RFCAUDIT: AI Agent for Auditing Protocol Implementations Against RFC Specifications

Mingwei Zheng[1], Chengpeng Wang[1], Xuwei Liu[1], Jinyao Guo[1], Shiwei Feng[1],and Xiangyu Zhang[1]
[1]Department of Computer Science, Purdue University, West Lafayette, USA
Email: {zheng618, wang6590, liu2598, guo846, feng292, xyzhang}@purdue.edu

***Abstract*—Functional correctness is critical for ensuring the reliability and security of network protocol implementations. Functional bugs, instances where implementations diverge from behaviors specified in RFC documents, can lead to severe consequences, including faulty routing, authentication bypasses, and service disruptions. Detecting these bugs requires deep semantic analysis across specification documents and source code, a task beyond the capabilities of traditional static analysis tools. This paper introduces RFCAUDIT, an autonomous agent that leverages large language models (LLMs) to detect functional bugs by checking conformance between network protocol implementations and their RFC specifications. Inspired by the human auditing procedure, RFCAUDIT comprises two key components: an indexing agent and a detection agent. The former hierarchically summarizes protocol code semantics, generating semantic indexes that enable the detection agent to narrow down the scanning scope. The latter employs demand-driven retrieval to iteratively collect additional relevant data structures and functions, eventually identifying potential inconsistencies with the RFC specifications effectively. We evaluate RFCAUDIT across six real-world network protocol implementations. RFCAUDIT identifies 47 functional bugs with 81.9% precision, of which 20 bugs have been confirmed or fixed by developers.**

I. INTRODUCTION

Network protocols are essential to digital communication, establishing standardized rules that govern data exchange between devices. They specify message formats and sequencing, as well as mechanisms for routing, connection management, and resource allocation. Due to their complexity, implementations of network protocols are susceptible to functional bugs—logical errors that cause deviations from intended behaviors, which undermines the reliability and security of network infrastructures. For example, the Zerologon vulnerability (CVE-2020-1472 [1]) in Microsoft's Netlogon Remote Protocol (MS-NRPC) [2] stemmed from improper initialization of a cryptographic vector, which allows attackers to impersonate any host, including domain controllers, and gain unauthorized domain administrator access. Due to its critical severity, it was assigned a maximum CVSS score of 10.0.

To enhance the reliability and security of network protocols, it is essential to identify functional bugs in their implementations. Manually detecting such bugs is notoriously difficult, as developers have to compare the implementation against every relevant clause in lengthy RFC documents—a process that demands significant human effort and domain expertise. However, automating functional bug detection remains a challenging task. First, the semantic properties depicting protocol functionality are highly diverse and expressed informally in natural language within RFC documents. In contrast to generic low-level safety properties [3], which can often be uniformly represented using logical constraints, application-specific high-level semantic properties [4]–[7] (or *semantic properties* in this paper), such as those found in network protocols, are considerably more difficult to formalize due to their reliance on domain knowledge and context-dependent behaviors. As shown in the example in Figure 1(a), such properties in the RFC document often require reasoning about the states of multiple protocol entities and the complex interactions among them, making encoding such properties inherently intricate. Second, the diversity of semantic properties significantly hinders the development of effective detection mechanisms. Conventional bug detection techniques, such as constraint-based methods (e.g., symbolic execution [8]–[12]) and graph-based reasoning approaches (e.g., data-flow reachability analysis [13], [14]), are inadequate for automatically validating the semantic properties of network protocols. These methods cannot automatically localize the functions relevant to specific properties and verify their compliance with the specification. For instance, it is challenging to localize the function `route_lost` shown in Figure 1(b) and leverage a uniform reasoning framework to detect violations of the associated semantic property.

RFC 8966 3.8.2.1. Avoiding Starvation

If a Babel node loses all feasible routes to a destination but still has unexpired **unfeasible** routes, it **must** send a **sequence number request** (seqno request) to attempt route recovery.

