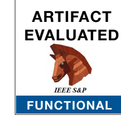


SpecAuditor: Generating Audit Specifications for LLM-Driven Bug Detection

Miaoqian Lin
The University of Hong Kong

Hao Chen
The University of Hong Kong

***Abstract*—Static analysis has been widely used for bug detection in real-world systems. However, traditional static analysis depends on manually written rules that match specific code forms. They often fail to handle semantically equivalent code variants and diverse bug patterns. Recent advances in large language models (LLMs) provide new opportunities for bug detection due to their capability to understand code semantics. However, directly applying LLMs to bug auditing is ineffective and expensive. Without clear guidance, LLMs fall back on memorized common patterns. They struggle with system-specific semantics and rare bug patterns that require domain knowledge. Therefore, it is necessary to provide high-quality audit specifications that clearly describe where to audit and under what conditions a bug occurs, for guiding LLMs to perform effective bug detection.**

In this paper, we propose SpecAuditor, an end-to-end framework that automatically generates and applies audit specifications for LLM-driven bug detection. SpecAuditor leverages historical bug patches to obtain specifications and then uses them to detect new bugs. Instead of directly extracting syntactic patterns from patches, SpecAuditor generalizes specifications at the semantic level to obtain new, broader specifications, thereby significantly extending the coverage of bug detection. In particular, SpecAuditor proceeds in three stages: (1) It extracts seed specifications from bug patches and validates them via differential checking. (2) It generalizes each seed specification to capture its underlying behavior and uses documentation-based semantic retrieval to identify other code entities performing similar behaviors, generating new specifications for them. (3) It performs LLM-driven bug detection by combining AST-based code search with LLM-based semantic auditing, followed by context-aware report pruning to reduce false positives. Our evaluation on the Linux kernel shows that SpecAuditor generates diverse specifications that go beyond syntactic patterns seen in patches. Using these specifications, SpecAuditor detects 71 long-latent new bugs, with an average lifetime of more than 7 years, including memory leaks, use-after-free, and out-of-bounds bugs. To date, 52 bugs have been confirmed by maintainers and 37 have been fixed. Moreover, 21 of the bug patches have been backported to the Linux stable trees for long-term release stability.

1. Introduction

Software bugs pose a serious threat to system security, making efficient bug detection crucial. Traditional static analysis tools (e.g., CodeQL [1], Clang Static Analyzer [2]) rely on manually crafted rules that are tied to specific code patterns. While these tools are effective for well-defined patterns, they operate mainly at the syntactic level and struggle to generalize across semantically equivalent yet structurally different code [3], [4], [5]. Moreover, developing such rules requires significant domain expertise, which limits scalability when facing new bug patterns [6], [7], [8].

Recent advances in large language models (LLMs) open new opportunities for bug detection due to their semantic reasoning capabilities [9], [10], [11], [12], [13], [14]. With strong capabilities in code understanding and reasoning, LLMs can overcome the semantic limitations of traditional static analysis [10]. However, directly applying LLMs for bug detection remains ineffective. Without explicit guidance, LLMs tend to rely on memorized knowledge of common patterns and struggle to capture bugs related to program-specific semantics, leading to shallow analysis [15]. Direct large-scale auditing with LLMs is also expensive.

These limitations highlight the necessity of explicit audit specifications that clearly define both where to check and what to check for. Such specifications help LLMs reason on critical code segments rather than surface-level code patterns. This enables LLMs to fully leverage their semantic understanding instead of relying solely on built-in knowledge. However, obtaining these specifications remains challenging: real-world documentation rarely provides specifications in sufficient detail, and manually developing such specifications at scale is infeasible [16], [17], [18], [19], [20], [21]. Consequently, the lack of high-quality specifications limits the coverage of LLM-driven bug detection.

To bridge this gap, we aim to automatically generate audit specifications that guide LLMs for effective bug detection. An audit specification defines how code should behave to remain correct. It consists of two elements: the *entity*, indicating the code locations to audit (e.g., security-sensitive function calls), and the *constraint*, which describes the condition that must hold when the entity is used. Violations of these specifications often indicate bugs. Real bug patches provide a natural source for obtaining such specifications [22], [23]. Each patch captures both the buggy behavior and the fixing: the pre-patch code exposes the incorrect behavior, and the post-patch code shows how the bug is fixed. By comparing the two, we can infer the underlying specification. Prior studies extract specifications from patches [22], [23], [24], but they capture only syntax-level patterns and can detect only bugs that share the same

syntactic details. Yet, many code entities never appear in patches, leaving their related bugs undiscovered.

Insights. Therefore, we aim to generate diverse audit specifications that go beyond the syntax-level patterns shown in patches and enable scalable LLM-driven bug detection. Our key insight is that the need for safety constraints arises from the underlying behaviors whose improper handling may cause bugs, not from the syntactic forms of the entities performing them. Thus, entities with similar vulnerable behaviors should share similar constraints, regardless of their syntactic differences. Building on this insight, we treat patch-extracted specifications as seed specifications and generalize them to other entities exhibiting similar behaviors. This allows us to synthesize new, semantically grounded specifications that go beyond the original patches and significantly broaden the coverage of LLM-driven bug detection. To realize this idea, we must address three key questions: ❶ *How to reliably extract specifications from patches?* LLMs may hallucinate or misinterpret patch semantics, so we need a way to ensure extracted specifications correctly reflect the behavior enforced by the patch. ❷ *How to generate specifications for code entities beyond those in patches?* In large codebases, the same behavior can be implemented in many different ways, so we must semantically identify entities performing similar behaviors rather than relying on surface code similarity. ❸ *How to efficiently apply audit specifications for new bug detection at scale?* Directly invoking LLMs across entire codebases is costly and error-prone, so we need to combine LLM reasoning with scalable analysis to control both cost and false positives.

Solutions. To address the above problems, we design SpecAuditor, an end-to-end framework that automatically generates audit specifications to guide LLM-driven bug detection. SpecAuditor consists of three stages: ❶ *Seed Specification Extraction.* SpecAuditor first extracts candidate specifications from historical patches and validates them by checking that they are violated in the pre-patch version and satisfied after the fix. After validation, SpecAuditor generalizes each specification by abstracting its underlying behavior and constraint while removing syntax-specific details. ❷ *Specification Generation.* From the generalized seed specifications, SpecAuditor identifies new code entities that perform similar operations. It first parses project documentation to build an entity–description mapping and then uses semantic search to identify related entities. For each retrieved entity, SpecAuditor examines its implementation and usage context to determine its actual behavior and whether the corresponding constraint applies. If so, SpecAuditor instantiates a concrete specification for that entity. ❸ *LLM-driven Bug Detection.* In this stage, SpecAuditor applies the obtained specifications to audit the codebase and detect potential violations that may indicate bugs. It adopts a hybrid detection strategy that combines Abstract Syntax Tree (AST)-based code search with the LLM's reasoning. The AST search efficiently locates code regions related to each specification's entity, while the LLM determines whether the corresponding constraint is violated. To further improve precision, SpecAuditor includes a report pruning module, where the LLM re-evaluates violation reports with adaptively supplied context, reducing false positives.

Results. We implement our approach in SpecAuditor, and evaluate it on the Linux kernel, a widely used and large-scale system with diverse and complex mechanisms. In our experiments on 100 real bug-fix patches, SpecAuditor successfully extracts reliable seed specifications and generalizes them into hundreds of new specifications that go beyond the syntactic forms of the original patches. Using these specifications, SpecAuditor discovers 71 previously unknown, long-latent bugs (average lifetime over seven years), spanning diverse types such as memory leaks, use-after-free, and out-of-bounds accesses. These bugs were missed by both the state-of-the-art LLM-assisted analyzer [6] and the LLM-driven bug detection method [10], demonstrating that SpecAuditor can uncover bugs beyond the reach of existing methods. To date, 52 of these bugs have been confirmed and 37 fixed by maintainers. In addition, 21 patches have been backported to the Linux stable trees, showing the real-world impact of SpecAuditor. These results further highlight the future potential of specification-guided LLM bug auditing. The artifact is available at https://github.com/Yuuoniy/SpecAuditor.

Contributions. Our contributions are as follows:

- **New Idea.** We propose a semantics-based approach for generating audit specifications to guide LLM-driven bug detection. By abstracting the underlying behaviors and constraints from real bug patches, our approach moves beyond syntax-level patterns shown in patches and enables semantic transfer across diverse code entities.
- **New Framework.** Based on the idea, we develop SpecAuditor, an end-to-end framework that integrates LLM reasoning, documentation-based semantic retrieval, and LLM-driven bug auditing to extract, validate, generalize, and apply specifications for LLM-driven bug detection.
- **New Discoveries.** Applied to the Linux kernel, SpecAuditor generates high-quality specifications and detects 71 long-latent new bugs covering diverse bug patterns, of which 52 are confirmed, 37 fixed, 21 patches backported, showing its real-world impact.

2. Background and Motivation

In this section, we introduce related work and analyze their limitations as background to motivate our work.

Traditional Static Bug Detection. Static bug detection analyzes source code without execution and has shown effectiveness in finding many bugs. Traditional static tools such as CodeQL [1] and Clang Static Analyzer [2] require domain knowledge to model different bug patterns into predefined queries or checkers that match specific code patterns to detect bugs. These methods operate on syntactic patterns rather than semantics. However, in practice, the same programming intention can be expressed in multiple ways, leading to diverse code variants. For example, a structure can be initialized using `memset()` or by directly initializing it as `var={}`. To handle such variants, developers must enumerate all possible cases and hard-code them into queries or checkers. This process is labor-intensive and

still struggles to cover semantically similar but syntactically diverse patterns, often resulting in inaccurate bug detection. Therefore, incorporating semantic reasoning capabilities is essential for more effective bug detection.

LLM for Bug Detection. Recent advances in large language models (LLMs) provide new opportunities for bug detection [14]. Their strong semantic understanding and generalization capabilities can complement traditional static analysis and improve detection effectiveness [13]. However, the effectiveness of LLM-based detection heavily depends on how the model is guided. When prompted only with general auditing instructions, LLMs tend to rely on internal, memorized knowledge, allowing them to detect only common and well-known bug patterns. They struggle with project-specific semantics, and less frequent bug patterns that require domain knowledge. Moreover, directly feeding large codebases into LLMs is prohibitively expensive and inefficient, making naive end-to-end auditing impractical. Therefore, to fully leverage LLMs' semantic reasoning capabilities, it is crucial to provide effective, targeted guidance, so the models can perform effective and scalable bug auditing.

Specifications Extraction for Bug Detection. In bug detection, specifications define the expected correct behavior of code, and any violation of these specifications often indicates a potential bug [16], [25], [26], [27]. Therefore, obtaining rich and accurate specifications is critical for effective bug detection. However, automatically obtaining such specifications remains challenging. Prior studies have explored specification extraction using different software artifacts, including source code [17], [26], documentation [27], [28], or bug patches [22], [23]. Most existing studies focus on API misuse detection and extract API specifications for checking. However, their underlying assumptions do not always hold, leaving many API specifications uncovered. Specifically, documentation-based methods [27], [28] can only capture explicitly documented specifications, which are incomplete. Source code–based methods [17], [26] often rely on statistically frequent API usage patterns. They lack semantic understanding and miss infrequent but critical specifications. Patch-based methods [22], [23] extract specifications from bug patches but rely on syntactic patterns or predefined templates, limiting generality. The specifications only apply to similar code or the same API. These limitations motivate a semantics-based approach to specification extraction that generalizes beyond specific syntax patterns in software artifacts. Achieving this goal requires leveraging the semantic reasoning capabilities of LLMs while mitigating their inherent hallucinations.

