Literature research, writing, and revision. T.N.: Literature research, writing, and revision. W.-B.L.: supervision. Q.Z.: supervision. All authors have read and agreed to the published version of the manuscript.

Funding

This work was fully supported by the Research Grants Council of Hong Kong (RGC) under Grants C1029-22G and in part by the Innovation and Technology Commission of Hong Kong (ITC) under Mainland-Hong Kong Joint Funding Scheme (MHKJFS) MHP/135/23. Any opinions, findings, and conclusions in this paper are those of the authors and are not necessarily of the supported organizations.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Mens, T.; Wermelinger, M.; Ducasse, S.; et al. Challenges in software evolution. In Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE'05), Lisbon, Portugal, 5–6 September 2005; pp. 13–22.
2. Li, H.; Kwon, H.; Kwon, J.; et al. Clorifi: software vulnerability discovery using code clone verification. *Concurr. Comput. Pract. Exp.* **2016**, *28*, 1900–1917.
3. Aggarwal, A.; Jalote, P. Integrating static and dynamic analysis for detecting vulnerabilities. In Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06), Chicaco, IL, USA, 17–21 September 2006; Volume 1, pp. 343–350.
4. Stechyshyn, A. Security Vulnerabilities in Financial Institutions. Master's Thesis, Utica College, Utica, NY, USA, 2015.
5. Williams, P.A.; Woodward, A.J. Cybersecurity vulnerabilities in medical devices: A complex environment and multifaceted problem. *Med. Devices Evid. Res.* **2015**, *8*, pp. 305–316.
6. Mathas, C.M.; Vassilakis, C.; Kolokotronis, N.; et al. On the design of iot security: Analysis of software vulnerabilities for smart grids. *Energies* **2021**, *14*, 2818.
7. Atanasov, A.; Chenane, R. Security Vulnerabilities in Next Generation Air Transportation System. Master's Thesis, Gothenburg, Sweden, 2014.
8. Shahintash, A.; Hajiyev, Y. e-Government Services Vulnerability. In Proceedings of the 2014 IEEE 8th International Conference on Application of Information and Communication Technologies (AICT), Astana, Kazakhstan, 15–17 October 2014; pp. 1–5.
9. Goseva-Popstojanova, K.; Perhinschi, A. On the capability of static code analysis to detect security vulnerabilities. *Inf. Softw. Technol.* **2015**, *68*, 18–33.
10. Siddiqui, S.; Metta, R.; Madhukar, K. Towards multi-language static code analysis. In Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), Florence, Italy, 9–12 October 2023; pp. 81–82.
11. Wang, J.; Huang, M.; Nie, Y.; et al. Static analysis of source code vulnerability using machine learning techniques: A survey. In Proceedings of the 2021 4th International Conference on Artificial Intelligence and Big Data (ICAIBD), Chengdu, China, 28–31 May 2021; pp. 76–86.
12. Chernis, B.; Verma, R.M. Machine learning methods for software vulnerability detection. In Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, Tempe, AZ, USA, 19–21 March 2018.
13. Pradel, M.; Sen, K. Deepbugs: A learning approach to name-based bug detection. In Proceedings of the ACM on Programming Languages, Boston, MA, USA, 4–9 November 2018; Volume 2, pp. 1–25.
14. Zou, D.; Zhu, Y.; Xu, S.; et al. Interpreting deep learning-based vulnerability detector predictions based on heuristic searching. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2021**, *30*, 1–31.
15. Ye, G.; Tang, Z.; Wang, H.; et al. Deep program structure modeling through multi-relational graph-based learning. In Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques, Atlanta, GA, USA, 3–7 October 2020; pp. 111–123.
16. Ahmed, T.; Devanbu, P. Few-shot training llms for project-specific code-summarization. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Rochester, MI, USA, 10–14 October 2022.
17. Sharma, P.; Dash, B. Impact of big data analytics and chatgpt on cybersecurity. In Proceedings of the 2023 4th International Conference on Computing and Communication Systems (I3CS), Shillong, India, 16–18 March 2023; pp. 1–6.
18. Wögerer, W. *A Survey of Static Program Analysis Techniques*; Technische Universität Wien: Vienna, Austria, 2005.
19. Gosain, A.; Sharma, G. A survey of dynamic program analysis techniques and tools. In Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA), Bhubaneswar, India, 14–15 November 2014; pp. 113–122.