(a) Babel specification

```
void route_lost(struct source *src, unsigned oldmetric) {
   new_route = find_best_route(src->prefix, src->plen, 1, NULL);
   if (new_route) { consider_route(new_route); }
   else {
-    if (oldmetric < INFINITY) {
+      unfeasible = find_best_route(src->prefix, src->plen, 0, NULL);
+    if (unfeasible && !route_expired(unfeasible))
+      send_request_resend(NULL, src->prefix, src->plen, …);
+    else if (oldmetric < INFINITY)
        send_update_resend(…);  ..
    } ..
```

(b) The bug fix in FRRouting

Fig. 1: An example functional bug in Babel and its fix

Although recent studies have proposed various techniques for bug detection in network protocol implementations, they remain inadequate to effectively identify functional bugs. In general, existing approaches fall into three categories. First, fuzzing techniques aim to uncover bugs by generating crash-

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00105

triggering inputs [15]–[17]. However, functional bugs typically do not cause crashes or other observable anomalies, making them difficult for fuzzing-based methods to detect. Second, differential analysis can reveal behavioral inconsistencies and discover potential functional bugs by comparing multiple independent implementations of the same protocol [18]–[20]. Yet, this approach depends on the availability of high-quality alternative implementations and does not guarantee conformance to the original specification, thereby limiting its recall for functional bug detection. Third, formal verification offers strong semantic guarantees but requires rigorous formalization of protocol properties and formal reasoning frameworks [21], [22]. These requirements may not be easily achievable for real-world protocols due to the informal and diverse specifications highlighted above. These limitations suggest the need for a new paradigm of bug detection that enables semantic reasoning on both informal specifications and implementation behavior, an essential capability for effectively detecting functional bugs.

This paper presents RFCAUDIT, the first LLM agent for detecting functional bugs in network protocol implementations. Our approach is motivated by the observation that LLMs exhibit strong capabilities in understanding both natural language specifications, such as RFC documents, and the semantics of code snippets when they are well scoped, such as those spanning just a few related functions. Notably, RFC documents are typically well-structured and articulate protocol specifications clearly in natural language. Meanwhile, protocol implementations often follow intuitive naming conventions that reflect the functionality of functions and modules. This suggests a promising opportunity: Given an informal property described in an RFC, we can leverage LLMs to align the natural language description of the property with the code structure. If the functions related to the property can be precisely identified by further code retrieval (e.g., along calling contexts), LLMs can perform comprehensive and accurate reasoning over these functions to discover potential functional inconsistencies, thereby enabling functional bug detection. Technically, RFCAUDIT consists of two collaborative agents, namely an ***indexing agent*** and a ***detection agent***, which perform the following core analyses, respectively.

- *Code Semantic Indexing.* To bridge informal properties in RFC documents with relevant code scope, the indexing agent constructs hierarchical semantic indexes of the protocol implementation using LLMs, which summarizes the semantics of functions, files, and directories into concise natural language descriptions. Such semantic indexes allow RFCAUDIT to narrow down the scope of functions relevant to the target property, significantly reducing token and time costs during the detection.
- *Retrieval-Guided Detection.* To collect sufficient contexts, the detection agent progressively retrieves additional relevant functions using symbolic tools driven by the LLM. If the currently available context suffices to validate or refute conformance to the semantic property, RFCAUDIT proceeds to the next property. Otherwise, it augments the context by retrieving callers or callees until a conclusion can be drawn. The retrieved functions can also support self-critics in the detection, during which the detection agent reviews its own reasoning steps upon the retrieved functions to mitigate hallucinations and improve the precision of the detection.