3. Audit Specification

In this section, we introduce the representation of audit specifications and present our key idea.

Audit Specification. To enable effective bug detection, we first define a specification representation that is expressive enough to capture diverse bug-related knowledge while remaining flexible across different code semantics. To this end, we adopt a semantics-based specification representation expressed in natural language. Specifically, we decompose

```
01 static int ovl_copy_up_tmpfile(struct ovl_copy_up_ctx *c){
02    struct file *tmpfile;
03    tmpfile = ovl_do_tmpfile(ofs, c->workdir, c->stat.mode);
04    ...
05    err = ovl_copy_up_file(ofs, c->dentry, tmpfile, ...);
06    if (err)
07 -      return err;
08 +      goto out_fput;
09
10 out_fput:
11   fput(tmpfile);
12   return err;
13 }
```

(a) Patch of CVE-2023-5300

Entity	Constraint
Calls to the function `ovl_do_tmpfile`.	*Within the lifetime of its returned struct file pointer, all exit paths should invoke* `fput` *to release it.*

(b) Summarized specification

Figure 1: Patch of CVE-2023-5300 and its specification.

LLM-driven bug detection into two key questions: (1) Where in the code should we check for potential bugs? (2) What constraints must hold at these locations to ensure security? Accordingly, each specification consists of two components:

- *Entity*: code element that performs security-sensitive or error-prone operations and may require further handling (e.g., call to sensitive functions, use sensitive structures).
- *Constraint*: the constraint that must hold when the entity appears, such as performing required checks, ensuring correct initialization, or handling errors properly.

Each specification is represented as an *Entity–Constraint* pair. Thus, when a piece of code matches the entity but violates its associated constraint, we consider it potentially buggy. The specifications are expressed in natural language, which provides strong generalization and is naturally suited to LLM reasoning. For example, Figure 1 shows the patch for CVE-2023-5300 and the specification derived from it. Specifically, the function `ovl_do_tmpfile` allocates a file resource, but one error path returns without calling `fput` to release it, resulting in a resource leak. The patch redirects this return path to the cleanup block, ensuring that `fput` is invoked and the file resource is properly released. From this patch, we obtain the specification: the entity is the function `ovl_do_tmpfile`, and the constraint is that any code path that returns the struct file pointer must invoke `fput` before exiting. Using this specification, we can guide LLMs to examine other call sites of `ovl_do_tmpfile` to check whether the constraint holds. If a path returns the file without releasing it, there may be a resource leak.

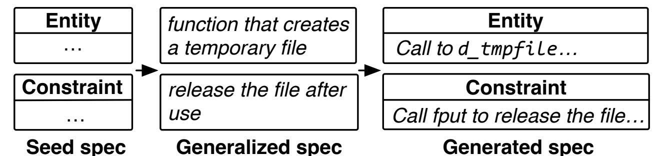


Figure 2: Generalizing the seed specification and generating the specification for `d_tmpfile`.

Key Idea. By analyzing patches, we can extract specifications to detect similar bugs. However, patch-extracted specifications are limited to the specific entities that appear

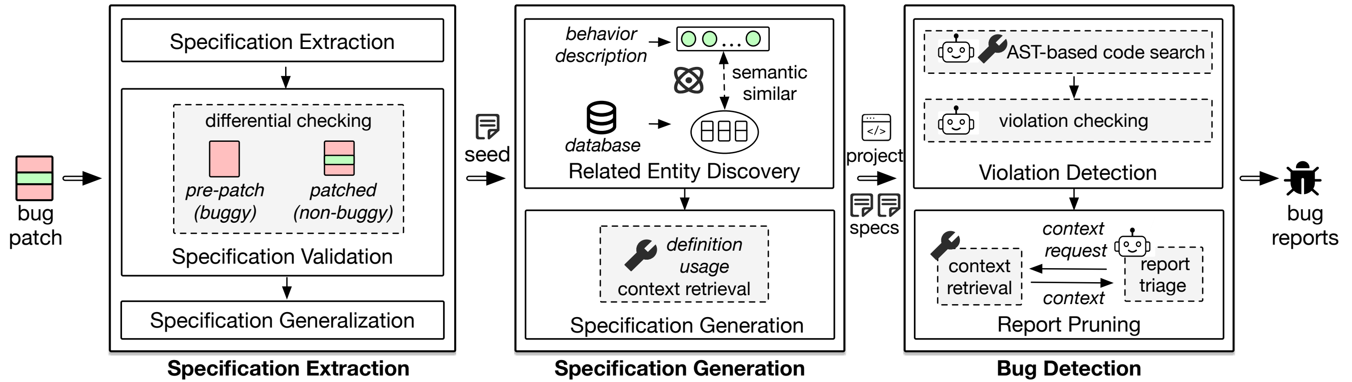


Figure 3: Overview of SpecAuditor.

in patches, and real-world patches are limited in number and diversity. As a result, the coverage of bug detection is restricted. To address this, we make a key observation: a constraint is required because of the underlying operation that may lead to a bug, not because of the entity's syntactic form. We refer to this operation as the *vulnerable behavior*, whose incorrect handling may lead to a bug. Thus, entities that exhibit similar vulnerable behaviors may share similar constraints, even if they look syntactically different. Based on this insight, we treat patch-extracted specifications as seeds that capture vulnerable behaviors and their required constraints, and we generalize these seeds to other entities that exhibit the same behaviors. This expands the coverage of bug detection beyond the entities appearing in patches. For example, in Figure 2, the patch-extracted seed specification states that any temporary file created by `ovl_do_tmpfile()` must be released using `fput`. This specification, if used directly, can only detect bugs related to `ovl_do_tmpfile`. In this example, the vulnerable behavior here is creating a temporary file that requires explicit release, regardless of the specific function name. Using this behavior, we identify another function, `d_tmpfile`, that performs the same operation and has the same constraint that its temporary file must be properly released. This helps to detect resource leaks involving `d_tmpfile`, even though it does not appear in the original patch.

4. Overview

Based on the above insight, we design SpecAuditor, an end-to-end framework that automatically extracts, generalizes, and applies audit specifications for LLM-driven bug detection. As shown in Figure 3, SpecAuditor consists of three stages: (1) *Seed Specification Extraction*: extracts specifications from patches as seeds. (2) *Specification Generation*: generates specifications for semantically similar entities from seed specifications. (3) *Bug Detection*: applies the obtained specifications for large-scale bug detection.

Specifically, in the *Seed Specification Extraction* stage, SpecAuditor analyzes bug patches to extract initial seed specifications, which describe where to check and what constraint must hold. SpecAuditor prompts the LLM to examine both patched code and its description to understand the bug's root cause and the patch behaviors. To ensure correctness, SpecAuditor performs differential validation: a valid specification must be violated in the pre-patch version but satisfied in the post-patch version. Only specifications that are validated are retained as reliable seeds. Validated specifications are then generalized into semantic-level descriptions that capture the underlying vulnerable behavior and its required constraint, without patch-specific details. In the *Specification Generation* stage, given these generalized seeds, SpecAuditor transfers them to new entities beyond those in patches. Using documentation-based semantic retrieval, SpecAuditor identifies entities exhibiting similar behaviors with the seed's underlying behavior. For each entity, SpecAuditor prompts the LLM to analyze its implementation and usage context and, when appropriate, specializes the generalized constraint into a concrete, entity-specific constraint. This process yields specifications for entities beyond those in the patches. In the *Bug Detection* stage, SpecAuditor applies the obtained specifications across the codebase to detect new bugs. It first uses AST-based queries to efficiently locate code snippets involving the specified entities. Then it examines each snippet to decide whether the corresponding constraint is violated, producing initial violation reports. The later report pruning stage re-examines each report with extra context to confirm whether the specification is truly violated and whether the violation can lead to an issue, and filters out irrelevant reports.

Working Example. Figure 4 illustrates a working example of SpecAuditor. As shown in Figure 4(a), the patch for CVE-2022-49878 fixes a memory leak related to `krealloc_array`. Specifically, the function attempts to reallocate an existing buffer. On success, it will free the old buffer and return a new pointer. On failure, it returns NULL while leaving the original buffer unchanged. Therefore, the caller must explicitly handle the original memory to avoid memory leak. However, in the pre-patch code, the return value of `krealloc_array` was assigned directly back to the original pointer. If reallocation fails, the pointer becomes `NULL`, losing the reference to the original buffer and causing a memory leak. The patch fixes this by storing the new pointer in a temporary variable and freeing the original buffer if allocation fails. From this patch, SpecAuditor extracts a seed specification stating that calls to `krealloc_array` must properly handle allocation failures. Next, SpecAuditor generalizes this specification,

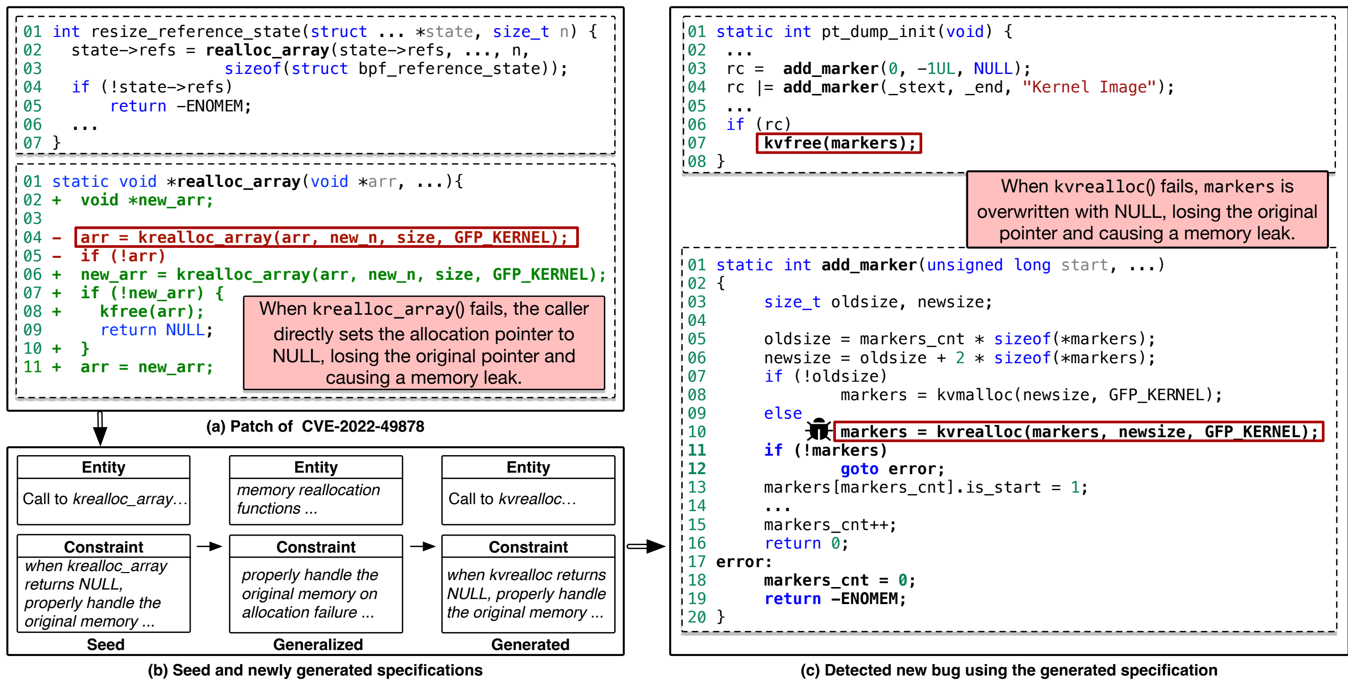


Figure 4: A working example of SpecAuditor. Starting from a bug patch, SpecAuditor extracts a seed specification, generalizes it to capture the underlying behavior, transfers it to a semantically similar entity and generates a corresponding specification, and then applies the newly generated specification to detect a new bug.