20. Baldoni, R.; Coppa, E.; D'elia, D.C.; et al. A survey of symbolic execution techniques. *ACM Comput. Surv.* **2018**, *51*, 1–39. https://doi.org/10.1145/3182657.
21. Xue, H.; Sun, S.; Venkataramani, G.; et al. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access* **2018**, *7*, 65889–65912.
22. Zhou, X.; Cao, S.; Sun, X.; et al. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. *arXiv* **2024**, arXiv:2404.02525.
23. Chen, Y.; Cui, M.; Wang, D.; et al. A survey of large language models for cyber threat detection. *Comput. Secur.* **2024**, *145*, 104016.
24. Wang, J.; Huang, Y.; Chen, C.; et al. Software testing with large language models: Survey, landscape, and vision. *IEEE Trans. Softw. Eng.* **2024**, *50*, 911–936.
25. Al-Karaki, J.; Khan, M.A.Z.; Omar, M. Exploring llms for malware detection: Review, framework design, and countermea-

sure approaches. *arXiv* **2024**, arXiv:2409.07587.
26. Xu, H.; Wang, S.; Li, N.; et al. Large language models for cyber security: A systematic literature review. *arXiv* **2024**, arXiv:2405.04760.
27. Hassanin, M.; Moustafa, N. A comprehensive overview of large language models (llms) for cyber defences: Opportunities and directions. *arXiv* **2024**, arXiv:2405.14487.
28. Sheng, Z.; Chen, Z.; Gu, S.; et al. Llms in software security: A survey of vulnerability detection techniques and insights. *arXiv* **2025**, arXiv:2502.07049.
29. Basic, E.; Giaretta, A. From vulnerabilities to remediation: A systematic literature review of llms in code security. *arXiv* **2025**, arXiv:2412.15004.
30. Nielson, F.; Nielson, H.R.; Hankin, C. *Principles of Program Analysis*; Springer: Berlin/Heidelberg, Germany, 2015.
31. Ashish, A.K.; Aghav, J. Automated techniques and tools for program analysis: Survey. In Proceedings of the 2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT), Tiruchengode, India, 4–6 July 2013; pp. 1–7.
32. Lösch, F. Instrumentation of Java Program Code for Control Flow Analysis. Ph.D. Thesis, Universitätsbibliothek der Universität Stuttgart, Stuttgart, Germany, 2005.
33. Vaswani, A. Attention is all you need. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017.
34. Douglas, M.R. Large language models. *Commun. ACM* **2023**, *66*, 7.
35. Touvron, H.; Lavril, T.; Izacard, G.; et al. Llama: Open and efficient foundation language models. *arXiv* **2023**, arXiv:2302.13971.
36. AI, M. Codellama: Open Code-Focused Language Models. 2023. Available online: https://ai.meta.com/research/code-llama (accessed on 12 October 2024).
37. Brown, T.; Mann, B.; Ryder, N.; et al. Language models are few-shot learners. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 1877–1901.
38. OpenAI. Gpt-4 Technical Report. 2023. Available online: https://openai.com/research/gpt-4 (accessed on 14 October 2024).
39. Li, H.; Hao, Y.; Zhai, Y.; et al. The hitchhiker's guide to program analysis: A journey with large language models. *arXiv* **2023**, arXiv:2308.00245.
40. Ye, J.; Fei, X.; de Carnavalet, X.D.C.; et al. Detecting command injection vulnerabilities in linux-based embedded firmware with llm-based taint analysis of library functions. *Comput. Secur.* **2024**, *144*, 103971.
41. Liu, P.; Sun, C.; Zheng, Y.; et al. Harnessing the power of llm to support binary taint analysis. *arXiv* **2023**, arXiv:2310.08275.
42. Liu, D.; Lu, Z.; Ji, S.; et al. Detecting kernel memory bugs through inconsistent memory management intention inferences. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, August 14–16 2024; pp. 4069–4086.
43. Wang, J.; Huang, Z.; Liu, H.; et al. Defecthunter: A novel llm-driven boosted-conformer-based code vulnerability detection mechanism. *arXiv* **2023**, arXiv:2309.15324.
44. Li, Z.; Dutta, S.; Naik, M. Llm-assisted static analysis for detecting security vulnerabilities. *arXiv* **2024**, arXiv:2405.17238.
