Neurosymbolic Code Auditing

Suggested Citation: Chengpeng Wang, Zhuo Zhang, Xiangzhe Xu, Mingwei Zheng, Guannan Wei and Xiangyu Zhang (2026), "Neurosymbolic Code Auditing", : Vol. xx, No. xx, pp 1–18. DOI: 10.1561/XXXXXXXXX.

Chengpeng Wang
Purdue University
wang6590@purdue.edu

Zhuo Zhang
Columbia University
zz@cs.columbia.edu

Xiangzhe Xu
Purdue University
xu1415@purdue.edu

Mingwei Zheng
Purdue University
zheng618@purdue.edu

Guannan Wei
Tufts University
guannan.wei@tufts.edu

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

now
the essence of knowledge
Boston — Delft

Contents

Neurosymbolic Code Auditing

Chengpeng Wang[1], Zhuo Zhang[2], Xiangzhe Xu[3], Mingwei Zheng[4], Guannan Wei[5] and Xiangyu Zhang[6]

[1] *Purdue University; wang6590@purdue.edu*
[2] *Columbia University; zz@cs.columbia.edu*
[3] *Purdue University; xu1415@purdue.edu*
[4] *Purdue University; zheng618@purdue.edu*
[5] *Tufts University; guannan.wei@tufts.edu*
[6] *Purdue University; xyzhang@purdue.edu*

ABSTRACT

Code auditing examines whether an implementation satisfies target properties such as reliability, security, performance, and design intent. Classical static analysis provides rigorous foundations for automated auditing, but it remains difficult to apply broadly in real-world software. The emergence of LLMs and agents creates new opportunities for leveraging semantic cues that are embedded in source code, documentation, and other software artifacts yet difficult for classical static analyzers to derive.

This survey presents a comprehensive study of *neurosymbolic code auditing*, an emerging synthesis that integrates LLM-driven reasoning with classical static analysis and other structured program reasoning techniques. By formalizing the cognitive processes underlying manual code auditing, we conceptualize an agent-centric framework that serves as a unifying foundation for neurosymbolic analysis. Specifically, we categorize LLM-enabled primitive analyses over both code and non-code artifacts, addressing the fundamental

ISSN ; DOI XXXXXXXX

reasoning tasks routinely performed by human auditors. Furthermore, we review a broad spectrum of neurosymbolic architectures that combine symbolic tools with LLM-based reasoning, illustrating how these systems replicate and extend the practical strategies employed by human auditors. Lastly, we synthesize the survey around three recurring bottlenecks, namely observation, verification, and memory, then outline future directions and the persistent technical challenges.

1

Introduction

Code auditing is a fundamental activity in software development in which developers, security analysts, and automated tools examine whether an implementation satisfies a set of expected properties. These properties typically concern security, reliability, performance, or compliance with project-specific specifications. Rather than merely inspecting code, auditing is a multi-step, often iterative, evidence-seeking process that involves gathering program facts from source code and surrounding artifacts and assessing whether those facts indicate violations of the target properties.

From the perspective of automated support, code auditing is commonly assisted by dynamic and static program-analysis techniques. Dynamic techniques, such as unit testing (Xie *et al.*, 2016; Lukasczyk and Fraser, 2022) and fuzzing (Gutmann, 2016; Zhu *et al.*, 2022), execute the program to expose concrete failures. They can be effective at uncovering certain classes of defects, but their applicability is constrained by the availability of an execution environment, adequate test harnesses, and sufficient input coverage. Static techniques, in contrast, analyze code without executing it and can therefore be applied earlier in the development process. In this survey, we focus on *static code auditing*,

This focus is particularly relevant in modern software development, where large codebases, complex dependencies, and AI-generated code increasingly create situations in which execution-based validation is impractical, but early and scalable auditing support is still required.

This chapter first provides preliminaries on code auditing, with particular attention to the new challenges that have emerged in the AI era. It then examines the opportunities afforded by large language models (LLMs) (Brown *et al.*, 2020; OpenAI, 2023) for code auditing, discussing their intrinsic capabilities and limitations. Finally, it introduces the scope of this survey and presents an overview of its organization.

1.1 Code Auditing in the AI Era

The rise of generative artificial intelligence (Vaswani *et al.*, 2017; Brown *et al.*, 2020; Ouyang *et al.*, 2022) has fundamentally reshaped modern software development (Chen *et al.*, 2021; Jiang *et al.*, 2026). AI coding agents powered by LLMs, such as Codex (OpenAI, 2026a) and Claude Code (Anthropic, 2026a), have been rapidly evolving and are now deeply embedded in mainstream development environments and workflows. Developers increasingly rely on these tools to generate program lines, complete functions, and even implement non-trivial program logic, substantially accelerating the pace of software production (Peng *et al.*, 2023; Ziegler *et al.*, 2022; Barke *et al.*, 2023).

However, AI coding agents also raise concerns that go beyond security, including reliability, performance, maintainability, and broader aspects of code quality. For example, recent studies have demonstrated that AI-generated code can harbor substantial security weaknesses (Xu *et al.*, 2025b; Fu *et al.*, 2025). Developer surveys further indicate that practitioners frequently expend considerable effort correcting "almost-right" outputs and remain skeptical of their correctness. As a result, code auditing has taken on heightened importance in the AI era. Because developers may possess only a partial understanding of generated implementations and the design decisions behind them, they must invest greater effort in defect detection, root-cause localization, and repair. Auditing therefore serves as a critical mechanism for establishing trust in AI programming (Roychoudhury *et al.*, 2026).

Over the past several decades, a broad spectrum of program-analysis techniques has been developed to support code auditing, including data-flow analysis (Kildall, 1973; Kam and Ullman, 1977; Reps *et al.*, 1995; Arzt *et al.*, 2014), abstract interpretation (Cousot and Cousot, 1977a; Cousot and Cousot, 1979; Blanchet *et al.*, 2003), symbolic execution (King, 1976; Cadar *et al.*, 2008), and model checking (Clarke and Emerson, 1982; Queille and Sifakis, 1982; Holzmann, 1997). These static analysis frameworks offer rigorous foundations for reasoning about program behavior and have achieved considerable success. However, fundamental gaps still remain that prevent them from serving as general-purpose auditing solutions, particularly in the context of AI-assisted software development.

First, AI-assisted coding workflows often produce intermediate code artifacts that are incomplete with respect to their project contexts. For example, developers may receive partial implementations that lack surrounding type definitions, imported dependencies, build configurations, or the caller and callee context needed to construct a complete analysis target. Many production static analyzers assume a well-formed and buildable project. Although modular and compositional analyses relax the need for whole-program availability, they still typically require explicit modules that may be absent in early AI-assisted coding scenarios. This limitation is not unique to AI-assisted workflows. Even in conventional software development, existing static analysis tools generally offer limited support when developers need to reason about in-progress code that is not yet integrated into a buildable project.

Second, many mature static analyzers are most effective when the target property can be stated as a relatively precise and recurring pattern, such as memory-safety violations and taint-style bugs. These classic bug categories remain important, but the quality concerns arising in AI-assisted programming are broader and more diverse. They include subtle logical errors, semantic inconsistencies with developer intent, violations of implicit design conventions, and performance anti-patterns. The difficulty is not that such properties are inexpressible in formal languages. Rather, their intended specifications are often unavailable, underspecified, or highly context-dependent. As a result, building sound and complete characterizations of these properties of-

ten becomes impractical, while natural-language descriptions remain accessible because developers can tolerate a certain degree of ambiguity when communicating intent.

These limitations reveal a mismatch between the assumptions of classical static analysis and the demands of modern AI-assisted software development. As AI coding agents assume an increasingly large role in software production, there is a pressing need for auditing techniques that can reason about partial code artifacts, recover or approximate missing context, and accommodate properties whose specifications are distributed across diverse software artifacts.

1.2 LLMs for Code Auditing

Beyond code generation, modern LLMs and AI coding agents are increasingly employed as assistants capable of inspecting, reasoning about, and acting upon complex software artifacts within integrated development environments (IDEs). Systems such as Codex (OpenAI, 2026a), Claude Code (Anthropic, 2026a), and other AI coding agents have demonstrated that LLMs, when equipped with appropriate tooling, can navigate multi-file codebases, follow cross-module call chains, interpret documentation, and respond to high-level auditing requests expressed in natural language. This emerging paradigm points to a fundamental shift in how code auditing may be conducted. Rather than relying exclusively on classical static analysis frameworks, auditing can increasingly draw on customizable, interactive, and goal-driven semantic reasoning over both code and its surrounding artifacts.

To illustrate this potential, we consider the motivating example in Figure 1.1, a simplified null pointer dereference bug drawn from `sofa-pbrpc`[1], an open-source RPC framework. The bug involves a null value that originates in the function `field2json`, escapes the function boundary, and is eventually dereferenced without a guard. When we provide Claude 3.5 Sonnet with the set of relevant functions and specify, in natural language, the auditing objective of detecting the null pointer dereference, the model is able to analyze the code and produce a

[1] https://github.com/baidu/sofa-pbrpc

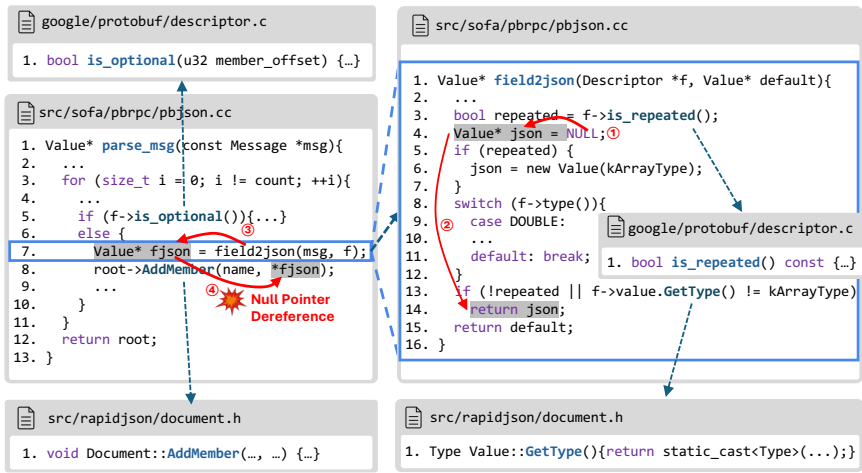

Figure 1.1: A simplified null pointer dereference bug drawn from the open-source RPC framework `sofa-pbrpc`. The function `field2json` may return null under certain branch conditions. The null value is propagated out of the function `field2json` before being dereferenced without a null check. Blue dashed arrows indicate call-graph edges, while red solid arrows trace the interprocedural data-flow path along which the null value propagates from its origin to the dereference site.

detailed explanation of how the null value is propagated through the call chain and eventually dereferenced. Notably, the model does not require constructing a complete intermediate representation of the project, resolving external dependencies, or writing formal specifications. Instead, the auditing intent can be specified directly in natural language, and the model responds by interpreting the code semantics, tracing data-flow facts, and generating a detailed bug explanation.

This interaction exemplifies several important departures from classical static analysis. First, auditing can be brought earlier into the development cycle, for example directly inside an IDE, without requiring a fully configured analysis environment or a successful build. Second, the auditing objective can be specified flexibly. Human auditors can ask about particular bug classes, project-specific invariants, or ad hoc concerns without implementing new checkers. Third, the output can take the form of an explanation rather than a bare warning, connecting root causes, propagation paths, and failure points in a manner similar to human audit reports. These properties suggest that LLMs can serve

as powerful building blocks for next-generation code-auditing systems, offering flexibility, accessibility, and semantic expressiveness that are difficult to obtain from classical static analysis alone. At the same time, LLMs remain insufficient as standalone auditors. In the following sections, we discuss both their intrinsic capabilities and the limitations that prevent them from scaling reliably to large-scale code auditing.

1.2.1 LLMs' Intrinsic Capabilities

Prior studies (Chen *et al.*, 2024; Cheng *et al.*, 2024; Yadavally *et al.*, 2024) have shown that LLMs can perform basic static-analysis tasks on small code fragments. This capability can complement classical static analysis, which typically operates over structured intermediate representations, such as Static Single Assignment (SSA) (Cytron *et al.*, 1991), and often requires substantial engineering to model program semantics, such as points-to properties and data dependencies. In contrast, LLMs can often conduct such reasoning within a limited analysis scope by leveraging code-understanding abilities acquired during pretraining and post-training, much like skilled human auditors. In what follows, we focus on three critical capabilities, including *reducing programs*, *handling pointers*, and *exploring feasible program paths*.

Capability I: Reducing Programs. Program reduction (Ferles *et al.*, 2017; Sun *et al.*, 2018) is essential for scalability in code auditing. Given a property, a set of initial program points, and a defined scope, program reduction identifies the subset of statements relevant to the property within the scope. These statements form a self-contained, smaller program, significantly reducing the number of paths to analyze. For instance, in null pointer dereference detection, program reduction targets how null values are propagated in the program and focuses on key statements such as null pointer assignments, conditional checks guarding propagation, and cross-function propagation (e.g., passing the pointer to a callee function via a parameter, returning it to a caller function, or writing it to a global variable). In a preliminary experiment (Figure 1.2), we fed the function `field2json` shown in Figure 1.1 to Claude 3.5 Sonnet and asked it to reduce the program with respect to

Please analyze the function *field2json* in Figure 1.1. We only care about the statements that affect the propagation of the NULL value at line 3. Please simplify the function *field2json* for program reduction.

Here is the program after reduction.

```
1. Value* field2json(Descriptor *f, Value* default){
2.    bool repeated = f->is_repeated();
3.    Value* json = NULL;
4.    if (repeated) {
5.       json = new Value(kArrayType);
6.    }
7.    if (!repeated || f->value.GetType() != kArrayType)
8.       return json;
9. }
```

Figure 1.2: An example of program reduction by LLMs. Given the function `field2json` from Figure 1.1 and the targeted null value at line 3, the model (Claude 3.5 Sonnet) retains only the statements relevant to how a null value is produced and returned, while removing irrelevant logic such as the `switch` block (lines 8–12 in the function `field2json`).

null-value propagation. The reduced program retains only the statements critical to how a null value is produced and returned, while removing irrelevant logic such as the `switch` block. Notably, human auditors often perform such reduction implicitly, facilitating complex code reasoning without the excessive path exploration that hampers classic static tools.

Capability II: Handling Pointers. Program pointers may point to different memory locations depending on their runtime values. Consequently, reads and writes via pointer dereferences create data-flow facts that dynamically vary based on pointer values. Determining the set of memory locations a pointer variable may point to is known as *points-to analysis* (Smaragdakis and Balatsouras, 2015), an important prerequisite of many downstream analyses, such as data-dependence analysis. Classical points-to analysis techniques are based on memory abstractions, treating concrete memory locations as abstract objects. For instance, an analysis may assume that a pointer referencing a heap object could potentially point to multiple heap objects, as precise heap addresses cannot typically be determined statically. This over-approximation leads to overly inflated dependence relations, complicating downstream analysis

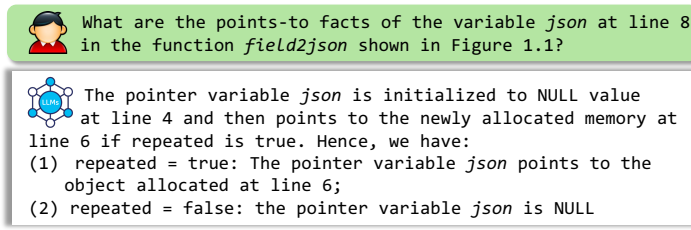

Figure 1.3: An example of pointer handling by LLMs. Given `field2json` shown in Figure 1.1, the model (Claude 3.5 Sonnet) identifies the two possible points-to targets of the return value, namely a heap-allocated `JSON` object and null, together with the path conditions under which each arises.

tasks. In contrast, human auditors can often intuitively and accurately determine the points-to facts of a specific pointer variable through their understanding of program semantics. Advanced LLMs exhibit similar capabilities, particularly within the scope of individual functions. As shown in Figure 1.3, we prompt Claude 3.5 Sonnet with the `field2json` function from Figure 1.1 and query the points-to targets of its return value. The model accurately identifies the two possible targets, including a heap object allocated at line 6 in the function `field2json` and null, and, crucially, reports the path conditions under which each arises. In contrast, representative static analyzers such as SVF (Sui and Xue, 2016) compute points-to facts without path conditions by default, conflating the two cases and losing precision.

Capability III: Exploring Feasible Program Paths. A bug report is often considered valid only if it provides a feasible program execution path from the root cause to the symptom as evidence. To determine feasibility, classical tools rely on modeling conditional checks along a path as symbolic constraints (e.g., in the form of first-order logic formulas) and querying a theorem prover (e.g., the SMT solver Z3 (Moura and Bjørner, 2008)) to check if an input can satisfy the constraints. This process is computationally expensive and prone to failure, as converting program paths to logic formulas requires exploring an exponentially large number of program paths and modeling a wide range of program behaviors that lack direct logical representations, such as loops, recursion, aliasing, pointer arithmetic, and unbounded string operations.

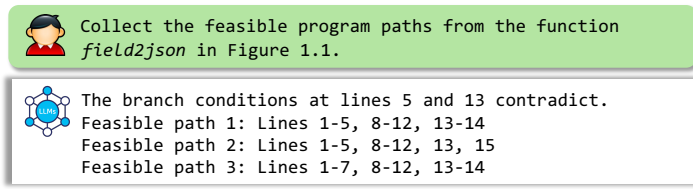

Figure 1.4: An example of feasible program path exploration by LLMs. Given `field2json` shown in Figure 1.1, the model (Claude 3.5 Sonnet) skips irrelevant branches (e.g., the `switch` block), detects that the conditions at lines 5 and 13 are contradictory, and enumerates the three feasible paths through the function.

In contrast, humans, as well as LLMs, rely on program reduction and intuitive logical reasoning to assess feasibility, an approach that is highly effective within a limited scope. For example, in Figure 1.4, we prompt Claude 3.5 Sonnet to enumerate the feasible paths through `field2json` from Figure 1.1. The model skips the irrelevant `switch` block, discovers that the branch conditions at lines 5 and 13 are contradictory, thereby ruling out paths that traverse both lines 6 and 14, and reports exactly three feasible paths. In contrast, a classical symbolic execution engine would need to encode every branch as a solver query, incurring substantially higher cost on real-world code with deeply nested control flow.

1.2.2 LLMs' Inadequacy

LLMs alone are not silver bullets for code auditing. Due to the inherent complexity of software engineering (Brooks, 1987) and the undecidability of static analysis problems (Rice, 1953), we cannot expect LLMs to discover all bugs in real-world codebases without false positives. Recent frontier models have substantially expanded the usable context window (Liu *et al.*, 2024a; Chen *et al.*, 2023; Peng *et al.*, 2024), making it possible to process larger code fragments. However, a large nominal context window does not by itself guarantee reliable code auditing. Reasoning over long contexts is still constrained by finite test-time compute and by the model's imperfect ability to retrieve, connect, and reason over relevant information scattered across the input. Notably, recent long-context evaluations show that model performance can vary

substantially across downstream tasks and may degrade as the context becomes longer (Yen *et al.*, 2024). Therefore, long-context modeling should be viewed as a useful substrate for code auditing rather than a complete solution.

To validate our speculation, we conducted a controlled experiment in which we prompted Claude 3.5 Sonnet with all five functions shown in Figure 1.1 and asked it to identify the null pointer dereference bug in a single end-to-end pass, following a methodology similar to a recent study (Fang *et al.*, 2024a). We call this a controlled experiment because, in practice, an auditor cannot assume knowledge of the complete set of functions relevant to a bug. The model exhibited substantial hallucinations, reporting that almost all dereferenced pointers have null values. Even when we improved the prompts by offering several few-shot examples and explanations of how a null pointer dereference bug occurs, the model still hallucinated, producing false positives and incorrect explanations. Therefore, an end-to-end prompting-based solution cannot offer high-quality auditing results.

This gap reflects a well-known structural challenge in manual code auditing: given scale, complexity, and limited time, even experienced human auditors cannot reliably identify all potential bugs in a large and complex software system through direct codebase inspection alone. This observation motivates the exploration of a more integrated approach to code auditing. In this survey, we refer to the systematic orchestration of LLMs and static analysis techniques within auditing workflows as *neurosymbolic code auditing*, an emerging synthesis that constitutes the central theme of this work.

1.3 Scope and Organization

This survey adopts a neurosymbolic perspective on code auditing. Rather than treating auditing as a single-shot prediction problem or a purely symbolic reasoning task, we view it as an evidence-seeking workflow. In this workflow, an auditor operates over a rich environment of code and non-code artifacts, interacts with external static-analysis tools, and incrementally accumulates the program facts needed for concrete auditing tasks. This perspective provides a unifying lens for recent

code-auditing techniques, clarifies the main bottlenecks they face, and delineates the distinct roles of different tool families.

Although the neurosymbolic auditing framework introduced here is, in principle, applicable to a broad range of coding tasks, including code understanding, refactoring, and transformation, this survey focuses primarily on *bug detection*, especially defects relevant to security and reliability. Specifically, bug detection is the application domain in which the convergence of LLMs and static-analysis techniques has been most actively explored, and it provides the concrete technical challenges around which the surveyed literature is organized. Although a small number of surveyed techniques incorporate dynamic components, such as concolic execution or hybrid fuzzing (Meng *et al.*, 2024a; Luo *et al.*, 2026), their primary reasoning remains static in nature. We include these techniques because they instantiate the same neurosymbolic mechanisms that unify this survey.

Under this lens, this survey covers work published between 2023 and 2026 across the programming languages, software engineering, computer security, machine learning, and natural language processing communities. For programming languages, software engineering, and computer security venues, we collected bibliographic metadata from conference proceedings and used Claude Code with Claude Sonnet 4.6 to screen paper abstracts for relevance to LLMs and code auditing. Because the machine learning and natural language processing literatures are substantially larger, we relied on citation-guided exploration in these communities and selected representative and influential works. We also include selected high-quality arXiv papers when they illuminate important emerging directions.

To ground the survey in the foundations of static analysis, we begin with representative works across major theoretical frameworks and application domains. Their limitations motivate our investigation of neurosymbolic approaches that combine classical static-analysis techniques with LLM capabilities. Overall, our goal is to provide broad coverage of both recent neurosymbolic code-auditing research and the classical static-analysis literature, while acknowledging that several relevant work may nevertheless have been omitted. The raw data collected for the survey, including venue `.bib` files, HTML pages containing accepted-paper

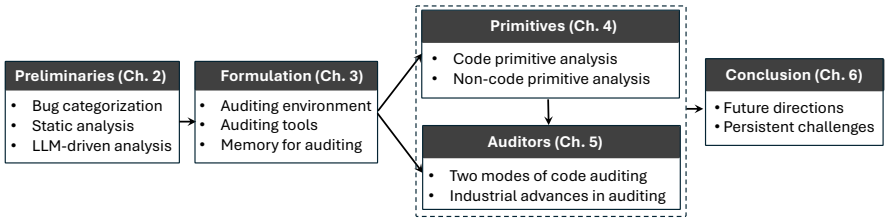

Figure 1.5: The organization of the survey

lists, and paper-classification scripts, are available online[2].

The remainder of this survey is organized as a progressive argument, which is demonstrated by Figure 1.5. Chapter 2 establishes the problem setting by revisiting the foundations of code auditing and common bug categories, examining why formally rigorous static-analysis techniques remain brittle in practice, and explaining why LLM-only approaches are insufficiently reliable for end-to-end auditing. Chapter 3 then formalizes code auditing from an *agentic* perspective, where an auditor analyzes code and non-code artifacts in the environment, augments its observations through tools such as static analyzers, and relies on memory to track evolving hypotheses, partial facts, and verification traces across codebase navigation and task decomposition. Building on this abstraction, Chapter 4 examines how LLMs can perform *primitive analyses* that extract primitive program properties and interpret non-code artifacts, positioning them as bounded-scope components rather than wholesale replacements for classical analyses. Chapter 5 shifts from primitive components to complete systems by surveying neurosymbolic auditing workflows that support or automate key aspects of human auditing. These include systems that simulate *manual auditing* by using LLMs to interpret program semantics, as well as *semi-automatic auditing* in which LLMs help customize and configure static analyzers and interpret their results. Finally, Chapter 6 synthesizes the survey around three coupled bottlenecks, including *observation*, *verification*, and *memory*, and reframes neurosymbolic auditing as an engineering discipline aimed at repeatedly producing *stable, actionable* security conclusions under real-world constraints. It closes by outlining future research directions,

[2] https://github.com/PurCL/ASE

including auditability-by-design interfaces, cost-aware orchestration, evidence-prioritized remediation loops, ecosystem-level auditing across configurations and dependencies, the translation of neural reasoning into reusable artifacts with explicit lifecycles and uncertainty, and the security of the auditor itself in adversarial environments.

2

Code Auditing and Its Challenges

In this survey, we consider bug detection the primary application scenario for code auditing, where the audited properties facilitate security analysis and inform subsequent actionable protection measures. To establish the necessary background, we first classify bugs according to the taxonomy defined by the Common Weakness Enumeration (CWE, 2025), whose categories have been extensively studied by the research community over the past several decades. We then revisit existing code auditing techniques, encompassing both classical static analysis and LLM-driven analysis, and examine their respective strengths and limitations. Finally, we introduce the key insights underlying neurosymbolic code auditing, which motivate the remaining chapters of this survey.

2.1 Bug Categorization

Bug categorization provides a fundamental lens for understanding both the challenges of code auditing and the strategies needed to address them. By categorizing bugs according to the relevant program expressions they involve and the associated semantic properties, we identify four major bug classes, namely numeric bugs, memory bugs, taint-style bugs, and functional bugs. This taxonomy highlights the diversity of reasoning

tasks involved in effective auditing and provides a basis for assessing how well existing techniques support different bug classes.

2.1.1 Numeric Bugs

Numeric bugs arise when arithmetic computations operate outside valid numeric domains or violate developer-intended numeric contracts, leading to exceptions, incorrect results, or security-relevant program states. Representative examples include divide-by-zero errors (Guo *et al.*, 2022), out-of-range conversions, integer overflow and underflow (Pomonis *et al.*, 2014; Dietz *et al.*, 2012), and floating-point errors. Floating-point bugs are especially subtle because floating-point numbers only approximate real values with finite precision. As a result, rounding errors and error amplification can cause computations to deviate from the intended mathematical behavior, as observed in machine learning systems (Zhang *et al.*, 2020). The core reasoning task in numeric bug detection is therefore to determine whether program values remain within valid and intended numeric ranges across arithmetic operations, type conversions, and control-flow paths.

Notably, numeric anomalies such as overflow and underflow are not intrinsically erroneous (Dietz *et al.*, 2012). Their status depends on both language-defined semantics and the developer's intent. For instance, signed integer overflow may constitute undefined behavior in several programming languages, whereas unsigned overflow or modular arithmetic may be well defined in others. Even when wrap-around behavior is well defined, it may be either intentional, as in hashing and checksum computations, or unintended, as in allocation-size calculations and bounds checks. Consequently, the mere presence of a numeric anomaly is insufficient to establish a bug. The anomaly must be interpreted in relation to the applicable execution semantics and the numeric contract expected by the surrounding program.

For these reasons, detecting numeric bugs requires reasoning that is both *semantics-aware* and *intent-aware*. On the semantics side, an auditor must track value ranges, path conditions, and floating-point error propagation with sufficient precision to determine whether an operation may exceed representable bounds, trigger exceptional behavior, or

accumulate round-off error relative to the intended real-number computation. Classical analyses address these checks using numerical abstract domains, such as affine equalities, intervals, polyhedra, octagons, and related relational domains (Miné, 2002; Miné, 2017; Jeannet and Miné, 2009), or using symbolic execution with constraint solving (Godefroid *et al.*, 2005; Sen *et al.*, 2005; Cadar *et al.*, 2008). For floating-point programs, the analysis must additionally account for the gap between finite-precision execution and the idealized real-number computation, including rounding, cancellation, and error amplification (Monniaux, 2008; Goubault and Putot, 2011; Darulova and Kuncak, 2014). On the intent side, the auditor often needs contextual evidence, such as type annotations, surrounding bit-level idioms, and other specifications, to determine whether wrap-around, truncation, or precision loss is intentional and acceptable. However, most existing code auditing techniques focus primarily on language-defined numeric semantics, flagging potential overflows or exceptional cases on the basis of abstract execution alone, while largely neglecting intent-aware distinctions. As a result, these techniques tend to over-report intentional computations.

2.1.2 Memory Bugs

Memory bugs arise when programs misuse memory objects or violate the spatial, temporal, or lifetime assumptions required for safe memory management. They are a major source of reliability failures and security vulnerabilities in low-level programs, especially C/C++ systems software. Representative examples include null-pointer dereferences (Xie *et al.*, 2003), spatial errors such as out-of-bounds reads and writes (Austin *et al.*, 1994; Nagarakatte *et al.*, 2009), temporal errors such as use-after-free and double-free bugs (Nagarakatte *et al.*, 2010; Serebryany *et al.*, 2012), and memory leaks caused by missing deallocation (Hastings and Joyce, 1992; Cherem *et al.*, 2007; Sui *et al.*, 2012; Fan *et al.*, 2019). Such bugs can lead to data corruption, information disclosure, or arbitrary code execution, and are therefore frequently exploited in real-world attacks. As a result, memory errors remain a persistent attack surface in real-world systems (Szekeres *et al.*, 2013).

From an auditing perspective, memory bugs require reasoning about

the validity of memory locations and the legality of operations performed on them. An auditor must determine, for instance, whether an object has been properly allocated and initialized, which pointers or references may refer to that object, and whether the object remains live at each use site. Accordingly, auditing for memory bugs requires collecting low-level primitive facts about pointer-related program behavior, including *points-to facts*, *aliasing facts*, and the locations of allocation, deallocation, and dereference operations. Section 4.1.1 discusses how LLM-driven pointer analysis can infer such pointer-related facts and thereby provide a foundation for memory-bug detection.

Out-of-bounds accesses are a particularly important subclass of memory bugs because they blur the boundary between memory reasoning and numeric reasoning. Although they manifest as memory bugs, their root causes often lie in incorrect index computations, erroneous pointer arithmetic, or inconsistent bounds checks. Consequently, evidence for an out-of-bounds access must connect two classes of facts, namely facts about the target memory object and its size, and numeric facts about indices and offsets. Importantly, out-of-bounds accesses are not a uniform class of local access errors, but rather a broad and heterogeneous family of bugs spanning both the spatial and temporal dimensions of memory safety. For such challenging memory bugs, auditors typically need to analyze pointers and numeric values precisely, because the two domains are often deeply intertwined. Section 4.1.4 presents the details of LLM-driven invariant generation, which can derive expressive program invariants over values from multiple domains, including both pointer and numeric domains.

2.1.3 Taint-Style Bugs

Taint-style bugs arise when data crosses implicit trust boundaries without adequate validation, sanitization, or authorization checks. In the common case, untrusted or adversarial data propagates from external inputs to security-sensitive operations. Conversely, confidential data may also flow to publicly observable outputs without proper protection. Such data, commonly referred to as *tainted values*, may originate from user inputs, network messages, file systems, or inter-process commu-

nication channels. Typical instances include command injection (Su and Wassermann, 2006), cross-site scripting (Su *et al.*, 2023), and SQL injection (Gowtham and Pramod, 2021). These bugs allow attackers to influence control flow, data access, or resource usage in unintended ways, often leading to privilege escalation, information disclosure, or malicious command execution. Unlike memory bugs, taint-style bugs usually do not cause immediate runtime failures. Instead, they compromise security guarantees by violating implicit trust boundaries and input validation assumptions embedded in the program.

Statically detecting taint-style bugs is typically formulated as a data-flow problem rather than as a classification of tainted values. An auditor must determine how tainted values are preserved, transformed, or terminated across possible executions, and whether the resulting flow violates the program's intended trust boundaries. In real-world codebases, this task requires three supporting analyses.