describing functions that perform memory reallocation and may return `NULL` must ensure that the original memory is properly handled. Using documentation-based semantic retrieval, SpecAuditor identifies related functions, such as `kvrealloc`, and confirms—based on their implementation and usage—that a similar constraint applies. A new specification related to `kvrealloc` is thus generated. Finally, applying this generated specification across the Linux kernel allows SpecAuditor to detect a previously unknown bug in `add_marker`. The function calls `kvrealloc` and directly overwrites the original pointer. If reallocation fails, the original buffer becomes unreachable and is never freed, causing a memory leak. The example shows that SpecAuditor can generalize new specifications beyond the original patch and leverage them to detect new bugs.

5. Design

Based on these ideas, SpecAuditor operates in three stages. It first extracts, validates, and generalizes the seed specifications. It then identifies semantically similar entities and generates new specifications for them. Finally, it combines AST-based search with LLM reasoning to detect new bugs. We describe each stage below.

5.1. Seed Specification Extraction

Given a bug patch, SpecAuditor extracts seed specifications in the form defined in Section 3. In the pre-patch code, the entity is used without satisfying the required constraint, leading to the bug. The patch fixes the bug by enforcing the required constraint. Therefore, by analyzing the patch, we can summarize the corresponding specifications. However, directly prompting an LLM to extract specifications may result in hallucination, particularly for complex patches. To ensure reliability, SpecAuditor validates each extracted specification and generalizes it by abstracting its underlying behavior and constraint. The prompt used in this phase is shown in Table 1. Next, we introduce the details.

Specification Extraction. SpecAuditor first instructs the LLM to extract a specification. To prevent the model from extracting specifications based on incidental syntactic details, SpecAuditor first guides it to understand the underlying buggy behavior and how the patch fixes it. Specifically, SpecAuditor prompts the LLM to (i) analyze the buggy behavior in the pre-patch code, and (ii) understand how the patch fixes the bug. From this reasoning process, the LLM identifies the entity involved in the bug, and the constraint that was previously violated and is enforced by the fix. To provide sufficient context, SpecAuditor supplies both the natural-language patch description and the patch in diff format. Rather than only presenting the modified lines, SpecAuditor provides the full function bodies containing the changes, because the cause of a bug often depends on surrounding control flow and state. This enables the LLM to understand the causal relationship between the bug and its fix more reliably. The prompt used for specification extraction is shown in Table 1.

Specification Validation. LLM-generated specifications may contain inaccuracies due to hallucination or instability. Using such incorrect specifications in later stages could

TABLE 1: Prompts used in the specification extraction phase.

Specification Extraction	Specification Validation	Specification Generalization
Task: You are a security analysis expert. Given a security patch, summarize a transferable, patch-grounded specification that captures the root cause and fixing logic. The specification will be used to detect similar bugs. **Input:** {`patch_desc`} {`patch_code`} **Output:** {"*entity*": "a syntactically detailed, specific natural-language description of the detection target.", "*constraint*": "a constraint that reflects the fixing intention behind the patch."}	**Task:** You are a security analysis expert. Your task is to analyze the given code and determine if it violates a given security specification. You need to understand the security specifications clearly, then examine the code for the potential violation. **Input:** {`specification`} {`pre-patch code/post-patch code`} **Output:** {"*decision*": "yes/no","*reason*": "detailed explanation"}	**Task:** You are a program analysis expert. Given a concrete specification, your task is to generalize it into a more abstract and semantically meaningful form. Generalize the concrete code entities appearing in the entity and constraint, and describe the key bug-relevant behaviors they represent. The generalized description will be used to identify semantically similar entities. **Input:** {`extracted_spec`} {`patch`} **Output:** {"*generalized_entity*": "generalized description of vulnerable behaviors.", "*generalized_constraint*": "generalized constraint corresponding to the behaviors." }

propagate and amplify errors during generalization or detection. To prevent this, SpecAuditor validates each extracted specification before use. A specification is considered valid only if it is grounded in the original patch: the pre-patch code should violate the specification, while the post-patch code should satisfy it, accurately reflecting the behavioral change enforced by the patch. We refer to this process as differential checking.

To perform this validation, SpecAuditor uses the LLM as the checker. For each extracted specification, SpecAuditor provides the model with the pre- and post-patch versions of the relevant function and asks whether each version satisfies or violates the specification. If the specification is valid, the pre-patch code should be judged as violating the specification, while the post-patch code should be judged as satisfying it. A specification is retained only if the LLM makes these judgments correctly. Specifications are filtered out if they either (i) contain incorrect logic that prevents LLMs from distinguishing pre- and post-patch behavior, or (ii) are correct but cannot be applied effectively by the LLM for bug detection, indicating that they lie beyond LLMs' reasoning capacity. This ensures that only reliable seed specifications are passed to later stages.

Seed Specification Generalization. Seed specifications extracted from patches often contain concrete syntax details in the patches such as specific function names or variable names, which limits their generality. To enable specification generation, SpecAuditor first generalizes each seed specification by removing these surface syntactic details and expressing it instead at the semantic level, which captures the underlying behavior of the entity and the constraint. To achieve this, SpecAuditor uses an LLM to generalize each seed specification. SpecAuditor prompts the LLM to capture the core operation performed by the entity and the constraint required to use it safely, while removing concrete syntactic details tied to specific identifiers or control-flow structures. The resulting semantic specifications serve as abstract yet accurate representations of critical behaviors, providing the foundation for new specification generation.

5.2. Specification Generation

After obtaining generalized seed specifications, the next step is to generate new specifications for other entities that exhibit similar behaviors beyond the patches. The key idea is that a code entity requires a constraint because of its underlying operation, not its specific syntax. Therefore, if another entity performs a similar operation, it should follow the same constraint, even if its implementation differs syntactically. To realize this, SpecAuditor uses documentation-based semantic retrieval to identify other entities in the project that perform similar operations. For each retrieved entity, SpecAuditor examines its implementation and usage context to determine whether the constraint applies and, if so, specializes it into a concrete new specification. This process effectively transfers the behavior–constraint relationship across semantically related entities, expanding the specification set beyond the original patches and enabling broader LLM-driven bug detection.

Related Entities Discovery. After obtaining the generalized underlying behavior from a seed specification, SpecAuditor identifies other entities that perform similar behaviors using documentation-based semantic retrieval. The project documentation provides explicit descriptions of program entities and their intended functionality, making it a more reliable source for semantic retrieval than code syntax alone. Specifically, SpecAuditor proceeds in two steps. First, SpecAuditor parses the documentation to construct an *(entity, description)* database covering functions, data structures, and other program entities. Each description is embedded into a vector using a pre-trained embedding model, and all vectors are stored in a vector database for efficient similarity search. Second, the generalized behavior description from the seed specification is encoded into a query vector using the same embedding model. SpecAuditor then performs semantic search to retrieve entities whose documented behavior is semantically similar. The distance is semantics-based and computed over semantic embeddings obtained from the embedding model. This yields a set of candidate entities for further specification generation.

New Specification Generation. After identifying candidate entities, SpecAuditor determines whether the constraint from the seed specification applies to each candidate and, if appropriate, generates a concrete specification for it. However, semantic similarity between entity descriptions alone does not guarantee that they require the same constraint. Some candidates may not actually perform the vulnerable operation, while others may already satisfy the constraint

TABLE 2: Prompts used for specification generation.

Specification Generation
Task: You are an experienced security researcher. Given a generalized specification consisting of an entity (the applicable behavior) and a constraint (the required constraint), determine whether a given code entity requires this constraint. Analyze the entity's implementation to see if its behavior matches the generalized behavior, examine its usage context, and decide whether callers must follow the constraint to avoid bugs. **Input:** `{generalized_spec}` `{entity_description}` `{entity_source_code}` `{entity_usage_examples}` `{seed_spec}` **Output**: { "*judgment*": "yes/no", "*reason*": ..., "*evidence*": ..., "*concretized_specification*": { "*entity*": a syntactically detailed description of the detection target in code that uses this entity, "*constraint*": concrete constraint that the code should follow } or null }

through internal safety mechanisms (e.g., built-in checks). Therefore, SpecAuditor examines the candidate's actual behavior to verify whether the constraint truly applies. If so, SpecAuditor specializes the abstract constraint to the entity's specific usage context to make it effective for bug detection.

To support this decision, SpecAuditor provides the LLM with relevant code context for each candidate entity. Specifically, SpecAuditor uses lightweight AST-based code search to extract the entity's function definition and usage examples. The definition reflects how the entity is actually implemented, while the usage examples show how it is used in practice. These code snippets are then combined with the entity's description and the generalized behavior description for the LLM. Given this information, the LLM first decides whether the entity should follow a similar constraint, based on whether its actual behavior aligns with the behavior described in the seed specification. If the constraint applies, the LLM then specializes the generalized constraint into a concrete specification according to the candidate's implementation and usage, thereby producing new specifications. The prompt used for this process is shown in Table 2.

Example. As shown in Figure 5, the patch fixes a memory leak due to incorrect handling of `krealloc_array`. During semantic generalization, SpecAuditor abstracts this concrete function into a behavior-level description such as *"functions that reallocate memory..."* Using this description, SpecAuditor retrieves related functions from the Linux kernel documentation, including `kvrealloc`, `krealloc`, and `devm_krealloc`. SpecAuditor then examines each function's definition and usage to determine whether the same constraint applies. For example, for `krealloc`, it generates a corresponding specification, extending detection beyond entities present in the original patch.