45. Cheng, Y.; Shar, L.K.; Zhang, T.; et al. Llm-enhanced static analysis for precise identification of vulnerable oss versions. *arXiv* **2024**, arXiv:2408.07321.
46. Mao, Z.; Li, J.; Jin, D.; et al. Multi-role consensus through llms discussions for vulnerability detection. *arXiv* **2024**, arXiv:2403.14274.
47. Yang, A.Z.; Tian, H.; Ye, H.; et al. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv* **2024**, arXiv:2406.05892.
48. Sun, Y.; Wu, D.; Xue, Y.; et al. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–13. http://dx.doi.org/10.1145/3597503.3639117.
49. Yang, Y. Iot software vulnerability detection techniques through large language model. In Proceedings of the Formal Methods and Software Engineering: 24th International Conference on Formal Engineering Methods, ICFEM 2023, Brisbane, QLD, Australia, 21–24 November 2023.
50. Mathews, N.S.; Brus, Y.; Aafer, Y.; et al. Llbezpeky: Leveraging large language models for vulnerability detection. *arXiv* **2024**, arXiv:2401.01269.
51. Mohajer, M.M.; Aleithan, R.; Harzevili, N.S.; et al. Skipanalyzer: A tool for static code analysis with large language models. *arXiv* **2023**, arXiv:2310.18532.
52. Yang, S.; Lin, X.; Chen, J.; et al. Hyperion: Unveiling dapp inconsistencies using llm and dataflow-guided symbolic execution. *arXiv* **2024**, arXiv:2408.06037.
53. Zhang, C.; Liu, H.; Zeng, J.; et al. Prompt-enhanced software vulnerability detection using chatgpt. *arXiv* **2024**, arXiv:2308.12697.
54. Hu, S.; Huang, T.; Ilhan, F.; et al. Large language model-powered smart contract vulnerability detection: New perspectives. *arXiv* **2023**, arXiv:2310.01152.

55. Xiang, J.; Fu, L.; Ye, T.; et al. Luataint: A static analysis system for web configuration interface vulnerability of internet of things devices. *arXiv* **2024**, arXiv:2402.16043.
56. Aslan, O.A.; Samet, R. A comprehensive review on malware detection approaches. *IEEE Access* **2020**, *8*, 6249–6271.
57. Fujii, S.; Yamagishi, R. Feasibility study for supporting static malware analysis using llm. *arXiv* **2024**, arXiv:2411.14905.
58. Post, M. A call for clarity in reporting bleu scores. *arXiv* **2018**, arXiv:1804.08771.
59. Ganesan, K. Rouge 2.0: Updated and improved measures for evaluation of summarization tasks. *arXiv* **2018**, arXiv:1803.01937.
60. Simion, C.A.; Balan, G.; Gavriluţ, D.T. Benchmarking out of the box open-source llms for malware detection based on api calls sequences. In Proceedings of the International Conference on Intelligent Data Engineering and Automated Learning—IDEAL 2024, Valencia, Spain, 20–22 November 2024.
61. Jiang, A.Q.; Sablayrolles, A.; Mensch, A.; et al. Mistral 7B. *arXiv* **2023**, arXiv:2310.06825.
62. Jiang, A.Q.; Sablayrolles, A.; Roux, A.; et al. Mixtral of experts. *arXiv* **2024**, arXiv:2401.04088.
63. Zahan, N.; Burckhardt, P.; Lysenko, M.; et al. Leveraging Large Language Models to Detect npm Malicious Packages. *arXiv* **2024**, arXiv:2403.12196.
64. GitHub. Codeql: Github's Static Analysis Engine for Code Vulnerabilities. 2025. Available Online: https://codeql.github.com/ (accessed on 15 January 2025).
65. Khan, I.; Kwon, Y.W. A structural-semantic approach integrating graph-based and large language models representation to detect android malware. In Proceedings of the IFIP International Conference on ICT Systems Security and Privacy Protection, Edinburgh, UK, 12–14 June 2024; pp. 279–293.
66. Feng, Z.; Guo, D.; Tang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
67. Zhao, W.; Wu, J.; Meng, Z. Apppoet: Large language model based android malware detection via multi-view prompt engineering. *arXiv* **2024**, arXiv:2404.18816.