First, to handle real-world programs, tainted values often need to propagate through mutable heap state and indirect memory accesses. For example, tainted values may be written into a field, array element, or heap object through one reference and later read through another aliased reference. Such flows are implicit with respect to syntactic variable names, and a def-use analysis that ignores aliasing may fail to connect the write and the read. Therefore, a precise taint analysis often relies on *pointer analysis* to identify points-to and aliasing relationships relevant to taint propagation. Missed aliases lead to false negatives, while spurious aliases inflate the set of tainted values. Section 4.1.1 discusses how neurosymbolic analysis can reason about pointer properties, including aliasing.

Second, taint-style bugs in real applications commonly depend on complex control flow. The propagation of a tainted value may be distributed across different functions, modules, framework callbacks, and library boundaries. Moreover, it can be governed not only by direct function calls but also by virtual dispatch, asynchronous callbacks, event-driven handlers, and other implicit control-transfer mechanisms. In languages where such bugs are prevalent, such as Java and JavaScript, these features obscure which code may execute and in what order, and hence which paths may carry tainted values. Therefore, taint analysis

requires precise interprocedural control-flow analysis. Call graph analysis is a central component of this reasoning because it approximates the caller-callee structure over which tainted values are propagated. In object-oriented programs, precise call graph construction further depends on points-to analysis to resolve dynamic dispatch. Imprecise control-flow or call graph analysis introduces spurious paths or caller–callee edges that cause tainted values to propagate along infeasible paths, inflating both the analysis cost and the false-positive rate. Section 4.1.3 surveys LLM-aided call graph analysis, a key instance of control-flow reasoning for improving the precision of taint-style bug detection in such settings.

Third, even when a candidate flow path is recovered, determining whether it constitutes a bug depends on the security meaning of the APIs and operations along that path. Auditors must know which program inputs should be treated as sources of untrusted data, which operations are security-sensitive sinks, and which checks or transformations neutralize taint. Therefore, accurate *taint specifications* are indispensable (Chibotaru *et al.*, 2019). Among these specifications, sanitizer identification is especially challenging. Unlike sources and sinks, which often follow recognizable API patterns, sanitization logic is typically application-specific. The absence of comprehensive sanitization specifications is a major contributor to false positives in practice, as legitimate flows that pass through custom validation are incorrectly flagged. Section 4.2.1 presents neurosymbolic techniques for inferring such specifications from documentation and other non-code artifacts.

In this sense, taint-style bugs are closely related to certain memory bugs, such as null pointer dereferences and use-after-free errors, because they all require reasoning about specific forms of data flow. In general, detecting such bugs benefits from pointer analysis, control-flow analysis, and specification inference, but the main difficulties differ because the relevant program features and bug semantics differ. For example, in memory-bug detection, specification inference often focuses on identifying memory-management functions rather than inferring sanitization specifications. Similarly, in C/C++ programs where memory leaks and other memory-management errors are common, function pointers and the induced indirect calls are often the central targets of call graph

analysis. Thus, the same primitive analyses support both taint-style and memory-bug auditing, while their concrete bottlenecks and specification targets are determined by the bug class under consideration.

2.1.4 Functional Bugs

Unlike the three bug classes discussed above, functional bugs are not primarily violations of low-level safety or security properties. Instead, they arise when an implementation deviates from its intended semantics, i.e., the functional-correctness specification of what a program, component, or service is expected to compute or guarantee. Such intent can be expressed explicitly at multiple abstraction levels. At the local program level, functions, methods, and classes may be specified using assertions, pre-/post-conditions, invariants, behavioral contracts (Meyer, 1997), and refinement or liquid types (Rondon *et al.*, 2008). At the application and service level, intent may be captured by domain rules, data-integrity constraints (Huang *et al.*, 2023a), and performance or latency expectations (Yang *et al.*, 2018). Such intent may also remain implicit in design assumptions and business logic.

A functional bug is a violation of these intended properties. At lower abstraction levels, functional bugs include incorrect return values, violated object invariants, missing precondition checks, or state updates that fail to establish required postconditions. At higher abstraction levels, they manifest as violations of domain or business logic, such as smart-contract logic vulnerabilities (Sun *et al.*, 2024), missing database integrity constraints (Huang *et al.*, 2023a), performance anti-patterns that violate expected response-time behavior (Yang *et al.*, 2018), and configuration or coordination mistakes that degrade service availability. In practice, functional bugs often manifest as service-level failures rather than localized faults. For instance, large-scale outages can be triggered by subtle errors in DNS resolution, configuration logic, or coordination protocols. The programs involved may remain memory-safe and type-correct, yet still violate higher-level assumptions about availability and system behavior. These cases illustrate that functional bugs can have severe real-world impact despite leaving classical safety properties intact.

This taxonomy should not be interpreted as a strict partition of bug

types. Rather, it highlights the dominant semantic property that makes each class difficult to audit. A bug may simultaneously involve low-level safety, security, and functional correctness concerns. For example, several numeric bugs, such as numerical instability in machine learning systems or scientific computing programs, may ultimately manifest as incorrect output values and hence as failures to implement the intended functionality. Similarly, a memory or taint-style bug may become functional when its externally visible consequence is a violation of service behavior, data integrity, or application-specific requirements. Functional bugs therefore provide a broad category for reasoning about intent violations, while still overlapping with the other bug classes when their effects are interpreted at the level of program functionality.

Detecting functional bugs requires reasoning over a broad and heterogeneous set of program properties, often spanning data flows within code, configuration files, and interactions with external systems. While violations of well-specified assertions or contracts can, in principle, be checked using conventional verification techniques, many functional bugs arise precisely because such specifications are incomplete, implicit, or informal. Consequently, *specification inference* becomes a central prerequisite. Auditors must recover the intended behavior, invariants, environmental assumptions, and domain-specific constraints against which the implementation should be judged. Such specifications are often scattered across design documents, comments, and commit messages, requiring auditors to reason over natural-language artifacts in addition to code. We discuss neurosymbolic techniques for extracting and structuring such specifications from non-code artifacts in Section 4.2. As a result, functional bug detection is challenging not because the resulting properties are inherently unverifiable, but because it demands integrated reasoning over diverse and often informal sources of intent, making it one of the most complex and least standardized areas of automated code auditing.

2.2 Static Analysis and Its Challenges

This section discusses several representative static analysis techniques that have been applied to code auditing in real-world software systems.

We organize the discussion around four representative automated static analysis techniques, namely symbolic execution, abstract interpretation, data-flow analysis, and pattern-matching-based analysis. For each category, we present an overview of its methods, summarize representative studies, and highlight the classes of bugs for which it is most effective, following the taxonomy introduced in Section 2.1.

2.2.1 Static Analysis Techniques

Symbolic execution is a static analysis technique that systematically explores execution paths by treating program inputs as symbolic variables rather than concrete values (King, 1976; Baldoni *et al.*, 2018; Boyer *et al.*, 1975; Howden, 1977). Along each explored path, the analysis accumulates branch conditions and encodes the semantics of program statements as logical constraints, thereby constructing symbolic states that represent program memory under these constraints. To assess whether a bug may occur at a particular program location, the triggering condition of the bug is conjoined with the corresponding symbolic state. Symbolic execution is well-suited for detecting bugs that manifest under specific input conditions, such as null pointer dereferences, buffer overflows, and divide-by-zero errors, by systematically exploring path conditions and solving constraints to generate triggering inputs. Tools such as KLEE (Cadar *et al.*, 2008) and Symbolic PathFinder (Havelund and Pressburger, 2000) have demonstrated effectiveness in automatically uncovering deep semantic errors in real-world software systems. Despite its precision and capacity to reason about complex execution paths, symbolic execution is fundamentally limited by the path explosion problem, which severely restricts its scalability. To address this challenge, a variety of techniques have been developed, including heuristic search strategies (He *et al.*, 2021), state merging (Kuznetsov *et al.*, 2012; Torlak and Bodík, 2014; Porncharoenwase *et al.*, 2022), compositional symbolic execution (Lööw *et al.*, 2025), parallelism (Wei *et al.*, 2023; Bucur *et al.*, 2011), and compilation (Wei *et al.*, 2020; Wei *et al.*, 2023; Poeplau and Francillon, 2020). Typically, symbolic execution can explore only partial program paths, especially in the presence of loops and recursive calls, yielding an *under-approximated* view of program behavior.

Abstract interpretation provides a principled framework for soundly approximating program semantics through the design of abstract domains and abstract transformers over program operations (Cousot and Cousot, 1977a; Cousot and Cousot, 2014). In contrast to symbolic execution, which reasons about a subset of feasible paths, abstract interpretation computes abstract states that collectively over-approximate the set of all reachable concrete states. This over-approximation enables analyzers to establish the absence of certain classes of bugs when no error state is reachable in the abstract semantics. It has been applied to a broad range of bug classes, including numeric errors such as divide-by-zero, memory bugs such as null pointer dereferences and buffer overflows, and selected forms of resource mismanagement such as file-descriptor leaks. Industrial analyzers such as Astrée demonstrate the practical impact of abstract interpretation (Cousot *et al.*, 2005), especially in safety-critical settings where soundness guarantees are central. The main challenge lies in balancing precision, scalability, and expressiveness. Coarse abstractions scale well but may introduce many spurious alarms, whereas precise abstractions can be computationally expensive and difficult to apply to large programs. Consequently, practical abstract interpreters often rely on domain-specific abstractions, widening/narrowing strategies, and refinement mechanisms tailored to particular properties or program classes. Techniques such as demand-driven abstraction refinement further improve this balance by selectively increasing precision only where it is needed to discharge alarms or prove target properties (Stein *et al.*, 2019).

Data-flow analysis (Kildall, 1973) comprises a broad family of program analysis techniques for reasoning about how values and program states propagate across instructions and procedures (Reps *et al.*, 1995; Sagiv *et al.*, 1996; Rountev *et al.*, 2008; Späth *et al.*, 2017). A data-flow analysis typically maintains a set of program facts and updates them according to the semantics of each instruction, often expressed through Gen/Kill functions that specify which facts are produced or invalidated. Classical instances include reaching definitions, constant propagation, and live-variable analysis, which have long served as core components of compiler optimization. Beyond these foundational analyses, specialized

forms of data-flow reasoning have been widely used for security and correctness checking. Taint analysis (Arzt *et al.*, 2014) tracks whether values derived from untrusted sources can reach security-sensitive sinks, thereby identifying potential injection vulnerabilities and various other forms of taint-style bugs. Typestate analysis (Fink *et al.*, 2006) instead reasons about whether objects are used according to valid operation protocols, such as ensuring that files are closed after being opened or that allocated resources are eventually released. A central practical challenge in precise data-flow analysis is pointer reasoning. Because data-flow facts often depend on aliasing relationships, many analyzers adopt layered designs in which points-to or alias information is computed before the target analysis (Sui and Xue, 2016). However, this design can lead to the so-called "pointer trap" (Shi *et al.*, 2018), i.e., improving pointer precision can substantially improve downstream data-flow accuracy, but the pointer analysis itself may dominate the overall cost. This creates a persistent tension between precision and scalability. To mitigate this tension, recent techniques employ demand-driven analysis, which computes aliasing or data-flow facts only when they are needed (Späth *et al.*, 2016), or exploit parallel and incremental design to scale data-flow analysis to large programs (Liu *et al.*, 2019).

Pattern-matching-based analysis detects bugs by searching specific syntactic or semantic patterns upon program representations, such as abstract syntax trees (ASTs) and control-flow graphs (CFGs). Tools such as PMD (PMD Project, 2026) and FindBugs/SpotBugs (SpotBugs Community, 2026) exemplify this family of techniques. Specifically, PMD applies extensible rules over source-level ASTs, whereas FindBugs and its successor SpotBugs identify recurring bug patterns in Java bytecode. Mainstream integrated development environments (IDEs), including JetBrains products, also provide structural-search facilities and inspection rules that can capture common coding idioms and recurring bugs. Compared with semantics-based analyses such as symbolic execution, abstract interpretation, and data-flow analysis, pattern-matching-based techniques are tailored to specific syntactic patterns, yet are lightweight, efficient, and easy to deploy. In principle, rules can be designed to target concrete patterns from any of the bug classes discussed above,

including numeric, memory, taint-style, and functional bugs. In practice, however, such rules often cover highly specific cases, and their effectiveness is bounded by the coverage, expressiveness, and quality of the predefined templates. Pure pattern matching is especially limited because it generally lacks deep interprocedural semantic reasoning about aliasing, path feasibility, and data dependencies. Consequently, pattern-matching-based approaches may produce substantial false positives when pattern matches do not correspond to semantically feasible bugs. They are therefore best viewed as practical complements to more principled program analyses, since they can efficiently detect common and recurring bugs but are less suitable for discovering bugs whose manifestation depends on complex semantic context.

Summary. The four categories of static analysis techniques discussed above collectively constitute the methodological foundation of automated code auditing. Each technique entails a distinct balance among precision, recall, efficiency, and scalability. Symbolic execution offers fine-grained, per-path reasoning and high precision, enabling the detection of deep semantic errors, although it generally under-approximates program behaviors due to the path explosion problem. Abstract interpretation, in contrast, provides sound over-approximation guarantees and has demonstrated industrial success in safety-critical domains, albeit at the cost of precision-efficiency trade-offs. Data-flow analysis occupies a versatile middle ground, supporting both classical compiler optimizations and advanced security analyses. Its effectiveness, however, is tightly coupled with the precision of pointer analysis, which often introduces significant computational overhead. Pattern-matching-based analysis, while lightweight and efficient, is limited to recurring, rule-specified patterns and therefore serves as a complement rather than a substitute for more principled analyses. Additionally, other classes of program analysis techniques, such as fuzzing and dynamic testing, fall outside the scope of this survey. These approaches rely on program execution to explore runtime behaviors and therefore differ fundamentally from our focus on practical static analysis methods that operate without program execution and are designed for scalable bug detection across real-world software systems.

Table 2.1 lists a representative set of static analysis tools, categorized by their underlying techniques and supported programming languages. In practice, industrial analyzers are rarely based on a single technique. For example, Coverity integrates abstract interpretation with symbolic execution and data-flow analysis for bug detection (Bessey *et al.*, 2010), illustrating that real-world static analyzers are fundamentally hybrid systems rather than instantiations of any single theoretical framework.

2.2.2 Common Challenges of Static Analysis

While static analysis has achieved significant progress, existing techniques continue to encounter fundamental limitations that hinder their effectiveness and widespread adoption. Well-known challenges, such as limited precision, poor scalability, and substantial analysis overhead, have been extensively studied in both academic research and industrial practice. Beyond these established concerns, however, several overlooked yet equally critical barriers remain unsolved. In particular, we emphasize three underappreciated obstacles: (1) intermediate representation reliance, (2) limited customizability, and (3) semantic barrier. These challenges profoundly impact the applicability and usability of static analysis tools in real-world software engineering environments.

Intermediate Representation (IR) Reliance. As demonstrated in Table 2.1, most static analysis techniques operate on a specific intermediate representations, such as the Jimple IR for Java programs (Bodden, 2012) or the LLVM IR for C/C++ programs (LLVM Project, 2026). Specifically, IRs provide normalized and structured representations of programs. They are typically used as internal data structures by compilers, eliminating unimportant syntactic details while preserving essential semantics. By operating on uniform IRs rather than unstructured textual source code, analysis can abstract away language-specific syntax and focuses on semantic properties relevant to bug detection. For instance, Soot performs analyses over the Jimple IR (Bodden, 2012), and Meta Infer intercepts compilation commands to extract the IR required for its checks (Facebook, 2022).

However, the requirement for IRs also introduces fragility into large-

Tool	Method	Language	IR Reliance	Customizability
KLEE	SymEx	C, C++	Yes	No
Symbolic PathFinder	SymEx	Java	Yes	No
Clang Static Analyzer	SymEx	C, C++	Partial	Partial
Astrée	AbsInt	C, C++	Yes	No
Infer	AbsInt	Multiple	Yes	No
Frama-C/Eva	AbsInt	C	Partial	No
FlowDroid	DFA	Java	Yes	No
SVF	DFA	C, C++	No	No
Pinpoint	DFA	C, C++	Yes	No
FindBugs/SpotBugs	PM	Multiple	Partial	Yes
PMD	PM	Multiple	No	Yes
Semgrep	DFA, PM	Multiple	No	Yes
CodeGuru	DFA, PM	Java, Python	Partial	No
CodeQL	DFA, PM	Multiple	Partial	Partial
Coverity	Hybrid	Multiple	Partial	Partial

Table 2.1: A comparison of representative static analysis tools. SymEx, AbsInt, DFA, and PM stand for symbolic execution, abstract interpretation, data-flow analysis, and pattern-matching-based analysis, respectively. *IR Reliance* indicates whether the tool relies on IRs beyond the AST. *Customizability* denotes the extent to which developers can define detectors for project-specific bugs without substantially modifying the analyzer. "Yes" indicates user-level customizability. "Partial" indicates customizability that requires substantial expertise. "No" indicates that the tool primarily targets fixed families of bugs.

scale and evolving software ecosystems. First, most IRs, except for ASTs, cannot be easily obtained during early stages of development, causing analyzers to miss opportunities to surface bugs in earlier, incomplete versions. Second, this requirement complicates analysis across diverse build systems and compilers. More critically, many analyzers are compatible only with specific IR formats, making them highly sensitive to compiler versions and build systems. When projects adopt newer compilers that emit updated IR formats, as in recent Linux kernel releases, existing tools may fail due to incompatibility. This phenomenon, often referred to as the IR version trap (Zhang *et al.*, 2024), poses a significant obstacle to the longevity and portability of static analysis tools. Finally, the rigid dependency on IRs also impedes the auditing of rapidly evolving or automatically generated code, such as AI-suggested code fragments. Although recent research has begun to explore strategies for loosening

this dependency, for example by reconstructing partial or incomplete programs into analyzable forms, mainstream analyzers remain brittle, with their effectiveness tightly coupled to the successful generation of the required IR.

Limited Customizability. Another major limitation of static analysis is its limited customizability. Existing analyzers often lack mechanisms that allow users to adapt analyses to new bug classes or project-specific semantics without substantial engineering effort. As discussed in Section 2.1, different bug classes impose distinct reasoning demands and require different forms of evidence. Real-world projects further specialize these demands through programming languages, frameworks, APIs, and coding practices, making project-specific customization essential. For example, the MISRA C++ standard enumerates a broad spectrum of bugs ranging from undefined behavior in pointer arithmetic to subtle resource management errors (MISRA, 2008), highlighting the diversity of bug patterns even within a single language. Yet most analyzers are built around fixed rule sets or narrowly scoped property families. FlowDroid, for instance, is primarily designed for taint-style bugs (Arzt *et al.*, 2014), whereas Meta Infer targets memory errors such as buffer overflows (Facebook, 2022).

Extending analyzers to new bug patterns typically requires custom rules or plugins, which in turn demand expertise in compiler infrastructures, IRs, and domain-specific rule languages. As a result, developers must either design new analyzers from scratch or substantially modify existing ones. Empirical evidence shows that more than 70% of developers who have used static analyzers express dissatisfaction with the limited support for desired project-specific adaptations (Johnson *et al.*, 2013). Although recent tools such as CodeQL provide query-based customization (Avgustinov *et al.*, 2016), developers must still master a domain-specific query language (CodeQL, 2024) and a large API ecosystem, thereby introducing a steep learning curve. Other tools, such as FlowDroid, SVF, and Pinpoint, allow developers to customize dataflow specifications, while such customization mainly varies parameters within an existing analysis rather than enabling developers to define new semantic properties across diverse bug classes.

Semantic Barrier. Another fundamental limitation of existing static analysis techniques lies in their requirement for explicit semantic modeling, often in the form of formal and well-defined semantics. Specifically, most analyzers depend on two prerequisites. The first is a formal model of bug-triggering conditions, which specifies the logical specification under which a bug may manifest, such as an arithmetic operation exceeding representable bounds, a dereference reaching a null or freed object, an untrusted value reaching a sensitive sink, or an implementation violating an intended functional property. The second is a formal semantics of program behavior, which defines the meaning (e.g., operational semantics) of program constructs, such as statements, instructions, or even APIs and libraries. The well-defined semantics indeed allow analyzers to collect and check evidence, including numeric ranges, aliasing and lifetime facts, data-flow paths, and inferred contracts. Collectively, these two components determine the semantic reasoning capacity of static analysis techniques.

In practice, however, both requirements can be difficult to fulfill. Many bug-triggering conditions are complex to formalize even for human experts, especially for functional bugs whose correctness criteria are informal or domain-specific (Zheng *et al.*, 2025a; Guo *et al.*, 2025a). Semantics integrity is also impeded by incomplete or unavailable code: A common example is the absence of third-party library implementations, which prevents tools from automatically modeling the semantics of external function calls. Although manually written library specifications are a widespread workaround, this process is labor-intensive, error-prone, and requires substantial domain expertise. Together, these limitations constrain the semantic understanding that existing auditing techniques can achieve, leaving many program bugs, particularly those involving complex logic or hidden dependencies, beyond their reach.

2.3 LLM-driven Code Analysis and Its Challenges

Recent advances in LLMs have reshaped how code auditing tasks can be approached. Rather than relying exclusively on explicitly engineered abstractions, LLM-driven code analysis explores the use of neural models to infer semantic properties from source code, documentation, and other

software artifacts. This capability is particularly relevant to the limited-customizability and semantic-barrier challenges discussed above, because many auditing tasks require project-specific evidence and informal intent that are difficult to encode manually. In this section, we first review LLM-driven code analysis techniques and then discuss the common challenges that limit their accuracy and scalability in practice.

2.3.1 LLM-driven Code Analysis Techniques

Early efforts toward LLM-driven code analysis focused on incorporating program structure into neural representations through pretraining. Models such as GraphCodeBERT extend language-model pretraining with explicit representations of program graphs (Guo *et al.*, 2020), enabling them to capture syntactic structure as well as semantic relations such as data-flow and control-flow edges among program elements (Chen *et al.*, 2024). By exposing models to these structured signals during pretraining, such approaches demonstrate that neural models can learn to predict nontrivial program properties. In parallel, reinforcement learning has been explored to guide models toward capturing additional semantic attributes, further strengthening the connection between learned representations and program semantics.

Building on these foundations, more recent work has shifted toward leveraging general-purpose, decoder-only LLMs for program analysis via prompting (Pei *et al.*, 2023; Wang *et al.*, 2024c; Cheng *et al.*, 2025). Instead of post-tuning models with task-specific supervision, these approaches rely on carefully designed prompts to elicit analytical behavior at inference time. Notably, chain-of-thought prompting enables LLMs to articulate intermediate reasoning steps, allowing them to simulate aspects of program execution, reason about control flow, and track variable states across code fragments (Fang *et al.*, 2024a). Through such prompting strategies, LLMs have been applied directly to source code for bug detection.

Compared with classical static analysis, LLM-driven code analysis offers several practical advantages. First, it does not require intermediate representations, making it applicable to evolving codebases during development. Second, analysis logic can be customized through natural-

language prompts, significantly lowering the barrier to adapting analyses to new bug patterns or property domains. Third, because LLMs are pretrained on large and diverse code and text corpora, they possess rich prior knowledge about programming idioms, common libraries, and API usage, enabling them to reason across the semantic barrier that is difficult for static analyzers to cross, such as calls to external libraries or undocumented framework behavior. These strengths make LLM-driven analysis a compelling complement to static analysis, while also raising important questions about its limitations, which we discuss next.

2.3.2 Common Challenges of LLM-driven Code Analysis

Despite these encouraging capabilities, LLMs alone are far from sufficient for effective code auditing. Their underlying neural architecture, training, and inference paradigm are fundamentally misaligned with several core requirements of code auditing. In what follows, we illustrate three representative failure modes that systematically limit the effectiveness of LLM-based code analysis in realistic settings.

Sensitivity to Nonlinear Program Structure. Unlike textual natural language, program behaviors are dynamic and induced by nonlinear/nonlocal control-flow and data-flow structures. Auditing tasks such as detecting resource leaks, null pointer dereferences, or use-after-free errors require reasoning about the reachability of specific program points and the propagation of values along particular execution paths. However, LLMs process code primarily as linear token sequences and do not explicitly construct or reason over CFGs or data-flow graphs (DFGs). As a result, they are prone to implicitly assuming "typical" execution patterns from the pretaining dataset rather than faithfully tracking feasible paths.

This limitation is illustrated by the example in Figure 2.1, where a `goto`-based error-handling structure causes execution to bypass resource deallocation. Correctly identifying the resulting memory leak requires recognizing that the jump skips the `free` statement on that particular path. In contrast, an LLM may hallucinate a more benign control flow in which the deallocation is executed, simply because such patterns are

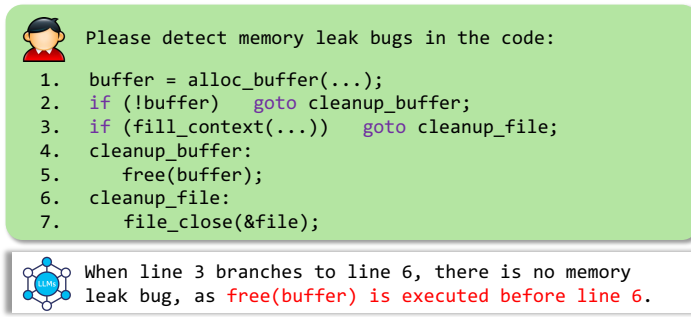

Figure 2.1: A false negative when analyzing multiple `if` and `goto` statements

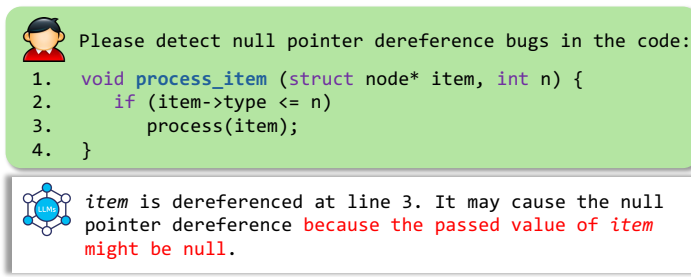

Figure 2.2: A false positive caused by the lack of calling context

common in similar code. This tendency leads to false negatives. Real bugs are missed because the model implicitly repairs or simplifies the control-flow structure in its internal reasoning.

Dependence on Missing Calling Contexts. As stated in Section 2.1, auditing complex properties requires interprocedural reasoning of the program. Whether a pointer can be `NULL`, whether a buffer index is in bounds, or whether a resource has been initialized often depends on how a function is called, the values of global variables, and protocol-level invariants that are not visible within a single function body. Human auditors routinely consult call sites, documentation, and usage patterns to understand such preconditions.

In contrast, LLMs are typically prompted with isolated code fragments. When crucial calling contexts are absent, the model cannot distinguish between values that are unconstrained and values that are

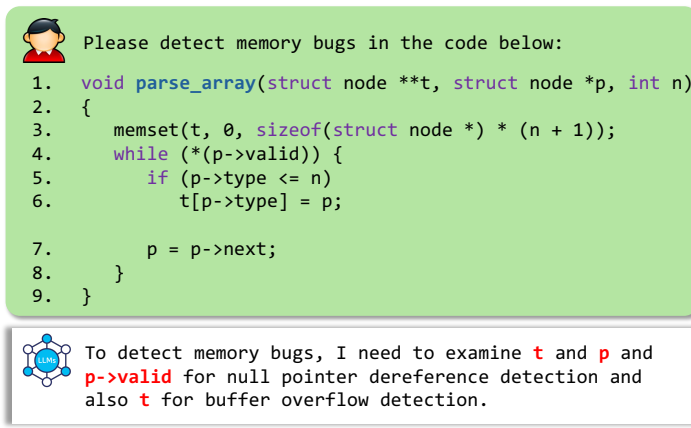

Figure 2.3: The incomplete reasoning caused by insufficient test-time compute

constrained by external guarantees. As shown in Figure 2.2, the pointer `item` is dereferenced inside `process_item`, but its nullability is determined entirely by its callers. Without access to these callers, an LLM tends to treat `item` as potentially null and will often report a null pointer dereference even when none can occur in any feasible execution. Such context-insensitive reasoning systematically produces false positives, undermining the practical usefulness of LLM-based analysis.

Insufficient Test-time Compute for Multi-fact Reasoning. Realistic auditing tasks rarely involve a single, isolated risk. A single function may simultaneously manipulate multiple pointers, indices, and buffers, each governed by different constraints that interact in subtle ways. Accurately reasoning about such code requires maintaining and updating several interdependent invariants, such as the relationships among array sizes, loop bounds, pointer validity, and data structure topology.

Although modern LLMs support extended context windows, their test-time compute remains limited. When faced with code such as the example in Figure 2.3, the model must track the values of `t`, `p`, `p->valid`, and `p->type`, while simultaneously reasoning about array bounds and pointer dereferences. In practice, the model tends to allocate its limited reasoning budget unevenly, often focusing on a subset of potential issues

while neglecting others. Consequently, some real bugs are overlooked, while other potential risks are conservatively over-approximated. This imbalance leads to both false negatives and false positives, and cannot be reliably remedied by simply increasing the length of prompts or the number of reasoning steps.

2.4 Summary

This chapter revisits code auditing from two complementary perspectives, namely static analysis and LLM-driven code analysis. Static analysis provides precise, structured, and formal reasoning over program semantics. However, its effectiveness is often constrained by several limitations, such as IR reliance, limited customizability, and the semantic barrier. In contrast, LLM-driven code analysis offers flexibility and adaptability. It can operate without IRs, be customized through natural language descriptions, and leverage rich prior knowledge to reason across incomplete code, external libraries, and non-code artifacts. Unfortunately, its reasoning inherently suffers from instability, hallucination, and the lack of formal guarantees.

These observations indicate that classical static analysis and LLM-driven code analysis should not be regarded as competing alternatives, but rather as complementary techniques whose relative advantages depend on the auditing context. For example, in early-stage development scenarios, where code may be incomplete or difficult to build, LLM-driven analysis can provide valuable semantic insights, while lightweight static analyses can supply structural grounding. Moreover, the ability of LLMs to perform semantic reasoning over multimodal software artifacts can help static analysis overcome the semantic barrier, for instance by interpreting library APIs and system-level specifications. Conversely, when the compilation pipeline is available and stable, static analyses perform deeper semantic reasoning over IRs, reducing the reliance on LLM-based inference for low-level properties such as control flow, aliasing, and numeric properties. Effective code auditing therefore requires selectively combining these techniques based on the constraints and objectives of the analysis scenario.

From a broader perspective, LLM-driven code analysis is closer to

how human auditors inspect code. It leverages an understanding of programming-language semantics and natural-language intent to reason about a restricted scope of code. Static analysis, in turn, provides the auxiliary tooling that human auditors rely on to systematically search for critical program constructs, derive relevant semantic properties, and validate or refute hypotheses with structured evidence. This analogy naturally motivates an *agentic* formulation of code auditing, in which an auditing agent orchestrates LLM-based reasoning and static analyses to emulate human auditing workflows across the software development lifecycle. Here, the agentic formulation does not presuppose full autonomy or a planning-based architecture. Instead, it provides a unifying abstraction for the diverse forms of orchestration observed in practice, including hand-designed fixed workflows and autonomous LLM agents. What makes these designs neurosymbolic in this survey is not merely the coexistence of an LLM and a static analyzer, but the multi-step coordination of heterogeneous tools, artifacts, and memory in an evidence-seeking workflow. Such an agent-centric view provides a unifying framework for understanding and designing diverse instances of neurosymbolic code auditing, which we develop in detail in Chapter 3.