5.3. Bug Detection

Given the validated specifications, each consisting of an *Entity–Constraint* pair, SpecAuditor aims to detect new bugs in large-scale codebases. A piece of code that matches the entity but violates its associated constraint is considered potentially buggy. A naive approach would be to provide the specification to an LLM and ask it to analyze the entire

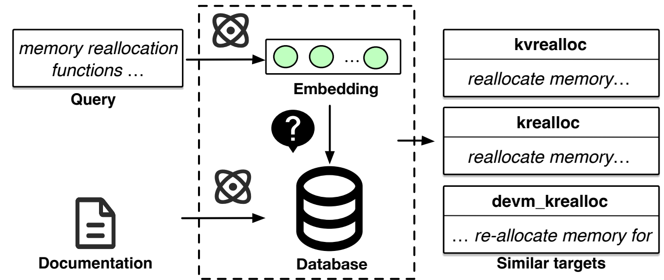


Figure 5: Example of related entities discovery.

codebase. However, this is infeasible for large projects due to prohibitive token costs and extremely low efficiency.

To support efficient LLM-driven auditing, SpecAuditor adopts a hybrid detection framework that leverages the scalability of AST-based code queries and the semantic reasoning capability of LLMs. AST-based (Abstract Syntax Tree) code queries are responsible for locating and retrieving the relevant code, while the LLM focuses on semantic reasoning for code auditing. This prevents the LLM from processing irrelevant code, reducing token usage and improving detection precision. Specifically, SpecAuditor locates candidate sites using AST queries, and identifies violations via LLM reasoning. To reduce false positives, SpecAuditor further performs context-aware pruning by re-evaluating reports with adaptive code context. The prompt used in this phase is shown in Table 3.

Violation Detection. Given a specification in the *Entity–Constraint* form, SpecAuditor first identifies code locations where the entity appears and then checks whether the associated constraint is violated. This stage combines efficient syntactic scoping with LLM-based semantic reasoning: syntactic analysis is used to locate relevant code, while the LLM focuses only on judging whether the constraint holds.

The entity typically contains recognizable syntactic details (e.g., function names, macros, and so on). SpecAuditor leverages these cues to perform AST-based code querying and retrieve relevant code snippets. Unlike string matching that treats code as plain text, AST queries are syntax-aware and can accurately distinguish and locate different code elements based on code structure rather than textual patterns. Moreover, it does not require compilation, enabling scalable analysis over large codebases. To support this step, SpecAuditor uses the LLM to translate the entity description into executable AST queries. These queries are applied across the target codebase to collect all functions that involve the relevant entity. Each retrieved function is treated as a candidate site for further auditing. Once candidate sites are identified, SpecAuditor evaluates whether they violate the specification. For each candidate function, the full function body is provided to the LLM along with the specification, and the model determines whether the required constraint is satisfied. If the entity is present but the constraint is not satisfied, SpecAuditor reports the function as a potential violation. This stage produces initial violation reports, each corresponding to a function that may violate the specification. These reports are then passed to the pruning stage.

TABLE 3: Prompts used in the bug detection phase.

AST Query Generation	Violation Checking	Report Pruning
Task: You are an expert in code analysis and proficient in writing Weggli queries. Your task is to generate a simple and effective AST-based query given the description of a target code location. Weggli query syntax: - Use $var for capturing variables - Use _ for wildcards - Use {} for code blocks ... Generate a single Weggli query that effectively matches the described code locations. **Input**: {`entity_description`} **Output**: generated Weggli query.	**Task:** You are a security expert skilled in bug auditing. Given a function and a specification, determine whether the function violates the specified constraint and leads to a bug. Analyze involved variable usage and data flows, including aliases and escaped values. Check all execution paths and semantically equivalent code forms. **Input**: {`func_code`} {`specification`} **Output**: {"*decision*": "yes/no", "*explanation*": "detailed reasoning of the analysis" }	**Task:** You are a senior bug analysis expert. Given a specification and a code snippet that appears to violate it, determine whether the snippet leads to a real bug. You can request extra context only when necessary. **Input**: {`specification`} {`violation_func`} Extra context (if any): {`prev_interactions`} **Output:** More context needed (when evidence is insufficient): {"*type*": "more_context", "*requests*":[{"*request_type*": "source_code","*entity_name*":...},{"*request_type*": "usage_code","*entity_name*": ...}]} Final decision (when you have enough evidence): {"*type*": "final_decision","*decision*": "yes/no" , "*explanation*": ... }

Report Pruning. This stage re-examines each report with additional context to confirm whether the specification is truly violated and whether the violation leads to an issue. SpecAuditor uses LLMs to emulate expert code review: starting from the reported site, LLMs inspect surrounding context, e.g., callers and callees of the reported functions, to decide whether the violation can lead to an issue.

To support expert-like review, SpecAuditor equips the LLM with a lightweight context retriever that supplies code on demand. For any program entity (function, struct, variable, etc.), the retriever can provide two complementary context types: *(1) Definition*: the entity's full definition (e.g., complete function body, struct definition), exposing internal implementation and semantics. *(2) Usage*: the code where the entity is used (e.g., function callers), revealing data-flow across call chains. For each violation report, SpecAuditor provides the specification and the full function code to the LLM, which first attempts to determine whether the constraint is indeed violated. If the available context is insufficient, the LLM can request additional information by specifying the relevant entity (e.g., a callee, a caller, or a referenced data structure) and the type of context needed (definition or usage). The context feeder then retrieves and supplies the requested code. This iterative process continues until the LLM reaches a conclusion or a preset request limit is met. The LLM finally determines whether the report corresponds to a real bug and provides a brief justification. Reports judged as benign are discarded, and the remaining confirmed cases form the final bug reports. By letting the LLM request only the context it needs, SpecAuditor mirrors human auditing practice, reduces false positives, and avoids unnecessary token consumption.

6. Implementation

We implemented a prototype of SpecAuditor, consisting of about 3.4K lines of Python code. Next, we introduce its key implementation details.

LLM Query. For all tasks involving LLM queries, we accessed the model through the online API. Each request included explicit formatting instructions, requiring the model to return results in JSON for structured parsing.

Specification Extraction. We used PyDriller to parse each patch and extract its description while removing metadata such as author and reviewer information. For the patch code, we retrieved the complete source code of the modified functions using the command `git diff -W`, which represents the patch in diff format with full function context. During specification validation, we parsed both the pre-patch and post-patch versions of the files to obtain the corresponding functions before and after the patch.

Specification Generation. In the specification generation, we built the retrieval corpus from the official documentation. We used the LangChain [29] to perform semantic search. We employed the embedding model to encode descriptions and stored them in the Chroma vector database for efficient similarity search. We computed distances using the Chroma's default L2 distance on normalized embeddings [30]. We used LangChain's `similarity_search_with_score` API for retrieval [31], and then ranked and filtered results using the score 1−distance with a threshold.

AST-based Code Retrieval. For code retrieval in both specification generation and bug detection, SpecAuditor uses Weggli [32], an AST-based semantic code search tool built on tree-sitter [33]. Weggli is efficient for large-scale code analysis. To automate context extraction, we design parameterized AST query templates for various code entities (e.g., functions, structures). At runtime, SpecAuditor instantiates these templates with the target entity and runs the queries to collect relevant code snippets. SpecAuditor is not tied to any specific programming language. Other AST-based search tools, such as Semgrep [34] or AST-Grep [35] can be adopted similarly with minor syntax changes.

7. Evaluation

Research Questions. In our evaluation, we explore the following research questions for SpecAuditor:

- **RQ1.** Can SpecAuditor detect new bugs in real-world, large-scale software? (see Section 7.1)
- **RQ2.** How effective are the individual components of SpecAuditor? (see Section 7.2)
- **RQ3.** How does SpecAuditor compare with state-of-the-art bug detection tools? (see Section 7.3)

Evaluation Dataset. We evaluate SpecAuditor on the widely used open-source system Linux kernel, which features rich semantics, complex program mechanisms, and a large-scale codebase. Such characteristics make it an ideal benchmark for assessing the effectiveness of SpecAuditor in real-world systems. Specifically, our experiments are conducted on the Linux kernel version v6.17-rc3, which contains 79 085 files and approximately 29 million lines of code. To construct the evaluation dataset, we collect seed patches from historical CVE bug patches to ensure both correctness and security. To further maintain diversity, we focus on ten common types and randomly select ten patches for each type, resulting in a dataset of 100 patches.

Configurations. In experiments, we use Claude Sonnet 4 (version 20250514) as the default LLM [36], with temperature = 0 to reduce randomness and improve consistency across runs, and all other parameters kept at default. For the related entity discovery in specification generation, we build the database from the official Linux kernel documentation [37]. For each indexed entry, we extract its associated description, resulting in a database of 14 710 entity–description pairs, including 11 221 functions (76.3%), 1763 structures (12.0%), 1173 macros (8.0%), 366 enumerations (2.5%), and 187 type definitions (1.3%). We use the `BAAI/bge-large-en-v1.5` as the embedding model [38], with similarity threshold 0.35, selected through lightweight tuning to balance recall and precision. Meanwhile, we set the retrieval size to 100 (`Top-k = 100`) to keep the entity set manageable for processing. The embedding model is accessed through the API provided by SiliconFlow [39]. For each candidate entity, up to 5 randomly selected usage examples are provided for LLMs when generating new specifications. Constraint variants across usages are semantically equivalent and handled via semantic reasoning during detection. In the report pruning stage of bug detection, we allow the LLM to request additional context up to 5 iterations.

Machine. All experiments were conducted on an Ubuntu machine equipped with an Intel Xeon Gold 5218 CPU @ 2.30GHz processor, 211 GB of memory, and 17 TB of storage. The LLM and embedding model requests were accessed through the public network.

TABLE 4: Distribution of patches dataset across bug types.

Bug Type	# Patches
Memory leak	10
Buffer overflow	10
Integer overflow	10
Improper input validation	10
Double free / UAF	10
Uninitialized use	10
NULL pointer dereference	10
Resource leak	10
Logic errors	10
Out-of-bounds access	10
Total	**100**

TABLE 5: Bug type distribution for newly detected bugs.

Bug Type	Count
Memory leak	6
Double free / UAF	6
Missing input validation	4
NULL pointer dereference	4
Out-of-bounds access	2
Logic confusion	6
Resource leak	43
Total	**71**

7.1. Effectiveness of Bug Detection

We evaluate the end-to-end performance of SpecAuditor in detecting new bugs in the Linux kernel. We manually reviewed the violation reports and confirmed 71 true bugs. Specifically, for each reported case, we review the original patch that generated the seed specification, the corresponding violated specification (seed or generated), the reported function, and the LLM's reasoning. We then manually inspect the code to see whether the violation indeed leads to a bug. This includes examining the control flow and data flow in the reported function, and the related context if necessary to confirm whether there is a bug. Table 5 summarizes the types and distribution of the detected bugs, spanning diverse types such as memory leaks, use-after-free, and out-of-bounds accesses. These results demonstrate that SpecAuditor can generate diverse specifications that capture project-specific behaviors and detect a wide range of bugs.

Results also show that SpecAuditor can detect bugs involving multiple entities. Among the 71 detected bugs, 6 are multi-entity cases. Figure 6 shows a case with a generated specification and the detected bug. The bug involves the interaction of `kstrdup`, `strsep`, and `kfree`. Specifically, the generated specification captures the interaction, using `kstrdup` as the main entity and describing other related entities in the constraint, as shown in Figure 6(a). Guided by this specification, SpecAuditor detected a new memory leak bug, as shown in Figure 6(b). The code allocates a buffer (`dup`) with `kstrdup` and passes it to `strsep`, which likely rewrites the pointer in place to a different value. As a result, the subsequent `kfree` frees the modified pointer instead of the original allocation, causing a leak.