68. Kozyrev, A.; Solovev, G.; Khramov, N.; et al. Coqpilot, a plugin for llm-based generation of proofs. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, Sacramento, CA, USA, 27 October–1 November 2024; pp. 2382–2385.
69. Zhang, L.; Lu, S.; Duan, N. Selene: Pioneering automated proof in software verification. *arXiv* **2024**, arXiv:2401.07663.
70. Chakraborty, S.; Lahiri, S.K.; Fakhoury, S.; et al. Ranking llm-generated loop invariants for program verification. *arXiv* **2024**, arXiv:2310.09342.
71. Janßen, C.; Richter, C.; Wehrheim, H. Can chatgpt support software verification? *arXiv* **2023**, arXiv:2311.02433.
72. Pirzada, M.A.; Reger, G.; Bhayat, A.; et al. Llm-generated invariants for bounded model checking without loop unrolling. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, Sacramento, CA, USA, 27 October–1 November 2024; pp. 1395–1407.
73. Wu, G.; Cao, W.; Yao, Y.; et al. Llm meets bounded model checking: Neuro-symbolic loop invariant inference. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, Sacramento, CA, USA, 27 October–1 November 2024; pp. 406–417.
74. Pei, K.; Bieber, D.; Shi, K.; et al. Can large language models reason about program invariants? In Proceedings of the 40th International Conference on Machine Learning, Honolulu, HI, USA, 23–29 July 2023.
75. Wen, C.; Cao, J.; Su, J.; et al. Enchanting program specification synthesis by large language models using static analysis and program verification. In Proceedings of the Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, 24–27 July 2024.
76. Wu, H.; Barrett, C.; Narodytska, N. Lemur: Integrating large language models in automated program verification. *arXiv* **2024**, arXiv:2310.04870.
77. Mukherjee, P.; Delaware, B. Towards automated verification of llm-synthesized c programs. *arXiv* **2024**, arXiv:2410.14835.
78. Liu, Y.; Xue, Y.; Wu, D.; et al. Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. *arXiv* **2024**, arXiv:2405.02580.
79. Wang, W.; Liu, K.; Chen, A.R.; et al. Python symbolic execution with llm-powered code generation. *arXiv* **2024**, arXiv:2409.09271.
80. Su, J.; Deng, L.; Wen, C.; et al. Cfstra: Enhancing configurable program analysis through llm-driven strategy selection based on code features. In Proceedings of the Theoretical Aspects of Software Engineering: 18th International Symposium, TASE 2024, Guiyang, China, 29 July–1 August 2024; pp. 374–391. https://doi.org/10.1007/978-3-031-64626-3_22.
81. Chapman, P.J.; Rubio-González, C.; Thakur, A.V. Interleaving static analysis and llm prompting. In Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, Copenhagen, Denmark, 25 June 2024; pp. 9–17. https://doi.org/10.1145/3652588.3663317.
82. Anthropic. Claude. 2025. Available online: https://www.anthropic.com/claude (accessed on 16 January 2025).
83. Czajka, Ł.; Kaliszyk, C. Hammer for coq: Automation for dependent type theory. *J. Autom. Reason.* **2018**, *61*, 423–453. https://doi.org/10.1007/s10817-018-9458-4.
84. Blaauwbroek, L.; Urban, J.; Geuvers, H. *The Tactician: A Seamless, Interactive Tactic Learner and Prover for Coq*;

Springer International Publishing: Berlin/Heidelberg, Germany, 2020; pp. 271–277. http://dx.doi.org/10.1007/978-3-030-53518-6_17.

85. Klein, G.; Andronick, J.; Elphinstone, K.; et al. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.* **2014**, *32*, 1–70. https://doi.org/10.1145/2560537.
86. Group, T.S. sel4: The World's First Operating-System Kernel with an End-to-End Proof of Implementation Correctness. Available online: https://sel4.systems/ (accessed on 18 January 2025).
87. Beyer, D. *Competition on Software Verification and Witness Validation: SV-COMP 2023*; Springer Nature: Cham, Switzerland, 2023; pp. 495–522.
88. Baudin, P.; Filliâtre, J.-C.; Marché, C.; et al. ACSL: ANSI/ISO C Specification Language. Available online: http://frama-c.com/download/acsl.pdf (accessed on 15 October 2024).