3

Code Auditing From an Agentic Perspective

By examining how human auditors conduct manual code auditing, we observe that auditing proceeds as a multi-step and often iterative reasoning process rather than a single-step reasoning process. Specifically, human auditors analyze multiple bug-relevant program properties and leverage these properties as evidence to determine the presence of bugs. Throughout this process, developers reason over multimodal software artifacts. In addition to source code, they routinely exploit non-code artifacts, such as library documentation, project histories, and community discussions, to infer API specifications, bug patterns, and heuristic strategies for auditing. They also rely on diverse tools, ranging from IDE-based navigation and search to existing auditing tools, to achieve effective code auditing.

Motivated by these observations, this chapter formulates an agentic perspective on code auditing. As shown in Figure 3.1, the auditor interacts with an environment of code and non-code artifacts through tool use and maintains auditing evidence in memory. This abstraction lays the foundation for Chapter 4, which introduces LLM-driven primitive analyses, and Chapter 5, which presents neurosymbolic designs that instantiate these primitives into end-to-end code auditing systems. In

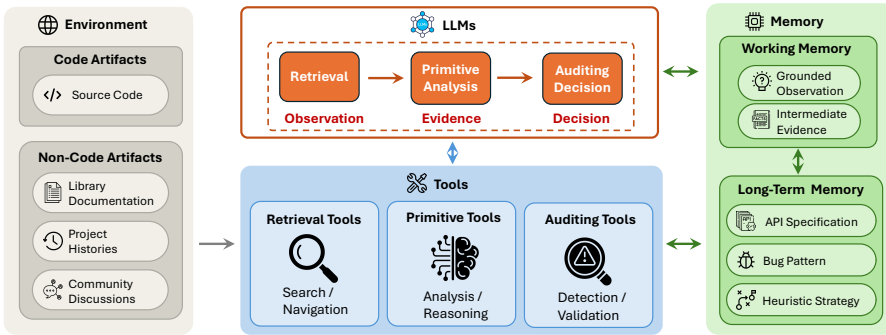

Figure 3.1: The workflow of neurosymbolic code auditing from an agentic perspective

this chapter, we introduce the characteristics of the environment, tool use, and memory for code auditing from an agentic perspective.

3.1 Environment of Code Auditing

Code auditing is inherently situated in a multimodal software environment that extends beyond source code alone. In practice, human auditors reason over a diverse range of software artifacts, which can be broadly categorized as *code artifacts* and *non-code artifacts*. Typically, human auditors directly analyze the source code of a software system during code auditing. Non-code artifacts, including library documentation, project histories (e.g., commit histories, issue reports, and patches), and community discussions (e.g., posts on Stack Overflow), provide complementary contextual information that is often indispensable for grounding auditing decisions in intended semantics and known failure patterns.

These categories of artifacts expose distinct yet complementary *primitive properties* that guide auditing decisions. Source code encodes rich semantic information about program behavior, such as data-flow facts (Bodden, 2012), typestate transitions (Fink *et al.*, 2006), and resource usage patterns (Wang *et al.*, 2023b), which together characterize how the program executes. Library documentation further refines this view by specifying API-level contracts, preconditions, and side effects that govern the correct use of external libraries. Project historical

data, such as patches and issue reports, capture recurring bug patterns, common failure modes, and the causal logic underlying past defects.

From this perspective, code auditing can be viewed as an iterative interaction between the auditor and the surrounding environment. Auditors continuously inspect, analyze, and collect primitive properties from multimodal artifacts, gradually forming evidence to support or refute the presence of bugs. The outcome of this process is not only a bug report but also an accompanying explanation linking derived program properties to violated specifications or known bug patterns. Notably, many of these properties cannot be inferred from an isolated artifact alone. Instead, they emerge only when code, documentation, usage context, and project history are examined together. The multimodal artifacts in the environment enable auditors to cross the semantic barrier imposed by any single artifact, such as source code alone, and thereby audit more effectively. To analyze different kinds of artifacts and extract these primitive properties, auditors rely on a range of analysis tools and maintain intermediate auditing state in memory across analysis steps. The roles of these tools and agentic memory are discussed in Section 3.2 and Section 3.3, respectively.

3.2 Tools for Code Auditing

To audit different forms of software artifacts effectively, human auditors rely on a range of tools to retrieve, inspect, and analyze the portions of the environment that are relevant to a bug candidate. Based on recurring behavioral patterns in manual code auditing, we categorize these tools into three classes, namely *retrieval tools*, *primitive tools*, and *auditing tools*. Specifically, retrieval tools acquire relevant context from artifacts, primitive tools derive semantic properties from retrieved contexts, and auditing tools generate bug reports at a larger scale. Together, they support observation and verification within the broader evidence-seeking workflow and accumulate the evidence needed for final bug detection decisions.

3.2.1 Retrieval Tools

In practice, developers often audit code by manually browsing and navigating a codebase across different stages of software development. As any single artifact exposes only a partial view of the program's behavior, missing context renders an incomplete view of the program's behavior. Retrieval tools are therefore essential for crossing these artifact boundaries and assembling the context required for reliable auditing decisions.

IDEs, such as IntelliJ and VS Code, provide essential navigation and querying capabilities that allow human auditors to retrieve precise program fragments on demand. Commonly used functionalities include *Definition Lookup*, which retrieves declarations of functions, types, and variables to establish their scope and intended role, *Call Graph Navigation*, which enables traversal between callers and callees to expose interprocedural dependencies, *Use-Site Analysis*, which identifies all references to a variable or function and supports reasoning about data-flow facts and object lifecycles, and *Textual and Structural Search*, which allows auditors to locate suspicious constructs using keyword-based or syntactic patterns. At a lower level, many of these capabilities are exposed through the Language Server Protocol (LSP) (Gunasinghe and Marcus, 2021), which provides a uniform interface for querying program structure and semantic information. As a result, any LSP-compliant implementation can serve as a retrieval tool, enabling agents to collect relevant program constructs and surrounding contexts in a manner analogous to IDE-assisted auditing.

Beyond local IDE support, *codebase-wide search platforms*, such as GitHub code search or Sourcegraph (Sourcegraph, Inc., 2026), enable human auditors to retrieve relevant code and documentation across large codebases. External web search plays a similar role for non-code artifacts when auditors need library documentation, issue discussions, bug reports, or usage examples that reside outside the local codebase. *Version control systems* further enrich retrieval by providing historical context. Commands such as `git blame` and `git log` allow auditors to trace the evolution of suspicious code, revealing when it was introduced, how it has changed over time, and whether it has been associated with

previous bug fixes or regressions.

Modern coding agents operate in a closely related way. Systems such as Claude Code (Anthropic, 2026a) and Codex (OpenAI, 2026a) typically do not reason over an entire codebase at once. Instead, they iteratively issue retrieval actions within the environment, including grep-like text search, file-system traversal, targeted file reads, version-control queries, and, when necessary, web retrieval for external documentation. In particular, these agents often depend on customized grep-like searches over code to quickly surface candidate definitions, call sites, and error paths. In practice, such lightweight retrieval operations are often more effective for interactive auditing than heavier static analyses or AST-based structural search, because they are easy to compose, require little setup, and adapt well to incomplete, evolving, or heterogeneous codebases.

3.2.2 Primitive Tools

Primitive tools capture recurring analytical subprocesses employed by human auditors during manual code auditing. Given the context retrieved through retrieval tools, auditors derive specific primitive program properties and incrementally store them in the memory of an auditor. This process is repeated until the accumulated properties are sufficient to determine whether the potential bug exists. We formalize this iterative analysis workflow as the use of *primitive tools*. Depending on the nature of the analyzed artifacts, we distinguish between primitive tools that operate on code artifacts and those that operate on non-code artifacts.

On code artifacts, primitive tools correspond to analyses that infer semantic properties of program behavior directly from source code. Examples include pointer analysis (Chen *et al.*, 2024), dependency analysis (Wang *et al.*, 2024c), call graph analysis (Cheng *et al.*, 2024), and invariant generation (Pei *et al.*, 2023), which support the detection of different types of bugs. For example, auditors reason about program dependencies to determine whether a pointer may be null at a dereference site and infer loop invariants to justify whether a given assertion always holds within a loop.

On non-code artifacts, primitive tools support the extraction of semantic specifications that are not explicitly encoded in the program itself. Auditors infer library API specifications from documentation, usage examples, issue reports, online discussions, and other descriptive artifacts to understand intended behavior, preconditions, and error-handling requirements. These inferred specifications are then used to interpret application code, identify semantic mismatches, and recover developer intent for bug detection.

Overall, regardless of whether they operate on code or non-code artifacts, primitive tools serve the same fundamental role. They infer primitive properties from available evidence and thereby support multi-step and even iterative code auditing workflows. The main difference lies in methodological maturity. For code artifacts, many primitive analyses have long been studied in static analysis, and substantial tool support already exists, although these techniques retain several common limitations, as discussed earlier in Section 2.2.2. For non-code artifacts, by contrast, systematic methods have historically been less mature and were often framed as traditional NLP tasks, such as semantic parsing (Zettlemoyer and Collins, 2005). With the rise of LLMs, however, primitive tools can be instantiated more effectively via neurosymbolic designs, as discussed in Chapter 4.

3.2.3 Auditing Tools

While manual code auditing is indispensable for uncovering subtle or context-dependent bugs, it is often labor-intensive and difficult to scale. In practice, human auditors therefore complement manual reasoning with automated auditing tools, most commonly static analysis tools, to identify candidate bugs across large codebases. For example, auditors routinely employ static analyzers such as Clang Static Analyzer (LLVM Project, 2026), Meta Infer (Facebook, 2022), and Coverity (Bessey *et al.*, 2010) to detect memory errors and certain logic flaws. These tools correspond to the static analysis techniques summarized in Section 2.2.1.

However, auditing tools are rarely usable out of the box. Auditors typically need to perform non-trivial setup and customization, including configuring build environments, supplying library specifications,

writing project-specific rules or queries, and even choosing appropriate parameters for performance tuning. After analysis completes, the generated reports require careful manual inspection to validate findings and discard spurious warnings. As a result, auditing tools must be integrated with retrieval and primitive tools to provide sufficient context for interpretation.

In this workflow, auditing tools function as candidate bug report generators rather than final decision makers. They direct human auditors' attention to suspicious program regions, but the final decision depends on the broader auditing workflow, which combines retrieved context, primitive analyses, and accumulated memory. We will revisit this interaction between automated auditing tools and human reasoning in Section 5.2.

3.3 Memory for Code Auditing

From an agent-centric perspective, code auditors critically depend on memory that supports multi-step evidence-seeking reasoning. We distinguish between two complementary forms of memory, namely *working memory* and *long-term memory*. Specifically, working memory stores intermediate results generated by tools during the auditing of a specific codebase, while long-term memory captures reusable knowledge that generalizes across auditing tasks, versions, projects, and domains.

3.3.1 Working Memory

Working memory maintains the intermediate results of an ongoing auditing task. These include program semantic properties derived from code and non-code artifacts as well as provisional bug hypotheses. Such artifacts are typically produced incrementally through repeated invocations of retrieval, primitive, and auditing tools, and are continuously updated as the analysis progresses.

By accumulating these intermediate results as evidence, working memory enables auditors to integrate information across different program locations and abstraction levels. This integration supports long-range reasoning that is essential for validating bugs whose evidence is

distributed across functions, files, or modules. For example, assessing a potential null pointer dereference requires retaining data-flow facts about pointer initialization and propagation across call chains, while confirming a resource leak depends on remembering allocation and deallocation events observed at different points in the program.

Working memory also plays a central role when auditors interact with static analysis tools. Bug reports generated by such tools are rarely self-contained. Interpreting them requires recalling previously derived intermediate results. Determining whether a warning is a true bug or a false positive often hinges on correlating multiple intermediate artifacts stored in working memory, such as path conditions, aliasing relationships, or prior sanitization steps.

3.3.2 Long-Term Memory

Long-term memory captures knowledge that persists across auditing problems and can be reused in different projects or contexts. This includes API specifications, domain-specific bug patterns, and established reasoning strategies for particular categories of bugs, which are typically acquired through prior audits.

Long-term memory is particularly important for compensating for missing or incomplete specifications in automated tools. When analyzers lack accurate models of library APIs or domain-specific behaviors, auditors rely on previously acquired specifications and patterns stored in long-term memory to interpret analysis results and guide further investigation. For example, recognizing misuse of a security-critical API or identifying a recurring resource-management pattern often depends on prior knowledge accumulated across multiple audits.

While working memory is task-specific, the boundary between the two forms of memory is not rigid. For example, inferred API specifications and reusable reasoning patterns may move from working memory to long-term memory for future reuse. Within a single project, previously derived program facts can also persist over time and support incremental auditing. More fundamentally, working memory extends the prior knowledge of LLMs during auditing. By persistently storing knowledge derived from past audits in long-term memory, an auditing

agent can reintroduce this information through prompts and retrieved notes, and can reorganize information acquired across different contexts and time spans. This mechanism enables reasoning beyond the semantic barrier, offering an important pathway toward more effective and scalable code auditing.

3.4 Summary

In this chapter, we present code auditing as an evidence-seeking workflow from an agentic perspective. In this view, the environment provides code and non-code artifacts, tools support retrieval, primitive analysis, and automated auditing, and memory stores grounded observations, intermediate evidence, and other facts. This view reflects a key feature of manual auditing. Auditors rarely make decisions based on a single artifact or a single reasoning step. Instead, they gather evidence, infer primitive properties, test candidate hypotheses, and relate program behavior to specifications and known bug patterns.

From an agentic perspective, we can also explain how to address the challenges of static analysis discussed in Section 2.2.2. Specifically, a rich multimodal environment helps auditors cross the semantic barrier imposed by source code alone. Tool use also extends auditing beyond classical static analysis. For example, lightweight code search and LSP-based navigation can reduce dependence on IRs and lessen the customization burden often required by static analyzers. Similarly, this perspective helps address the limitations of standalone LLM-based code analysis. Instead of asking an LLM to infer all relevant semantics from a limited context window, we can retrieve missing context from the environment, use tools to derive or verify primitive properties, and store useful evidence in working or long-term memory. These mechanisms help mitigate context limitations and reduce hallucinations by grounding model reasoning in external artifacts and tool-supported observations. In addition, iterative auditing can help overcome limited test-time compute and thus scale better to large real-world programs.

Lastly, we clarify the scope of the term *agentic* in this chapter. We do not use the term to suggest that every code auditing system must be a fully autonomous agent with open-ended planning. Rather, the agentic

perspective is a broad abstraction for auditing workflows that interact with environments, use tools, and maintain memory across multiple reasoning steps. Under this interpretation, both LLM agents with planning ability and fixed auditing workflows can be regarded as agentic systems. In Chapter 5, we discuss different neurosymbolic approaches, some of which rely on the planning ability of LLMs, while others follow fixed workflows that combine LLM-based primitive analysis with classical static analysis. For this reason, we use the term *neurosymbolic code auditing* rather than *agentic code auditing* in this survey.

4

LLM-driven Primitive Analysis

In this chapter, we explore how LLMs can be leveraged to perform a series of *primitive analyses* to extract program properties relevant to different bug types. By primitive analyses, we refer to static analyses that concern basic program properties (*e.g.*, points-to, aliasing, data dependence, control dependence, caller-callee relationships, program invariants, etc.) and serve as building blocks for a concrete auditing problem. These analyses are also often combined and integrated with other tools, ultimately enabling the detection of bugs during code auditing.

As discussed in Section 3.1, the auditing environment consists of both code artifacts (i.e., source code) and non-code artifacts (e.g., documents and bug reports). Therefore, our discussion of primitive analyses in this chapter is structured along these two dimensions, targeting both code and non-code artifacts, respectively.

4.1 Primitive Analysis over Code Artifacts

LLMs have increasingly been applied to primitive code analysis tasks, where they infer program properties without necessarily executing the code. We organize these analyses according to a progression of program

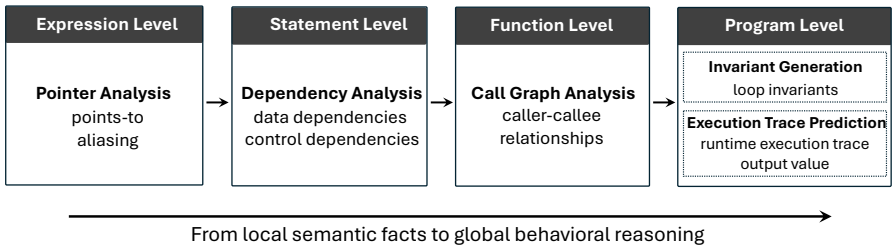

Figure 4.1: Overview of LLM-driven primitive code analysis

granularity, reflecting how relevant evidence is often accumulated during auditing. At the *expression* level, we summarize the LLM-driven analysis of pointer expressions, which discovers facts about points-to and aliasing relationships (Section 4.1.1). At the *statement* level, dependency analysis identifies data and control dependencies among statements (Section 4.1.2). At the *function* level, call graph analysis targets potential caller-callee relationships and supports interprocedural reasoning (Section 4.1.3). Finally, at the broader *program* level, invariant generation and execution trace prediction reason about properties that hold across all executions or about concrete behaviors along specific paths (Sections 4.1.4 and 4.1.5).

4.1.1 Pointer Analysis

Establishing pointer-related properties, such as points-to facts and aliasing facts, is fundamental building block in static analysis. They describe which memory locations pointers may refer to and which pointers may refer to the same memory locations. Such information supports many downstream analyses used in code auditing, including program slicing and data-flow bug detection. When inspecting pointer-heavy code, auditors do not usually compute points-to sets or alias sets explicitly, but they nevertheless maintain a mental model of which objects may be referenced and which buffers or memory regions may overlap or be disjoint with others.

In static analysis, *pointer analysis* (Smaragdakis and Balatsouras, 2015) systematically derives such pointer-related properties by computing an approximation of the memory locations that pointer expressions

may refer to. Its precision and scalability are determined by the choice of abstraction granularity, including flow sensitivity (Ye *et al.*, 2014), context sensitivity (Li *et al.*, 2020), and heap abstraction (Kanvar and Khedker, 2016). Although these analyses provide principled semantic abstractions of program statements, they require language-specific analysis infrastructure and can be difficult to apply to library functions without available implementations or to scale to programs involving sophisticated memory operations or deep calling contexts (Bastani *et al.*, 2018; Eberhardt *et al.*, 2019; Wang *et al.*, 2024a).

Recent work has therefore explored neural and LLM-assisted approaches for recovering pointer-related facts. One line of work reduces pointer reasoning to a classification problem. For example, Chen *et al.* (2024) evaluate neural code models on alias prediction. Given two pointers in source code, the model determines whether they must alias, may alias, or must not alias. Their results suggest that neural code models can capture certain local regularities of pointer semantics from real-world code, although the predicted labels do not constitute sound guarantees of aliasing facts. Similarly, LLMs have been used to infer pointer-related properties such as the nullability of C function arguments (Tullsen *et al.*, 2024). Such nullability information is useful, for example, for C-to-Rust translation, where nullable C pointers may need to be translated into Rust option types rather than ordinary references.

A complementary line of work uses LLMs not to replace static pointer analysis, but to improve the semantic modeling required by conventional analyses. For example, Cheng *et al.* (2025) use LLMs to help identify custom allocation functions in C/C++ programs. By combining static pointer analysis with LLM-based reasoning about more complex allocation patterns, their approach enriches the heap-allocation model used by a downstream pointer analysis. This illustrates a neurosymbolic use case in which the static analyzer preserves the global analysis workflow, while LLMs assist at semantic bottlenecks that are difficult to cover with hand-written rules.

Discussion. Pointer analysis is a setting where soundness matters especially strongly, because missed aliases or points-to targets can invalidate downstream reasoning about memory updates, data flow, and verifica-

tion conditions. LLM-driven pointer analysis is therefore most useful when it predicts candidate facts or enriches semantic models, rather than when it is treated as a standalone substitute for a conventional pointer analysis. In program verification, for example, LLM-derived pointer facts need to be checked or embedded inside a sound procedure before they can support proof obligations.

4.1.2 Dependency Analysis

At the statement level, code auditing often requires developers to reason about how program statements depend on one another. Given a concrete program value, auditors trace which statements produce, propagate, or use the value and which branch conditions control the reachability of these statements. *Dependency analysis* automates this reasoning by identifying data dependencies, which capture value flow from definitions to uses, and control dependencies, which capture how statement execution depends on control-flow decisions.

Static dependency analyses typically compute these relationships over dependence representations such as program dependence graphs and their interprocedural variants (Click and Paleczny, 1995; Bracevac *et al.*, 2023; Ferrante *et al.*, 1987). These representations make data and control dependencies explicit and provide a structured basis for slicing and other downstream analyses. However, they often assume that the analyzed code is complete, well-formed, and amenable to reliable discovery of def-use relations and control-flow structure (Tip, 1994; Horwitz *et al.*, 1988). These assumptions can be hard to satisfy in auditing scenarios involving partial code fragments, uncompilable code, and sophisticated language features. This motivates learning-based dependency analysis methods that approximate statement-level dependencies without requiring a fully constructed dependence graph. For example, GraphCodeBERT shows that code models can benefit from dependence-aware pretraining objectives, suggesting that models can internalize dependency information from large code corpora (Guo *et al.*, 2020).

Building on this observation, recent LLM-driven dependency analysis can be grouped into two streams, namely fine-tuned neural dependency

analysis and prompting-based dependency analysis. First, fine-tuned neural techniques use smaller code models to directly predict program dependencies, such as data/control dependency prediction (Yadavally *et al.*, 2023; Yadavally *et al.*, 2024) and sequence-to-sequence slicing with constrained outputs (He *et al.*, 2025). These methods typically formulate dependency analysis as a supervised prediction problem over statements, making them suitable for incomplete or fragment-style code where classical dependence-graph construction may fail. Second, prompting-based techniques utilize general LLMs with structured decomposition to recover data dependencies and control dependencies. These methods often combine LLM outputs with external tooling or multi-step procedures to reduce hallucination and enforce semantic constraints, as exemplified by LLMDFA (Wang *et al.*, 2024b) and NESA (Wang *et al.*, 2026). Existing empirical evidence shows that LLMs can produce approximate data/control dependencies, but the outcomes vary substantially with prompting choices and exhibit recurring error patterns (Shahandashti *et al.*, 2025). Hence, explicit verification and refinement loops are also introduced to improve the accuracy and reliability of dependency-analysis results (Chang *et al.*, 2025).

Discussion. Similar to the pointer analysis discussed in Section 4.1.1, LLM-driven dependency analysis lacks the theoretical guarantees provided by classical static analysis. Moreover, statement-level dependencies often depend on expression-level semantic facts, such as aliasing. For example, if an LLM can more accurately understand pointer-related properties within a given context, it should also be better positioned to recover data/control dependencies. Therefore, designing appropriate training tasks that help LLMs better understand memory states would be a feasible way to improve LLM-driven dependency analysis.

4.1.3 Call Graph Analysis

When analyzing a real-world codebase, human auditors routinely reason about interprocedural behavior by asking which functions may invoke which others and how execution flows across function boundaries. Call graphs provide a compact abstraction for this reasoning by

summarizing potential caller-callee relationships. To obtain all possible caller-callee pairs, *call graph analysis* needs to resolve sophisticated program constructs that may induce indirect calls, such as function pointers, higher-order functions, and dynamic dispatch. Conversely, incomplete semantic modeling may lead to missing edges and hence unsound downstream analyses.

Unlike classical static analysis, recent LLM-assisted approaches typically do not construct entire call graphs from scratch. Instead, they integrate LLM-driven semantic reasoning into conventional call graph analysis pipelines. One role is *precision-oriented pruning*. SEA (Cheng *et al.*, 2024) first obtains an initial target set for each indirect call using static analysis, and then asks an LLM to summarize the indirect-call context and candidate target functions. These summaries are then compared to filter out semantically implausible caller-callee pairs, thereby improving precision while retaining the candidate space supplied by static analysis. AutoPruner (Le-Cong *et al.*, 2022) follows a pruning-oriented view, but formulates call-graph pruning as a supervised edge classification task, i.e., given a baseline static call graph, it combines LLM-derived semantic representations of caller and callee code with graph-structural features to predict whether an edge is likely to be a false positive. In higher-order control-flow analysis, Wei *et al.* (2025) propose to use LLMs as the abstract address allocator to improve the analysis precision via context-sensitivity, meanwhile without introducing unsoundness to the analysis.

A complementary role is *recall-oriented augmentation*. CALLME targets JavaScript dynamic property-access calls (Wang *et al.*, 2025c), which are often ignored or under-approximated by classical static call graph analysis due to the difficulty of enumerating possible object and property values. Rather than merely pruning an existing graph, CALLME combines a custom static analyzer with an LLM to recover likely dynamic call edges. The language model exploits semantic signals such as variable names, natural language documentation, and calling contexts, which are difficult to encode with fixed static-analysis rules but can be informative for resolving dynamic property calls.

At the same time, empirical studies caution against treating LLMs as standalone replacements for dedicated call-graph analyzers. In a large-

scale study over Python and JavaScript, Venkatesh *et al.* (2024) report that traditional tools such as PyCG (Salis *et al.*, 2021) remain stronger at call-graph generation, while LLMs perform comparatively better on related semantic tasks such as Python type inference (Venkatesh *et al.*, 2024). This contrast is intuitive. Call graph analysis requires broad coverage of interprocedural structure and careful treatment of language and runtime semantics, whereas LLMs are strongest when a bounded context contains enough semantic, naming, and documentation cues to support local disambiguation.

Discussion. Compared with the pointer analysis and dependency analysis discussed above, call graph analysis targets a function-level program property. However, its precision often highly depends on points-to and data dependency information, including, but not limited to, aliasing relationships among function pointers and the types of specific expressions. For example, recent LLM-assisted approaches analyze function pointers to support more precise call graph analysis for the Linux kernel (Cheng *et al.*, 2024). Meanwhile, constructing an accurate call graph can also improve context retrieval, which is important for interprocedural analyses, including pointer analysis and dependency analysis.

4.1.4 Invariant Generation

When manually auditing code, human auditors reason about *program invariants*: logical properties expected to hold at designated program points across all relevant executions. Invariants operate at complementary levels of granularity. At the function level, *pre- and postconditions* characterize what a function assumes about its inputs and guarantees about its outputs, abstracting away from the concrete function body. These specifications are the cornerstone of modular code auditing, as they allow an auditor to reason about a function in isolation: a caller can be verified against the callee's postcondition without inspecting its implementation, and the callee can be verified against its precondition without examining its call sites.

At the intra-procedural level, *loop invariants* describe properties that remain true before and after each loop iteration, allowing auditors

to abstract away from individual iterations and reason about repetitive behavior at a higher level. A sufficiently strong invariant, whether at the function or loop level, can be used to establish that required assertions continue to hold across the executions it summarizes, such as bounds checks, resource constraints, and data-structure consistency properties. When such an invariant cannot be established or does not imply the desired property, auditors are directed to the remaining proof obligations or suspicious behaviors that require closer inspection. In this sense, *invariant generation* reflects a core form of human reasoning used to explain, summarize, and validate program behavior, and it further enables the detection of a wide range of bugs related to functions and loops.

Concretely, a loop invariant is a logical assertion that remains true before and after each loop iteration. It serves as a central proof artifact in program verification (Flanagan and Leino, 2001; Cousot and Cousot, 1977b; Cousot and Cousot, 1979; Cousot and Cousot, 1977a; Floyd, 1993; Hoare, 1969; Leino, 2010). In abstract interpretation, an analysis iteratively computes an approximation of reachable states until a fixpoint is reached, with the precision of the resulting invariant depending on the chosen abstract domain and widening/narrowing strategies (Cousot and Cousot, 1977b). Additionally, automated program verification may proceed by guessing candidate invariants and checking them with a verifier. Specifically, a pool of candidate predicates is generated and then checked for inductiveness and adequacy against the verification conditions (Si *et al.*, 2018). By contrast, dynamic invariant generation methods infer likely invariants from concrete executions (e.g., from trace observations) and are often useful for documentation, test augmentation, or heuristic guidance. However, such dynamically inferred invariants do not guarantee inductiveness without additional formal verification (Ernst *et al.*, 2007).

To mitigate this bottleneck, a growing body of recent work explores LLM-assisted invariant inference as a complement to classical program verification techniques (Pei *et al.*, 2023; Chakraborty *et al.*, 2023; Liu *et al.*, 2024b). These approaches leverage the pattern-recognition and semantic generalization capabilities of LLMs to propose candidate invariants that are difficult to synthesize using purely symbolic methods.

Recent studies demonstrate that LLMs can generate plausible loop invariants from program text, execution traces, or verification conditions, and can often capture non-trivial relationships involving control-flow structure (Pei *et al.*, 2023) and memory-manipulating code (Liu *et al.*, 2024b). More recent work integrates LLM-generated invariants into verification pipelines, where candidates are subsequently checked, ranked, or refined using SMT solvers or bounded model checking, enabling a synergistic combination of neural hypothesis generation and symbolic validation (Chakraborty *et al.*, 2023). This integration reflects an emerging workflow in which LLMs assist in expanding the hypothesis space, while formal reasoning components ensure correctness.

Despite these advances, existing LLM-driven approaches typically do not address the fundamental challenges of invariant or specification inference. Generated invariants may be incomplete, overly specific, or spurious, and their effectiveness often depends on prompt design, model choice, and downstream filtering strategies. Moreover, most current techniques focus on local invariants or bounded reasoning and still struggle with codebase-wide analysis.

Discussion. Intuitively, invariant generation resembles the way human auditors reason about programs with complex control flow. For example, when analyzing a function, auditors often expect it to satisfy a specification, such as ensuring that a postcondition holds under a given precondition. To validate such a hypothesis, human auditors may first conjecture properties at specific program points and then use rigorous manual reasoning to verify them against a complete function specification. This intermediate process of conjecturing program invariants is closely aligned with current uses of LLMs for invariant generation. It is also worth noting that the properties captured by program invariants vary across applications. For example, when verifying heap-manipulating programs, the corresponding invariants may characterize complex properties over heap memory (Liu *et al.*, 2024b), including, but not limited to, points-to facts and aliasing facts studied in pointer analysis.

4.1.5 Execution Trace Prediction

Human developers often need to reason about how a program would execute under particular inputs, mentally simulating control-flow decisions and memory state updates to determine whether the program behaves as intended. However, it is not always feasible to run the program due to missing software dependencies, unsupported platforms, or prohibitive cost. *Execution trace prediction* infers dynamic program behaviors, including executed statement sequences and intermediate variable states, without actually running the program (Ni *et al.*, 2024; Xu *et al.*, 2025a). Such predictions support explaining program behavior, validating assumptions about runtime states, and identifying bugs that only manifest along specific execution paths.

Predicting execution traces is particularly challenging because runtime behavior is jointly determined by path conditions, evolving program states, and the external environment. Errors in execution trace prediction are highly non-local. A single incorrect branch decision or value update can invalidate the entire subsequent trace. This challenge is further exacerbated in real-world code by partial programs, dynamic typing, and implicit flows, where many semantic facts required for execution are difficult to discover through purely static reasoning.

Recent work explores execution trace prediction along two broad directions. The first direction injects execution semantics into the model through *trace-aware learning*. This includes approaches that condition reasoning on concrete execution traces or variable-state sequences, as well as self-training schemes that distill execution-aware rationales into the model, allowing it to internalize runtime behavior patterns beyond surface code structure (Ni *et al.*, 2024; Ding *et al.*, 2024). The second direction introduces *inference-time neurosymbolic grounding*, where symbolic structure or executable tools are used to constrain the model's prediction. This direction can take several forms. PredEx adopts an analysis-guided decomposition strategy (Li *et al.*, 2025d), i.e., lightweight program analysis fixes deterministic execution structure whenever possible, while predictive backward slicing decomposes state prediction into smaller variable-level subproblems handled by the model. Other work uses CFG-grounded or multimodal prompting to align

reasoning with feasible control-flow paths (Le Chi *et al.*, 2025). Tool-augmented approaches further execute deterministic fragments and rely on the model only for unsupported, underspecified, or semantically ambiguous parts (Li *et al.*, 2024). Across these designs, symbolic or executable scaffolds serve as external anchors that reduce hallucinated paths and improve consistency with program semantics.