We implement RFCAUDIT as a prototype [23], powered by Claude 3.5 Sonnet. We apply RFCAUDIT to the implementations of six widely-used network protocols, including Babel, DHCP, and IGMP, and assess its ability to detect functional bugs according to the corresponding RFC documents. It is shown that RFCAUDIT successfully identifies 47 zero-day functional bugs with a high precision of 81.9%. In comparison, existing advanced agents, such as Copilot powered with three advanced models: Claude 3.7 Sonnet, Claude 3.5 Sonnet, and GPT-4o, only identify 26 bugs on average, while exhibiting a substantially higher average false positive rate of 77.5%. We further conduct comprehensive ablation studies, which highlight the significant precision gains attributed to the designs of code semantic indexing and retrieval-guided detection. In summary, our work makes the following contributions:

- We propose RFCAUDIT, the first LLM agent that checks the compliance of network protocol implementation and RFC documents for functional bug detection.
- We introduce a multi-agent design that summarizes code functionalities as semantic indexes and leverages them for demand-driven code retrieval, facilitating functional bug detection with high precision.
- We evaluate RFCAUDIT on six network protocol implementations and identify 47 functional bugs with 81.9% precision, outperforming existing LLM-based baselines in both precision and semantic coverage. 20 of the detected bugs have been confirmed or fixed by the respective developers.

II. OVERVIEW

In this section, we begin with a real-world functional bug in a network protocol implementation and discuss the limitations of existing approaches (Section II-A). Then we provide the key ideas of our approach (Section II-B) with illustrative examples.

A. Motivating Example

Figure 1 shows a real-world functional bug in FRRouting, a collaborative project under the Linux Foundation with over 400 contributors, powering major open-source large-scale networking platforms like Microsoft's SONiC and Amazon's DENT. According to the RFC document shown in Figure 1(a), when a node loses all feasible routes to a destination but still holds unexpired unfeasible routes, it MUST send a sequence number (seqno) request to trigger route recovery. An unfeasible route means a neighbor recently advertised a path that is not currently usable, but might become valid again. Sending a seqno request prompts that neighbor to re-announce the route, potentially restoring connectivity. However, the original implementation fails to check for unfeasible routes, which can leave the router in a starved state with no way to recover, causing persistent connectivity loss in dynamic networks. As shown in Figure 1(b), the fix adds an explicit

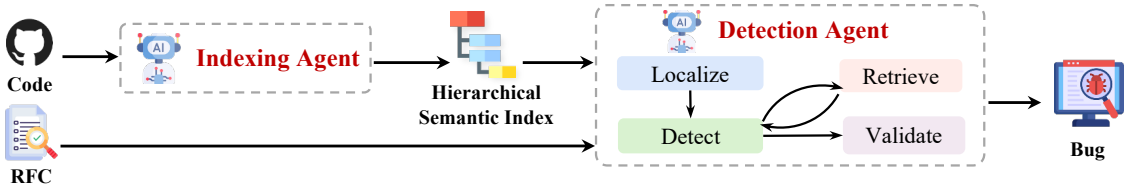


Fig. 2: The workflow of RFCAUDIT

check for unexpired unfeasible routes and issues a seqno request when such a route is found, ensuring RFC compliance and preventing persistent routing failure. This bug has been confirmed and fixed by the developers.

However, detecting functional bugs is extremely challenging [18], [24]. Traditional bug detection methods, such as fuzzing [15], [25], [26] and static analysis [27], [28], primarily focus on identifying low-level issues like memory corruption. Consequently, these approaches are ineffective against non-crashing protocol non-compliance, such as the bug illustrated in Figure 1. Model checking techniques also encounter significant difficulties in detecting this bug, primarily due to their dependence on explicitly specified and formally defined properties. For instance, tools such as CMC [22] and UPPAAL [29] efficiently verify high-level behaviors like state transitions and message sequencing, while struggling to capture diverse and nuanced semantic properties, exemplified in Figure 1(a). Moreover, even when semantic properties can be explicitly specified, model checking requires extensive manual abstraction efforts, which are often labor-intensive and error-prone, constraining their practical applicability. Lastly, recent studies such as ParDiff [18] and ParCleanse [30] target non-crashing parsing bugs by inferring valid packet formats from code or RFCs. Although effective in identifying parser-related issues, these approaches fall short in detecting functional bugs that involve more complex semantic properties in components beyond network packet parsers.