89. Baudin, P.; Bobot, F.; Bühler, D.; et al. The dogged pursuit of bug-free c programs: the frama-c software analysis platform. *Commun. ACM* **2021**, *64*, 56–68. https://doi.org/10.1145/3470569.
90. Beyer, D.; Keremoglu, M.E. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 184–190.
91. Ernst, M.D.; Perkins, J.H.; Guo, P.J.; et al. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* **2007**, *69*, 35–45.
92. Menezes, R.; Aldughaim, M.; Farias, B.; et al. Esbmc v7.4: Harnessing the power of intervals. *arXiv* **2023**, arXiv:2312.14746.
93. Gurfinkel, A.; Kahsai, T.; Komuravelli, A.; et al. The seahorn verification framework. In *Computer Aided Verification*; Springer International Publishing: Cham, Switzerland, 2015; pp. 343–361.
94. Darke, P.; Agrawal, S.; Venkatesh, R. Veriabs: A tool for scalable verification by abstraction (competition contribution). In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Luxembourg, 27 March–1 April 2021. https://doi.org/10.1007/978-3-030-72013-1_32.
95. Kroening, D.; Tautschnig, M. Cbmc—C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 389–391.
96. Heizmann, M.; Christ, J.; Dietsch, D.; et al. Ultimate automizer with smtinterpol. In *Tools and Algorithms for the Construction and Analysis of Systems*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 641–643.
97. De Moura, L.; Bjørner, N. Z3: an efficient smt solver. In Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary, 29 March–6 April 2008; pp. 337–340.
98. Lu, J.; Yu, L.; Li, X.; et al. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), Florence, Italy, 9–12 October 2023; pp. 647–658.
99. Dhulipala, H.; Yadavally, A.; Nguyen, T.N. Planning to guide llm for code coverage prediction. In Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering, Lisbon, Portugal, 14 April 2024; pp. 24–34. https://doi.org/10.1145/3650105.3652292.
100. Hu, P.; Liang, R.; Chen, K. Degpt: Optimizing decompiler output with llm. In Proceedings of the 2024 Network and Distributed System Security Symposium, San Diego, CA, USA, 26 February–1 March 2024.
101. Yan, J.; Huang, J.; Fang, C.; et al. Better debugging: Combining static analysis and llms for explainable crashing fault localization. *arXiv* **2024**, arXiv:2408.12070.
102. Pomian, D.; Bellur, A.; Dilhara, M.; et al. Together we go further: Llms and ide static analysis for extract method refactoring. *arXiv* **2024**, arXiv:2401.15298.
103. Rong, H.; Duan, Y.; Zhang, H.; et al. Disassembling obfuscated executables with llm. *arXiv* **2024**, arXiv:2407.08924.
104. Wang, H.; Wang, Z.; Liu, P. A hybrid llm workflow can help identify user privilege related variables in programs of any size. *arXiv* **2024**, arXiv:2403.15723.
105. Wen, C.; Cai, Y.; Zhang, B.; et al. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Trans. Knowl. Discov. Data* **2024**, *18*, 1–34. https://doi.org/10.1145/3653718.
106. Flynn, L.; Klieber, W. Using Llms to Automate Static-Analysis Adjudication and Rationales. Available online: https://insights.sei.cmu.edu/library/using-llms-to-automate-static-analysis-adjudication-and-rationales/ (accessed on 2 October 2024).
107. Hao, Y.; Chen, W.; Zhou, Z.; et al. E&v: Prompting large language models to perform static analysis by pseudo-code execution and verification. *arXiv* **2023**, arXiv:2312.08477.
108. Yan, P.; Tan, S.; Wang, M.; et al. Prompt engineering-assisted malware dynamic analysis using gpt-4. *arXiv* **2023**, arXiv:2312.08317.
109. Sun, Y.S.; Chen, Z.K.; Huang, Y.T.; et al. Unleashing Malware Analysis and Understanding With Generative AI. *IEEE Secur. Priv.* **2024**, *22*, 12–23.
110. Sánchez, P.M.S.; Celdrán, A.H.; Bovet, G.; et al. Transfer learning in pre-trained large language models for malware detection based on system calls. *arXiv* **2024**, arXiv:2405.09318.

111. Li, Y.; Fang, S.; Zhang, T.; et al. Enhancing android malware detection: The influence of chatgpt on decision-centric task. *arXiv* **2024**, arXiv:2410.04352.