Execution trace prediction also admits several variants. For example, RepoReason evaluates whether an assertion at a specific program location holds under a given execution path (Li *et al.*, 2026). Beck *et al.* (2026) also propose the concept of a *neuro debugger*, which aims not only to use LLMs to predict execution paths and outputs under given inputs, but also to support a reverse form of execution trace prediction, i.e., inferring possible prior states or input parameters from a given current state. These techniques and variants can potentially facilitate debugging during development, especially when code fragments and environment configurations are incomplete. At the same time, evaluating LLMs on execution trace prediction provides a measure of their ability to understand code semantics and can offer useful insights for downstream tasks such as bug exploitation (Fang *et al.*, 2024b).

Discussion. Unlike invariant generation, as discussed in Section 4.1.4, execution trace prediction focuses on the runtime behavior of a program along a specific execution path under a concrete input. An accurate execution trace prediction reflects a concrete program execution. In contrast, invariant generation is concerned with whether a property holds for all inputs, or for all inputs satisfying certain conditions. Nevertheless, both techniques offer useful observations for concrete code auditing applications and are important for bug detection and diagnosis.

4.1.6 Comparisons with Classical Static Analysis

The ability of LLMs to perform primitive code analysis stems from a combination of complementary mechanisms. First, large-scale pretraining on code and code-related text exposes models to the statistical regularities of software. Second, code-specific pretraining objectives can make such semantic signals more explicit. For example, Graph-

CodeBERT incorporates data-flow and control-flow prediction (Guo *et al.*, 2020). CodeT5 introduces identifier-aware objectives to exploit the semantics conveyed by developer-assigned identifiers (Wang *et al.*, 2021). In addition, TRACED injects dynamic semantics by pretraining on execution traces (Ding *et al.*, 2024). Empirical probing studies also show that pretrained code models encode nontrivial syntactic and semantic program structures, although their semantic understanding remains uneven across tasks and models (Ma *et al.*, 2024). At the same time, LLMs can exploit informal semantic cues embedded in symbol names, API choices, comments, and coding conventions, which often reflect developer intent and engineering practice (Chen *et al.*, 2022). As a result, LLM-driven code primitive analyses are not solely based on surface-level natural-language similarity, but on a joint representation of local program structure, developer intent cues, and statistically correlated semantic patterns observed during pretraining.

When compared with classical static analyses, LLM-driven primitive code analyses exhibit fundamentally different strengths and limitations. Static analysis methods derive semantic properties by constructing explicit program representations, such as dependence graphs and abstract program states, and applying rigorous reasoning to ensure formal guarantees at a chosen abstraction level. These methods are well suited for analyses that require principled over-approximation, but they rely heavily on structured IRs, well-formalized program semantics, and carefully engineered analysis frameworks. In contrast, LLM-based approaches reformulate primitive analyses as prediction or generation problems over a restricted scope of code fragments. This design allows them to operate on incomplete, uncompilable, or fragment-level code and to approximate semantic relations that are difficult to encode with static rules. However, because their reasoning occurs in latent space, LLMs behave as black boxes without guarantees of soundness and completeness, and their errors can propagate non-locally. Consequently, LLM-driven analyses are best understood not as replacements for static analyses, but as complementary mechanisms that provide flexibility and semantic intuition where formal modeling is costly or brittle.

These observations motivate a neurosymbolic design perspective for practical code auditing tasks. Real-world auditing rarely consists

of a single monolithic analysis. Instead, it involves solving a sequence of primitive subtasks, such as identifying relevant program slices, resolving ambiguous indirect calls, or reasoning about potential runtime behaviors under partial information. In this setting, static analysis plays a role analogous to the specialized tools available to human auditors, offering structured, reusable, and verifiable results whenever formal reasoning is feasible. LLM-driven code analysis, in turn, corresponds to the primitive analysis tools formulated in Chapter 3, providing judgments and hypotheses when rules are incomplete or intent needs to be inferred. By carefully deciding how static analysis and LLM-driven analysis are combined at the level of individual primitive subtasks, these components can be composed into a coherent neurosymbolic auditing workflow. Such workflows leverage static analysis to anchor reasoning in program semantics while using LLMs to address the semantic barrier, ultimately enabling scalable and effective auditing across real-world codebases. Chapter 5 will further elaborate on this design space and its instantiations in practical code auditing systems.

4.2 Primitive Analysis over Non-Code Artifacts

As discussed in Section 3.1, effective code auditing requires reasoning not only over source code, but also over a diverse range of non-code artifacts. This section surveys primitive analyses that leverage LLMs and lightweight natural language processing techniques to derive auditing evidence from such artifacts. We organize the discussion according to the types of non-code artifacts.

4.2.1 Library Documentation

Documentation provides a major source of semantic description for code auditing, especially when analyzing code that interacts with external libraries. Many auditing tasks require knowledge of library behaviors, yet such semantics are often unavailable in a formal form or too costly to specify manually. Under this view, documentation is not merely auxiliary context for developers. Instead, it serves as a primary source of semantic description from which library specifications can be inferred.

Documentation-driven specification extraction therefore provides an important bridge between informal, developer-facing library knowledge and automated auditing of application code.

This idea predates recent LLM-based systems. Earlier NLP-based approaches already explored how to infer various forms of API specifications from natural-language documentation (Zhong *et al.*, 2009; Pandita *et al.*, 2012; Zhai *et al.*, 2016). However, these systems typically relied on task-specific linguistic patterns, semantic parsers, or hand-engineered extraction rules. Recent neural and LLM-based approaches inherit the same goal but shift the central design problem. The challenge is no longer only how to recognize particular textual patterns, but how to ground natural language understanding into stable specifications that can be consumed by downstream auditors.

Among existing studies, DAInfer is a representative example of documentation-guided specification inference designed for code auditing (Wang *et al.*, 2024a). It targets API aliasing specifications, which are crucial for library-aware data-flow analysis but are notoriously difficult to recover from library code alone. DAInfer interprets API documentation to classify memory-related operations, such as access, insertion, and deletion, and to identify the semantic entities manipulated by these APIs. On top of these abstractions, it formulates API aliasing specification inference as a neurosymbolic optimization problem, selecting a globally consistent set of aliasing facts induced by memory operations and semantic entities. This design is important because the LLM is not asked to directly emit final specifications. Instead, documentation-derived observations are converted into structured constraints, making the inferred semantics more stable and easier to integrate into existing code auditors.

Similarly, ChatDetector targets resource-management API specifications (Yang *et al.*, 2025b). It uses LLMs to recover allocation-release protocols from documentation and then constructs resource-management API constraints for misuse detection. Unlike DAInfer, ChatDetector decomposes documentation understanding into smaller and more verifiable steps and further introduces cross-validation and inconsistency checking to mitigate hallucination before the recovered constraints are used by an auditor. Compared with DAInfer (Wang *et al.*, 2024a), ChatDetector

illustrates a complementary design philosophy in which reliability is enforced through staged reasoning and validation of intermediate results rather than through a global semantic objective.

Beyond specifications for library APIs, another line of related work studies whether LLMs can extract more general formal specifications from general software documentation. Although these specifications are not always library API contracts, they expose a similar challenge that informal documentation must be converted into precise, checkable properties before it can support automated testing or auditing. For example, Li *et al.* (2025b) empirically studied the use of LLMs to translate natural-language documents into formal specifications for test automation. They found that out-of-the-box LLMs can often identify useful rules but also tend to oversimplify conditions and fabricate unsupported requirements. To mitigate these problems, they proposed a two-stage annotation-then-conversion approach, in which the model first highlights relevant facts in the text and then translates them into formal statements. This result illustrates both the promise and the current limitations of directly inferring precise formal specifications from documentation.

Discussion. Across these documentation-guided specification inference techniques, the most significant differences lie not merely in the specific specification targets, but in how language-derived outputs are grounded. DAInfer (Wang *et al.*, 2024a) relies on global consistency enforced by optimization, whereas ChatDetector (Yang *et al.*, 2025b) relies on procedural grounding through task decomposition, cross-validation, and intermediate consistency checks. The formal-specification extraction setting further suggests that even when LLMs can identify relevant semantic facts, direct end-to-end translation remains fragile without explicit grounding steps. Thus, the central design question is not whether LLMs can read documentation, but how their linguistic judgments are constrained, validated, and translated into stable artifacts consumed by downstream code auditors. Compared with classical NLP-based approaches, the main advantage of recent LLM-based systems is their ability to handle the semantic diversity of real documentation, reducing reliance on task-specific semantic templates.

4.2.2 Protocol Specifications

Protocol specifications can be viewed as a special form of documentation. Compared with general software documentation, protocol specifications are domain-specific standards that are usually more normative, structured, and authoritative, and are most commonly published as RFCs (Request for Comments). Typically, they prescribe protocol behavior, including message formats, state transitions, error handling, and protocol-level constraints that conforming implementations are expected to satisfy. As a result, protocol specifications provide a particularly valuable semantic source for auditing protocol implementations. Once extracted into machine-readable forms, they can serve as input grammars, state machines, conformance oracles, or protocol-specific constraints for code auditing.

Despite their relatively standardized form, protocol specifications remain natural-language documents. Their requirements are often distributed across long documents through section hierarchies, cross-references, conditional statements, and normative keywords such as "MUST" and "SHOULD". Understanding whether an implementation conforms to a protocol therefore requires more than sentence-level interpretation. An auditor must recover how message formats, field constraints, and stateful interaction rules compose into coherent protocol-level behavior. Traditionally, this process has relied heavily on manual reasoning by domain experts, making specification analysis time-consuming, error-prone, and difficult to scale.

Recent work shows that LLMs provide a promising new avenue for extracting machine-readable protocol knowledge from natural-language standards. Existing LLM-based approaches can be roughly divided according to the source from which protocol knowledge is obtained. Several approaches elicit latent protocol knowledge from pretrained models, while others explicitly ground the model's reasoning in RFC documents and extract protocol constraints, message formats, or state machines from the standard text.

A representative example of the first line is ChatAFL, an LLM-guided fuzzer for network protocol implementations (Meng *et al.*, 2024b). ChatAFL is motivated by the observation that modern LLMs are

pretrained on large corpora that include substantial protocol-related text and therefore may implicitly encode knowledge about message formats and valid interaction patterns. For a given text-based protocol, such as RTSP or SMTP, ChatAFL queries LLMs to generate message grammars describing the structure of protocol messages and uses these grammars to guide mutation-based fuzzing. It also uses LLMs to enrich the initial seed corpus by increasing the diversity of recorded message sequences. However, ChatAFL does not synthesize a complete protocol state machine directly with LLMs. Instead, it retains the execution-feedback loop and uses LLMs only as local sources of guidance. When fuzzing enters a coverage plateau, ChatAFL prompts an LLM with the recent client-server communication history and asks it to generate the next client request that may induce a new state transition. The generated request is then inserted into the current message sequence and executed against the implementation. If the resulting sequence increases coverage, ChatAFL uses the server response and state feedback to update the inferred protocol state machine. This design illustrates how pretrained protocol knowledge can be useful for fuzzing, while also suggesting a limitation of such knowledge-only grounding: the model may omit or invent protocol details when the target protocol is underrepresented in pretraining data. More broadly, although ChatAFL is designed for protocol fuzzing, it suggests that protocol knowledge acquired during pretraining can provide useful semantic hints for downstream program analysis tasks, including bug detection, when combined with additional validation mechanisms.

The second line of work targets network protocols whose specifications are well documented in RFC documents. Specifically, these approaches ground extraction in the authoritative specification and aim to recover protocol constraints, message formats, state machines, or other structured artifacts from RFC documents. They differ in the kinds of artifacts they extract and in how these artifacts support auditing. Several systems focus on message formats and parser-level constraints. For example, 3DGen (Fakhoury *et al.*, 2025) uses specification knowledge to support the generation of binary-format parsers, while ParCleanse (Zheng *et al.*, 2025b) and ParVAL (Zheng *et al.*, 2025c) use RFC-derived information to validate network protocol parsers. Although

these techniques differ in their specific targets, they share a common goal of converting natural-language protocol standards into structured knowledge that can assist verified code generation, program verification, and bug detection.

Grounding LLMs in RFC documents improves fidelity to the standard, but it also introduces a new challenge. Concretely, protocol specifications are often long, internally cross-referenced, and globally constrained. Context-window limitations and context-collapse effects can cause LLMs to miss important constraints or apply them inconsistently across sections. Existing approaches therefore commonly adopt divide-and-conquer strategies that decompose specifications into smaller, structurally coherent units aligned with section hierarchies. For example, ParCleanse constructs a document tree to compose message formats extracted from individual sections (Zheng *et al.*, 2025b), whereas AutoSpec performs section-wise behavioral extraction and deterministically merges extracted states, transitions, and constraints into a protocol-wide state-transition graph (Liu *et al.*, 2025). These designs show that the central difficulty is not merely extracting isolated facts from RFCs, but maintaining cross-section and cross-state consistency when converting long standards into auditable protocol models.

Discussion. The analysis of semi-structured protocol documentation has a structural analogy to call graph analysis in classical static analysis. The analogy is not that document sections behave like program functions, but that both settings require recovering a graph-structured representation from distributed local evidence. In static analysis, call graph analysis links call sites and potential callees to support interprocedural reasoning. In protocol-specification analysis, references among terms, fields, sections, and state descriptions can similarly be used to construct a document-level graph. Such a graph helps organize long documents, propagate constraints across related sections, and support consistent extraction of protocol-level semantics.

4.2.3 Bug Knowledge Sources

Beyond developer-facing documentation and protocol standards, LLM-based code auditing can also exploit a broader class of *bug knowledge sources*. These sources are produced during bug classification, disclosure, discussion, and repair. Unlike library documentation or protocol specifications, they are usually not authoritative descriptions of how a system should behave. Instead, they encode how software bugs are conceptualized, observed, explained, and fixed in practice. As a result, they provide complementary semantic evidence for code auditing, which helps define what counts as a bug, suggests what program behaviors are security-relevant, or supplies concrete examples from which reusable bug patterns can be learned.

Bug knowledge sources differ in their level of curation and proximity to code. Curated bug taxonomies and disclosure records, such as CWE (CWE, 2025) and CVE (CVE, 2025), provide high-level security concepts and concrete bug instances. Bug reports, commits, and patches provide code-grounded evidence of how defects manifest and how developers repair them. Online technical discussions provide community knowledge about API usage, debugging practices, and security-relevant edge cases. These sources therefore support different forms of grounding for LLM-aided code auditors.

Curated bug records, such as the Common Weakness Enumeration (CWE) and Common Vulnerabilities and Exposures (CVE), are widely used sources of curated bug knowledge (CWE, 2025; CVE, 2025). CWE organizes weaknesses into abstract classes, often describing their causes, consequences, examples, and mitigations, whereas CVE assigns identifiers and descriptions to concrete publicly disclosed bugs. Together, they form a bridge between high-level bug concepts and real-world failure instances. One line of work uses CWE/CVE-style knowledge for proactive bug discovery. In this setting, the goal is not merely to classify code by a known weakness label, but to convert bug knowledge into specifications or analysis guidance that can be applied to unseen code. For example, IRIS (Li *et al.*, 2025f) takes a target project and a CWE class as input, uses LLMs to infer CWE-specific taint specifications such as sources and sinks, and integrates the inferred specifications with CodeQL for

codebase-wide analysis. Vul-RAG (Du *et al.*, 2026) constructs a knowledge base from existing bug instances by distilling information about functionality, bug causes, and fixes, and retrieves relevant knowledge to guide LLM-based bug detection. These systems illustrate a common design principle that curated bug records are not used as final answers, but are transformed into intermediate specifications, retrieved knowledge, or reasoning instructions that ground downstream code analysis. Another line of work focuses on post-disclosure bug localization. Here, the bug is already known, and the auditing task is to identify where the disclosed bug appears in the corresponding codebase. CVE descriptions are therefore treated as natural-language bug specifications that must be aligned with source code. VFFinder (Wu *et al.*, 2024), for example, uses CVE descriptions and source code to identify buggy functions. This setting emphasizes that the auditor must connect a bug report to the concrete functions or code regions that realize the root cause.

Project histories provide a localized form of security evidence for code auditing. They include issue reports, commit records, code review discussions, and bug-fixing patches that document how security-relevant defects have appeared, been diagnosed, and been repaired within a particular codebase or program domain. Unlike CWE/CVE records, which organize bug knowledge across projects and software ecosystems, project histories preserve local development experience. Their value therefore lies not only in individual bug-fix examples, but also in the recurring patterns exposed by a sequence of related development activities. In the same project or domain, bugs often recur because related code shares similar abstractions, implementation conventions, and operational assumptions. Project histories can therefore act as localized bug memory, enabling auditors to recognize recurring defect patterns that may not be fully captured by general bug taxonomies.

A common strategy is to learn bug patterns from representative historical examples and generalize them to new code within the same project or program domain. BugScope (Guo *et al.*, 2025a) follows this intuition by emulating how human auditors learn from buggy and non-buggy examples. Given representative examples, it synthesizes a retrieval strategy to collect relevant program context and constructs a

tailored detection prompt for LLM-based auditing. This design treats buggy and non-buggy examples not simply as isolated labels, but as demonstrations of domain-specific bug patterns that can guide the search for similar defects. A complementary strategy is to distill patch knowledge into reusable symbolic detectors. KNighter (Yang *et al.*, 2025a) synthesizes static analysis checkers from historical patches in the Linux kernel. Rather than asking an LLM to directly scan the entire Linux kernel codebase, KNighter uses LLMs to extract a bug pattern from patches, generate a specialized checker, and validate or refine the checker against the original patch context. This design converts patch-derived knowledge into a scalable static-analysis artifact, allowing it to scale to large codebases.

Community technical discussions, including Stack Overflow questions, project mailing lists, issue discussions, and technical forums, provide a less curated but highly practical source of software knowledge. These discussions often capture developer-facing experience that may be absent from official documentation, such as common API misuse patterns, debugging strategies, and workarounds for version-specific behavior. For code auditing, such knowledge can provide useful hypotheses about where defects arise and what contextual assumptions developers commonly miss. However, community discussions are weaker evidence than curated bug records or code patches. They may be outdated, incomplete, contradictory, or specific to a particular environment. LLMs pretrained on web-scale corpora may implicitly encode parts of this community knowledge, but because it is stored latently in model parameters, it is difficult to attribute, update, or verify. Recent systems therefore increasingly favor retrieval-augmented designs that ground generated answers in explicitly retrieved community-authored content. For example, Stack Overflow's AI Assist (Sajor, 2025) uses a Retrieval-Augmented Generation (RAG) pipeline over Stack Overflow and Stack Exchange content, prioritizing human-validated answers and providing attribution. Although such systems are not themselves code auditors, they illustrate a broader grounding pattern that is relevant to auditing, highlighting that community knowledge is most useful when retrieved, attributed, and treated as supporting evidence rather than as unverified

model memory.

Discussion. Bug knowledge sources differ from documentation-based sources in both authority and evidential structure. Library documentation and protocol specifications describe intended behavior, whereas bug records, patches, and discussions describe security knowledge accumulated through failures, repairs, and community experience. This makes them valuable but also noisy. The central design question is therefore how to turn heterogeneous bug evidence into auditable artifacts, such as bug specifications, detection prompts, static-analysis checkers, or localization queries. Across the systems above, LLMs are most reliable when their outputs are grounded in curated records, validated against code or patches, linked to retrieved evidence, or compiled into analyzable artifacts. In this sense, the role of LLMs is not to replace security expertise, but to operationalize distributed bug knowledge so that downstream auditors can use it systematically.

4.3 Summary

This chapter surveys LLM-driven primitive analyses over both code and non-code artifacts. Within the agentic view developed in Chapter 3, these analyses correspond to the primitive tools that transform retrieved artifacts into intermediate evidence for auditing. Their outputs are not final bug reports, but formal/informal semantic specifications. In this sense, primitive analysis provides the bridge between raw software artifacts and the higher-level auditing decisions studied in the next chapter.

For code artifacts, the surveyed techniques show that LLMs are most useful at semantic bottlenecks where classical static analyses are costly, brittle, or incomplete. Pointer analysis and dependency analysis recover memory-related and data-flow facts. Call graph analysis exposes interprocedural structure needed for cross-function reasoning. Invariant generation summarizes stable properties of program states, while execution trace prediction reveals path-specific runtime behavior. These primitive facts directly support the bug classes discussed in Chapter 2. For example, memory bugs require aliasing, lifetime, and nullability

facts. Taint-style bugs require data-flow, call-graph, and specification facts. Numeric bugs often rely on value invariants and path conditions. Functional bugs depend on recovering intended behavior. However, LLM-derived code facts remain fallible. They can reduce dependence on full build environments and often operate without structured IRs, but they do not provide the soundness, completeness, or consistency ensured by classical static analysis. As a result, the LLM-driven primitive code analyses demonstrate that LLMs are most effective when used to approximate, prioritize, or refine semantic properties, rather than to replace classical static analyses wholesale.

For non-code artifacts, LLMs address the complementary side of the semantic barrier problem. Many auditing-relevant specifications are not present in code at all, but are distributed across library documentation, protocol standards, bug records, project histories, patches, and community discussions. The systems surveyed in this chapter use LLMs to convert such informal or semi-structured evidence into auditable artifacts, which can be viewed as a form of externalized long-term auditing knowledge, i.e., once extracted, they help auditors interpret code behavior, instantiate bug-specific analyses, and reuse knowledge across different projects, versions, and auditing tasks.

Taken together, the primitive analyses surveyed in this chapter are composed and orchestrated according to the demands of specific auditing subtasks. Rather than committing to static analysis tools or LLMs only, practical auditors dynamically combine static tools and LLM-driven reasoning, switching between static analysis tools and semantic intuition offered by LLMs. The next chapter will build on this foundation by examining how these primitive analyses are instantiated and coordinated in neurosymbolic auditing systems, illustrating how they collectively enable scalable code auditing across real-world codebases.

5

Auditing Code as Human Auditors

As discussed in Chapter 2, classical static analysis faces fundamental challenges in code auditing due to IR reliance, limited customizability, and the semantic barrier. To address these challenges, Chapter 3 introduces the agent-centric perspective, highlighting how human auditors use various tools, software artifacts in the auditing environment, and memory to reason effectively about code behavior. Chapter 4 further elaborates on LLM-driven primitive analyses, demonstrating their capability to analyze both code and non-code artifacts.

Building upon these foundations, this chapter explores how neurosymbolic approaches leverage LLMs and symbolic analysis tools, including classical static analyzers, to closely simulate human auditing behavior. As shown in Figure 5.1, we start from two typical modes of human code auditing, namely manual code auditing and semi-automatic code auditing, and examine how existing approaches effectively simulate these two auditing modes in Section 5.1 and Section 5.2, respectively. Beyond these academic efforts, we also systematically discuss recent industrial advances in code auditing in Section 5.3 before summarizing this chapter in Section 5.4.

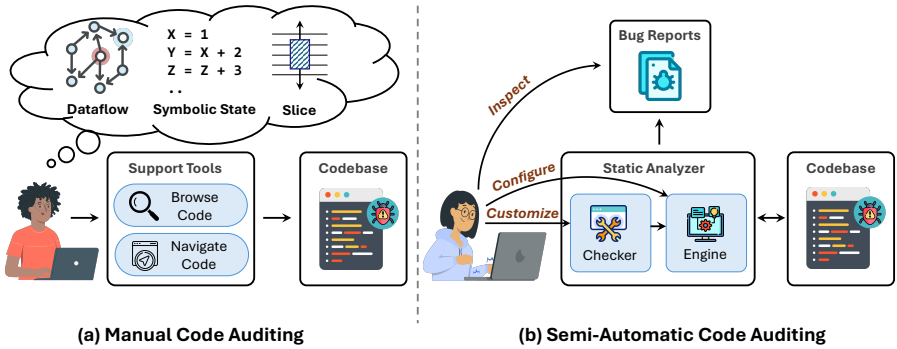

Figure 5.1: The two modes of code auditing

5.1 Simulating Manual Code Auditing

When auditing source code, human auditors may either rely on automated auditing tools or perform manual inspection. Although manual auditing is often characterized as an informal, experience-driven activity, in practice it often mirrors well-established program analysis principles, such as data-flow analysis, symbolic execution, and program slicing. Concretely, human auditors routinely trace the propagation of specific program values, reason about memory states associated with those values, and focus their attention on statements that are relevant to the auditing target. During this process, they may also leverage built-in static analysis capabilities in their development environment, such as IDE-supported code navigation, reference lookup, and lightweight semantic analysis, to inspect code across the codebase more effectively.

In this section, we focus on simulating this manual code auditing process by integrating LLMs with these tools. In our setting, LLMs play the role of human auditors, performing high-level reasoning and decision making, while static analysis tools provide structured support, including control-flow analysis, call graph analysis, and various lightweight static analyses that ground and steer the reasoning process. Building on the primitive analysis instances introduced in Chapter 4, we investigate how LLMs can instantiate program analysis frameworks to emulate the cognitive and procedural behavior of human auditors. Guided by program abstractions such as data-flow summaries, memory states, and

program slices, LLMs can simulate human-style auditing workflows and flexibly detect diverse classes of bugs.

5.1.1 Data-Flow Analysis

Data-flow bugs constitute a class of software defects that arise from the unsafe or unintended propagation of faulty values within a program. Representative examples include taint-style bugs and certain memory-related bugs, such as null pointer dereferences and memory leaks, as discussed in Section 2.1. Detecting such bugs requires precise reasoning about data dependencies that span multiple functions and files. This reasoning, in turn, relies on pointer analysis and interprocedural data-flow analysis to track how values flow through data structures and across function boundaries.

In real-world software systems, however, practical data-flow bug detection remains challenging. First, pointer analysis often becomes overly imprecise when applied to complex heap-allocated data structures, such as linked lists, trees, and other container types (Wang *et al.*, 2022). Second, deep call chains involving library functions significantly hinder scalability, as the number of contexts and execution paths to explore grows rapidly (Arzt and Bodden, 2016). Third, many data-flow bugs, such as null pointer dereferences, can occur only under specific path conditions (Shi *et al.*, 2021). Imprecise reasoning about these path conditions further increases both false positives and false negatives. Fourth, existing code auditing tools often face the semantic-barrier issue discussed in Section 2.2.2. Although many tools expose source, sink, and propagation-rule specifications, they have limited practical customizability because these specifications still require manual adaptation based on substantial domain knowledge and familiarity with the underlying analysis framework (Chibotaru *et al.*, 2019). As a result, existing data-flow bug detectors, such as SVF (Sui and Xue, 2016) and FlowDroid (Arzt *et al.*, 2014), still struggle to strike an effective balance among precision, scalability, and usability when applied to large-scale, real-world codebases.

Neurosymbolic Approach. Human auditors typically identify data-flow bugs in large codebases through a structured reasoning process that can be decomposed into three components. First, based on their understanding of the target bug class, auditors identify sources that introduce untrusted, invalid, or otherwise faulty values, as well as sinks that correspond to operands of security-critical or potentially dangerous operations. Second, they trace how these values propagate both within individual functions and across function boundaries, determining whether values originating from sources may reach sinks along feasible execution paths. Third, they perform an end-to-end inspection of the resulting propagation paths, i.e., data-flow paths, to assess whether the inferred flows are semantically valid or instead depend on hidden assumptions, implicit constraints, or overlooked guarding conditions.

Motivated by this manual auditing workflow, our recent works, such as LLMDFA (Wang *et al.*, 2024b) and RepoAudit (Guo *et al.*, 2025b), explicitly decompose automated data-flow bug detection into three corresponding subproblems, namely source and sink identification, data-flow summarization, and path feasibility validation. This decomposition closely mirrors the reasoning process of human auditors and allows each subproblem to be addressed through the complementary strengths of static analysis and LLM-driven semantic reasoning. As a result, it provides a principled and modular foundation for scalable, codebase-wide neurosymbolic auditing of data-flow bugs.

Component I: Source and Sink Identification. To identify sources and sinks within a codebase, LLMDFA adopts a programming-by-example paradigm and leverages LLMs to synthesize source and sink extractors from user-provided examples that characterize the desired patterns. These examples specify concrete instances of sources and sinks, based on which LLMs synthesize extractors that invoke parsing libraries to accurately recognize the corresponding program constructs. Once synthesized, the extractors can be efficiently applied across the entire codebase. Compared with prompt engineering, this synthesis-driven approach is both more precise and more efficient. Specifically, the synthesis incurs only a one-time cost. For a given bug type, the same extractors can be reused across different codebases without additional

manual effort. As a result, developers are relieved of manually implementing domain-specific extractors, substantially reducing the human effort required to adapt the analysis to new projects.

Component II: Data-Flow Summarization. To detect data-flow bugs induced by data flows across multiple functions, LLMDFA and RepoAudit follow the principle of summary-based interprocedural data-flow analysis. Specifically, they leverage the dependency-analysis capability of LLMs discussed in Section 4.1.2 to summarize how values propagate within individual functions, capturing data flows among parameters, arguments, return values, and output values. After obtaining these intraprocedural summaries, they compose them symbolically based on the call graph, with interprocedural reachability computed via CFL-reachability, thereby facilitating context-sensitive data-flow analysis (Reps *et al.*, 1995). In LLMDFA and RepoAudit, the call graph is constructed based on function names and parameter/argument counts. Applying the LLM-driven call graph analysis discussed in Section 4.1.3 would enable data-flow analysis to discover more accurate caller-callee relationships.

Component III: Path Feasibility Validation. To eliminate spurious data-flow paths induced by unsatisfiable path conditions, both LLMDFA and RepoAudit reason about path conditions and solve the resulting satisfiability problems. Specifically, LLMDFA synthesizes Python scripts that encode the path conditions of data-flow paths and then invokes the Z3 SMT solver (Moura and Bjørner, 2008) via its Python bindings to check satisfiability. In this design, LLMs focus only on semantic understanding and constraint synthesis, mitigating LLM hallucinations during constraint solving. With the emergence of large reasoning models (e.g., OpenAI o1 and Claude 3.7 Sonnet), RepoAudit further simplifies this pipeline by directly prompting reasoning models with detailed path information and asking them to assess path feasibility. Empirical results from RepoAudit demonstrate that such models are highly effective at pruning infeasible paths, achieving substantial reductions in false positives with minimal loss of truly feasible flows.

Discussion. As one of the earliest works that directly leverage LLMs to reason about data flow, LLMDFA demonstrates a principle that LLMs should not be used as black-box generators of program facts, but rather as program synthesizers to create tools. This principle is instantiated through the synthesis of source and sink extractors, as well as the generation of executable scripts that encode path conditions for downstream satisfiability checking. Instead of hard-coding analysis logic, LLMDFA delegates the construction of these tools to LLMs, while relying on the synthesized tools to generate, validate, or compose program facts. This design makes analysis steps explicit, executable, and inspectable. More broadly, it reflects an emerging trend in neurosymbolic systems in which LLMs are increasingly employed to synthesize reusable analysis tools, rather than being asked to directly produce final answers. Similar ideas have also been explored in other reasoning domains, such as code search (Jiang *et al.*, 2025; Wang *et al.*, 2023a) and mathematical reasoning (Ye *et al.*, 2023), where LLMs generate query programs or solver-friendly formulations to interface with symbolic backends.

Beyond tool synthesis, LLMDFA and RepoAudit also demonstrate that prompt-based data-flow summarization can instantiate classical interprocedural data-flow analysis. Instead of relying on explicit pointer analysis and applying transfer functions, these neurosymbolic systems prompt LLMs to generate function-level data-flow summaries from local code contexts and compose them across function boundaries to recover interprocedural data-flow paths. This design avoids requiring LLMs to reason over an entire codebase at once, while still enabling codebase-wide analysis through symbolic composition.