B. RFCAUDIT *in a Nutshell*

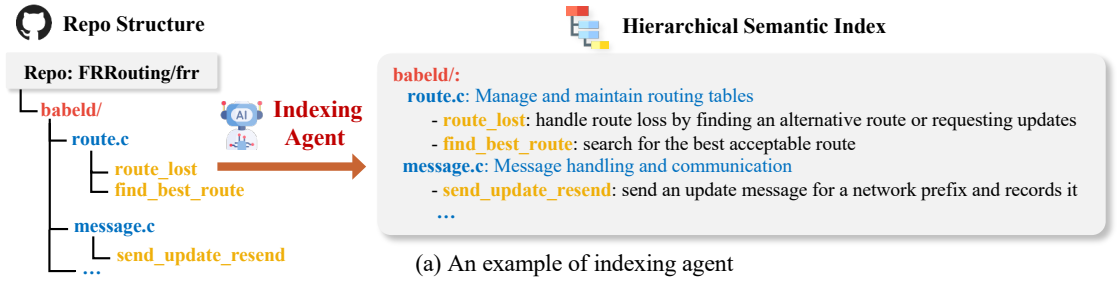
To bridge the gap, we propose RFCAUDIT, an LLM-powered autonomous agent that identifies semantic inconsistencies between source code and RFC documents. Our approach is motivated by the observation that LLMs, having been pre-trained on vast corpora of natural language and programming code, exhibit strong capabilities in understanding both the specification offered by the RFC document and program semantics in the code. For instance, given the semantic property shown in Figure 1(a) and the buggy implementation in Figure 1(b), LLMs can potentially identify the inconsistency between the RFC documents and the corresponding implementation logic. However, applying LLMs for functional bug detection in network protocols is far from trivial, requiring us to resolve two critical technical challenges.

- Due to the lack of direct correspondence between RFC segments and source code, identifying where a high-level property is realized in the implementation is quite difficult. For instance, checking the property in Figure 1(a) requires finding the related function `route_lost` in Figure 1(b), which involves exploring a large codebase without clear guidance from the RFC document.
- Even after locating a relevant function, LLMs may still struggle to determine whether the implementation satisfies the intended property. Since a semantic requirement is often implemented through multiple interconnected functions spread across files or modules, LLMs typically need additional contexts, such as the functions `find_best_route` and `send_update_resend` in Figure 1.

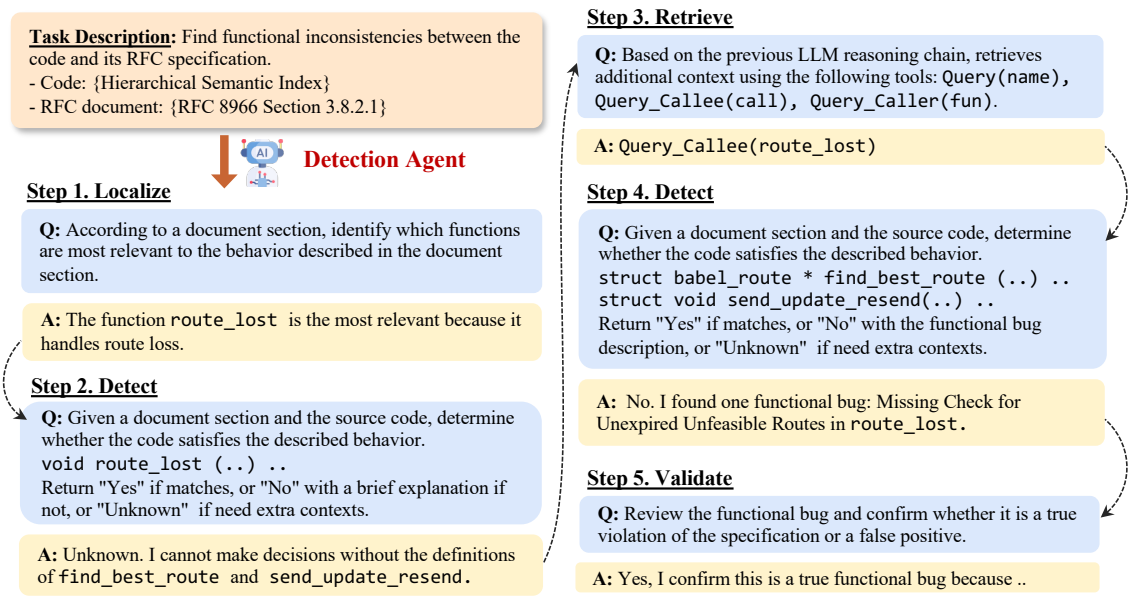
To tackle these challenges, we draw inspiration from the way human developers typically audit functional correctness against protocol specifications. When verifying whether a particular semantic property is correctly implemented, experienced developers first perform a preliminary walkthrough of the codebase to become familiar with the roles of different functions, files, and modules. They then progressively gather additional program constructs, such as related functions or data structures, to reason about the correctness of the implementation and identify potential violations.