These 71 detected bugs correspond to 31 different specifications in total, where 15 are seed specifications directly extracted from the patches, and the remaining 16 are generated specifications that do not appear in the original patches. This shows that SpecAuditor generalizes patch-extracted specifications to semantically similar entities, allowing it to detect bugs beyond those observable in patches. We further present case studies of the detected bugs in Section 7.1.2.

Security Implications of Detected Bugs. The detected bugs cover diverse types and can cause resource exhaustion, crashes, and consequently Denial-of-Service or unintended program behavior. Among them, resource leaks are the most prevalent, which involve various types of system resources such as reference counters and file descriptors. These resources are often managed through custom functions, whose

Entity

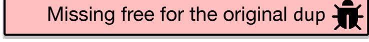
A call to kstrdup that allocates memory for a string duplication.

Constraint

The original pointer returned by kstrdup() must be preserved and freed if strsep() operations are performed on the pointer.

(a) Generated specification

```
01 void dsp_hwec_enable(struct dsp *dsp, ...)
02 {
03     ...
04     dup = kstrdup(arg, GFP_ATOMIC);
05
06     while ((tok = strsep(&dup, ","))) {
07         if (!strlen(tok))
08             continue;
09         ...
10     }
11     kfree(dup);
12 }
```



(b) New memory leak bug detected

Figure 6: Example of a detected multi-entity bug.

patterns vary across subsystems, making them difficult for pattern-based analyzers. SpecAuditor also detects memory leaks and double-free/use-after-free arising from uncommon allocation patterns or custom cleanup logic. In addition, SpecAuditor detects missing input validation bugs that allow arbitrary values from userspace and may cause excessive memory allocation, NULL pointer dereferences caused by missing checks, and logic flaws where implemented behavior deviates from intended semantics.

Bug Disclosure. For all 71 detected bugs, we prepared and submitted corresponding patches to the upstream communities. To date, 52 bugs have been confirmed by the developers, among which 37 have been fixed. Moreover, 21 bug patches have been backported to the Linux kernel stable trees to improve long-term release stability. For the remaining 15 confirmed but unfixed bugs, we are working with maintainers to refine the patches. The other 19 bugs are still pending and awaiting feedback from the developers.

Existence Time of Detected Bugs. To measure the existence time of the detected bugs, we computed the time between when each bug was introduced and when SpecAuditor identified the bug. As shown in Figure 7, the bug lifetimes exhibit substantial variance. The average lifetime is 7.6 years, with a median of 5.9 years. 8 bugs (11%) existed for less than one year, while 39 (55%) remained in the codebase for more than 5 years. 19 bugs (24%) persisted for over ten years, and 2 lasted for more than twenty years. These results indicate that SpecAuditor can detect long-latent bugs overlooked by previous bug detection tools.

7.1.1. Reports Analysis. SpecAuditor produces 297 violation reports, including 71 true bugs. These true bugs correspond to cases where the code violates the specification, leading to potential security risks or incorrect behavior, and therefore require fixes. From a strict bug detection perspective, the remaining 226 reports are false positives. Given the complexity of large-scale systems, we believe the false positives are reasonable and comparable to prior static

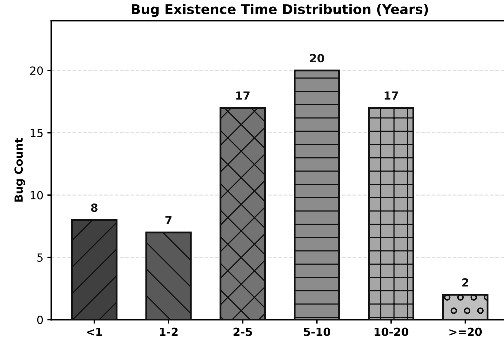


Figure 7: Existence time distribution of detected bugs.

detectors [17], [40]. Moreover, reports include the violated specifications, facilitating manual validation. Meanwhile, many false positives still indicate code quality issues. We categorize false positives into two types: *code warnings* (198 cases) and *invalid reports* (28 cases). Specifically, *code warnings* refer to reports that reflect code patterns similar to those in the original patches, such as missing robustness checks. They are not strict bugs, but fixing them can improve code quality, so we refer to them as code warnings. As system complexity increases, such warnings are valuable for long-term maintainability and may eventually evolve into real bugs. In contrast, *invalid reports* refer to reports that appear to violate the specification, but further inspection shows that the code is actually correct, and modifying it based on the report could even introduce new issues.

We manually analyzed the invalid reports and identified two main causes. (1) *Inaccurate specifications*. Some generated specifications do not correctly represent the core behavior captured in the original patch, leading to unrelated entities being incorrectly included and resulting in false positives. (2) *Constraint satisfied through different ways*. In some cases, the code already satisfies the intended constraint but does so through a different implementation than the specification describes. Such correct variants are flagged as violations due to the LLM's limited understanding.

7.1.2. Case Studies for Diverse Bug Patterns. The results show that SpecAuditor can derive accurate and diverse audit specifications, enabling effective bug detection beyond traditional static analysis. We next introduce three bug patterns covered by SpecAuditor. The corresponding detected bugs have been confirmed by developers and fixed.

Bug Pattern ①: Reference leak caused by missing release for custom reference-management functions. SpecAuditor covers a reference leak pattern where a function acquires a reference to a firmware or device node, but fails to release it after use. As shown in Figure 8(a), which shows the patch of CVE-2024-36955, the function `fwnode_get_named_child_node` returns a referenced firmware node. To correctly manage the reference count, the patch adds a corresponding `fwnode_handle_put` call, ensuring the obtained reference is released after use. Based on this patch, SpecAuditor extracts the seed specification and generalizes it to

other semantically similar functions that acquire references to firmware nodes, such as `of_slim_get_device`, and then generates their corresponding specifications. Using the new specification, SpecAuditor successfully detects a new reference leak bug in the Linux kernel, as shown in Figure 8(b). Here, `of_slim_get_device` obtains a reference to `sbdev`, but the reference is never released, leading to a reference count leak. Traditional static analysis tools typically lack knowledge about reference semantics of custom device management functions. More importantly, functions such as `of_slim_get_device` do not explicitly document that the returned object carries an owned reference that must be released. In contrast, SpecAuditor extracts this specification from real patches and generalizes it to semantically similar functions, enabling it to detect such bugs.

```
01 bool is_link_enabled(struct fwnode_handle *fw_node, u8 idx) {
02
03  link = fwnode_get_named_child_node(fw_node, name);
04
05  fwnode_property_read_u32(link,
06              "intel-quirk-mask",...);
07
08  +   fwnode_handle_put(link);
09  ...
10 }
```
"function that acquires a reference to a firmware node"

(a) Patch of CVE-2024-36955

```
01 void qcom_slim_ngd_notify_slaves(struct ... *ctrl) {
02  struct slim_device *sbdev;
03  ...
04  for_each_child_of_node(ctrl->ngd->pdev->dev.of_node, node) {
05      sbdev = of_slim_get_device(&ctrl->ctrl, node);
06
07      if (slim_get_logical_addr(sbdev))
08          dev_err(ctrl->dev, "Failed to ...\n");
09  }
10 }
```
Missing release of the reference to sbdev

(b) New reference leak bug detected

Figure 8: Example of a reference leak pattern.

```
01 ssize_t tracing_cpumask_write(struct file *filp,
02  const char __user *ubuf, size_t count, loff_t *ppos) {
03  ...
04  + if (count == 0 || count > KMALLOC_MAX_SIZE)
05  +     return -EINVAL;
06
07  err = cpumask_parse_user(ubuf, count, tracing_cpumask_new);
08  if (err)
09  ...
10 }
```
"function that copies user-provided data with size parameters"

(a) Patch of CVE-2024-56763

```
01 static ssize_t proc_write_simdisk(..., const char __user *buf,
02          size_t count, loff_t *ppos)
03 {
04  char *tmp = memdup_user_nul(buf, count);
05  struct simdisk *dev = pde_data(file_inode(file));
06
07  if (IS_ERR(tmp))
08     return PTR_ERR(tmp);
09  ...
10 }
```
Missing check for user-provided count

(b) New missing input validation bug detected

Figure 9: Example of a missing input validation pattern.

Bug Pattern ②: Missing validation of user-supplied size before memory duplication from user space. This pattern illustrates a representative pattern of missing input validation, where user-supplied variables are passed to sensitive

```
01 static int proc_thermal_pci_probe(struct pci_dev *pdev, ...) {
02
03  ret = pcim_enable_device(pdev);
04
05  pci_info->tzone = thermal_zone_device_register_with_trips(...);
06  if (IS_ERR(pci_info->tzone))
08    goto err_del_legacy;
09  ...
10 err_del_legacy:
11 -  pci_disable_device(pdev);
12  return ret;
13 }
```

(a) Patch of CVE-2024-50093

```
01 static int cdnsp_pci_probe(struct pci_dev *pdev, ...){
02
03  ret = pcim_enable_device(pdev);
04
05  cdnsp = kzalloc(sizeof(*cdnsp), GFP_KERNEL);
06  if (!cdnsp)
07    goto disable_pci;
08
09 disable_pci
10  pci_disable_device(pdev);
11 }
```
Double release for pdev

(b) New double release bug detected

Figure 10: Example of a double release pattern.

functions without proper bounds checking, potentially leading to uncontrolled memory allocation and kernel crashes. For example, in the patch of CVE-2024-56763 shown in Figure 9(a), the vulnerable function was accessible through a driver interface and directly used the user-provided variable count without validation. This value was passed to `cpumask_parse_user`, a helper that duplicates data from user space into kernel space. Without checking the size of count, a malicious user could specify an excessively large value, triggering uncontrolled memory allocation and kernel crashes. The patch addresses this issue by adding validation logic for count before invoking `cpumask_parse_user`. From this patch, SpecAuditor extracted the specification: user-supplied size must be validated before memory duplication from user space, and generalized it to detect similar entities. Using this specification, SpecAuditor identified a new bug in `proc_write_simdisk`, as shown in Figure 9(b), where `memdup_user_nul` is called without verifying the parameter count, leading to potential excessive allocation.

Bug Pattern ③: Double resource release due to system-specific indirect call mechanism. SpecAuditor identifies non-typical double-free bugs that are closely tied to system-specific resource management mechanisms. These issues typically arise from conflicts between automatic and manual resource release logic. Specifically, in the Linux kernel, there exists a class of resource allocation functions that automatically clean up resources. Such functions not only allocate resources but also register the device within a management framework, which implicitly ensures that the resources are safely released at specific lifecycle points (e.g., during driver unloading). However, when developers fail to fully understand this implicit contract, they may mistakenly invoke the corresponding manual release function in the error-handling path after calling such an "automatic" function, leading to a double-free. For example, in the patch for CVE-2024-50093 (see Figure 10(a)), the original code called `pcim_enable_device` and then explicitly invoked `pci_disable_device` in the error-handling path, resulting in a double release of the same resource. The patch

fixes the issue by removing the redundant manual release call. These bugs are hard for traditional static analysis tools to detect because the code does not exhibit an obvious double-free data flow. One release is explicit in the code, while the other is implicitly triggered by the framework. From the code surface, it does not appear as a double release. In contrast, SpecAuditor performs semantic reasoning and can detect this kind of double-release bugs.