Despite their effectiveness in scaling LLM-driven data-flow analysis to large codebases, LLMDFA and RepoAudit still exhibit several limitations. First, when inferring intraprocedural data-flow facts, LLMs often struggle with data flows induced by complex data structures. For example, in a function such as `foo(Packet* a)`, the object `Packet` may contain multiple fields, some of which may themselves be pointers to other memory locations. In such cases, long-standing challenges in classical data-flow analysis, including aliasing explosion (Yao *et al.*, 2024) and imprecise memory abstraction (Kanvar and Khedker, 2016), remain largely unresolved. Consequently, LLM-inferred data-flow summaries

may be incomplete or unsound. They may miss valid flows, conflate distinct memory locations, or infer spurious dependencies that do not hold semantically.

Second, even within a single function, LLMs may hallucinate data-flow facts in the presence of complex control-flow structures, such as nested branches and loops. A common failure mode arises when an LLM overlooks the precise execution order or guarding conditions of statements and therefore infers a data-flow relation that does not occur along any feasible execution path. To mitigate this problem, recent work such as LLMSAN (Wang *et al.*, 2024c) introduces lightweight static analyses to validate or filter LLM-generated data-flow facts, thereby reducing hallucinations during the data-flow summarization.

Third, given their target problem formulations, LLMDFA and RepoAudit focus primarily on data-flow bugs. More general classes of bugs often require reasoning about richer program properties that are difficult to express solely as source-to-sink reachability problems, thereby restricting the applicability of such neurosymbolic code auditing systems. A more recent line of work, represented by NESA (Wang *et al.*, 2026), extends this direction by instantiating a Datalog-style program analysis in which relations representing program facts can be derived either through deterministic symbolic analyses or through LLMs for semantically challenging predicates. Users can define Datalog-style analysis policies that specify how syntactic and semantic relations should be combined to infer target program properties. For example, source and sink predicates can be derived from both syntactic patterns, such as AST structures, and semantic properties, such as the intended role or security relevance of a value. Similarly, path-validation logic can be instantiated in flexible ways to support more general semantic checks, such as loop termination, rather than being restricted to CFL-reachability or path constraints. As a result, NESA broadens the scope of LLM-driven data-flow analysis beyond data-flow bug detection toward more customizable reasoning over diverse program behavior.

5.1.2 Symbolic Execution

Many software bugs, especially security bugs, manifest only along specific program paths, making symbolic execution a natural fit for path-sensitive bug detection. Classical symbolic execution represents program inputs and memory states symbolically, decomposes program behavior along control-flow paths, and discharges path-specific verification conditions using theorem provers or SMT solvers. However, real-world symbolic execution remains difficult to scale. Modern programs interact with libraries, files, networks, and other external environments that are hard to model precisely. In addition, the path-explosion problem and expensive constraint solving can quickly overwhelm symbolic execution engines, especially when the constraints involve strings, heap objects, or other source-level semantics that are awkward to encode in solver theories.

Neurosymbolic Approach. Recent LLM-assisted symbolic execution techniques revisit this design point by combining the path-decomposition discipline of symbolic execution with the semantic reasoning capability of LLMs (Li *et al.*, 2025e; Luo *et al.*, 2026). Rather than treating an LLM as a monolithic bug detector, these systems decompose bug detection into smaller path- or slice-level reasoning tasks that are more suitable for LLMs. Meanwhile, they retain conventional program-analysis workflows, such as path partitioning, program slicing, and solver- or execution-based validation. In this way, LLMs are used to reason about source-level semantics that are difficult to encode precisely, while symbolic analysis components provide the structural grounding needed to constrain and validate the reasoning process.

Component I: Path-Constraint Abstraction. Classical symbolic execution typically encodes path conditions within memory states as logical constraints over symbolic variables. Although such formulas are precise, they often obscure source-level intent, library semantics, and specification-level information that human auditors can more naturally reason about from code. LLM-assisted approaches therefore attempt to expose path constraints to the model in more LLM-friendly forms. AutoBug (Li *et al.*, 2025e) is one typical example. Given a Hoare-

logic-style specification $\{P\}C\{Q\}$, it constructs a CFG, enumerates coverage-based path partitions, and derives a truncated slice S for each partition with respect to the post-condition. The slice acts as an untranslated strongest-postcondition-style path constraint expressed in ordinary source code. This representation preserves the operational structure of the program while removing irrelevant statements. As a result, LLMs are not asked to reason over the entire program, but only over a compact path-level abstraction that summarizes the behavior relevant to the target property. Importantly, such a slice can summarize a set of paths, including paths through unbounded loops, without requiring all concrete executions to be explicitly enumerated.

Component II: Verification-Condition Reasoning. To reason about path-constraint abstractions, existing neurosymbolic techniques leverage LLMs to replace or augment solver-side reasoning by acting as an approximate semantic reasoner over source-level path summaries. In AutoBug, each truncated slice is rendered as a verification-condition prompt of the form $\{P\}S\{Q\}$. LLMs are then asked to determine whether the post-condition follows from the pre-condition and the sliced program behavior, or whether the slice exposes a possible counterexample. Thus, AutoBug does not use LLMs to flip branches, enumerate paths, or generate new dynamic test inputs. Instead, LLMs serve as a semantic solver over source-level path summaries. This design shifts the burden from low-level constraint solving to high-level semantic judgment. ConcoLLMic uses LLM reasoning in a different but related way (Luo *et al.*, 2026). Its summarization agent produces constraints in natural language, source-level code, or SMT-style notation. A separate solving agent then uses these constraints to synthesize a concrete test harness or input, optionally invoking tools such as Z3. In this setting, LLMs not only interpret a path abstraction but also help to bridge the gap between an observed execution, an inferred constraint, and a concrete input capable of exercising the desired behavior.

Component III: Path-Level Grounding. To ground LLM reasoning, AutoBug provides a truncated slice and a Hoare-style verification condition. The prompt therefore contains only the program statements that are relevant to the target post-condition under the selected path

abstraction. This focused formulation reduces irrelevant context and improves practical accuracy. However, the final PASS/FAIL judgment still comes from an approximate LLM oracle rather than a sound SMT proof. Consequently, AutoBug improves the applicability of symbolic reasoning to source-level semantic properties, but it does not provide the same formal guarantees as classical deductive verification. For ConcoLLMic, LLM reasoning is grounded through concrete execution traces, tool feedback, and re-execution checks. Generated constraints, inputs, or test harnesses are validated by running the program again, and a candidate test is retained only when it reaches the intended target lines or increases coverage. This execution-based grounding filters many incorrect LLM suggestions and provides practical evidence that the generated input matches the intended path behavior.

Discussion. These works illustrate a promising synergy between LLMs and classical symbolic execution techniques. A unifying theme is that LLMs are not treated as infallible oracles, but as assistants embedded within a larger analysis pipeline. In contrast to naive end-to-end LLM-based bug detectors, neurosymbolic designs ask LLMs to produce intermediate results, such as source-level path summaries, truncated slices, and natural-language constraints. This key insight is similar to the one used in LLMDFA (Wang *et al.*, 2024b) and RepoAudit (Guo *et al.*, 2025b) discussed in Section 5.1.1, where LLMs generate candidate solutions to local subproblems, such as data-flow summaries for single functions, acting as heuristic engines within a broader analysis. Because these intermediate results are localized to a path, slice, or function, they are easier to inspect, refine, or validate than a global bug report produced by an unconstrained LLM.

Recent Hoare-style neurosymbolic verification techniques extend this line of work by broadening the specification interface. Hoare-Prompt (Bouras *et al.*, 2025) uses strongest-postcondition-style reasoning in natural language. Instead of requiring a fully formal postcondition, it prompts LLMs to propagate natural-language descriptions of reachable program states and check whether they entail a natural-language requirement. FM-Agent (Ding *et al.*, 2026) pushes this idea toward large systems by deriving function-level natural-language pre/postconditions

from caller expectations and checking implementations using compositional Hoare-style reasoning. By allowing semantic intent to be expressed as natural-language requirements or function contracts, HoarePrompt and FM-Agent make higher-level functional bugs more amenable to neurosymbolic analysis. These works effectively enlarge the scope of bug types handled by existing symbolic execution techniques.

Nevertheless, these approaches still face two fundamental limitations. First, performance remains a key challenge. Compared with classical symbolic execution, LLM-driven reasoning incurs non-trivial time and monetary costs, especially when it is applied repeatedly across paths, slices, or functions. As a result, their overall throughput remains limited, motivating the need to reduce redundant reasoning and amortize neural inference costs. Second, reliability remains bounded by the quality of the intermediate results produced by LLMs. For HoarePrompt and FM-Agent, natural-language specifications increase expressiveness, but also raise the risk of ambiguity and LLM hallucinations in the semantic reasoning. Thus, richer semantic reasoning improves applicability, but does not by itself recover the formal guarantees of classical verification. These limitations suggest several promising directions for future work. First, mechanisms such as bi-abduction (Calcagno *et al.*, 2011) and compositional symbolic execution (Lööw *et al.*, 2025) could help infer, cache, and reuse function-level specifications across different execution contexts, thereby avoiding repeated LLM reasoning over the same code. A second direction is to develop more principled mechanisms for validating and refining natural-language specifications. Existing systems such as HoarePrompt and FM-Agent provide early evidence that natural-language specifications can express richer semantic intent, but future analyzers still need better ways to reconcile such specifications with formal constraints and concrete execution results.

5.1.3 Slicing-guided Pattern Matching

While the previous subsection focuses on path-sensitive bugs whose specifications can be well defined, many domain-specific bugs arise from a different source of complexity. These bugs are often caused by violations of domain-dependent semantics, including API usage conventions (Li

et al., 2021), protocol requirements (Zheng *et al.*, 2025c), and system-specific invariants (Huang *et al.*, 2023b; Thijm *et al.*, 2023). Unlike the bugs targeted by data-flow analysis and symbolic execution, which can often be characterized uniformly, domain-specific bugs usually require checking whether an implementation is semantically consistent with external domain knowledge. Such knowledge may be implicit in historical buggy examples or explicitly described in domain documents, such as RFCs for network protocols.

To detect domain-specific bugs, a common practice in industry is to manually implement domain-specific symbolic checkers, such as pattern-based linters. While such checkers can be effective for narrowly scoped targets, they are expensive to develop and maintain, require substantial domain expertise, and usually have limited expressiveness when the target semantics are too complex to be precisely encoded (Johnson *et al.*, 2013). In parallel, learning-based approaches can leverage domain-specific data to predict certain bug classes, but they typically require large, high-quality labeled datasets and often need to be retrained for each bug category. As a result, their applicability is limited in domains where labeled data is scarce, expensive, or highly specialized.

Neurosymbolic Approach. Recent studies explore neurosymbolic approaches that perform general compliance validation against domain-specific specifications, such as BugScope (Guo *et al.*, 2025a) and RFCAudit (Zheng *et al.*, 2025a). Specifically, BugScope treats domain-specific bugs as recurring anti-patterns and derives detection logic from historical buggy examples. RFCAudit targets network protocols and audits implementations against semantic properties extracted from RFC documents. Despite relying on different sources of domain knowledge, they share a similar key idea. Instead of asking LLMs to inspect an entire codebase end-to-end, these systems first obtain an external semantic specification, then retrieve the code fragments relevant to that specification, and finally use LLMs to validate whether the retrieved code is semantically consistent with the specification.

Component I: Semantic Specification Acquisition. The first component is to obtain a usable semantic specification without requiring

developers to manually encode a dedicated checker. Given historical buggy instances and, when available, their corresponding non-buggy or fixed versions, BugScope uses an LLM to infer a compact description of the underlying anti-pattern. This description captures the salient operations, relevant program entities, and semantic conditions that distinguish buggy behavior from legitimate code. Conceptually, this component uses prior failures as demonstrations of domain knowledge. Instead of formalizing the bug class from first principles, the system infers what should be checked from concrete examples. Unlike BugScope, RFCAudit derives specifications from domain documents, such as RFCs, and extracts mandatory requirements that protocol implementations are expected to satisfy. These requirements are usually written in natural language and may involve protocol states, message fields, and error-handling behavior. RFCAudit therefore translates normative textual requirements into semantic properties that can guide implementation auditing. Importantly, the output of this component is not a complete formal specification. Rather, it is a set of checkable semantic obligations that can later be checked against implementation evidence. Overall, BugScope and RFCAudit show that domain-specific auditing can obtain actionable semantic specifications from different external sources, including historical buggy examples and software documents. This observation also connects this line of work to the primitive analysis over non-code artifacts discussed in Section 4.2, since such artifacts can provide useful semantic signals for constructing the semantic specifications that guide subsequent auditing.

Component II: Context Retrieval. After obtaining the semantic specification, the next challenge is to collect the program evidence needed to check it. This step is crucial because domain-specific semantic validation often depends on a small portion of a large implementation. Presenting entire files, modules, or an entire codebase to LLMs would introduce excessive irrelevant context and make the reasoning process unreliable. Both BugScope and RFCAudit therefore use slicing-guided context retrieval to construct a bounded yet semantically concentrated reasoning scope.

BugScope performs statement-level slicing around bug-relevant ex-

pressions or operations. Depending on the anti-pattern being checked, it can generate forward or backward slices across function boundaries. A backward slice may identify statements that influence a suspicious operation, such as value definitions, guards, or earlier API calls. A forward slice may identify how the result of an operation is later used and whether the subsequent behavior matches the inferred buggy pattern. By restricting the reasoning scope to the derived slice, BugScope presents LLMs with the program statements most relevant to the candidate bug, rather than the surrounding implementation noise.

RFCAudit performs context retrieval at a coarser granularity. Given a semantic property extracted from an RFC, it first localizes functions that are likely to implement the corresponding protocol behavior. It then expands the context in a demand-driven manner by selecting relevant callers, callees, and definitions of used data structures. This function-level retrieval process is especially useful for protocol implementations, where the satisfaction or violation of the target property may depend on interactions among parsing logic, state transitions, validation checks, and error-handling routines.

Although BugScope and RFCAudit operate at different granularities, they share the same insight that slicing is used not merely to reduce input length, but to collect sufficient evidence for a semantic claim. BugScope uses statement-level slices to detect low-level semantic anti-patterns, whereas RFCAudit uses function-level slices to audit higher-level protocol requirements. In both cases, the retrieved context acts as the evidential basis on which LLMs perform subsequent semantic validation.

Component III: Semantic Consistency Validation. With the relevant context retrieved, both systems perform semantic consistency validation. BugScope checks whether the program slice exhibits an anti-pattern consistent with the historical buggy examples. This requires LLMs to compare the retrieved implementation behavior with the inferred bug specification and determine whether the slice contains the same semantic defect, rather than merely a superficial syntactic resemblance. RFCAudit checks whether the localized implementation evidence satisfies or violates the mandatory requirements extracted from

RFC documents. This requires LLMs to align natural-language protocol obligations with code. Because both settings involve semantically rich requirements that are only partially formalized, the final validation cannot be reduced to simple syntactic matching. Instead, LLMs are used as semantic verifiers that interpret code evidence in light of the acquired specification. To reduce hallucinations and improve reliability, these systems further incorporate validation mechanisms such as self-consistency checks and self-critique. Such mechanisms help filter unstable model judgments and reduce false positives or false negatives. As a result, slicing-guided pattern matching can handle domain-specific bugs whose semantics are difficult to encode in classical static checkers, while still avoiding fully unconstrained LLM-based codebase inspection.

Discussion. Slicing-guided pattern matching highlights a different style of LLM-assisted auditing. Its workflow is closer to how experienced human auditors inspect domain-specific bugs in practice. Rather than starting from the entire codebase or a fully formalized property, an auditor often begins with external semantic clues, such as a known buggy case, a recurring anti-pattern, or a requirement stated in the system specification. The auditor then asks where this semantic obligation is implemented, identifies critical statements or functions, and follows program dependencies or caller-callee relations to collect the code fragments needed for deeper inspection. Slicing-guided pattern matching uses LLMs to instantiate the step-by-step reasoning process of human auditors and combines this reasoning with call graph analysis to collect the necessary context, thereby enabling the auditing of diverse program properties.

This design fully exploits the strengths of LLMs as general code auditors over a restricted program scope. Earlier neurosymbolic auditing workflows often imitate the structure of traditional analyses. In LLMDFA (Wang *et al.*, 2024b) and RepoAudit (Guo *et al.*, 2025b), for example, LLMs are asked to propagate data-flow facts, summarize intraprocedural data flows, or reason about path constraints. In AutoBug (Li *et al.*, 2025e), program slices are used to construct Hoare-style verification conditions, so LLMs mainly reason about whether a path-level abstraction entails a target postcondition. Slicing-guided pattern

matching places greater emphasis on broader semantic comprehension. LLMs read a bounded but potentially long code context, understand the intent of several statements or functions together, and judge whether their combined behavior is consistent with an external semantic specification. The reasoning target is not limited to data-flow reachability, numeric constraints, or other program facts that are commonly targeted by existing work. Instead, it can also involve API usage conventions, protocol states, and other general program properties that are difficult to formulate as graph properties or logical formulas.

From this perspective, slicing plays a role similar to the evidence collection step in manual auditing. Here, program slicing is used in a broader sense. It is not limited to classical statement-level slicing, but also includes collecting relevant functions connected to a target function through the call graph. The collected code fragments enable the auditor to narrow the auditing scope and pay sufficient attention to the surrounding implementation context, thereby supporting a more careful examination of the semantic consistency between the implementation and the specification.

Nevertheless, the effectiveness of this paradigm depends on both the quality of the semantic specification and the completeness of the retrieved context. If the historical examples do not cover important variants of the bug, the inferred anti-pattern may be too narrow or may overfit incidental implementation details. If the document-derived requirement is ambiguous or contains implicit exceptions, the extracted semantic obligation may be unstable. Similarly, if the slice omits a relevant caller, callee, state update, or error-handling path, LLMs may make a judgment based on incomplete evidence. Thus, future systems need adaptive context retrieval mechanisms that can expand the audit context when the current evidence is insufficient, as well as lightweight validation procedures that can sanity-check LLM semantic judgments.

5.2 Simulating Semi-Automatic Code Auditing

This section investigates how LLMs can be integrated with existing static bug detectors to support *semi-automatic code auditing*. Specifically, we identify three primary modes in which LLMs act as human-auditor-

like assistants by configuring and customizing static bug detectors, as well as interpreting their analysis results. First, LLMs can support *bug detector synthesis* by formalizing bug specifications from examples, documentation, or code changes into bug-detection logic, mirroring how human auditors translate informal external artifacts into auditable properties. Second, LLMs can assist in *bug detector tuning* by adjusting the parameters and abstraction choices of the static analysis engine, such as context sensitivity and memory abstraction, mirroring how human auditors tune the analysis backend to balance accuracy and scalability. Third, LLMs can be applied to *bug report post-processing*, where they analyze, filter, cluster, or explain analysis results, simulating how human auditors inspect bug reports and remove false positives. In what follows, we discuss these three modes of semi-automatic code auditing in detail.

5.2.1 Bug Detector Synthesis

A central strength of static bug detectors lies in their ability to rigorously audit program behavior against well-defined bug specifications. In practice, however, effective auditing often requires customized bug detectors that encode bug-specific and domain-specific semantics. Specifically, different classes of bugs, such as data-flow bugs and other domain-specific bugs, demand distinct detection logic, e.g., setting different forms of sources and sinks. Moreover, bugs in real-world programs often involve external libraries and frameworks, whose semantics are rarely fully captured by default bug detectors. To bridge this gap, human auditors are typically required to manually formalize library behavior through auxiliary specifications, such as API usage protocols, and integrate these specifications into bug detectors (Antoniadis *et al.*, 2020). As a result, bug detectors are not standalone analyzers. Instead, they require manual customization that embeds expert knowledge about both target bug patterns and the surrounding software ecosystem.

Unfortunately, existing static bug detection infrastructures often provide limited practical customizability. Writing customized bug detectors in systems such as CodeQL (Avgustinov *et al.*, 2016) or LLVM IR-based detection frameworks (Sui and Xue, 2016; Shi *et al.*, 2018)

often requires deep familiarity with complex IRs, extensive utility libraries, and nontrivial analysis interfaces, making the process largely inaccessible to non-experts. These customizability limitations are further amplified in library semantic modeling, where the diversity and scale of third-party APIs render manual specification both labor-intensive and error-prone (Bastani *et al.*, 2018; Chibotaru *et al.*, 2019). Recent advances in LLMs, however, offer a promising alternative. Specifically, LLMs can interpret natural-language descriptions, documentation, and examples, and synthesize structured rules or analysis logic with minimal human intervention. This capability has motivated a growing line of work on bug detector synthesis, in which LLMs assist with or automate the generation of bug detectors and specifications, improving practical customizability while preserving the rigor of symbolic reasoning.

Neurosymbolic Approach. To synthesize bug detectors, existing techniques typically follow a three-stage workflow: (*i*) inferring or recovering bug- or domain-specific analysis specifications from external artifacts, (*ii*) synthesizing bug detector candidates under the constraints imposed by a bug-detection engine, and (*iii*) iteratively refining detector candidates using execution feedback. In what follows, we use QLCoder (Wang *et al.*, 2025a), MoCQ (Li *et al.*, 2025c), KNighter (Yang *et al.*, 2025a), and IRIS (Li *et al.*, 2025f) as representative systems to illustrate this workflow. While they target different symbolic bug-detection infrastructures, e.g., CodeQL (Avgustinov *et al.*, 2016) and Clang Static Analyzer (LLVM Project, 2026), their underlying techniques are largely infrastructure-independent.

Component I: Bug Specification Inference. Most approaches begin by grounding bug detector synthesis in bug-centric evidence. QLCoder and MoCQ leverage public artifacts such as CVE reports, GitHub issues, and patch commits to distill bug semantics and to support evaluation against both true positives and false negatives. KNighter similarly relies on historical fixes from the Linux kernel ecosystem, where patches encode recurring bug patterns and domain-specific conventions. In contrast, IRIS does not derive specifications from external non-code artifacts. Instead, it extracts invoked library APIs from the target

program and infers source and sink specifications to enable taint-style bug detection. Benefiting from pretraining on large-scale code and documentation corpora, LLMs can effectively model the semantic roles of library APIs. Overall, bug specification inference determines the detection logic of the bug detector and serves as a prerequisite for subsequent detector generation and refinement.

Component II: Detector Candidate Generation. Given inferred specifications, a valid bug detector is expected to detect buggy instances while excluding non-buggy ones. Existing techniques employ several strategies to generate such candidates. The most common design is to directly synthesize detector implementations using LLMs guided by predefined detector templates. When multiple buggy examples are available, MoCQ synthesizes per-example detectors and then merges them into a unified one, simplifying the generation problem in each round and increasing the likelihood of producing a valid detector candidate. To further accelerate synthesis, KNighter and QLCoder retrieve relevant prior detectors and documentation from a knowledge base, enabling LLMs to start from structurally similar checkers. Collectively, these strategies provide effective heuristics for exploring the search space and accelerating the synthesis of valid bug detectors.

Component III: Detector Candidate Refinement. Once detector candidates are generated, existing approaches rely on iterative refinement to correct errors when necessary. If a detector candidate is syntactically invalid or violates the inferred specifications, LLMs are prompted to repair it using tool feedback, such as compilation errors or missing predicates. Beyond syntactic repair, MoCQ explicitly emphasizes *generalization* to avoid overfitting to individual instances, for example by removing irrelevant constants or overly specific conditions. To further improve detector quality, KNighter performs dry runs of synthesized detectors and applies LLM-based triage to assess whether reported alarms are likely false positives. The resulting feedback is used to iteratively refine detection logic, such as by adding guards or refining state tracking, thereby improving detection accuracy in practice.

Discussion. These neurosymbolic systems use LLMs primarily for bug specification inference and detector generation/repair, closely mirroring how human auditors interact with static bug detectors in practice, i.e., first formalizing analysis requirements and then writing, debugging, and refining custom detectors. In this sense, LLMs act as a surrogate for human expertise in the most labor-intensive components of static bug detection, while the static bug detector retains responsibility for rigorous and scalable auditing.

It is worth noting that bug detector synthesis itself is not a new problem. Prior to the emergence of LLMs, substantial research explored the synthesis of Datalog programs (Li *et al.*, 2021) and other domain-specific bug detectors (Naik *et al.*, 2021; Garg and Sengamedu, 2022), which typically generate static bug detectors from examples and share problem settings similar to those of the works discussed above. However, compared with this line of work, LLM-driven bug detector synthesis offers two notable advantages. First, LLMs can autonomously derive bug specifications and better capture customized analysis requirements, especially when such requirements are embedded in unstructured non-code artifacts such as GitHub issues, patch descriptions, or natural-language bug reports. Second, LLMs bring strong prior knowledge about programming languages, APIs, and common bug patterns, which significantly increases the likelihood of quickly synthesizing valid detector candidates during generation and refinement. In contrast, non-LLM approaches, even those augmented with classical machine learning techniques, typically rely on syntax-guided enumerative search. Although these methods incorporate various heuristics to prune the search space, they remain fundamentally limited by the combinatorial complexity of the problem. Empirically, LLM-guided synthesis is substantially more effective at navigating this space, often converging to valid detectors with far fewer iterations, which helps explain the practical success of LLM-driven synthesis of static bug detectors.

Additionally, the problem setting of bug detector synthesis shares notable similarities with the slicing-guided pattern matching discussed in Section 5.1.3. In particular, BugScope (Guo *et al.*, 2025a) likewise derives semantic specifications from examples and natural-language descriptions, and strategically collects program contexts to support

bug detection over a deliberately limited analysis scope. Compared with BugScope, however, KNighter, MoCQ, and QLCoder externalize the auditing logic into synthesized bug detectors. This design choice is central to achieving scalable code auditing over large-scale codebases. Once the bug detectors are synthesized, analysis proceeds entirely within the static bug-detection engine, without further LLM invocations. As a result, these approaches offer substantial advantages in efficiency, in contrast to BugScope that relies on LLM-driven code semantic reasoning. That said, this reliance on underlying static analysis infrastructures also constrains the expressiveness of the synthesized bug detectors. In practice, the techniques discussed above primarily target data-flow bugs and remain limited in their ability to address more general bug categories, such as divide-by-zero, buffer overflows, or functional bugs. By contrast, systems such as BugScope (Guo *et al.*, 2025a) and RFCAudit (Zheng *et al.*, 2025a) discussed in Section 5.1.3 exploit the general reasoning capabilities of LLMs to cover a much wider range of bug types, albeit at the cost of potentially higher token consumption during analysis.

5.2.2 Bug Detector Tuning

Beyond synthesizing new static bug detectors, effective code auditing often requires careful *tuning* of existing bug detectors. In practice, static bug detectors expose a wide range of configurable parameters (Cha *et al.*, 2022; Jeon *et al.*, 2021) that directly affect analysis precision and scalability. Many of these parameters determine the granularity of the abstractions used by the analyzer. For example, in numeric abstract interpretation, an analyzer may approximate program states using intervals, octagons, polyhedra, or other abstract domains. A coarse interval abstraction may record only facts such as $0 \leq x \leq 10$, whereas a more relational domain may additionally capture constraints such as $x - y \leq 3$. In interprocedural analyses, parameters such as context sensitivity, field sensitivity, and heap abstraction determine whether different call sites, fields, or allocation sites are distinguished or merged.

These examples illustrate the central trade-off behind bug detector tuning. More precise configurations typically preserve finer-grained semantic distinctions and reduce false positives, but they also increase the

number of abstract states, paths, or value-flow facts that the analyzer must maintain. Conversely, coarser configurations improve scalability but may introduce false positives in the bug detection. Selecting appropriate configurations is therefore rarely straightforward and typically relies on substantial human expertise and iterative experimentation. As a result, bug detector tuning constitutes an essential yet labor-intensive component of semi-automatic code auditing.

From the perspective of human auditors, tuning static bug detectors is an adaptive process guided by both domain knowledge and empirical feedback. For example, when an analysis produces excessive false positives, human auditors may increase precision by selecting a more expressive abstract domain, distinguishing more calling contexts, and preserving field-level distinctions. Conversely, when an analysis fails to scale to large programs, they may relax these abstractions by using a coarser numeric domain, reducing context sensitivity, merging heap objects by allocation site, or applying widening more aggressively. These decisions are inherently heuristic and program-dependent, and are difficult to encode as fixed symbolic rules or universally optimal configurations.

Neurosymbolic Approach. Recent neurosymbolic approaches leverage LLMs to support this tuning process by acting as high-level advisors that guide abstraction choices, while leaving soundness-critical reasoning to the underlying static bug detectors. A representative example is ABSINT-AI (Wang *et al.*, 2025b), which integrates LLMs into an abstract interpretation framework to determine *where* and *how* abstraction should be applied. Rather than performing program analysis directly, LLMs are queried to decide which variables, objects, or fields should be abstracted together and which ones should remain distinguished. Crucially, within a sound over-approximating analysis framework, these decisions affect only analysis precision and efficiency. Specifically, an overly coarse abstraction may introduce false positives by merging semantically different program states, while an overly precise abstraction may delay convergence and degrade scalability by preserving distinctions that are irrelevant to the target bug specification. Neither case compromises the soundness of the underlying abstract interpretation, as

long as the chosen abstraction remains a sound over-approximation of the concrete semantics. This asymmetric risk profile makes abstraction tuning particularly well suited for LLM guidance.

Discussion. More broadly, LLM-driven bug detector tuning can be viewed as a form of *adaptive semantic abstraction*. By exploiting natural-language cues such as variable names, type names, API names, and developer comments, LLMs can infer which program elements are conceptually similar, which distinctions are likely irrelevant to the auditing task, and which values are likely to evolve unboundedly across executions. For instance, an LLM may infer that a loop counter, an array index, and a container size should not be merged in an array-bounds analysis, whereas several temporary logging fields may be safely abstracted together. These judgments closely mirror how human auditors reason about abstraction choices, yet they are difficult to formalize within classical static analysis.

Notably, bug detector tuning differs from bug detector synthesis in both scope and cost. While synthesis externalizes auditing logic into newly generated rules, tuning operates within the parameter-configuration space of an existing bug detector. This design enables lightweight interaction between LLMs and static analysis tools, often requiring only limited contextual information and avoiding repeated detector generation and refinement. As demonstrated in systems such as ABSINT-AI, LLM-guided tuning can substantially reduce false positives and, in several cases, improve runtime by preventing state-space explosion caused by overly precise abstractions.

5.2.3 Bug Report Post-Processing

False positives are pervasive in automated code auditing and often impose substantial manual inspection effort on human auditors. These false alarms often stem from two fundamental limitations of static analysis. First, due to the semantic barrier discussed in Section 2.2.2, taint-style bug detectors, for example, frequently fail to recognize sanitization logic, leading to spurious reports even when taint flows are properly sanitized. Second, imprecise program abstractions, such as

flow-insensitive or context-insensitive analyses, may over-approximate program behavior and introduce additional false positives. As a result, many reported bugs require substantial manual inspection before they can be confirmed or dismissed.

As discussed earlier, LLMs acquire rich semantic knowledge and excel at reasoning about code semantics within a limited scope. This capability makes LLMs particularly well suited for post-processing static bug-detection results, where the analysis scope is already narrowed to a small set of suspicious code fragments. Consequently, leveraging LLMs to refine and validate bug reports produced by static bug detectors has emerged as a natural and effective direction.