112. Qiu, F.; Ji, P.; Hua, B.; et al. Chemfuzz: Large language models-assisted fuzzing for quantum chemistry software bug detection. In Proceedings of the 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), Chiang Mai, Thailand, 22–26 October 2023; pp. 103–112.
113. Anthropic. Claude-2. 2023. Available online: https://www.anthropic.com/index/claude-2 (accessed on 9 December 2024).
114. Google. Bard. 2023. Available online: https://bard.google.com (accessed on 9 December 2024).
115. Eom, J.; Jeong, S.; Kwon, T. Covrl: Fuzzing javascript engines with coverage-guided reinforcement learning for llm-based mutation. *arXiv* **2024**, arXiv:2402.12222.
116. Zhang, H.; Rong, Y.; He, Y.; et al. Llamafuzz: Large language model enhanced greybox fuzzing. *arXiv* **2024**, arXiv:2406.07714.
117. Deng, Y.; Xia, C.S.; Yang, C.; et al. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv* **2023**, arXiv:2304.02014.
118. Hu, J.; Zhang, Q.; Yin, H. Augmenting greybox fuzzing with generative ai. *arXiv* **2023**, arXiv:2306.06782.
119. Lemieux, C.; Inala, J.P.; Lahiri, S.K.; et al. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In Proceedings of the 45th International Conference on Software Engineering, Melbourne, Australia, 14–20 May 2023; pp. 919–931. https://doi.org/10.1109/ICSE48619.2023.00085.
120. Meng, R.; Mirchev, M.; Böhme, M.; et al. Large language model guided protocol fuzzing. In Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 26 February–1 March 2024.
121. Oliinyk, Y.; Scott, M.; Tsang, R.; et al. Fuzzing busybox: Leveraging llm and crash reuse for embedded bug unearthing. *arXiv* **2024**, arXiv:2403.03897.
122. Xia, C.S.; Paltenghi, M.; Le Tian, J.; et al. Fuzz4all: Universal fuzzing with large language models. *arXiv* **2024**, arXiv:2308.04748.
123. ICMAB-CSIC. Siesta. 2023. Available online: https://departments.icmab.es/leem/siesta/ (accessed on 9 December 2024).
124. Sparck Jones, K. A statistical interpretation of term specificity and its application in retrieval. *J. Doc.* **1972**, *28*, 11–21.
125. Chen, M.; Tworek, J.; Jun, H.; et al. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374.
126. Nijkamp, E.; Pang, B.; Hayashi, H.; et al. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv* **2023**, arXiv:2203.13474.
127. Fioraldi, A.; Maier, D.; Eißfeldt, H.; et al. {AFL++}: Combining Incremental Steps of Fuzzing Research. In Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20), Online, 11 August 2020. Available online: https://www.usenix.org/conference/woot20/presentation/fioraldi (accessed on 11 November 2024).
128. McMinn, P. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.* **2004**, *14*, 105–156.
129. Wells, N. Busybox: A swiss army knife for linux. *Linux J.* **2000**, *78*, 10.
130. Arkin, B.; Stender, S.; McGraw, G. Software penetration testing. *IEEE Secur. Priv.* **2005**, *3*, 84–87.
131. Deng, G.; Liu, Y.; Mayoral-Vilches, V.; et al. {PentestGPT}: Evaluating and harnessing large language models for automated penetration testing. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–16 August 2024; pp. 847–864.
132. Huang, J.; Zhu, Q. Penheal: A Two-Stage Llm Framework for Automated Pentesting and Optimal Remediation. Available online: https://synthical.com/article/655e0b6b-8ece-4830-bb82-649bac33bd5e (accessed on 18 June 2024).
133. Goyal, D.; Subramanian, S.; Peela, A. Hacking, the lazy way: Llm augmented pentesting. *arXiv* **2024**, arXiv:2409.09493.
134. Bianou, S.G.; Batogna, R.G. Pentest-ai, an llm-powered multi-agents framework for penetration testing automation leveraging mitre attack. In Proceedings of the 2024 IEEE International Conference on Cyber Security and Resilience (CSR), London, UK, 2–4 September 2024; pp. 763–770.