7.2. Effectiveness of Components

7.2.1. Effectiveness of Specification Extraction. SpecAuditor initially extracted 100 seed specifications from bug patches. After validation, 81 specifications were retained. We manually examined the retained specifications to determine their validity. A specification is considered valid if (i) the pre-patch code violates it, (ii) the post-patch code satisfies it, and (iii) it correctly describes the bug's behavior being fixed. Results show that all 81 specifications are valid. In contrast, before validation, the specification validity rate was 93%, indicating that the validation phase effectively filters out invalid specifications. The initial accuracy is high because SpecAuditor leverages both the patch code and its accompanying description. The description often explains the bug's root cause and the intent of the fix, enabling the model to extract specifications from explicit semantic cues rather than inferring the bug logic from scratch.

The validation filters out specifications that are inherently incorrect or that cannot be applied reliably due to LLM reasoning limitations. Specifically, 19 specifications were filtered out for two main reasons: ❶ *Invalid specifications (7 cases)*: These specifications failed to reflect the actual bug-related behavior. Some misidentified the relevant code, causing the LLM to focus on unrelated locations. Others described the constraint incorrectly or too vaguely, which made the buggy and fixed versions appear indistinguishable. ❷ *Model reasoning limitations (12 cases)*: These specifications were conceptually correct but could not be reliably applied by the LLM. In some cases, the model failed to recognize small semantic differences between the buggy and fixed versions. In others, it focused on irrelevant code fragments and overlooked key bug-related statements. Overall, the validation ensures that only specifications reliable for LLM-driven bug detection are retained.

Behavior Breakdown Analysis. The generalized behaviors can be classified into six semantic types, each associated with typical safety constraints: ❶ *Memory Management*: covers memory allocation, release, and related operations. These behaviors require proper pairing and return-value checks to prevent memory leaks and use-after-free issues. ❷ *Resource Management*: includes management of non-memory resources such as reference-counted objects, files, and device handles. Correct acquire–release pairing is necessary to avoid resource leaks or double releases. ❸ *User Input and Data Transfer*: involves reading or copying data between user and kernel space. Boundary and validity checks are required to prevent out-of-bounds accesses and excessive resource consumption. ❹ *Arithmetic and Bitwise Operations*: covers shifts, divisions, and size or index calculations. Proper overflow, range, and type checks are needed to avoid incorrect boundary computations or integer overflows. ❺ *Error-Value Returning*: returning error values requires proper propagation and handling of failure conditions. Missing or incorrect error handling may lead to null-pointer dereferences or inconsistent program state. ❻ *String and Buffer Processing*: includes string copying, formatting, and length handling. Boundary checks and proper length validation are required to prevent issues such as buffer overflows.

7.2.2. Effectiveness of Specification Generation. Starting from 81 generalized specifications, SpecAuditor generalizes them to semantically similar entities. Using a database of 14 710 entity–description pairs, it retrieves 2964 candidate entities and finally produces 314 new concrete specifications. The reduction arises because many retrieved entities do not actually exhibit the intended vulnerable behavior. Moreover, entities with similar behaviors may not share the same constraints, as differences in their semantics can affect whether constraints apply. For example, both `kzalloc` and `devm_kzalloc` allocate memory, `kzalloc` requires an explicit `kfree`, but `devm_kzalloc` requires no manual release as it's auto-managed.

We further analyze the validity of the inferred specifications in terms of the soundness and completeness of the constraints associated with each entity. For soundness, a specification is sound if its constraint correctly reflects entity semantics, such that violating it can lead to incorrect behavior or security impact in some context. Otherwise, unsound specifications directly lead to false positives in bug detection. Based on manual inspection of the generated specifications and their associated code, 84% of them are sound, which are effective for bug detection. The remaining unsound cases arise for two main reasons. First, due to hallucination, the LLM generates specifications for entities unrelated to the original vulnerable behaviors in seed patches. Second, insufficient context during generalization leads to incorrect abstractions, where the specification fails to capture critical system-level knowledge or cross-entity interactions, leading to biases.

For completeness, a specification is complete if it captures all conditions under which the constraint should apply, so that it applies correctly across diverse contexts. Completeness is difficult to measure, as it is impractical to enumerate all possible cases. In practice, incompleteness of inferred specifications arises for two reasons. First, SpecAuditor transfers valid constraints from seed patches, but may miss additional conditions required when applying them to other cases. Second, SpecAuditor utilizes limited usage examples for specification generation, so it cannot cover all forms of constraints. For example, an allocated resource may be released in different ways, but the specification only describes one way. Despite this, the impact of incompleteness is mitigated during bug detection. SpecAuditor does not report all violations, it performs context checking and semantic reasoning to reduce invalid reports.

TABLE 6: Distribution of entity types in specifications.

Entity Type	Description	Ratio
Function	These correspond to functions (including function-like macros) whose specifications constrain required properties of the function and its usages. Their constraints are tied to the function's specific semantics and vary widely.	82.5%
Data structure	These correspond to program objects (e.g., structs, arrays). Their constraints include proper initialization before exposure and correct termination (e.g., sentinel-terminated arrays), etc.	16.2%
Control-flow	These relate to execution-path logic. Constraints typically ensure proper traversal and termination, such as ensuring iterations stop at the correct boundaries.	1.0%
Others	Covers processing workflows (e.g., protocol decoding) where the constraint applies to the entire execution sequence.	0.3%

Distribution of Entity Types. For the obtained specifications, we analyze their entity types. For specifications involving multiple entities, each is counted under its main entity type. Table 6 shows the distribution of entity types, along with brief descriptions of their constraints. As shown in Table 6, function entities account for 82.5%, followed by data structures. This distribution also mirrors the prevalence of entity types in the Linux kernel documentation. These entities encode rich semantics and behaviors (e.g., structure access, numeric manipulation, and sensitive-state handling), which determine corresponding constraints.

Complementary to Official Documentation. To evaluate the added value of SpecAuditor's generated specifications, we compared them against the official documentation. We focused on specifications whose constraints require nontrivial specialization, beyond simple NULL or boundary checks, and therefore should have been explicitly documented. We randomly sampled 50 correctly generated specifications involving custom resource allocators. In these cases, knowing the vulnerable operation is insufficient because the correct constraint depends on a custom release operation that may differ from the one in the seed patch. For each specification, we located the corresponding entity in the documentation and checked whether the required constraint was described. We found that 42 out of 50 (84%) lacked any such documentation, indicating that documentation alone is insufficient for obtaining these specifications. In contrast, SpecAuditor generalizes behavior-level semantics from seed specifications, uses documentation only to locate related entities, and specializes constraints from the source code. This enables SpecAuditor to generate undocumented specifications and detect the associated bugs effectively.

7.3. Comparison with Related Work

In our comparative experiments, we evaluate SpecAuditor against four types of related methods. Specifically, we consider: (1) LLM-only detection, which directly uses an LLM to judge whether a given code snippet is buggy. (2) LLM-assisted bug detection, which leverages LLMs to enhance traditional static analyzers. (3) LLM-driven bug detection, which directly employs LLMs for bug auditing and reasoning. (4) API misuse detection, which extracts API usage specifications to detect API misuses. The goal of this evaluation is to examine whether SpecAuditor can complement these methods in detecting new bugs. Specifically, we analyze whether the bugs detected by SpecAuditor are already covered by existing tools or represent new cases. The overall results are shown in Table 7, which reports the recall of each method on the bugs detected by SpecAuditor.

TABLE 7: Bug detection results of compared methods.

Type	Method	# Bugs (out of 71)
LLM-only	function-level [41]	7
	file-level [41]	1
LLM-assisted	KNighter [6]	0
LLM-driven	RepoAudit [10]	0
API misuse detection	AppMiner [17]	0

Compared with LLM-only Bug Detection. We compare SpecAuditor with directly using Claude Sonnet 4 for bug detection. We adopt the prompt from prior work [15] and let the LLM judge whether a given code snippet is vulnerable (see Table 8). We evaluate two input settings: function-level (providing the buggy function) and file-level (providing the full file). On the bugs detected by SpecAuditor, the LLM recalls only 7 cases at the function level and 1 case at the file level, missing most bugs. Without effective guidance, the LLM struggles to focus on relevant code regions and easily overlooks bugs. When given the full file, the larger context makes it harder to focus relevant code, resulting in lower recall. In contrast, SpecAuditor incorporates project-specific knowledge to guide LLMs and detect missed bugs.

TABLE 8: Prompt used for LLM-only detection.

You are an expert C programmer who can carefully analyze the provided C code. The goal is to judge if the provided code is vulnerable or not. Your answer should be concise, with a yes or no to represent the code's type. If it is vulnerable, then yes; otherwise, no. Also, please explain concisely why you made the decision.

Compared with LLM-assisted Bug Detection. We compare SpecAuditor with KNighter [6], a state-of-the-art approach that synthesizes static-analysis checkers from bug patches. To reduce token consumption and cost, we provided KNighter only with the bug patches corresponding to the bugs detected by SpecAuditor. Results show that KNighter fails to detect any of the bugs detected by SpecAuditor. This indicates that SpecAuditor is complementary to existing LLM-assisted bug detection methods. We further analyzed why KNighter did not detect the bugs identified by SpecAuditor. A key reason is the limited semantic expressiveness of checker templates: some bugs detected by SpecAuditor involve diverse patterns that cannot be effectively captured by KNighter's checker templates. Additionally, KNighter's generated checkers tend to encode syntax details directly from the patches, resulting in limited generalization.

Compared with LLM-driven Bug Detection. We compare SpecAuditor with RepoAudit [10], a state-of-the-art LLM-driven bug detection tool. RepoAudit uses LLM-driven data-flow analysis and supports three bug types: null-pointer dereference, memory leak, and use-after-free. We evaluate whether RepoAudit can detect the 16 new bugs identified by SpecAuditor that fall within these types. Following the setup described in RepoAudit's paper, we treat each bug's corresponding subsystem as an independent repository and run RepoAudit using the same LLM as SpecAuditor while keeping all other settings at their defaults. To control token usage and cost, we restrict RepoAudit to analyzing only the locations associated with the 16 bugs. Results show that RepoAudit fails to detect any of the bugs identified by SpecAuditor. The main reason is that RepoAudit initiates data-flow analysis from a predefined set of generic audit entities such as standard allocators. This design makes it less effective for project-specific resource management functions that are not included in the anchor list. Unlike RepoAudit, SpecAuditor does not rely on a predefined set of entities for audit. Instead, SpecAuditor extracts specifications from patches and generalizes them to semantically similar entities. This enables SpecAuditor to detect diverse bugs that general LLM-driven methods overlook.