Neurosymbolic Approaches. A growing body of work explores neurosymbolic designs for bug report post-processing. IRIS exemplifies this paradigm in the context of taint-style bug detection (Li *et al.*, 2025f). It first invokes CodeQL to identify potential bugs and extract the corresponding functions involved in the reported taint flows. These functions are then provided to an LLM for semantic validation. In particular, if the LLM identifies sanitization logic within any function along the reported flow, the bug report can be safely ruled out. Moreover, false positives caused by the lack of context sensitivity in CodeQL can be effectively filtered, as LLMs can reason about calling contexts and semantic constraints beyond the abstraction of the static bug detector. A similar design is adopted by LLM4SA, which feeds localized buggy contexts into LLMs and employs diverse prompting strategies to interpret and validate bug reports (Wen *et al.*, 2024).

LLM4PFA focuses on reducing false positives arising from infeasible execution paths. Rather than merely presenting the relevant functions and path constraints to LLMs, LLM4PFA introduces an LLM agent that performs lightweight range analysis to approximate argument values and function return values (Du *et al.*, 2025). This additional reasoning step enables more precise modeling of path conditions, especially when feasibility depends on data-dependent predicates. As a result, LLMs can more accurately determine whether a reported path is executable in practice.

LLMSAN adopts a complementary strategy by decomposing bug

reports into syntactic properties and semantic properties (Wang *et al.*, 2024c). The former are verified soundly using lightweight static analysis, while the latter are checked by LLMs over a restricted code scope to mitigate hallucination. Although LLMSAN primarily targets LLM-generated bug reports, its core idea, i.e., separating sound static analysis from flexible LLM-driven semantic validation, has influenced post-processing techniques for general bug reports. For instance, similar principles are adopted in other techniques, such as BugLens (Li *et al.*, 2025a), to validate static bug-detection reports more reliably.

Discussion. Across these approaches, neurosymbolic designs typically combine lightweight context retrieval with LLM-based reasoning. Common retrieval mechanisms include call graph analysis and control-flow analysis, which extract a small but relevant subset of code related to a reported bug. This workflow closely mirrors how human auditors rely on IDE-provided static-analysis support, such as *go to definition*, *find usages*, and *call hierarchy navigation*, to navigate a codebase and inspect the code fragments relevant to a bug report.

Because the analysis scope is usually limited to only a few functions and LLMs are tasked with validating specific semantic properties, bug report post-processing often achieves substantial precision improvements with relatively low computational cost. In contrast to the techniques in Section 5.1, which simulate manual code auditing and therefore place LLMs in a dominant role for semantic reasoning while using symbolic analysis as a lightweight auxiliary component, bug report post-processing lies at the opposite end of the neurosymbolic spectrum. Here, heavyweight static bug detectors are responsible for codebase-wide bug discovery, while LLMs focus on semantic validation of the reported code fragments.

This architectural distinction fundamentally shapes the strengths and limitations of the approach. Since LLMs are applied only to a small number of reports, token usage and runtime overhead remain modest, making these techniques easier to scale to large codebases. However, their reliance on classical static bug detectors also means that they inherit longstanding limitations, such as dependence on specific IRs and restricted support for certain bug classes. As a result, bug report

post-processing improves precision and usability but does not directly resolve the inherent limitations of static bug detectors.

5.3 Industrial Advances in Neurosymbolic Code Auditing

Recent industrial advances provide a concrete view of how neurosymbolic code auditing has evolved alongside advances in LLMs and AI coding agents. Early industrial adoption did not immediately produce open-ended codebase-wide auditors. Because earlier LLMs had limited reasoning capability, constrained context windows, and non-trivial inference costs, industrial efforts mainly followed two practical routes. One route placed LLMs in change-centered review, where the observation scope is restricted to a pull request, recent edit, or local code region. The other route coupled LLMs with existing static-analysis infrastructure, where symbolic tools provide intermediate findings or bug reports, and LLMs assist with customization, interpretation, and remediation. These two routes developed largely in parallel because both reduce the reasoning burden placed on the model.

As LLMs became stronger and coding agents gained better tool-use and orchestration abilities, industrial code auditing systems began to expand from local changes and analyzer-centered workflows to multimodal codebase-wide security auditing. In this setting, an auditing system reasons not only over source code, but also over non-code artifacts, as discussed in Section 4.2. More recent systems further strengthen generated hypotheses with executable or checkable witnesses, such as proof-of-concept inputs, fuzzing harnesses, and sandbox reproductions. This progression should not be interpreted as a fixed architecture that every industrial auditor must implement end-to-end. Rather, it reflects a temporal and methodological evolution in the role of LLMs in code auditing. In practice, different forms of assistance can be deployed at different points in the workflow, e.g., review assistants in IDEs and pull requests, LLM-powered static-analysis customization in the CI/CD workflow or security dashboards, multimodal codebase-wide security auditing in security review campaigns, and executable validation for high-risk candidates.

5.3.1 Change-Centered Code Review

One early route embeds LLMs into pull-request or IDE-based review. The system starts from a diff and retrieves a bounded amount of surrounding context, then asks LLMs to identify possible violations of correctness, security, or project-specific expectations. This route is natural under early constraints on capability and cost because the auditing scope is deliberately small. Instead of requiring the model to maintain a whole-codebase view or perform long-range interprocedural reasoning, the system presents the model with a localized review task that resembles ordinary human code review.

In terms of the agentic framework introduced in Chapter 3, this route mainly combines retrieval tools with LLM-driven semantic judgment over a narrow working context. Cursor BugBot (Anysphere, 2026), Claude Code security review (Anthropic, 2026b; Anthropic, 2026c), and Uber uReview (Choudhary *et al.*, 2025) instantiate this auditing mechanism. Although these systems differ in interface, they share a demand-driven observation model: the analysis surface is narrowed to recently changed code, and additional context is retrieved only when needed for semantic review, which follows insights similar to those of slicing-guided pattern matching methods discussed in Section 5.1.3. Codebase guidance files, such as `BUGBOT.md` or `AGENTS.md`, further act as lightweight project-specific specifications, allowing teams to encode security assumptions or review priorities in a form that can be consumed by the agent.

This route fits existing developer workflows and can provide immediate feedback before a change is merged. Its limitation is that the evidence remains local. It is useful for local regressions, missing checks, API misuse, and suspicious edits, but weaker for bugs whose evidence is distributed across files, modules, and frameworks. Thus, change-centered review demonstrates the usability of LLMs as semantic reviewers, while also revealing why stronger tool grounding and broader observation scopes are needed for deeper auditing.

5.3.2 LLM-Powered Static Analysis Augmentation

A parallel early route augments existing static-analysis infrastructure with LLMs. This route was also shaped by the limited reasoning ability and high inference cost of early LLMs, but it reduces the model's burden in a different way. Instead of asking the LLM to inspect a large codebase directly, the static analyzer provides the main analysis structure. The analyzer scans the codebase, generates candidate findings, exposes data-flow or control-flow evidence, and offers a configurable framework for recurring bug patterns. The LLM is then placed around this symbolic core to perform the kinds of auxiliary reasoning that human auditors usually conduct when using static analyzers, such as configuring analyzers, interpreting warnings, prioritizing reports, explaining evidence, and suggesting fixes.

This route is closely related to the semi-automatic auditing paradigm discussed in Section 5.2. In both cases, the static analyzer remains responsible for scalable codebase-wide reasoning, while the LLM augments the surrounding human-in-the-loop workflow. GitHub Copilot Autofix with CodeQL (GitHub, 2024) is a representative example. Specifically, CodeQL provides structured findings such as alert locations and data-flow paths, while the LLM constructs explanations and targeted fix suggestions. Amazon Q Developer (Tandon *et al.*, 2024) and Amazon Inspector Code Security (Desai *et al.*, 2025) follow a similar pattern by combining static scanning with generative remediation support. CodeRabbit (CodeRabbit, 2026; Giannini and Gill, 2025) also partially fits this direction because it combines PR review with static scanners, project memory, and LLM-based reasoning.

The main advantage of this route is that LLM reasoning is grounded in analyzer-produced evidence and, when possible, externalized into analyzer-compatible artifacts. Compared with change-centered review, this provides stronger traceability and inherits the scalability of symbolic static analysis tools. However, the approach remains bounded by the analyzer's semantic coverage. As remarked at the end of Section 5.2.1, if bug types are beyond the scope of the static analyzer, such a code auditing workflow cannot detect them effectively.

5.3.3 Multimodal Agentic Auditing Orchestration

With stronger model reasoning and more mature tool-use capabilities, industrial systems began to move beyond the two early constrained routes. Instead of limiting LLMs to local changes or analyzer-produced warnings, these systems use agents to orchestrate evidence collection across a multimodal auditing environment that contains source code and various forms of non-code artifacts. This broader observation scope enables security-oriented tasks such as entry-point discovery, trust-boundary analysis, threat modeling, variant analysis, and cross-component vulnerability search.

OpenAI Codex Security (OpenAI, 2026b) and Aardvark (OpenAI, 2025) illustrate this direction by constructing codebase-specific threat models, inspecting project history, identifying candidate bugs, validating them in isolated environments, and surfacing patches for human review. Microsoft MDASH (Microsoft, 2026) further exemplifies orchestration at scale by coordinating specialized agents for preparation, scanning, validation, deduplication, and exploit evidence construction.

The methodological shift is from bounded assistance to agentic orchestration. The system no longer only comments on a recent change or explains an analyzer warning. Instead, it maintains hypotheses, retrieves evidence from heterogeneous artifacts, invokes tools, and iteratively refines the audit trajectory. This makes the approach suitable for bugs whose evidence is distributed across the software environment, such as business-logic bugs, authorization errors, variant bugs, and subtle memory-safety issues.

5.3.4 Executable Validation as Evidence Grounding

The most recent direction strengthens agentic auditing by grounding candidate findings in executable or otherwise checkable artifacts. This direction should not be treated as a separate line of dynamic analysis. Rather, execution plays the role of evidence grounding. It converts a hypothesis produced by an LLM, a static analyzer, or an AI agent into an auditable witness, e.g., a proof-of-concept input. Compared with natural-language explanations or analyzer warnings, these artifacts provide stronger evidence because they connect a suspected bug to

observable program behavior.

Several industrial systems instantiate this principle. Google's OSS-Fuzz work (Liu *et al.*, 2023; Chang *et al.*, 2024) uses LLMs to generate or improve fuzz targets, while compilation feedback, coverage, crashes, sanitizers, and triage provide the actual validation evidence. The same idea appears in Big Sleep (Google, 2024) and CodeMender (Google, 2025), where codebase-wide reasoning is connected to bug evidence and patch validation. OpenAI Codex Security (OpenAI, 2026b) attempts to reproduce candidate vulnerabilities in clean containers before marking them as validated. Microsoft MDASH (Microsoft, 2026) includes a proof stage that constructs and executes triggering inputs when possible.

This direction changes the epistemic status of an LLM-generated finding. A plausible explanation or analyzer warning becomes more convincing when it is accompanied by an executable witness. However, such validation remains difficult for most current models. In practice, attempts to synthesize PoC inputs or reproduce vulnerabilities often fail because the agent cannot properly configure the environment, construct a faithful harness, or determine whether the intended bug has actually been triggered. Thus, executable validation strengthens evidence, but also introduces substantial engineering cost. Recent results from Anthropic's Claude Mythos model (Carlini *et al.*, 2026; Anthropic, 2026d) suggest that it may substantially improve end-to-end bug discovery and exploit construction. In the future, a key practical challenge is how to obtain such executable evidence reliably and at lower cost, rather than relying on expensive, highly capable models.

5.4 Summary

This chapter examines neurosymbolic code auditing through the lens of human auditing behavior. Rather than treating LLMs as standalone bug detectors, the surveyed systems instantiate auditing agents that emulate how human auditors inspect code, interpret external artifacts, configure tools, and validate findings. Following the two auditing modes introduced at the beginning of this chapter, we discuss two broad families of designs. The first family simulates manual code auditing, where LLMs perform semantic reasoning over focused program contexts

constructed by data-flow analysis, symbolic execution, or slicing-guided retrieval. The second family simulates semi-automatic code auditing, where existing static bug detectors remain the main codebase-wide analysis engines, while LLMs assist in synthesizing detectors, tuning analysis configurations, and post-processing bug reports. Overall, these designs show that the central role of LLMs in neurosymbolic auditing is not merely to produce final bug reports, but to discover and interpret semantic information that is difficult for classical symbolic analysis to capture directly.

Across the representative systems, a recurring design pattern emerges. Most systems decompose code auditing into several smaller stages, for example, acquiring a semantic specification, retrieving relevant program evidence, performing localized LLM-based reasoning, and grounding or validating the resulting judgment. Concretely, LLMDFA (Wang *et al.*, 2024b) and RepoAudit (Guo *et al.*, 2025b) decompose data-flow bug detection into source and sink identification, function-level data-flow summarization, and path feasibility checking. BugScope (Guo *et al.*, 2025a) and RFCAudit (Zheng *et al.*, 2025a) start from external domain knowledge, such as historical buggy examples or RFC requirements, and then use slicing-guided retrieval to collect the code evidence needed for semantic consistency checking. IRIS (Li *et al.*, 2025f), KNighter (Yang *et al.*, 2025a), MoCQ (Li *et al.*, 2025c), and QLCoder (Wang *et al.*, 2025a) instead place LLMs around static-analysis infrastructures, using them to infer specifications, synthesize analyzer-compatible rules, refine detector candidates, or validate analyzer-produced reports. Industrial systems such as MDASH (Microsoft, 2026) follow a similar principle at the orchestration level, coordinating specialized agents for preparation, scanning, validation, deduplication, and exploit-evidence construction. These systems differ in implementation details, but they share the same high-level insight that LLMs are most effective when they reason over bounded, semantically concentrated evidence, while symbolic tools provide structure, scalability, and grounding.

The surveyed systems also differ substantially in how they validate LLM-produced intermediate results. Some approaches rely on solver-based validation. For instance, LLMDFA (Wang *et al.*, 2024b) synthesizes constraints from inferred data-flow paths and delegates

satisfiability checking to Z3 (Moura and Bjørner, 2008), thereby using symbolic solving to reduce hallucinated path judgments. Other approaches rely more heavily on reasoning-model validation, as in RepoAudit (Guo *et al.*, 2025b), where large reasoning models assess the feasibility of candidate data-flow paths. Systems such as BugScope (Guo *et al.*, 2025a) and RFCAudit (Zheng *et al.*, 2025a) validate semantic consistency through LLM-based reasoning over retrieved code evidence and external specifications. In contrast, detector-synthesis systems such as KNighter (Yang *et al.*, 2025a) and MoCQ (Li *et al.*, 2025c) validate generated detectors through dry runs and alarm triage during iterative refinement. Industrial systems further extend this validation spectrum by incorporating workflow-level and executable evidence. For example, MDASH (Microsoft, 2026) coordinates validation, deduplication, and exploit-evidence construction, and includes a proof stage that constructs and executes triggering inputs when possible. Thus, validation in neurosymbolic code auditing is not a single mechanism, but a spectrum ranging from formal solvers and static analyzers to LLM self-checking and concrete execution.

These cross-system observations reveal a broader design space for neurosymbolic code auditing. When symbolic analysis dominates the pipeline, as in KNighter (Yang *et al.*, 2025a), MoCQ (Li *et al.*, 2025c), and IRIS (Li *et al.*, 2025f), auditing logic is externalized into explicit rules, specifications, or analyzer-produced traces. Such designs are typically efficient, scalable, and interpretable, but their coverage remains constrained by the expressiveness of the underlying symbolic infrastructure. When LLM-driven reasoning dominates, such as BugScope (Guo *et al.*, 2025a), RFCAudit (Zheng *et al.*, 2025a), and OpenAI Codex Security (OpenAI, 2026b), the code auditing systems can handle more diverse and less formally specified bug types, including domain-specific bugs, protocol violations, business-logic errors, and other semantic inconsistencies. However, these designs usually incur higher token and runtime costs, and their reasoning steps are less transparent and harder to validate. The most promising systems therefore occupy the middle of this spectrum. Specifically, they should use LLMs to interpret informal semantics from code, documentation, patches, issue reports, and project-specific guidance, while using symbolic analysis, retrieval, execution, or

tool feedback to constrain and validate the reasoning process. In this sense, neurosymbolic code auditing mitigates the drawbacks of classical static analysis discussed in Section 2.2.2 not by replacing static analysis, but by expanding the sources of semantic evidence available to the auditing agent and by grounding LLM-driven semantic reasoning in explicit, inspectable analysis workflows.

6

Conclusion and Future Directions

This survey has treated code auditing as an evidence-seeking workflow, rather than as a single prediction task. Chapter 2 first explains why this workflow is difficult. Classical static analysis provides structure and validation, but it often relies on buildable programs, reusable specifications, and stable intermediate representations. LLM-driven analysis is more flexible, but it lacks the reliability needed for high-stakes auditing. Chapter 3 then presents an agent-centric view, where an auditor observes a rich environment, invokes tools, and stores evidence in memory. Chapter 4 and Chapter 5 then show this workflow in concrete systems. LLMDFA (Wang *et al.*, 2024b) and RepoAudit (Guo *et al.*, 2025b) decompose data-flow auditing into source and sink identification, summary construction, and path validation. DAInfer (Wang *et al.*, 2024a) and ChatDetector (Yang *et al.*, 2025b) recover specifications from documentation. BugScope (Guo *et al.*, 2025a) and RFCAudit (Zheng *et al.*, 2025a) ground domain-specific checks in examples and RFCs. KNighter (Yang *et al.*, 2025a), MoCQ (Li *et al.*, 2025c), QLCoder (Wang *et al.*, 2025a), and IRIS (Li *et al.*, 2025f) compile LLM-derived reasoning into analyzer-ready artifacts. Aardvark (OpenAI, 2025), OpenAI Codex Security (OpenAI, 2026b), and MDASH (Microsoft, 2026) connect re-

ports to validation-oriented industrial workflows. These systems differ in target properties and engineering choices, but they point to the same division of labor. LLMs interpret local semantics and non-code artifacts. Symbolic tools provide structure, scale, and validation.

Taken together, these chapters suggest a natural organizing principle for future work. Practical auditing is limited by three coupled bottlenecks. The first is **observation**, which concerns what the auditor can recover from code, documentation, configuration, history, and tool outputs. The second is **verification**, which concerns what the auditor can confirm strongly enough to justify triage, remediation, or regression guards. The third is **memory**, which concerns which program facts, assumptions, and evidence can be retained across files, commits, and audit tasks.

This view also clarifies why stronger models alone are not enough. The barriers discussed in Section 2.2.2, including IR reliance, limited customizability, and the semantic barrier, do not disappear in agentic systems. They reappear as questions about what the auditor can observe, what it can afford to verify, and what it can preserve as durable evidence. As neurosymbolic auditing moves from prompt-based bug spotting to repository-scale workflows, the central research problem therefore shifts. The key question is no longer whether an LLM can produce a plausible explanation. It is whether an auditing stack can repeatedly produce stable, actionable, and reviewable security conclusions.

The remainder of this chapter is organized around these bottlenecks, and each direction extends a concrete line of work surveyed earlier rather than introducing a separate agenda. Section 6.1 targets observation by making software and its ecosystem more auditable. Section 6.2 targets verification and memory, first by treating auditing as a budgeted process and then by compiling its intermediate results into reusable artifacts. Two further concerns then decide whether progress on these bottlenecks can be trusted, namely whether it can be measured honestly and whether it survives an adversarial environment. Because both cut across all three bottlenecks rather than forming a fourth, Section 6.3 treats evaluation realism and auditor robustness on their own. Section 6.4 then summarizes the chapter.

6.1 Auditable Software and Ecosystem Boundaries

The observation bottleneck starts before an analyzer runs. The specification recovery systems in Section 4.2 and Section 5.1.3 show the immediate motivation. Many security properties are not written in a form that any tool can directly consume. They are scattered across code idioms, comments, documentation, issue discussions, configuration files, and deployment conventions. LLMs are useful because they can interpret these weak signals. However, inference from weak signals should not be the long-term foundation of reliable auditing. Future systems should make more of the security-relevant environment explicit.

6.1.1 Auditability-by-Design Interfaces

A promising direction is to make software more auditable by design. The goal is not to require full formal specifications for every function. That burden is too high for most active codebases. The goal is narrower. Security-critical boundaries should expose enough intent that an auditor can check claims, rather than reconstructing them from conventions.

Such interfaces may include lightweight contracts at parsers, privilege checks, serialization boundaries, and other sensitive seams. They may also record effects such as file-system access, network behavior, persistence, and trust-boundary crossings. For taint-style analysis, they may identify sources, sinks, sanitizers, and declassification points. These examples are not meant to define a universal schema. They illustrate a design principle. The most valuable interface is a small and stable semantic surface that makes high-impact assumptions visible to humans and tools.

This direction connects directly to the agent-centric environment in Chapter 3. If the environment contains explicit semantic surfaces, the agent no longer has to infer all intent from raw source code. Its task becomes specification completion and validation. This is already visible in the systems surveyed in Section 4.2 and Section 5.1.3. DAInfer and ChatDetector infer library specifications from documentation and then ground them through optimization, decomposition, or consistency checking. BugScope and RFCAudit derive domain specifications from examples and RFCs, then check whether retrieved code evidence satisfies those

expectations. These systems already instantiate the loop we expect to become central. The LLM proposes a semantic claim. A symbolic, structural, or evidence-grounded step decides how much trust that claim deserves. Auditability-by-design therefore asks future repositories to expose the semantic hooks that these systems currently recover after the fact.

This framing also suggests a notion of auditability debt. A project accumulates this debt when security assumptions are repeatedly rediscovered rather than recorded. Examples include undocumented sanitization responsibilities, unclear ownership of validation, and configuration flags whose security meaning is known only to a few maintainers. Reducing this debt would make future neurosymbolic auditors more effective for a reason that is independent of model scale. The software itself would become easier to audit.

6.1.2 Ecosystem-Level Auditing

Modern bugs often arise at ecosystem boundaries rather than inside a single function. A vulnerability may depend on how a library is configured, how a service is deployed, how a dependency changed, or how data crosses a language boundary. This extends the semantic-barrier problem from Section 2.2.2. The relevant semantics may lie outside the source file, outside the programming language, and sometimes outside the repository.

Future auditors should therefore treat the audit target as a graph of interacting artifacts. Source code remains central, but configuration, build scripts, lockfiles, API documents, deployment manifests, and patch histories are also semantic inputs. A local build failure should not end the audit if shipped behavior is determined by CI-only toolchains or feature flags. Similarly, a dependency update should not be summarized only as a version change. The relevant question is how the update changes reachable behavior in the host system.

LLMs can help connect these heterogeneous artifacts because they can read natural language and project conventions. Symbolic tools remain necessary because the final claim must be grounded in reachability, data flow, control flow, or executable evidence. A useful artifact in this setting is a boundary summary. It records what a component assumes

about inputs, outputs, privileges, authentication context, encoding, or resource ownership. If such summaries can be checked and invalidated under change, ecosystem-level auditing becomes more than broad retrieval. It becomes cross-component reasoning with explicit evidence.

6.1.3 Specification Completion under Scarcity

Specification scarcity is a recurring obstacle from Chapter 2 through Chapter 5. Classical analyzers often need precise specifications before they can search for a property. LLM-based systems can infer approximate specifications, but these inferred specifications are unstable if they are not grounded. The examples above show why future work should treat specification recovery as a first-class auditing task, not as a hidden prompting step.

The desired output is not merely a natural-language statement of intent. It should be an artifact that separates what is known, what is inferred, and what remains unverified. This point is the shared lesson behind documentation-driven systems such as DAInfer and ChatDetector, and domain-driven systems such as BugScope and RFCAudit. For example, an auditor may infer that a wrapper sanitizes path traversal inputs. It should also record the files that support this claim, the cases not covered by the evidence, and the tests or constraints needed to validate the claim. This is especially important for functional and domain-specific bugs, where expected behavior is often distributed across documents, reports, and historical patches.

The open problem is maintenance. When code evolves, the auditor should know which claims may have become stale. When a claim is uncertain, the auditor should know whether to ask for human review, run a test, invoke a solver, or postpone the claim. This turns missing specifications from an implicit weakness into an explicit object of auditing.

6.2 Cost-Aware Neurosymbolic Workflows

The verification bottleneck is partly a matter of cost. This is the lesson made concrete by the data-flow systems in Section 5.1.1 and by the industrial workflows in Section 5.3. A practical auditor cannot run

every expensive analysis, inspect every file, or ask a human to review every uncertain hypothesis. It must decide which evidence is worth collecting. This makes neurosymbolic auditing a budgeted decision process. The central question is not only whether the system can find bugs. It is whether the system can allocate verification effort like a good auditor under realistic constraints. The same budget also governs memory, because an auditor that reuses earlier results spends less effort rediscovering them. This section therefore addresses the verification and memory bottlenecks together.

6.2.1 Budgeted Orchestration

The systems surveyed in Chapter 5 already decompose auditing into stages. LLMDFA and RepoAudit use this structure for data-flow bugs. MDASH uses it for industrial-scale preparation, scanning, validation, deduplication, and proof construction. Other systems retrieve code slices, infer primitive facts, check path feasibility, synthesize rules, tune detectors, and post-process reports. Future work should make the orchestration policy explicit. Each tool invocation should be viewed as an action with a cost, an expected information gain, and a possible failure mode.

The data-flow systems in Section 5.1.1 are a useful concrete model. LLMDFA and RepoAudit separate source and sink identification, function-level summary construction, and path feasibility checking. This staged design is not just an implementation detail. It is a budget allocation strategy. Cheap semantic steps narrow the search space. More expensive validation is reserved for paths that remain plausible after earlier filters. Seen this way, future orchestration generalizes the LLMDFA and RepoAudit pipeline, rather than replacing it with a generic agent loop.

This perspective changes what evaluation should reward. A system should not simply maximize the number of warnings. It should maximize credible evidence under a fixed budget. An LLM may cheaply propose candidate sources and sinks. A static analyzer may then ground the candidates in a call graph or data-flow slice. A solver, reasoning model, or executable harness may be reserved for the few paths whose impact and plausibility justify the cost. The best policy depends on uncertainty, risk, and available resources.

Budgeted orchestration also clarifies the role of memory. A system that repeatedly re-derives the same facts is not scalable. It should cache validated summaries, tool outputs, boundary labels, and failed hypotheses. It should also invalidate them when relevant code, dependencies, or toolchain settings change. This is the practical route from one-off prompting to repository-scale auditing.

6.2.2 Evidence-Grounded Remediation

Auditing does not end at detection. The industrial systems in Section 5.3 make this point explicit by treating reproduction, isolation, and proof construction as part of the auditing workflow. A useful finding must support triage, repair, and regression prevention. For this reason, future neurosymbolic systems should close the loop from bug finding to remediation. This does not mean that a model should autonomously land patches. It means that the system should preserve enough evidence for a human or tool-assisted workflow to decide whether a patch is appropriate.

A patch is credible only when it addresses the cause of the bug under the relevant context. A superficial guard may silence a warning while leaving the real invariant broken. Conversely, a correct fix may require changes outside the function where the symptom appears. This is why remediation needs the same evidence structure as detection. The auditor should connect the reported bug to a feasible path, a violated specification, and the program context that makes the violation actionable.

The industrial systems in Section 5.3 provide early precedents. Aardvark (OpenAI, 2025), OpenAI Codex Security (OpenAI, 2026b), and MDASH (Microsoft, 2026) all move beyond narrative reports by using isolation, reproduction attempts, validation stages, or executable evidence. These systems show why remediation should be grounded in witnesses. A proposed fix is easier to trust when it is tied to a triggering input, a checked path, or a reproducible environment.

The output should also describe the validation envelope of a fix. It should state which paths, inputs, configurations, and dependencies were checked, and which were not. LLMs can help propose patches, tests, and explanations. Symbolic tools, tests, and review should decide whether the proposed repair satisfies the required property. In this workflow, the

model is a repair assistant rather than a repair authority.

6.2.3 Reusable Auditing Artifacts

The memory bottleneck suggests that neural reasoning should be compiled into reusable artifacts whenever possible. This is not only a future aspiration. It is already the organizing idea behind the detector-synthesis systems in Section 5.2.1. A final bug report is only one possible output. Other valuable outputs include analyzer rules, function summaries, inferred effects, test harnesses, path constraints, source and sink specifications, and structured provenance records. These artifacts make LLM-derived reasoning inspectable and repeatable. They also allow future audits to reuse earlier work rather than restart from raw prompts.

The strongest current evidence comes from bug detector synthesis in Section 5.2.1. KNighter, MoCQ, QLCoder, and IRIS already turn LLM-derived evidence into analyzer-ready rules, specifications, or detector candidates. Once these artifacts are generated, the static-analysis infrastructure can run them at codebase scale. This is the strongest evidence in the survey for the claim that LLM reasoning should be compiled into durable artifacts. It is also a stronger claim than prompt-based bug finding. The model does not merely express a judgment. It helps produce an artifact that another system can execute, test, and refine.

This direction follows naturally from Chapter 4. Primitive analyses produce intermediate semantic facts. In current systems, many such facts still remain inside a prompt transcript or a model response. Future systems should externalize them into representations that can be checked, versioned, diffed, and invalidated. The important point is not whether the community adopts a single audit intermediate representation. The important point is that auditing knowledge should have a lifecycle.

A lifecycle-aware artifact records what it claims, what evidence supports the claim, and when the claim must be rechecked. This is essential for CI integration. A generated CodeQL rule, a sanitizer summary, or a validated data-flow slice is useful only if the project can tell when it no longer applies. Without such maintenance, neurosymbolic auditing may create its own form of auditability debt.

6.3 Realistic Evaluation and Auditor Robustness

Evaluation determines which systems the community believes are making progress. This concern applies directly to the systems surveyed in Chapter 4 and Chapter 5, because they differ not only in detection accuracy but also in context requirements, validation strength, and cost. For neurosymbolic auditing, simple benchmark accuracy is too narrow. A future evaluation methodology should measure these properties directly. It should also account for the fact that an auditing agent may operate in an adversarial environment.

6.3.1 Benchmarks, Datasets, and Evaluation Realism

Recent benchmark studies make the need for caution concrete. PrimeVul shows that label quality, duplication, data leakage, chronological splitting, and realistic metrics can materially change conclusions about code language models for vulnerability detection (Ding *et al.*, 2025). VulnPatchPairs and the broader patch-pair evaluation of Risse and Böhme raise a complementary concern. Strong benchmark scores may not mean that a model can distinguish a vulnerable function from its patched version or reason about the context that makes a vulnerability exploitable (Risse and Böhme, 2024).

These results should not be read as a rejection of benchmarks. They show that benchmark claims need sharper interpretation. An evaluation should state what context is available to the auditor, how labels were obtained, how near-duplicates and temporal leakage were controlled, and what evidence counts as a correct answer. A warning label alone is often insufficient. For code auditing, a more realistic output is an actionable finding with a path, a specification link, a triggering condition, or a validated remediation obligation.

Future benchmarks should therefore stress repository-scale behavior. They should include cross-file context, configuration-dependent behavior, evolving dependencies, and realistic time budgets. They should report not only precision and recall, but also time to first actionable evidence, verification cost, stability under code changes, and the marginal value of additional budget. These measures better match the objective of

auditing. The objective is not to guess labels on isolated snippets. It is to produce credible security conclusions under constraints.

6.3.2 Path Validation and Repository-Scale Reasoning

Path validation remains one of the most important technical bottlenecks identified in Chapter 5. The split between LLMDFA and RepoAudit shows why this bottleneck has no single easy solution. LLMs can often explain a plausible path through a small code fragment. This is useful, but plausibility is not feasibility. A practical auditor must determine whether the path can occur under the relevant calling context, configuration, and environment. This is difficult even for symbolic execution, especially when paths cross library boundaries, involve heap aliasing, depend on strings, or require unavailable build environments.

The contrast between LLMDFA and RepoAudit makes this tension concrete. As discussed in Section 5.1.1, LLMDFA synthesizes path constraints and delegates satisfiability checking to Z3. RepoAudit instead uses large reasoning models to assess candidate path feasibility. These designs occupy different points in the same space. Solver-backed validation is more explicit when constraints are well formed. Reasoning-model validation is more flexible when constraints are difficult to encode. Future systems should treat this as a choice to be made under budget and risk, not as a fixed design preference.