135. Zych, M.; Mavroeidis, V. Enhancing the stix representation of mitre attack for group filtering and technique prioritization. *arXiv* **2022**, arXiv:2204.11368.
136. Muzsai, L.; Imolai, D.; Lukács, A. Hacksynth: Llm agent and evaluation framework for autonomous penetration testing. *arXiv* **2024**, arXiv:2412.01778.
137. LGioacchini, L.; Mellia, M.; Drago, I.; et al. Autopenbench: Benchmarking generative agents for penetration testing. *arXiv* **2024**, arXiv:2410.03225.
138. Shen, X.; Wang, L.; Li, Z.; et al. Pentestagent: Incorporating llm agents to automated penetration testing. *arXiv* **2024**, arXiv:2411.05185.
139. Bhatia, S.; Gandhi, T.; Kumar, D.; et al. Unit test generation using generative ai: A comparative performance analysis of autogeneration tools. In Proceedings of the 1st International Workshop on Large Language Models for Code, Lisbon, Portugal, 20 April 2024; pp. 54–61. https://doi.org/10.1145/3643795.3648396.
140. Lukasczyk, S.; Fraser, G. Pynguin: automated unit test generation for python. In Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, Pittsburgh, PA, USA, 21–29 May 2022. http://dx.doi.org/10.1145/3510454.3516829.

141. Pan, R.; Kim, M.; Krishna, R.; et al. Multi-language unit test generation using llms. *arXiv* **2024**, arXiv:2409.03093.
142. Chen, Y.; Hu, Z.; Zhi, C.; et al. Chatunitest: A framework for llm-based test generation. In Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, 15–19 July 2024. https://doi.org/10.1145/3663529.3663801.
143. Zhang, Z.; Liu, X.; Lin, Y.; et al. Llm-based unit test generation via property retrieval. *arXiv* **2024**, arXiv:2410.13542.
144. Gu, S.; Fang, C.; Zhang, Q.; et al. Testart: Improving llm-based unit testing via co-evolution of automated generation and repair iteration. *arXiv* **2024**, arXiv:2408.03095.
145. Yuan, Z.; Lou, Y.; Liu, M.; et al. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv* **2024**, arXiv:2305.04207.
146. Wang, Z.; Liu, K.; Li, G.; et al. Hits: High-coverage llm-based unit test generation via method slicing. *arXiv* **2024**, arXiv:2408.11324.
147. Lops, A.; Narducci, F.; Ragone, A.; et al. A system for automated unit test generation using large language models and assessment of generated test suites. *arXiv* **2024**, arXiv:2408.07846.
148. Nunez, A.; Islam, N.T.; Jha, S.K.; et al. Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing. *arXiv* **2024**, arXiv:2409.10737.
149. Pizzorno, J.A.; Berger, E.D. Coverup: Coverage-guided llm-based test generation. *arXiv* **2024**, arXiv:2403.16218.
150. Kumar, R.; Xiaosong, Z.; Khan, R.U.; et al. Effective and explainable detection of android malware based on machine learning algorithms. In Proceedings of the 2018 International Conference on Computing and Artificial Intelligence, Chengdu China, 12–14 March 2018. https://doi.org/10.1145/3194452.3194465.
151. Onwuzurike, L.; Mariconti, E.; Andriotis, P.; et al. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *arXiv* **2019**, arXiv:1711.07477.
152. Wu, B.; Chen, S.; Gao, C.; et al. Why an android app is classified as malware? towards malware classification interpretation. *arXiv* **2020**, arXiv:2004.11516.
153. Williamson, A.Q.; Beauparlant, M. Malware reverse engineering with large language model for superior code comprehensibility and ioc recommendations. *Research Square* **2024**, https://doi.org/10.21203/rs.3.rs-4471373/v1.
154. Glazunov, S.; Brand, M. Project Naptime: Evaluating Offensive Security Capabilities of Large Language Models. Available online: https://googleprojectzero.blogspot.com/2024/06/project-naptime.html (accessed on 16 October 2024).
155. Bhatt, M.; Chennabasappa, S.; Li, Y.; et al. Cyberseceval 2: A wide-ranging cybersecurity evaluation suite for large language models. *arXiv* **2024**, arXiv:2404.13161.
156. Liu, A.; Feng, B.; Xue, B.; et al. Deepseek-v3 technical report. *arXiv* **2024**, arXiv:2412.19437.