Compared with API Misuse Detection. We compare with AppMiner, a recent API misuse detector that mines frequent API path patterns from code and detects violations. Other tools are excluded because they cannot be automatically applied to the Linux kernel [42], [43]. We run AppMiner on the full Linux kernel, which produces over ten thousand violation reports. We cross-check whether it detects the bugs identified by SpecAuditor, and find that it detects none of them. Specifically, AppMiner relies on syntax-level co-occurrence statistics, making it sensitive to irrelevant statements and missing less frequent but critical patterns. In our analysis, over 85% of its mined patterns are reduced to simple checks for single APIs. In contrast, SpecAuditor leverages semantic transfer to generate specifications beyond existing API pattern mining and detecting new API misuses.

Failure Analysis for SpecAuditor. The results show that SpecAuditor can complement compared methods in detecting new bugs. Figure 8 presents a bug missed by all compared methods, as they lack knowledge of relevant entities or fail to leverage them during detection. Correspondingly, SpecAuditor can miss bugs detected by existing methods. It focuses on entity-related bugs, i.e., violations of specifications tied to specific entities, and is less effective for bugs without explicit entities. For example, Figure 11 shows a bug caused by missing checks on the addition of an offset and a length [44]. Both values are provided by userspace, and the lack of validation can lead to integer overflow. In such cases, the required specifications are generic (e.g., validating arithmetic on user inputs) rather than entity-specific ones, making them hard for SpecAuditor to capture. In contrast, methods such as LLM-only detection may still detect such bugs, as they do not rely on entity-specific specifications and can reason over general code patterns.

x86/sgx: Add overflow check in sgx_validate_offset_length()
sgx_validate_offset_length() function verifies "offset" and "length" arguments provided by userspace, but was missing an overflow check on their addition. Add it.

```
01 int sgx_validate_offset_length(struct sgx_encl *encl,
02   ...
03   if (!length || !IS_ALIGNED(length, PAGE_SIZE))
04     return -EINVAL;
05
06 +  if (offset + length < offset)
07 +    return -EINVAL;
08 +
09   if (offset + length - PAGE_SIZE >= encl->size)
10     return -EINVAL;
```

Figure 11: Example of a bug that SpecAuditor fails to detect.

7.4. Performance and Cost

To assess the real-world applicability of SpecAuditor, we measured the token usage, API cost, and runtime of each LLM-driven stage using Claude Sonnet 4 (version 20250514) and its official pricing. As shown in Table 9, the total token consumption is 41.35 million tokens, corresponding to an estimated cost of $164.73. The end-to-end analysis completes in 9.1 hours, demonstrating that SpecAuditor can efficiently analyze a large real-world codebase within a practical time and cost budget.

TABLE 9: Token usage, cost, and time of each stage.

Stage	Tokens (In/Out)	Cost	Time
Specification Extraction	0.53M / 0.06M	$2.49	0.7h
Specification Generation	4.11M / 0.44M	$18.93	2.1h
Bug Detection	33.32M / 2.89M	$143.31	6.3h
Total	41.35M	$164.73	9.1h

8. Discussion and Limitations

Scope and Assumption. SpecAuditor is designed to detect entity-related bugs. SpecAuditor does not handle generic specifications without specific entities and thus cannot detect the related bugs. The evaluation shows that SpecAuditor generates specifications for entities such as functions and data structures. Nevertheless, the effectiveness of SpecAuditor on other types of entities (e.g., sensitive states or structure fields) remains unclear and requires further study. In particular, SpecAuditor leverages entity semantics and behaviors to generate specifications. When constraints largely depend on context-specific logic, they are harder to abstract and generalize from the entity alone, making SpecAuditor less effective. In addition, SpecAuditor assumes that the bug root cause and its fix lie within the same function when extracting specifications from patches. In practice, some bugs have root causes outside the patched function, making the patched function alone insufficient. Handling such cases requires carefully selected context and remains future work.

Documentation for Specification Generation. SpecAuditor uses official documentation to retrieve similar entities. Since documentation may omit critical behavioral descriptions, semantic retrieval may fail to capture all relevant entities. To mitigate this, SpecAuditor can be extended with an additional behavior-summarization module, where an LLM analyzes each entity's source code and generates a concise summary of its behavior. These summaries can be incorporated into the vector database to enhance semantic retrieval. It is a one-time offline preprocessing step per project. We conduct a preliminary study to evaluate the effectiveness of LLM-generated summaries. To this end, we construct a dataset of 100 related and 100 irrelevant entities, randomly sampled and labeled according to documentation-based retrieval results. We then replace documentation with LLM-generated summaries produced by Claude Sonnet 4. Using these summaries, the retrieval achieves up to 0.91 recall and 0.80 precision, with an F1 score of 0.85 under an adjusted threshold of 0.59. These results suggest that LLM-generated summaries can complement documentation. In practice, the threshold for retrieval may differ, as LLM-generated summaries focus closely on behavior while documentation may provide diverse descriptions.

Mitigating False Positives. As a static auditing tool, SpecAuditor inevitably suffers from false positives. We mitigate them with additional strategies. First, we can perform manual review in advance to filter out inaccurate specifications, eliminating the resulting false positives. Second, false positives can follow similar patterns. We may design prompts that describe these patterns and use the LLM to filter them. In addition, directed fuzzing can be used to validate bugs with static reports as guidance [45].

LLM Capabilities and Limitations. SpecAuditor utilizes LLMs for understanding code semantics and generating diverse specifications that are difficult for static analyzers to capture. The capability of SpecAuditor is also influenced by the reasoning capacity of LLMs. SpecAuditor naturally benefits as LLMs improve, since stronger models provide more accurate semantic reasoning. However, current LLMs remain prone to hallucination and instability, which may lead to incorrect specifications or misclassified violations. Future work may further enhance reliability by integrating LLM reasoning with traditional program analysis techniques, such as data-flow or path-sensitivity checking [46].

9. Related Work

Our work lies at the intersection of LLM-assisted bug detection, LLM-driven bug detection, and specification extraction for bug detection.

LLM-assisted Static Analysis. Recent studies explored using LLMs to enhance traditional static analysis [6], [9], [42], [47], [48], [49], [50]. In these studies, LLMs are not used to directly detect bugs, but instead assist specific sub-tasks such as extracting critical information or generating checker for static analyzers. For example, GPTAID [42] uses LLMs to generate API parameter specifications, and InferROI [47] infers resource-related operations. IRIS [48] infers taint specifications from source code and integrates them with CodeQL to enhance taint analysis. Further, KNighter [6] automatically generates new static analysis checkers. In contrast, SpecAuditor performs LLM-driven detection based on semantic reasoning, allowing it to identify diverse bug patterns that static analyzers struggle to capture.

LLM-driven Bug Detection. Beyond assisting static analysis, recent work explores using LLMs for direct bug detection [10], [51], [52]. For example, RepoAudit [10] introduces an LLM-based agent that autonomously audits entire repositories. Its reasoning process is constrained by the model's memorized knowledge. Besides, LLM-based auditing of large systems like the Linux kernel is computationally expensive and hard to scale. BUGSCOPE [51] constructs procedural prompts from manually selected good examples and executes them directly with the LLM, which restricts the detection to those exact patterns. In contrast, SpecAuditor generates diverse specifications for LLM-driven bug detection that goes beyond the model's built-in knowledge. Meanwhile, some studies [53], [54], [55] focus on known benchmarks and treat detection as a binary classification task, i.e., directly judging whether given code snippets are buggy. They cannot be directly applied to large-scale bug detection that require effective searching. Different from them, SpecAuditor performs codebase-wide analysis and aims to discover new classes of bugs.

Specification Extraction for Bug Detection. To improve bug detection, prior studies have explored automatic specification extraction from different software artifacts, including source code [17], [42], documentation [28], or patches [23]. Specifically, source code–based methods often infer specifications by mining common usage patterns from codebases. [17], [26], [40]. For example, APP-Miner [17] mines recurring API usage subgraphs to infer usage specifications. Documentation-based methods extract specifications from official documents [9], [27], [56]. For example, ChatDetector [9] extracts memory resource management APIs from documents. Patch-based methods extract specifications directly from bug patches [22], [23]. For instance, APHP [22] focuses on API post-handling specifications and extract them from patches. In contrast, SpecAuditor abstracts from syntactic patterns to behavioral semantics, thus generates specifications that go beyond the original artifacts.

10. Conclusion

In conclusion, SpecAuditor generates audit specifications that guide LLM-driven bug detection and complement the limitations of traditional static analysis. By learning from bug patches and generalizing them into semantically similar specifications, SpecAuditor enables scalable and semantically grounded LLM-driven bug detection on large codebases. Our evaluation on the Linux kernel shows that SpecAuditor detects 71 new bugs that were missed by prior methods. To date, 52 of these bugs have been confirmed, including 37 that have already been fixed. The results highlight the promise of specification-guided LLM auditing for future bug detection.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd for their constructive feedback and efforts that help improve the paper. We are grateful to the Linux kernel maintainers for their valuable feedback and collaboration during bug patching. We also thank Haoyu Xiao for insightful suggestions on this work.

References

[1] GitHub, Inc., "Codeql: A semantic code-analysis engine for querying code as data," https://codeql.github.com/, 2025, accessed: 2025-10-23.

[2] LLVM Project, "Clang Static Analyzer: A source code analysis tool for C, C++, and Objective-C," https://clang-analyzer.llvm.org/, 2025, accessed: 2025-10-05.

[3] J. Mao, Y. Chen, Q. Xiao, and Y. Shi, "RID: Finding Reference Count Bugs with Inconsistent Path Pair Checking," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[4] J. Liu, L. Yi, W. Chen, C. Song, Z. Qian, and Q. Yi, "LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution," in *Proceedings of the 31st USENIX Security Symposium (Security)*, 2022.

[5] D. Liu, Q. Wu, S. Ji, K. Lu, Z. Liu, J. Chen, and Q. He, "Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, 2021.

[6] C. Yang, Z. Zhao, Z. Xie, H. Li, and L. Zhang, "Knighter: Transforming static analysis with llm-synthesized checkers," *ArXiv*, vol. abs/2503.09002, 2025.

[7] Y. Lyu, Y. Fang, Y. Zhang, Q. Sun, S. Ma, E. Bertino, K. Lu, and J. Li, "Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 2096–2113.

[8] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning," in *Proceedings of the 28th Network and Distributed System Security Symposium (NDSS)*, 2021.

[9] Y. Yang, J. Liu, K. Chen, and M. Lin, "The midas touch: Triggering the capability of llms for rm-api misuse detection," *ArXiv*, vol. abs/2409.09380, 2024.

[10] J. Guo, C. Wang, X. Xu, Z. Su, and X. Zhang, "RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing," in *International Conference on Machine Learning*. PMLR, 2025, pp. 21 083–21 100.

[11] C. Wang, W. Zhang, Z. Su, X. Xu, and X. Zhang, "Sanitizing Large Language Models in Bug Detection with Data-Flow," in *Conference on Empirical Methods in Natural Language Processing*, 2024.

[12] A. Lekssays, H. Mouhcine, K. Tran, T. Yu, and I. M. Khalil, "LLMx-CPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models," *ArXiv*, vol. abs/2507.16585, 2025.