Executable validation gives another anchor. The industrial practices surveyed in Section 5.3 show that Aardvark, OpenAI Codex Security, and MDASH try to reproduce findings or construct triggering inputs when possible. This does not replace static reasoning. It gives selected findings a stronger evidentiary status by tying a report to an executable witness or a failed reproduction attempt. A mature auditor should therefore combine approximate and strong validation deliberately. LLMs can reduce paths and propose summaries. Static analysis can provide over-approximated reachability and data flow. Solvers, tests, and concrete execution can validate selected obligations when the cost is justified.

Long-context models do not remove this problem. Larger contexts help auditors read more code, but they do not guarantee that the model retrieves the right evidence, composes it correctly, or reasons

consistently across distant files. This is why repository-scale auditing should not rely on a single long prompt. It needs retrieval, summaries, tool-grounded slices, and persistent memory that can be updated as the repository changes. Long context is a useful substrate, not a substitute for evidence management.

6.3.3 Security of the Auditor

Auditor security is a newer topic than the main systems surveyed in Chapter 4 and Chapter 5. We include it here because it becomes unavoidable once auditing agents read heterogeneous artifacts, retrieve text, execute tools, and propose patches. At that point, the auditor itself becomes part of the security boundary. This is not merely another accuracy problem. It is a threat-modeling problem for the auditing stack.

If an auditor reads code, documentation, issues, and tool outputs, the environment may contain adversarial comments, misleading identifiers, poisoned documentation, or outputs crafted to steer the agent away from the real issue. A robust design should separate instructions from evidence. Project text may describe a policy, but it should not override the auditor's policy. A comment may claim that a sanitizer is safe, but the auditor should require corroboration from code, tests, or prior validated artifacts. The codebase should be treated as evidence, not as an instruction source.

Tool execution also needs explicit safeguards. Compiling code, running harnesses, and executing generated tests should happen in controlled environments. Sandboxing is necessary, but it is not enough. The report should include provenance that allows a human to reconstruct the evidence path. Robustness benchmarks should therefore include adversarial artifacts, retrieval traps, misleading names, and provenance-tampering attempts. A steerable auditor is not only inaccurate. In security settings, it can become part of the attack surface.

6.4 Summary

This chapter has reframed future work around three bottlenecks that recur across the manuscript. Observation can be improved by making

security intent more explicit and by treating ecosystem artifacts as part of program semantics, following the specification-recovery path opened by DAInfer, ChatDetector, BugScope, and RFCAudit. Verification can be improved by cost-aware orchestration, path validation, and evidence-grounded remediation, following the staged pipelines in LLMDFA, RepoAudit, Aardvark, OpenAI Codex Security, and MDASH. Memory can be improved by compiling LLM-derived reasoning into reusable artifacts with provenance, uncertainty, and invalidation rules, following the artifact-oriented direction of KNighter, MoCQ, QLCoder, and IRIS.

The same bottlenecks also expose the most persistent technical challenges ahead. Specification scarcity still limits both classical analyzers and LLM-based auditors, even when systems such as DAInfer, ChatDetector, BugScope, and RFCAudit recover useful partial specifications. Path validation remains fragile when feasibility depends on aliases, strings, build environments, or cross-file context, as the LLMDFA and RepoAudit contrast makes clear. Long-context reasoning improves coverage, but it does not solve repository-scale evidence selection or composition. Reusable artifacts make auditing cumulative, as KNighter, MoCQ, QLCoder, and IRIS suggest, but only if those artifacts can survive semantic drift. Finally, agentic auditors must be robust in adversarial settings. These challenges are difficult because they are not isolated model weaknesses. They are system-level problems at the boundary between software semantics, evidence management, and security practice.

The long-term goal is therefore not an LLM that simply finds more bugs. It is an auditing stack that can transform messy software reality into durable, verifiable security knowledge. LLMs matter in that stack as semantic accelerators. They help produce the interfaces, summaries, specifications, witnesses, and checkers that trustworthy auditing ultimately requires.

References

Anthropic. (2026a). “Claude Code Overview”. https://docs.anthropic.com/en/docs/claude-code/overview.

Anthropic. (2026b). “Claude Code Review”. https://code.claude.com/docs/en/code-review.

Anthropic. (2026c). “Claude Code Security Reviewer”. https://github.com/anthropics/claude-code-security-review.

Anthropic. (2026d). “Project Glasswing: Securing Critical Software for the AI Era”. https://www.anthropic.com/glasswing.

Antoniadis, A., N. Filippakis, P. Krishnan, R. Ramesh, N. Allen, and Y. Smaragdakis. (2020). “Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room”. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by A. F. Donaldson and E. Torlak. ACM. 794–807. DOI: 10.1145/3385412.3386026. URL: https://doi.org/10.1145/3385412.3386026.

Anysphere. (2026). “Cursor Bugbot”. https://cursor.com/bugbot.

Arzt, S. and E. Bodden. (2016). "StubDroid: automatic inference of precise data-flow summaries for the android framework". In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. Ed. by L. K. Dillon, W. Visser, and L. A. Williams. ACM. 725–735. DOI: 10.1145/2884781.2884816. URL: https://doi.org/10.1145/2884781.2884816.

Arzt, S., S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel. (2014). "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by M. F. P. O'Boyle and K. Pingali. ACM. 259–269. DOI: 10.1145/2594291.2594299. URL: https://doi.org/10.1145/2594291.2594299.

Austin, T. M., S. E. Breach, and G. S. Sohi. (1994). "Efficient Detection of All Pointer and Array Access Errors". In: *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20–24, 1994*. Ed. by V. Sarkar, B. G. Ryder, and M. L. Soffa. ACM. 290–301. DOI: 10.1145/178243.178446. URL: https://doi.org/10.1145/178243.178446.

Avgustinov, P., O. de Moor, M. P. Jones, and M. Schäfer. (2016). "QL: Object-oriented Queries on Relational Data". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Ed. by S. Krishnamurthi and B. S. Lerner. Vol. 56. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 2:1–2:25. DOI: 10.4230/LIPIcs.ECOOP.2016.2. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2016.2.

Baldoni, R., E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. (2018). "A Survey of Symbolic Execution Techniques". *ACM Comput. Surv.* 51(3): 50:1–50:39.

Barke, S., M. B. James, and N. Polikarpova. (2023). "Grounded copilot: How programmers interact with code-generating models". *Proceedings of the ACM on Programming Languages*. 7(OOPSLA1): 85–111.

Bastani, O., R. Sharma, A. Aiken, and P. Liang. (2018). "Active learning of points-to specifications". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by J. S. Foster and D. Grossman. ACM. 678–692. DOI: 10.1145/3192366.3192383. URL: https://doi.org/10.1145/3192366.3192383.

Beck, M., J. Gehring, J. Kossen, and G. Synnaeve. (2026). "Towards a Neural Debugger for Python". *CoRR*. abs/2603.09951. DOI: 10.48550/ARXIV.2603.09951. arXiv: 2603.09951. URL: https://doi.org/10.48550/arXiv.2603.09951.

Bessey, A., K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Gros, A. Kamsky, S. McPeak, and D. R. Engler. (2010). "A few billion lines of code later: using static analysis to find bugs in the real world". *Commun. ACM*. 53(2): 66–75. DOI: 10.1145/1646353.1646374. URL: https://doi.org/10.1145/1646353.1646374.

Blanchet, B., P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. (2003). "A Static Analyzer for Large Safety-Critical Software". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. PLDI 2003*. New York, NY, USA: Association for Computing Machinery. 196–207. DOI: 10.1145/781131.781153. URL: https://doi.org/10.1145/781131.781153.

Bodden, E. (2012). "Inter-procedural data-flow analysis with IFDS/IDE and Soot". In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*. Ed. by E. Bodden, L. J. Hendren, P. Lam, and E. Sherman. ACM. 3–8. DOI: 10.1145/2259051.2259052. URL: https://doi.org/10.1145/2259051.2259052.

Bouras, D. S., Y. Dai, T. Wang, Y. Xiong, and S. Mechtaev. (2025). "HoarePrompt: Structural Reasoning About Program Correctness in Natural Language". *CoRR*. abs/2503.19599. DOI: 10.48550/ARXIV.2503.19599. arXiv: 2503.19599. URL: https://doi.org/10.48550/arXiv.2503.19599.

Boyer, R. S., B. Elspas, and K. N. Levitt. (1975). "SELECT - a formal system for testing and debugging programs by symbolic execution". In: *Reliable Software*. ACM. 234–245.

Bracevac, O., G. Wei, S. Jia, S. Abeysinghe, Y. Jiang, Y. Bao, and T. Rompf. (2023). "Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies". *Proc. ACM Program. Lang.* 7(OOPSLA2): 400–430.

Brooks, F. P. (1987). "No Silver Bullet - Essence and Accidents of Software Engineering". *Computer*. 20(4): 10–19. DOI: 10.1109/MC.1987.1663532. URL: https://doi.org/10.1109/MC.1987.1663532.

Brown, T. B., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. (2020). "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Bucur, S., V. Ureche, C. Zamfir, and G. Candea. (2011). "Parallel symbolic execution for automated real-world software testing". In: *EuroSys*. ACM. 183–198.

Cadar, C., D. Dunbar, and D. R. Engler. (2008). "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by R. Draves and R. van Renesse. USENIX Association. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf.

Calcagno, C., D. Distefano, P. W. O'Hearn, and H. Yang. (2011). "Compositional Shape Analysis by Means of Bi-Abduction". *J. ACM*. 58(6): 26:1–26:66. DOI: 10.1145/2049697.2049700. URL: https://doi.org/10.1145/2049697.2049700.

Carlini, N., N. Cheng, K. Lucas, M. Moore, M. Nasr, V. Prabhushankar, W. X. H. Angulu, E. Ben Asher, J. Bow, K. Bradwell, B. Buchanan, D. Forsythe, D. Freeman, A. Gaynor, X. Ge, L. Graham, K. Guru, H. Lakhani, M. McNiece, M. Mehrara, R. Nichol, A. Pirzada, S. Porter, A. Terzis, and K. Troy. (2026). "Assessing Claude Mythos Preview's Cybersecurity Capabilities". https://red.anthropic.com/2026/mythos-preview/.

Cha, S., M. Lee, S. Lee, and H. Oh. (2022). "SYMTUNER: Maximizing the Power of Symbolic Execution by Adaptively Tuning External Parameters". In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM. 2068–2079. DOI: 10.1145/3510003.3510185. URL: https://doi.org/10.1145/3510003.3510185.

Chakraborty, S., S. K. Lahiri, S. Fakhoury, A. Lal, M. Musuvathi, A. Rastogi, A. Senthilnathan, R. Sharma, and N. Swamy. (2023). "Ranking LLM-Generated Loop Invariants for Program Verification". In: *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*. Ed. by H. Bouamor, J. Pino, and K. Bali. Association for Computational Linguistics. 9164–9175. DOI: 10.18653/V1/2023.FINDINGS-EMNLP.614. URL: https://doi.org/10.18653/v1/2023.findings-emnlp.614.

Chang, J., J. Shi, Y. Lyu, X. Zhou, L. Wang, Z. Yang, B. Li, and D. Lo. (2025). "SLICEMATE: Accurate and Scalable Static Program Slicing via LLM-Powered Agents". *arXiv preprint arXiv:2507.18957*.

Chang, O., D. Liu, and J. Metzman. (2024). "Leveling Up Fuzzing: Finding More Vulnerabilities with AI". https://security.googleblog.com/2024/11/leveling-up-fuzzing-finding-more.html.

Chen, M., J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. (2021). "Evaluating Large Language Models Trained on Code". *CoRR*. abs/2107.03374. arXiv: 2107.03374. URL: https://arxiv.org/abs/2107.03374.

Chen, Q., C. Yu, R. Liu, C. Zhang, Y. Wang, K. Wang, T. Su, and L. Wang. (2024). "Evaluating the Effectiveness of Deep Learning Models for Foundational Program Analysis Tasks". *Proc. ACM Program. Lang.* 8(OOPSLA1): 500–528. DOI: 10.1145/3649829. URL: https://doi.org/10.1145/3649829.

Chen, Q., J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. L. Goues. (2022). "VarCLR: Variable Semantic Representation Pre-training via Contrastive Learning". In: *Proceedings of the 44th International Conference on Software Engineering*. Association for Computing Machinery. 2327–2339. DOI: 10.1145/3510003.3510162.

Chen, S., S. Wong, L. Chen, and Y. Tian. (2023). "Extending Context Window of Large Language Models via Positional Interpolation". *arXiv preprint arXiv:2306.15595*.

Cheng, B., K. Wang, L. Shi, H. Wang, P. Di, D. Li, X. Chen, and Y. Guo. (2025). "Boosting Pointer Analysis With LLM-Enhanced Allocation Function Detection". In: URL: https://api.semanticscholar.org/CorpusID:281659287.

Cheng, B., C. Zhang, K. Wang, L. Shi, Y. Liu, H. Wang, Y. Guo, D. Li, and X. Chen. (2024). "Semantic-enhanced indirect call analysis with large language models". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 430–442.

Cherem, S., L. Princehouse, and R. Rugina. (2007). "Practical Memory Leak Detection Using Guarded Value-Flow Analysis". In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, PLDI 2007, San Diego, California, USA, June 10–13, 2007*. Ed. by J. Ferrante and K. S. McKinley. ACM. 480–491. DOI: 10.1145/1250734.1250789. URL: https://doi.org/10.1145/1250734.1250789.

Chibotaru, V., B. Bichsel, V. Raychev, and M. T. Vechev. (2019). "Scalable taint specification inference with big code". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by K. S. McKinley and K. Fisher. ACM. 760–774. DOI: 10.1145/3314221.3314648. URL: https://doi.org/10.1145/3314221.3314648.

Choudhary, S. R., S. Mahajan, W. Bond, J. Wang, and A. Utture. (2025). "uReview: Scalable, Trustworthy GenAI for Code Review at Uber". https://www.uber.com/us/en/blog/ureview/.

Clarke, E. M. and E. A. Emerson. (1982). "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs*. Ed. by D. Kozen. Vol. 131. *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer. 52–71. DOI: 10.1007/BFb0025774. URL: https://doi.org/10.1007/BFb0025774.

Click, C. and M. Paleczny. (1995). "A Simple Graph-Based Intermediate Representation". In: *Intermediate Representations Workshop*. ACM. 35–49.

CodeQL. (2024). "CodeQL for Java". https://codeql.github.com/docs/codeql-language-guides/codeql-for-java/.

CodeRabbit. (2026). "CodeRabbit Documentation: AI Code Review". https://docs.coderabbit.ai/.

Le-Cong, T., H. J. Kang, T. G. Nguyen, S. A. Haryono, D. Lo, X.-B. D. Le, and Q. T. Huynh. (2022). "Autopruner: transformer-based call graph pruning". In: *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*. 520–532.

Cousot, P. and R. Cousot. (1977a). "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by R. M. Graham, M. A. Harrison, and R. Sethi. ACM. 238–252. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

Cousot, P. and R. Cousot. (1977b). "Automatic synthesis of optimal invariant assertions: Mathematical foundations". In: *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: Association for Computing Machinery. 1–12. DOI: 10.1145/800228.806926. URL: https://doi.org/10.1145/800228.806926.

Cousot, P. and R. Cousot. (1979). "Systematic Design of Program Analysis Frameworks". In: *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*. Ed. by A. V. Aho, S. N. Zilles, and B. K. Rosen. ACM Press. 269–282. DOI: 10.1145/567752.567778. URL: https://doi.org/10.1145/567752.567778.

Cousot, P. and R. Cousot. (2014). "Abstract interpretation: past, present and future". In: *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*. Ed. by T. A. Henzinger and D. Miller. ACM. 2:1–2:10. DOI: 10.1145/2603088.2603165. URL: https://doi.org/10.1145/2603088.2603165.

Cousot, P., R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. (2005). "The ASTRÉE analyzer". In: *European Symposium on Programming*. Springer. 21–30.

CVE. (2025). "Common Vulnerabilities and Exposures". https://www.cve.org.

CWE. (2025). "Common Weakness Enumeration". https://cwe.mitre.org.

Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. (1991). "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". *ACM Trans. Program. Lang. Syst.* 13(4): 451–490. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.

Darulova, E. and V. Kuncak. (2014). "Sound Compilation of Reals". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, January 20–21, 2014*. ACM. 235–248. DOI: 10.1145/2535838.2535874. URL: https://doi.org/10.1145/2535838.2535874.

Desai, N., D. Beghimer, and D. Cortegaca. (2025). "Beyond Compute: Shifting Vulnerability Detection Left with Amazon Inspector Code Security Capabilities". https://aws.amazon.com/blogs/security/shifting-vulnerability-detection-left-with-amazon-inspector-code-security-capabilities/.

Dietz, W., P. Li, J. Regehr, and V. S. Adve. (2012). "Understanding integer overflow in C/C++". In: *ICSE*. IEEE Computer Society. 760–770.

Ding, H., Z. Wang, and H. Chen. (2026). "FM-Agent: Scaling Formal Methods to Large Systems via LLM-Based Hoare-Style Reasoning". *CoRR*. abs/2604.11556. DOI: 10.48550/ARXIV.2604.11556. arXiv: 2604.11556. URL: https://doi.org/10.48550/arXiv.2604.11556.

Ding, Y., Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. A. Wagner, B. Ray, and Y. Chen. (2025). "Vulnerability Detection with Code Language Models: How Far are We?" In: *ICSE*. IEEE. 1729–1741.

Ding, Y., B. Steenhoek, K. Pei, G. Kaiser, W. Le, and B. Ray. (2024). "Traced: Execution-aware pre-training for source code". In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

Du, X., K. Yu, C. Wang, Y. Zou, W. Deng, Z. Ou, X. Peng, L. Zhang, and Y. Lou. (2025). "Minimizing False Positives in Static Bug Detection via LLM-Enhanced Path Feasibility Analysis". *CoRR*. abs/2506.10322. DOI: 10.48550/ARXIV.2506.10322. arXiv: 2506.10322. URL: https://doi.org/10.48550/arXiv.2506.10322.

Du, X., G. Zheng, K. Wang, Y. Zou, Y. Wang, W. Deng, J. Feng, M. Liu, B. Chen, X. Peng, T. Ma, and Y. Lou. (2026). "Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG". *ACM Transactions on Software Engineering and Methodology*. DOI: 10.1145/3797277. URL: https://doi.org/10.1145/3797277.

Eberhardt, J., S. Steffen, V. Raychev, and M. T. Vechev. (2019). "Unsupervised learning of API alias specifications". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by K. S. McKinley and K. Fisher. ACM. 745–759. DOI: 10.1145/3314221.3314640. URL: https://doi.org/10.1145/3314221.3314640.

Ernst, M. D., J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. (2007). "The Daikon system for dynamic detection of likely invariants". *Sci. Comput. Program.* 69: 35–45. URL: https://api.semanticscholar.org/CorpusID:17620776.

Facebook. (2022). "Infer Static Analyzer". https://fbinfer.com/.

Fakhoury, S., M. Kuppe, S. K. Lahiri, T. Ramananandro, and N. Swamy. (2025). "3DGen: AI-Assisted Generation of Provably Correct Binary Format Parsers". In: *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE. 2535–2547. DOI: 10.1109/ICSE55347.2025.00173. URL: https://doi.org/10.1109/ICSE55347.2025.00173.

Fan, G., R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang. (2019). "Smoke: scalable path-sensitive memory leak detection for millions of lines of code". In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. Ed. by J. M. Atlee, T. Bultan, and J. Whittle. IEEE / ACM. 72–82. DOI: 10.1109/ICSE.2019.00025. URL: https://doi.org/10.1109/ICSE.2019.00025.

Fang, C., N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, Asmita, R. Tsang, N. Nazari, H. Wang, and H. Homayoun. (2024a). "Large Language Models for Code Analysis: Do LLMs Really Do Their Job?" In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. Ed. by D. Balzarotti and W. Xu. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/fang.

Fang, R., R. Bindu, A. Gupta, Q. Zhan, and D. Kang. (2024b). "Teams of LLM Agents can Exploit Zero-Day Vulnerabilities". *arXiv preprint arXiv:2406.01637*.

Ferles, K., V. Wüstholz, M. Christakis, and I. Dillig. (2017). "Failure-Directed Program Trimming". In: *Proceedings of the 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ESEC/FSE 2017*. Paderborn, Germany: Association for Computing Machinery. 174–185. DOI: 10.1145/3106237.3106249. URL: https://doi.org/10.1145/3106237.3106249.

Ferrante, J., K. J. Ottenstein, and J. D. Warren. (1987). "The Program Dependence Graph and Its Use in Optimization". *ACM Trans. Program. Lang. Syst.* 9(3): 319–349.

Fink, S. J., E. Yahav, N. Dor, G. Ramalingam, and E. Geay. (2006). "Effective typestate verification in the presence of alias". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*. Ed. by L. L. Pollock and M. Pezzè. ACM. 133–144. DOI: 10.1145/1146238.1146254. URL: https://doi.org/10.1145/1146238.1146254.

Flanagan, C. and K. R. M. Leino. (2001). "Houdini, an Annotation Assistant for ESC/Java". In: *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity. FME '01*. Berlin, Heidelberg: Springer-Verlag. 500–517.

Floyd, R. W. (1993). "Assigning Meanings to Programs". In: *Program Verification. Studies in Cognitive Systems*. Springer Netherlands. 65–81.

Fu, Y., P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen. (2025). "Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study". *ACM Trans. Softw. Eng. Methodol.* 34(8): 218:1–218:34. DOI: 10.1145/3716848. URL: https://doi.org/10.1145/3716848.

Garg, P. and S. H. Sengamedu. (2022). "Synthesizing code quality rules from examples". *Proc. ACM Program. Lang.* 6(OOPSLA2): 1757–1787. DOI: 10.1145/3563350. URL: https://doi.org/10.1145/3563350.

Giannini, S. and H. Gill. (2025). "50% Faster Merge and 50% Fewer Bugs: How CodeRabbit Built Its AI Code Review Agent with Google Cloud Run". https://cloud.google.com/blog/products/ai-machine-learning/how-coderabbit-built-its-ai-code-review-agent-with-google-cloud-run.

GitHub. (2024). "Copilot Autofix for CodeQL Code Scanning Alerts Is Now Generally Available". https://github.blog/changelog/2024-08-14-copilot-autofix-for-codeql-code-scanning-alerts-is-now-generally-available/.

Godefroid, P., N. Klarlund, and K. Sen. (2005). "DART: Directed Automated Random Testing". In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, PLDI 2005, Chicago, IL, USA, June 12–15, 2005*. ACM. 213–223. DOI: 10.1145/1065010.1065036. URL: https://doi.org/10.1145/1065010.1065036.

Google. (2024). "From Naptime to Big Sleep: Using Large Language Models to Catch Vulnerabilities in Real-World Code". https://projectzero.google/2024/10/from-naptime-to-big-sleep.html.

Google. (2025). "Introducing CodeMender: An AI Agent for Code Security". https://deepmind.google/blog/introducing-codemender-an-ai-agent-for-code-security/.

Goubault, E. and S. Putot. (2011). "Static Analysis of Finite Precision Computations". In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23–25, 2011. Proceedings*. Ed. by R. Jhala and D. A. Schmidt. Vol. 6538. *Lecture Notes in Computer Science*. Springer. 232–247. DOI: 10.1007/978-3-642-18275-4_17. URL: https://doi.org/10.1007/978-3-642-18275-4_17.

Gowtham, M. and H. Pramod. (2021). "Semantic query-featured ensemble learning model for SQL-injection attack detection in IoT-ecosystems". *IEEE Transactions on Reliability*. 71(2): 1057–1074.

Gunasinghe, N. and N. Marcus. (2021). *Language server protocol and implementation*. Springer.

Guo, D., S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, *et al.* (2020). "GraphCodeBERT: Pre-training Code Representations with Data Flow". In: *International Conference on Learning Representations*.

Guo, J., C. Wang, D. Deluca, J. Liu, Z. Zhang, and X. Zhang. (2025a). "BugScope: Learn to Find Bugs Like Human". *CoRR*. abs/2507.15671. DOI: 10.48550/ARXIV.2507.15671. arXiv: 2507.15671. URL: https://doi.org/10.48550/arXiv.2507.15671.

Guo, J., C. Wang, X. Xu, Z. Su, and X. Zhang. (2025b). "RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing". *arXiv preprint arXiv:2501.18160*.

Guo, Y., J. Zhou, P. Yao, Q. Shi, and C. Zhang. (2022). "Precise Divide-By-Zero Detection with Affirmative Evidence". In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM. 1718–1729. DOI: 10.1145/3510003.3510066. URL: https://doi.org/10.1145/3510003.3510066.

Gutmann, P. (2016). "Fuzzing Code with AFL". *login Usenix Mag.* 41(2). URL: https://www.usenix.org/publications/login/summer2016/gutmann.

Hastings, R. and B. Joyce. (1992). "Purify: Fast Detection of Memory Leaks and Access Errors". In: *Proceedings of the Winter 1992 USENIX Conference*. 125–136. URL: https://web.stanford.edu/class/cs343/resources/purify.pdf.

Havelund, K. and T. Pressburger. (2000). "Model Checking JAVA Programs using JAVA PathFinder". *Int. J. Softw. Tools Technol. Transf.* 2(4): 366–381. DOI: 10.1007/S100090050043. URL: https://doi.org/10.1007/s100090050043.

He, J., G. Sivanrupan, P. Tsankov, and M. Vechev. (2021). "Learning to Explore Paths for Symbolic Execution". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. CCS '21*. Virtual Event, Republic of Korea: Association for Computing Machinery. 2526–2540. DOI: 10.1145/3460120.3484813. URL: https://doi.org/10.1145/3460120.3484813.

He, P., S. Wang, and T.-H. Chen. (2025). "SLICET5: Static Program Slicing using Language Models with Copy Mechanism and Constrained Decoding". *arXiv preprint arXiv:2509.17338*.

Hoare, C. A. R. (1969). "An Axiomatic Basis for Computer Programming". *Commun. ACM*. 12(10): 576–580.

Holzmann, G. J. (1997). "The Model Checker SPIN". *IEEE Transactions on Software Engineering*. 23(5): 279–295. DOI: 10.1109/32.588521. URL: https://doi.org/10.1109/32.588521.

Horwitz, S., T. Reps, and D. Binkley. (1988). "Interprocedural slicing using dependence graphs". *SIGPLAN Not.* 23(7): 35–46. ISSN: 0362-1340. DOI: 10.1145/960116.53994. URL: https://doi.org/10.1145/960116.53994.

Howden, W. E. (1977). "Symbolic Testing and the DISSECT Symbolic Evaluation System". *IEEE Trans. Software Eng.* 3(4): 266–278.

Huang, H., B. Shen, L. Zhong, and Y. Zhou. (2023a). "Protecting Data Integrity of Web Applications with Database Constraints Inferred from Application Code". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. ASPLOS 2023*. Vancouver, BC, Canada: Association for Computing Machinery. 632–645. DOI: 10.1145/3575693.3575699. URL: https://doi.org/10.1145/3575693.3575699.

Huang, H., B. Shen, L. Zhong, and Y. Zhou. (2023b). "Protecting Data Integrity of Web Applications with Database Constraints Inferred from Application Code". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. Ed. by T. M. Aamodt, N. D. E. Jerger, and M. M. Swift. ACM. 632–645. DOI: 10.1145/3575693.3575699. URL: https://doi.org/10.1145/3575693.3575699.

Jeannet, B. and A. Miné. (2009). "Apron: A Library of Numerical Abstract Domains for Static Analysis". In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Ed. by A. Bouajjani and O. Maler. Vol. 5643. *Lecture Notes in Computer Science*. Springer. 661–667. DOI: 10.1007/978-3-642-02658-4_52. URL: https://doi.org/10.1007/978-3-642-02658-4%5C_52.

Jeon, D., M. Jeon, and H. Oh. (2021). "A practical algorithm for learning disjunctive abstraction heuristics in static program analysis". *Inf. Softw. Technol.* 135: 106564. DOI: 10.1016/J.INFSOF.2021.106564. URL: https://doi.org/10.1016/j.infsof.2021.106564.

Jiang, J., F. Wang, J. Shen, S. Kim, and S. H. Kim. (2026). "A Survey on Large Language Models for Code Generation". *ACM Trans. Softw. Eng. Methodol.* 35(2): 58:1–58:72. DOI: 10.1145/3747588. URL: https://doi.org/10.1145/3747588.

Jiang, Z., M. Wen, G. Wen, and H. Jin. (2025). "Fact-Aligned and Template-Constrained Static Analyzer Rule Enhancement with LLMs". In: *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea*. IEEE.

Johnson, B., Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge. (2013). "Why don't software developers use static analysis tools to find bugs?" In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by D. Notkin, B. H. C. Cheng, and K. Pohl. IEEE Computer Society. 672–681. DOI: 10.1109/ICSE.2013.6606613. URL: https://doi.org/10.1109/ICSE.2013.6606613.

Kam, J. B. and J. D. Ullman. (1977). "Monotone Data Flow Analysis Frameworks". *Acta Informatica*. 7(3): 305–317. DOI: 10.1007/BF00290339. URL: https://doi.org/10.1007/BF00290339.

Kanvar, V. and U. P. Khedker. (2016). "Heap Abstractions for Static Analysis". *ACM Comput. Surv.* 49(2): 29:1–29:47. DOI: 10.1145/2931098. URL: https://doi.org/10.1145/2931098.

Kildall, G. A. (1973). "A Unified Approach to Global Program Optimization". In: *POPL*. ACM Press. 194–206.

King, J. C. (1976). "Symbolic Execution and Program Testing". *Commun. ACM*. 19(7): 385–394. DOI: 10.1145/360248.360252.

Kuznetsov, V., J. Kinder, S. Bucur, and G. Candea. (2012). "Efficient state merging in symbolic execution". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '12*. Beijing, China: Association for Computing Machinery. 193–204. DOI: 10.1145/2254064.2254088. URL: https://doi.org/10.1145/2254064.2254088.

Le Chi, C., C. T. V. Hoang, P. N. Huy, D. D. Le, T. N. Nguyen, and N. D. Bui. (2025). "Visualcoder: Guiding large language models in code execution with fine-grained multimodal chain-of-thought reasoning". In: *Findings of the Association for Computational Linguistics: NAACL 2025*. 6628–6645.

Leino, K. R. M. (2010). "Dafny: An Automatic Program Verifier for Functional Correctness". In: *LPAR (Dakar). Lecture Notes in Computer Science*. Springer. 348–370.

Li, C., J. Liang, A. Zeng, X. Chen, K. Hausman, D. Sadigh, S. Levine, L. Fei-Fei, F. Xia, and B. Ichter. (2024). "Chain of Code: Reasoning with a Language Model-Augmented Code Emulator". In: *International Conference on Machine Learning*. PMLR. 28259–28277.

Li, H., H. Zhang, K. Pei, and Z. Qian. (2025a). "Towards More Accurate Static Analysis for Taint-Style Bug Detection in Linux Kernel". In: *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*. IEEE. 380–392. DOI: 10.1109/ASE63991.2025.00039. URL: https://doi.org/10.1109/ASE63991.2025.00039.

Li, H., Z. Dong, S. Wang, H. Zhang, L. Shen, X. Peng, and D. She. (2025b). "Extracting Formal Specifications from Documents Using LLMs for Test Automation". In: *Proceedings of the 2025 IEEE/ACM 33rd International Conference on Program Comprehension, ICPC 2025*. IEEE Computer Society. 310–321. DOI: 10.1109/ICPC66645.2025.00039. URL: https://doi.org/10.1109/ICPC66645.2025.00039.

Li, J., Y. Su, and M. R. Lyu. (2026). "From Laboratory to Real-World Applications: Benchmarking Agentic Code Reasoning at the Repository Level". *CoRR*. abs/2601.03731. DOI: 10.48550/ARXIV.2601.03731. arXiv: 2601.03731. URL: https://doi.org/10.48550/arXiv.2601.03731.

Li, P., S. Yao, J. S. Korich, C. Luo, J. Yu, Y. Cao, and J. Yang. (2025c). "Automated Static Vulnerability Detection via a Holistic Neuro-symbolic Approach". *CoRR*. abs/2504.16057. DOI: 10.48550/ARXIV.2504.16057. arXiv: 2504.16057. URL: https://doi.org/10.48550/arXiv.2504.16057.

Li, Y., H. Dhulipala, A. Yadavally, X. Rong, S. Wang, and T. N. Nguyen. (2025d). "Blended Analysis for Predictive Execution". *Proc. ACM Softw. Eng.* 2(FSE). DOI: 10.1145/3729402. URL: https://doi.org/10.1145/3729402.

Li, Y., R. Meng, and G. J. Duck. (2025e). "Large language model powered symbolic execution". *Proceedings of the ACM on Programming Languages*. 9(OOPSLA2): 3148–3176.

Li, Y., T. Tan, A. Møller, and Y. Smaragdakis. (2020). "A Principled Approach to Selective Context Sensitivity for Pointer Analysis". *ACM Trans. Program. Lang. Syst.* 42(2): 10:1–10:40. DOI: 10.1145/3381915. URL: https://doi.org/10.1145/3381915.

Li, Z., S. Dutta, and M. Naik. (2025f). "IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities". In: *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. URL: https://openreview.net/forum?id=9LdJDU7E91.

Li, Z., A. Machiry, B. Chen, M. Naik, K. Wang, and L. Song. (2021). "ARBITRAR: User-Guided API Misuse Detection". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE. 1400–1415. DOI: 10.1109/SP40001.2021.00090. URL: https://doi.org/10.1109/SP40001.2021.00090.

Liu, A., B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, *et al.* (2024a). "Deepseek-v3 technical report". *arXiv preprint arXiv:2412.19437*.

Liu, B., J. Huang, and L. Rauchwerger. (2019). "Rethinking Incremental and Parallel Pointer Analysis". *ACM Trans. Program. Lang. Syst.* 41(1): 6:1–6:31. DOI: 10.1145/3293606. URL: https://doi.org/10.1145/3293606.

Liu, C., X. Wu, Y. Feng, Q. Cao, and J. Yan. (2024b). "Towards General Loop Invariant Generation: A Benchmark of Programs with Memory Manipulation". In: *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*. Ed. by A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang. URL: http://papers.nips.cc/paper%5C_files/paper/2024/hash/e93b673c55d6768cdd39ce90de8c4d4c-Abstract-Datasets%5C_and%5C_Benchmarks%5C_Track.html.

Liu, D., J. Metzman, and O. Chang. (2023). "AI-Powered Fuzzing: Breaking the Bug Hunting Barrier". https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html.

Liu, K., D. Chakraborty, A. Liggesmeyer, and A. Zeller. (2025). "Synthesizing Precise Protocol Specs from Natural Language for Effective Test Generation". *arXiv preprint arXiv:2511.17977*.

LLVM Project. (2026). "Clang Static Analyzer". https://clang-analyzer.llvm.org/.

Lööw, A., S. H. Park, D. Nantes-Sobrinho, S.-É. Ayoun, O. Sjöstedt, and P. Gardner. (2025). "Compositional Symbolic Execution for the Next 700 Memory Models". *Proc. ACM Program. Lang.* 9(OOPSLA2). DOI: 10.1145/3763151. URL: https://doi.org/10.1145/3763151.

Lukasczyk, S. and G. Fraser. (2022). "Pynguin: Automated Unit Test Generation for Python". In: *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE. 168–172. DOI: 10.1145/3510454.3516829. URL: https://doi.org/10.1145/3510454.3516829.

Luo, Z., H. Zhao, D. Wolff, C. Cadar, and A. Roychoudhury. (2026). "Agentic Concolic Execution". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 1–19.

Ma, W., S. Liu, M. Zhao, X. Xie, W. Wang, Q. Hu, J. Zhang, and Y. Liu. (2024). "Unveiling Code Pre-Trained Models: Investigating Syntax and Semantics Capacities". *ACM Trans. Softw. Eng. Methodol.* 33(7): 169:1–169:29. DOI: 10.1145/3664606. URL: https://doi.org/10.1145/3664606.

Meng, R., G. J. Duck, and A. Roychoudhury. (2024a). "Large language model assisted hybrid fuzzing". *arXiv preprint arXiv:2412.15931*.

Meng, R., M. Mirchev, M. Böhme, and A. Roychoudhury. (2024b). "Large Language Model guided Protocol Fuzzing". In: *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society. URL: https://www.ndss-symposium.org/ndss-paper/large-language-model-guided-protocol-fuzzing/.

Meyer, B. (1997). *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall.

Microsoft. (2026). "Defense at AI Speed: Microsoft's New Multi-Model Agentic Security System Tops Leading Industry Benchmark". https://www.microsoft.com/en-us/security/blog/2026/05/12/defense-at-ai-speed-microsofts-new-multi-model-agentic-security-system-tops-leading-industry-benchmark/.

Miné, A. (2002). "A Few Graph-Based Relational Numerical Abstract Domains". In: *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*. Ed. by M. V. Hermenegildo and G. Puebla. *Lecture Notes in Computer Science*. Springer. 117–132. DOI: 10.1007/3-540-45789-5_11. URL: https://doi.org/10.1007/3-540-45789-5%5C_11.

Miné, A. (2017). "Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation". *Foundations and Trends in Programming Languages*. 4(3-4): 120–372. ISSN: 2325-1107. DOI: 10.1561/2500000034. eprint: https://www.emerald.com/ftpgl/article-pdf/4/3-4/120/11044131/2500000034en.pdf. URL: https://doi.org/10.1561/2500000034.

MISRA. (2008). *MISRA C++:2008: Guidelines for the Use of the C++ Language in Critical Systems*. Nuneaton, UK: MIRA Limited. URL: https://misra.org.uk/misra-c-plus-plus/.

Monniaux, D. (2008). "The Pitfalls of Verifying Floating-Point Computations". *ACM Transactions on Programming Languages and Systems*. 30(3): 12:1–12:41. DOI: 10.1145/1353445.1353446. URL: https://doi.org/10.1145/1353445.1353446.

Moura, L. M. de and N. Bjørner. (2008). "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and J. Rehof. Vol. 4963. *Lecture Notes in Computer Science*. Springer. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3%5C_24.

Nagarakatte, S., J. Zhao, M. M. K. Martin, and S. Zdancewic. (2009). "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15–21, 2009*. Ed. by M. Hind and A. Diwan. ACM. 245–258. DOI: 10.1145/1542476.1542504. URL: https://doi.org/10.1145/1542476.1542504.

Nagarakatte, S., J. Zhao, M. M. K. Martin, and S. Zdancewic. (2010). "CETS: Compiler Enforced Temporal Safety for C". In: *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5–6, 2010*. Ed. by J. Vitek and D. Lea. ACM. 31–40. DOI: 10.1145/1806651.1806657. URL: https://doi.org/10.1145/1806651.1806657.

Naik, A., J. Mendelson, N. Sands, Y. Wang, M. Naik, and M. Raghothaman. (2021). "Sporq: An Interactive Environment for Exploring Code using Query-by-Example". In: *UIST '21: The 34th Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 10-14, 2021*. Ed. by J. Nichols, R. Kumar, and M. Nebeling. ACM. 84–99. DOI: 10.1145/3472749.3474737. URL: https://doi.org/10.1145/3472749.3474737.

Ni, A., M. Allamanis, A. Cohan, Y. Deng, K. Shi, C. Sutton, and P. Yin. (2024). "NExT: Teaching Large Language Models to Reason about Code Execution". In: *Forty-first International Conference on Machine Learning*.

OpenAI. (2023). "GPT-4 Technical Report". *CoRR*. abs/2303.08774. DOI: 10.48550/ARXIV.2303.08774. arXiv: 2303.08774.

OpenAI. (2025). “Introducing Aardvark: OpenAI’s Agentic Security Researcher”. https://openai.com/index/introducing-aardvark/.

OpenAI. (2026a). “Codex”. https://openai.com/codex.

OpenAI. (2026b). “Codex Security: Now in Research Preview”. https://openai.com/index/codex-security-now-in-research-preview/.

Ouyang, L., J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe. (2022). “Training language models to follow instructions with human feedback”. In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. URL: http://papers.nips.cc/paper%5C_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html.

Pandita, R., X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. (2012). “Inferring Method Specifications from Natural Language API Descriptions”. In: *Proceedings of the 34th International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland, June 2–9, 2012*. IEEE Computer Society. 815–825. DOI: 10.1109/ICSE.2012.6227137. URL: https://doi.org/10.1109/ICSE.2012.6227137.

Pei, K., D. Bieber, K. Shi, C. Sutton, and P. Yin. (2023). “Can Large Language Models Reason about Program Invariants?” In: *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*. Ed. by A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett. Vol. 202. *Proceedings of Machine Learning Research*. PMLR. 27496–27520. URL: https://proceedings.mlr.press/v202/pei23a.html.

Peng, B., J. Quesnelle, H. Fan, and E. Shippole. (2024). “YaRN: Efficient Context Window Extension of Large Language Models”. In: *The Twelfth International Conference on Learning Representations*.

Peng, S., E. Kalliamvakou, P. Cihon, and M. Demirer. (2023). “The Impact of AI on Developer Productivity: Evidence from GitHub Copilot”. *CoRR*. abs/2302.06590. DOI: 10.48550/arXiv.2302.06590. arXiv: 2302.06590 [cs.SE]. URL: https://arxiv.org/abs/2302.06590.

PMD Project. (2026). “PMD Source Code Analyzer”. https://pmd.github.io/pmd/index.html.

Poeplau, S. and A. Francillon. (2020). “Symbolic execution with SymCC: Don’t interpret, compile!” In: *USENIX Security Symposium*. USENIX Association. 181–198.

Pomonis, M., T. Petsios, K. Jee, M. Polychronakis, and A. D. Keromytis. (2014). “IntFlow: improving the accuracy of arithmetic error detection using information flow tracking”. In: *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*. Ed. by C. N. P. Jr., A. Hahn, K. R. B. Butler, and M. Sherr. ACM. 416–425. DOI: 10.1145/2664243.2664282. URL: https://doi.org/10.1145/2664243.2664282.

Porncharoenwase, S., L. Nelson, X. Wang, and E. Torlak. (2022). “A formal foundation for symbolic evaluation with merging”. *Proc. ACM Program. Lang.* 6(POPL): 1–28.

Queille, J.-P. and J. Sifakis. (1982). “Specification and Verification of Concurrent Systems in CESAR”. In: *International Symposium on Programming*. Vol. 137. *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer. 337–351. DOI: 10.1007/3-540-11494-7_22. URL: https://doi.org/10.1007/3-540-11494-7_22.

Reps, T. W., S. Horwitz, and S. Sagiv. (1995). “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In: *Conference Record of POPL’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. Ed. by R. K. Cytron and P. Lee. ACM Press. 49–61. DOI: 10.1145/199448.199462. URL: https://doi.org/10.1145/199448.199462.

Rice, H. G. (1953). “Classes of recursively enumerable sets and their decision problems”. *Transactions of the American Mathematical Society*. 74(2): 358–366.

Risse, N. and M. Böhme. (2024). “Uncovering the limits of machine learning for automatic vulnerability detection”. In: *33rd USENIX security symposium (USENIX Security 24)*. 4247–4264.

Rondon, P. M., M. Kawaguchi, and R. Jhala. (2008). “Liquid Types”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 159–169. DOI: 10.1145/1375581.1375602.

Rountev, A., M. Sharp, and G. Xu. (2008). “IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries”. In: *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*. Ed. by L. J. Hendren. Vol. 4959. *Lecture Notes in Computer Science*. Springer. 53–68. DOI: 10.1007/978-3-540-78791-4_4. URL: https://doi.org/10.1007/978-3-540-78791-4%5C_4.

Roychoudhury, A., C. Păsăreanu, M. Pradel, and B. Ray. (2026). “Agentic AI Software Engineers: Programming with Trust”. *Commun. ACM*. Apr. ISSN: 0001-0782. DOI: 10.1145/3769314. URL: https://doi.org/10.1145/3769314.

Sagiv, S., T. W. Reps, and S. Horwitz. (1996). “Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation”. *Theor. Comput. Sci.* 167(1&2): 131–170. DOI: 10.1016/0304-3975(96)00072-2. URL: https://doi.org/10.1016/0304-3975(96)00072-2.

Sajor, P. (2025). “Introducing Stack Overflow AI Assist—a tool for the modern developer”. https://stackoverflow.blog/2025/12/02/introducing-stack-overflow-ai-assist-a-tool-for-the-modern-developer.

Salis, V., T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos. (2021). “Pycg: Practical call graph generation in python”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 1646–1657.

Sen, K., D. Marinov, and G. Agha. (2005). “CUTE: A Concolic Unit Testing Engine for C”. In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE 2005, Lisbon, Portugal, September 5–9, 2005*. ACM. 263–272. DOI: 10.1145/1081706.1081750. URL: https://doi.org/10.1145/1081706.1081750.

Serebryany, K., D. Bruening, A. Potapenko, and D. Vyukov. (2012). "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13–15, 2012*. Ed. by G. Heiser and W. C. Hsieh. USENIX Association. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

Shahandashti, K. K., M. M. Mohajer, A. B. Belle, S. Wang, and H. H. Lassonde. (2025). "Program slicing in the era of large language models". In: *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE. 1224–1233.

Shi, Q., X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang. (2018). "Pinpoint: fast and precise sparse value flow analysis for million lines of code". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by J. S. Foster and D. Grossman. ACM. 693–706. DOI: 10.1145/3192366.3192418. URL: https://doi.org/10.1145/3192366.3192418.

Shi, Q., P. Yao, R. Wu, and C. Zhang. (2021). "Path-sensitive sparse analysis without path conditions". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by S. N. Freund and E. Yahav. ACM. 930–943. DOI: 10.1145/3453483.3454086. URL: https://doi.org/10.1145/3453483.3454086.

Si, X., H. Dai, M. Raghothaman, M. Naik, and L. Song. (2018). "Learning Loop Invariants for Program Verification". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. 7762–7773. URL: https://proceedings.neurips.cc/paper/2018/hash/65b1e92c585fd4c2159d5f33b5030ff2-Abstract.html.

Smaragdakis, Y. and G. Balatsouras. (2015). "Pointer Analysis". *Found. Trends Program. Lang.* 2(1): 1–69.

Sourcegraph, Inc. (2026). "Sourcegraph: Code Understanding, Oversight and Evolution". https://sourcegraph.com/.

Späth, J., K. Ali, and E. Bodden. (2017). "IDE^{al}: efficient and precise alias-aware dataflow analysis". *Proc. ACM Program. Lang.* 1(OOPSLA): 99:1–99:27. DOI: 10.1145/3133923. URL: https://doi.org/10.1145/3133923.

Späth, J., L. N. Q. Do, K. Ali, and E. Bodden. (2016). "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Ed. by S. Krishnamurthi and B. S. Lerner. Vol. 56. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 22:1–22:26. DOI: 10.4230/LIPIcs.ECOOP.2016.22. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2016.22.

SpotBugs Community. (2026). "SpotBugs: Find Bugs in Java Programs". https://spotbugs.github.io/.

Stein, B., B. B. Nielsen, B.-Y. E. Chang, and A. Møller. (2019). "Static analysis with demand-driven value refinement". *Proc. ACM Program. Lang.* 3(OOPSLA). DOI: 10.1145/3360566. URL: https://doi.org/10.1145/3360566.

Su, H., F. Li, L. Xu, W. Hu, Y. Sun, Q. Sun, H. Chao, and W. Huo. (2023). "Splendor: Static Detection of Stored XSS in Modern Web Applications". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*. Ed. by R. Just and G. Fraser. ACM. 1043–1054. DOI: 10.1145/3597926.3598116. URL: https://doi.org/10.1145/3597926.3598116.

Su, Z. and G. Wassermann. (2006). "The essence of command injection attacks in web applications". In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '06*. Charleston, South Carolina, USA: Association for Computing Machinery. 372–382. DOI: 10.1145/1111037.1111070. URL: https://doi.org/10.1145/1111037.1111070.

Sui, Y. and J. Xue. (2016). "SVF: interprocedural static value-flow analysis in LLVM". In: *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. Ed. by A. Zaks and M. V. Hermenegildo. ACM. 265–266. DOI: 10.1145/2892208.2892235. URL: https://doi.org/10.1145/2892208.2892235.

Sui, Y., D. Ye, and J. Xue. (2012). "Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis". In: *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15–20, 2012*. Ed. by M. P. E. Heimdahl and Z. Su. ACM. 254–264. DOI: 10.1145/2338965.2336784. URL: https://doi.org/10.1145/2338965.2336784.

Sun, C., Y. Li, Q. Zhang, T. Gu, and Z. Su. (2018). "Perses: Syntax-Guided Program Reduction". In: *Proceedings of the 40th International Conference on Software Engineering. ICSE '18*. Gothenburg, Sweden: Association for Computing Machinery. 361–371. DOI: 10.1145/3180155.3180236. URL: https://doi.org/10.1145/3180155.3180236.

Sun, Y., D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu. (2024). "GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis". In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM. 166:1–166:13. DOI: 10.1145/3597503.3639117. URL: https://doi.org/10.1145/3597503.3639117.

Szekeres, L., M. Payer, T. Wei, and D. Song. (2013). "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19–22, 2013*. IEEE Computer Society. 48–62. DOI: 10.1109/SP.2013.13. URL: https://doi.org/10.1109/SP.2013.13.

Tandon, S., C. Mishra, Z. Patterson, and P. Garg. (2024). "Code Security Scanning with Amazon Q Developer". https://aws.amazon.com/blogs/devops/code-security-scanning-with-amazon-q-developer/.

Thijm, T. A., R. Beckett, A. Gupta, and D. Walker. (2023). "Modular Control Plane Verification via Temporal Invariants". *Proc. ACM Program. Lang.* 7(PLDI): 50–75. DOI: 10.1145/3591222. URL: https://doi.org/10.1145/3591222.

Tip, F. (1994). "A survey of program slicing techniques". *J. Program. Lang.* 3. URL: https://api.semanticscholar.org/CorpusID:9882901.

Torlak, E. and R. Bodík. (2014). "A lightweight symbolic virtual machine for solver-aided host languages". In: *PLDI*. ACM. 530–541.

Tullsen, M., S. Pernsteiner, and M. Dodds. (2024). "Function Argument Nullability Using an LLM". Galois Inc., https://www.galois.com/articles/function-argument-nullability-using-an-llm.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. (2017). "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett. 5998–6008. URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Venkatesh, A. P. S., R. Sunil, S. Sabu, A. M. Mir, S. Reis, and E. Bodden. (2024). "An Empirical Study of Large Language Models for Type and Call Graph Analysis". *arXiv preprint arXiv:2410.00603*.

Wang, C., Y. Gao, W. Zhang, X. Liu, J. Guo, M. Zheng, Q. Shi, and X. Zhang. (2026). "NESA: Relational Neuro-Symbolic Static Program Analysis". *Proc. ACM Softw. Eng.* 3(FSE): FSE154. DOI: 10.1145/3808161. URL: https://doi.org/10.1145/3808161.

Wang, C., W. Wang, P. Yao, Q. Shi, J. Zhou, X. Xiao, and C. Zhang. (2022). "Anchor: Fast and Precise Value-Flow Analysis for Containers via Memory Orientation". *ACM Transactions on Software Engineering and Methodology*. DOI: 10.1145/3565800.

Wang, C., P. Yao, W. Tang, G. Fan, and C. Zhang. (2023a). "Synthesizing Conjunctive Queries for Code Search". In: *37th European Conference on Object-Oriented Programming, ECOOP 2023, Seattle, Washington, United States, July 17-21, 2023*. Ed. by K. Ali and G. Salvaneschi. Vol. 263. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 36:1–36:30. DOI: 10.4230/LIPICS.ECOOP.2023.36. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2023.36.

Wang, C., J. Zhang, R. Wu, and C. Zhang. (2024a). "DAInfer: Inferring API Aliasing Specifications from Library Documentation via Neurosymbolic Optimization". *Proc. ACM Softw. Eng.* 1(FSE): 2469–2492. DOI: 10.1145/3660816. URL: https://doi.org/10.1145/3660816.

Wang, C., W. Zhang, Z. Su, X. Xu, X. Xie, and X. Zhang. (2024b). "LLMDFA: Analyzing Dataflow in Code with Large Language Models". In: *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*. Ed. by A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang. URL: http://papers.nips.cc/paper%5C_files/paper/2024/hash/ed9dcde1eb9c597f68c1d375bbecf3fc-Abstract-Conference.html.

Wang, C., W. Zhang, Z. Su, X. Xu, and X. Zhang. (2024c). "Sanitizing Large Language Models in Bug Detection with Data-Flow". In: *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*. Ed. by Y. Al-Onaizan, M. Bansal, and Y. Chen. Association for Computational Linguistics. 3790–3805. DOI: 10.18653/V1/2024.FINDINGS-EMNLP.217. URL: https://doi.org/10.18653/v1/2024.findings-emnlp.217.

Wang, C., J. Liu, X. Peng, Y. Liu, and Y. Lou. (2023b). "Boosting Static Resource Leak Detection via LLM-based Resource-Oriented Intention Inference". *CoRR*. abs/2311.04448. DOI: 10.48550/ARXIV.2311.04448. arXiv: 2311.04448. URL: https://doi.org/10.48550/arXiv.2311.04448.

Wang, C., Z. Li, S. Dutta, and M. Naik. (2025a). "QLCoder: A Query Synthesizer For Static Analysis of Security Vulnerabilities". *CoRR*. abs/2511.08462. DOI: 10.48550/ARXIV.2511.08462. arXiv: 2511.08462. URL: https://doi.org/10.48550/arXiv.2511.08462.

Wang, M., K. Pei, and A. Solar-Lezama. (2025b). "ABSINT-AI: Language Models for Abstract Interpretation". In: *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*.

Wang, M., K. Pei, and A. Solar-Lezama. (2025c). "CALLME: Call Graph Augmentation with Large Language Models for Javascript". In: *Second Conference on Language Modeling*.

Wang, Y., W. Wang, S. R. Joty, and S. C. H. Hoi. (2021). "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Ed. by M. Moens, X. Huang, L. Specia, and S. W. Yih. Association for Computational Linguistics. 8696–8708. DOI: 10.18653/V1/2021.EMNLP-MAIN.685.

Wei, G., O. Bracevac, S. Tan, and T. Rompf. (2020). "Compiling symbolic execution with staging and algebraic effects". *Proc. ACM Program. Lang.* 4(OOPSLA): 164:1–164:33.

Wei, G., S. Jia, R. Gao, H. Deng, S. Tan, O. Bracevac, and T. Rompf. (2023). "Compiling Parallel Symbolic Execution with Continuations". In: *ICSE*. IEEE. 1316–1328.

Wei, G., Z. Zhang, and C. Urban. (2025). "Hallucination-Resilient LLM-Driven Sound and Tunable Static Analysis: A Case of Higher-Order Control-Flow Analysis". In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages. LMPL '25*. Singapore, Singapore: Association for Computing Machinery. 6–11. DOI: 10.1145/3759425.3763378. URL: https://doi.org/10.1145/3759425.3763378.

Wen, C., Y. Cai, B. Zhang, J. Su, Z. Xu, D. Liu, S. Qin, Z. Ming, and C. Tian. (2024). "Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We?" *ACM Trans. Knowl. Discov. Data.* 18(7): 168. DOI: 10.1145/3653718. URL: https://doi.org/10.1145/3653718.

Wu, Y., M. Wen, Z. Yu, X. Guo, and H. Jin. (2024). "Effective Vulnerable Function Identification based on CVE Description Empowered by Large Language Models". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*. Ed. by V. Filkov, B. Ray, and M. Zhou. ACM. 393–405. DOI: 10.1145/3691620.3695013. URL: https://doi.org/10.1145/3691620.3695013.

Xie, T., N. Tillmann, and P. Lakshman. (2016). "Advances in unit testing: theory and practice". In: *Proceedings of the 38th International Conference on Software Engineering Companion. ICSE '16*. Austin, Texas: Association for Computing Machinery. 904–905. DOI: 10.1145/2889160.2891056. URL: https://doi.org/10.1145/2889160.2891056.

Xie, Y., A. Chou, and D. R. Engler. (2003). "ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors". In: *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1–5, 2003*. Ed. by J. Paakki and P. Inverardi. ACM. 327–336. DOI: 10.1145/940071.940115. URL: https://doi.org/10.1145/940071.940115.

Xu, R., J. Cao, Y. Lu, M. Wen, H. Lin, X. Han, B. He, S. Cheung, and L. Sun. (2025a). "CRUXEVAL-X: A Benchmark for Multilingual Code Reasoning, Understanding and Execution". In: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*. Ed. by W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar. Association for Computational Linguistics. 23762–23779. URL: https://aclanthology.org/2025.acl-long.1158/.

Xu, X., G. Shen, Z. Su, S. Cheng, H. Guo, L. Yan, X. Chen, J. Jiang, X. Jin, C. Wang, Z. Zhang, and X. Zhang. (2025b). "ASTRA: Autonomous Spatial-Temporal Red-teaming for AI Software Assistants". *CoRR*. abs/2508.03936. DOI: 10.48550/ARXIV.2508.03936. arXiv: 2508.03936. URL: https://doi.org/10.48550/arXiv.2508.03936.

Yadavally, A., Y. Li, S. Wang, and T. N. Nguyen. (2024). "A Learning-Based Approach to Static Program Slicing". *Proc. ACM Program. Lang.* 8(OOPSLA1). DOI: 10.1145/3649814. URL: https://doi.org/10.1145/3649814.

Yadavally, A., T. N. Nguyen, W. Wang, and S. Wang. (2023). "(Partial) Program Dependence Learning". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2501–2513. DOI: 10.1109/ICSE48619.2023.00209.

Yang, C., Z. Zhao, Z. Xie, H. Li, and L. Zhang. (2025a). "KNighter: Transforming Static Analysis with LLM-Synthesized Checkers". In: *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*. Ed. by Y. Won, Y. Kwon, D. Yuan, and R. Isaacs. ACM. 655–669. DOI: 10.1145/3731569.3764827. URL: https://doi.org/10.1145/3731569.3764827.

Yang, J., P. Subramaniam, S. Lu, C. Yan, and A. Cheung. (2018). "How not to structure your database-backed web applications: a study of performance bugs in the wild". In: *Proceedings of the 40th International Conference on Software Engineering. ICSE '18*. Gothenburg, Sweden: Association for Computing Machinery. 800–810. DOI: 10.1145/3180155.3180194. URL: https://doi.org/10.1145/3180155.3180194.

Yang, Y., J. Liu, K. Chen, and M. Lin. (2025b). "The Midas Touch: Triggering the Capability of LLMs for RM-API Misuse Detection". In: *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24–28, 2025*. The Internet Society. URL: https://www.ndss-symposium.org/ndss-paper/the-midas-touch-triggering-the-capability-of-llms-for-rm-api-misuse-detection/.

Yao, P., J. Zhou, X. Xiao, Q. Shi, R. Wu, and C. Zhang. (2024). "Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis". *Proc. ACM Program. Lang.* 8(PLDI): 567–592. DOI: 10.1145/3656400. URL: https://doi.org/10.1145/3656400.

Ye, S., Y. Sui, and J. Xue. (2014). "Region-Based Selective Flow-Sensitive Pointer Analysis". In: *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*. Ed. by M. Müller-Olm and H. Seidl. *Lecture Notes in Computer Science*. Springer. 319–336. DOI: 10.1007/978-3-319-10936-7_20. URL: https://doi.org/10.1007/978-3-319-10936-7%5C_20.

Ye, X., Q. Chen, I. Dillig, and G. Durrett. (2023). "Satisfiability-Aided Language Models Using Declarative Prompting". *CoRR*. abs/2305.09656. DOI: 10.48550/arXiv.2305.09656. arXiv: 2305.09656. URL: https://doi.org/10.48550/arXiv.2305.09656.

Yen, H., T. Gao, M. Hou, K. Ding, D. Fleischer, P. Izsak, M. Wasserblat, and D. Chen. (2024). "Helmet: How to evaluate long-context language models effectively and thoroughly". *arXiv preprint arXiv:2410.02694*.

Zettlemoyer, L. S. and M. Collins. (2005). "Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars". In: *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*. AUAI Press. 658–666. URL: https://dl.acm.org/doi/10.5555/3020336.3020416.

Zhai, J., J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin. (2016). "Automatic model generation from documentation for Java API functions". In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016*. Ed. by L. K. Dillon, W. Visser, and L. A. Williams. ACM. 380–391. DOI: 10.1145/2884781.2884881. URL: https://doi.org/10.1145/2884781.2884881.

Zhang, B., W. Chen, P. Yao, C. Wang, W. Tang, and C. Zhang. (2024). "SIRO: Empowering Version Compatibility in Intermediate Representations via Program Synthesis". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*. Ed. by R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir. ACM. 882–899. DOI: 10.1145/3620666.3651366.

Zhang, Y., L. Ren, L. Chen, Y. Xiong, S. Cheung, and T. Xie. (2020). "Detecting numerical bugs in neural network architectures". In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by P. Devanbu, M. B. Cohen, and T. Zimmermann. ACM. 826–837. DOI: 10.1145/3368089.3409720. URL: https://doi.org/10.1145/3368089.3409720.

Zheng, M., C. Wang, X. Liu, J. Guo, S. Feng, and X. Zhang. (2025a). "RFCAudit: AI Agent for Auditing Protocol Implementations Against RFC Specifications". In: *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea*. IEEE.

Zheng, M., D. Xie, Q. Shi, C. Wang, and X. Zhang. (2025b). "Validating Network Protocol Parsers with Traceable RFC Document Interpretation". *Proc. ACM Softw. Eng.* 2(ISSTA): 1772–1794. DOI: 10.1145/3728955. URL: https://doi.org/10.1145/3728955.

Zheng, M., D. Xie, and X. Zhang. (2025c). "Large Language Models for Validating Network Protocol Parsers". In: *2025 IEEE Security and Privacy, SP 2025 - Workshops, San Francisco, CA, USA, May 15, 2025*. Ed. by M. Blanton, W. Enck, and C. Nita-Rotaru. IEEE. 56–64. DOI: 10.1109/SPW67851.2025.00009. URL: https://doi.org/10.1109/SPW67851.2025.00009.

Zhong, H., L. Zhang, T. Xie, and H. Mei. (2009). "Inferring Resource Specifications from Natural Language API Documentation". In: *24th IEEE/ACM International Conference on Automated Software Engineering, ASE 2009, Auckland, New Zealand, November 16–20, 2009*. IEEE Computer Society. 307–318. DOI: 10.1109/ASE.2009.94. URL: https://doi.org/10.1109/ASE.2009.94.

Zhu, X., S. Wen, S. Camtepe, and Y. Xiang. (2022). "Fuzzing: A Survey for Roadmap". *ACM Comput. Surv.* 54(11s): 230:1–230:36. DOI: 10.1145/3512345. URL: https://doi.org/10.1145/3512345.

Ziegler, A., E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian. (2022). "Productivity Assessment of Neural Code Completion". In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming. MAPS 2022*. New York, NY, USA: Association for Computing Machinery. 21–29. DOI: 10.1145/3520312.3534864. URL: https://doi.org/10.1145/3520312.3534864.