[13] Y. Li, P. Branco, A. M. Hoole, M. Marwah, H. M. Koduvely, G.-V. Jourdan, and S. Jou, "Sv-trusteval-c: Evaluating structure and semantic reasoning in large language models for source code vulnerability analysis," *2025 IEEE Symposium on Security and Privacy (SP)*, pp. 3014–3032, 2025.

[14] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 2048–2060, 2023.

[15] J. Lin and D. Mohaisen, "From large to mammoth: A comparative evaluation of large language models in zero-shot vulnerability detection," *Proceedings 2025 Network and Distributed System Security Symposium*, 2025.

[16] M. Lin, K. Chen, Y. Yang, and J. Liu, "Uncovering the iceberg from the tip: Generating API Specifications for Bug Detection via Specification Propagation Analysis," in *Proceedings of the 32rd Network and Distributed System Security Symposium (NDSS)*, 2025.

[17] J. Jiang, J. Wu, X. Ling, T. Luo, S. Qu, and Y. Wu, "APP-Miner: Detecting API Misuses via Automatically Mining API Path Patterns," *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 4034–4052, 2024.

[18] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically Detecting Error Handling Bugs Using Error Specifications," in *Proceedings of the 25th USENIX Security Symposium (Security)*, 2016.

[19] P. Hu, R. Liang, Y. Cao, K. Chen, and R. Zhang, "AURC: Detecting errors in program code and documentation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[20] Q. Wu, A. Pakki, N. Emamdoost, S. McCamant, and K. Lu, "Understanding and detecting disordered error handling with precise function pairing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[21] N. Dossche and B. Coppens, "Inference of Error Specifications and Bug Detection Using Structural Similarities," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1885–1902.

[22] M. Lin, K. Chen, and Y. Xiao, "Detecting api post-handling bugs using code and description in patches," in *USENIX Security Symposium*, 2023.

[23] W. Chen, B. Zhang, C. Wang, W. Tang, and C. Zhang, "Seal: Towards diverse specification inference for linux interfaces from security patches," *Proceedings of the Twentieth European Conference on Computer Systems*, 2025.

[24] Q. Wu, Y. Xiao, D. Kirat, K. Eykholt, J. Jang, and D. L. Schales, "One Bug, Hundreds Behind: LLMs for Large-Scale Bug Discovery," *arXiv preprint arXiv:2510.14036*, 2025.

[25] P. Bian, B. Liang, Y. Zhang, C. Yang, W. Shi, and Y. Cai, "Detecting bugs by discovering expectations and their violations," *IEEE Transactions on Software Engineering*, vol. 45, pp. 984–1001, 2019.

[26] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "APISan: Sanitizing API Usages through Semantic Cross-Checking," in *Proceedings of the 25th USENIX Security Symposium (Security)*, 2016.

[27] T. Lv, R. Li, Y. Yang, K. Chen, X. Liao, X. Wang, P. Hu, and L. Xing, "RTFM! Automatic Assumption Discovery and Verification Derivation from Library Document for API Misuse Detection," *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[28] X. Wang and L. Zhao, "Apicad: Augmenting api misuse detection through specifications from code and documents," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[29] LangChain Developers, "LangChain Documentation," https://python.langchain.com/docs, accessed: 2025-10-05.

[30] Chroma, "Chroma documentation: Configure collections," https://docs.trychroma.com/docs/collections/configure, 2026.

[31] LangChain, "Chroma API," https://reference.langchain.com/python/langchain-chroma/vectorstores/Chroma/similarity_search_with_score, 2026.

[32] Weggli contributors, "Weggli: Fast and robust semantic search for code using tree-sitter," https://github.com/weggli-rs/weggli, 2025, accessed: 2025-10-23.

[33] tree sitter, "tree-sitter," https://tree-sitter.github.io/tree-sitter/, 2025.

[34] Semgrep Contributors, "Semgrep: Lightweight static analysis for many languages," https://github.com/semgrep/semgrep, 2025, accessed: 2025-10-23.

[35] ast-grep Contributors, “ast-grep: A cli tool for code structural search, lint, and rewriting,” https://github.com/ast-grep/ast-grep, 2025, accessed: 2025-10-23.

[36] Anthropic, “System card: Claude opus 4 & claude sonnet 4,” Anthropic, Tech. Rep., May 2025, pDF. [Online]. Available: https://www-cdn.anthropic.com/07b2a3f9902ee19fe39a36ca638e5ae987bc64dd.pdf

[37] The Linux Kernel Project, “Index — the linux kernel documentation,” https://www.kernel.org/doc/html/latest/genindex.html, accessed: 2025-10-05.

[38] S. Xiao, Z. Liu, P. Zhang, and N. Muennighoff, “C-pack: Packaged resources to advance general chinese embedding,” 2023.

[39] siliconflow, “siliconflow,” https://www.siliconflow.com/, 2025.

[40] K. Lu, A. Pakki, and Q. Wu, “Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences,” in *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[41] J. Lin, D. Mohaisen *et al.*, “From large to mammoth: A comparative evaluation of large language models in vulnerability detection.” in *Proceedings of the 32rd Network and Distributed System Security Symposium (NDSS)*, 2025.

[42] J. Liu, Y. Yang, K. Chen, and M. Lin, “Generating api parameter security rules with llm for api misuse detection,” *ArXiv*, vol. abs/2409.09288, 2024.

[43] Z. Li, A. Machiry, B. Chen, M. Naik, K. Wang, and L. Song, “Arbitrar: User-guided api misuse detection,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1400–1415.

[44] NIST National Vulnerability Database (NVD), “CVE-2022-49785 — NVD Detail,” https://nvd.nist.gov/vuln/detail/CVE-2022-49785, 2026.

[45] A. Bao, W. Zhao, Y. Wang, Y. Cheng, S. McCamant, and P.-C. Yew, “From alarms to real bugs: Multi-target multi-step directed greybox fuzzing for static analysis result verification,” in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 6977–6997.

[46] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and Discovering Vulnerabilities with Code Property Graphs,” in *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*, 2014.

[47] C. Wang, J. Liu, X. Peng, Y. Liu, and Y. Lou, “Boosting static resource leak detection via llm-based resource-oriented intention inference,” *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 2905–2917, 2025.

[48] Z. Li, S. Dutta, and M. Naik, “Llm-assisted static analysis for detecting security vulnerabilities,” *ArXiv*, vol. abs/2405.17238, 2024.

[49] P. Li, S. Yao, J. S. Korich, C. Luo, J. Yu, Y. Cao, and J. Yang, “Automated Static Vulnerability Detection via a Holistic Neurosymbolic Approach,” *ArXiv*, vol. abs/2504.16057, 2025.

[50] C. Wang, Z. Li, S. Dutta, and M. Naik, “QLCoder: A Query Synthesizer For Static Analysis of Security Vulnerabilities,” *arXiv preprint arXiv:2511.08462*, 2025.

[51] J. Guo, C. Wang, D. Deluca, J. Liu, Z. Zhang, and X. Zhang, “BugScope: Learn to Find Bugs Like Human,” *ArXiv*, vol. abs/2507.15671, 2025.

[52] Y. Xia, Z. Xie, P. Liu, K. Lu, Y. Liu, W. Wang, and S. Ji, “Beyond Static Pattern Matching? Rethinking Automatic Cryptographic API Misuse Detection in the Era of LLMs,” *Proc. ACM Softw. Eng.*, vol. 2, pp. 113–136, 2025.

[53] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, “LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights,” *ACM Computing Surveys*, vol. 58, pp. 1 – 35, 2025.

[54] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” *Advances in neural information processing systems*, vol. 32, 2019.

[55] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, “Vulnerability detection with code language models: How far are we?” *arXiv preprint arXiv:2403.18624*, 2024.

[56] M. Zheng, C. Wang, X. Liu, J. Guo, S. Feng, and X. Zhang, “An llm agent for functional bug detection in network protocols,” *ArXiv*, vol. abs/2506.00714, 2025.

[57] DeepSeek-AI, A. Liu, B. Feng, B. Xue, B.-L. Wang *et al.*, “Deepseek-v3 technical report,” *ArXiv*, vol. abs/2412.19437, 2024.

[58] Q. Team, “Qwen3 technical report,” *ArXiv*, vol. abs/2505.09388, 2025.

Appendix A. Ethics Considerations

Our work detected 71 previously unknown bugs. For all of them, we have developed and submitted corresponding patches to disclose the bugs to the community. So far, 52 bugs have been confirmed by the developers, among which 37 have been fixed. For the remaining 15 confirmed but unfixed cases, we are collaborating with maintainers to refine the patches. The other 19 reports are still pending and awaiting feedback from the developers.

Appendix B. LLM Usage Considerations

Originality. LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.
Transparency. LLMs were also employed as part of our research methodology. We have explicitly introduce the prompts, configurations, and model used in our paper. The LLM accessed through a public API reachable from the open network, ensuring full reproducibility. During all experiments, the temperature parameter was fixed at 0 to minimize randomness for reproducibility. Our released artifact allows users to configure alternative LLM endpoints, including both open-source and closed-source models with comparable capabilities, which can achieve similar results under equivalent settings. Users can also rerun the pipeline multiple times to assess reproducibility.
Responsibility. Our work involves no model training and therefore introduces no additional environmental or ethical concerns. All LLM-based experiments were conducted using publicly available APIs.

Appendix C. Specification Generation with Open-Source LLMs

SpecAuditor is model-agnostic and works with different LLMs. In the main experiments, we use Claude Sonnet 4 as the default model. To study model differences, we explore the performance of two open-source models, DeepSeek-V3 (deepseek-v3-250324) [57] and Qwen3-235B (Qwen3-235B-A22B) [58], on specification generation. Out of 100

seed specifications, DeepSeek-V3 and Qwen3-235B retain 59 and 43 seed specifications, compared to 81 with Claude Sonnet 4. This gap reflects weaker model capability, as the two models extract more inaccurate specifications and more often fail to distinguish buggy from non-buggy code. They generate 298 and 226 specifications, respectively. We randomly sample 50 specifications from each model and manually examine whether they are valid for bug detection. The valid rates are 76% and 62% for DeepSeek-V3 and Qwen3-235B. Compared to Claude Sonnet 4, the two models are more prone to hallucinations, such as incorrectly transferring constraints to entities that should not have them.

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

This paper presents SpecAuditor, an LLM-based system that generates specifications from patches and uses the generated specifications to identify other vulnerabilities that violate the specification.

D.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.

D.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. The paper presents an interesting approach to eliciting specifications from patches. It generalizes patches using the entity-constraint insight and uses it to generate specifications on the corresponding entity. Although the use of LLMs to analyze patches is known, SpecAuditor explores the entity aspect, which can be generalized to other large service-oriented codebases, e.g., SQL servers.
2) Furthermore, the paper identifies impactful vulnerabilities and creates a new tool to enable future science. The research has identified 71 new bugs in the Linux Kernel, many of which are undetected over several years. These bugs could have a serious security impact (e.g., spatial and temporal safety violations, resource leakage, etc.).

D.4. Noteworthy Concerns

None.