RFCAUDIT mirrors this human auditing process through two key technical components: ***code semantic indexing*** and ***retrieval-guided detection***. The former creates a semantic map of the codebase by summarizing the functionality of functions, files, and directories in natural language, allowing fast and informed localization of potentially relevant code. The latter incrementally augments the reasoning context with related program constructs, enabling the system to validate a property when its implementation spans multiple functions or modules. These two techniques are realized through two collaborative agents in RFCAUDIT, the ***indexing agent*** and the ***detection agent***, whose respective workflows are illustrated in Figure 2.

- ***Indexing Agent.*** The indexing agent performs semantic summarization and generates *hierarchical semantic indexes* at the directory, file, and function levels. As shown in Figure 3(a), for the Babel protocol, files like `route.c` and `message.c`, as well as functions like `route_lost`, `find_best_route`, and `send_update_resend`, are annotated with concise summaries of their functionality. These semantic descriptions are later utilized by the detection agent to navigate and localize relevant code.
- ***Detection Agent.*** Given a semantic property, the detection agent uses hierarchical indexes to identify inconsistencies. As shown in Figure 3(b), it proceeds with four actions in five steps: (1) Use the semantic indexes to localize a relevant function, such as `route_lost`; (2) Detect inconsistencies based on this context; (3) Retrieve additional relevant functions, such as `find_best_route` and



(a) An example of indexing agent



(b) An example of detection agent

Fig. 3: Illustrative examples of the two agents in RFCAUDIT

send_update_resend, if the context is insufficient; (4) Draw a conclusion about a potential violation; and (5) Apply a self-critique strategy to reassess the reasoning for hallucination mitigation.

III. OUR APPROACH

RFCAUDIT emulates the reasoning process of human developers when auditing for functional bugs and consists of two collaborative agents, the indexing agent and the detection agent, which support scalable and effective functional bug detection upon network protocol implementations. In what follows, we present more technical details of the two agents with concrete prompts and illustrative examples.

A. Phase 1: Code Semantic Indexing

To bridge RFC documents with relevant code segments, we follow the behavior of human developers by constructing semantic indexes over the protocol implementation. This approach enables us to capture the semantics of the code in a manner analogous to the way developers form a high-level understanding of the implementation. Basically, a simple indexing design is to utilize the names of functions, files, and directories as summaries, as they often suggest the intended functionality [31], [32]. However, such names are not always informative enough to capture precise semantics. For example, in the Babel implementation shown in Figure 1, both babeld.c and babel_main.c reside in the same directory, yet their names do not clearly reveal their distinct functionalities. To generate more informative semantic indexes, we leverage LLMs to summarize the contents of functions, files, and directories, producing concise natural language descriptions that form a hierarchical semantic view of the implementation.

As shown in Figure 4, the indexing agent constructs hierarchical semantic indexes in a bottom-up manner. It begins at the *function level*, where each function definition is passed to the LLM with a prompt, which asks for a concise summary of its semantics. At the *file level*, summaries of all functions within the same source file are aggregated to prompt the LLM to generate a description of the file's overall functionality, capturing how individual behaviors contribute to the role of the file in the protocol implementation. This process continues at the *directory level*, where the agent combines summaries of all files and subdirectories to generate a higher-level description

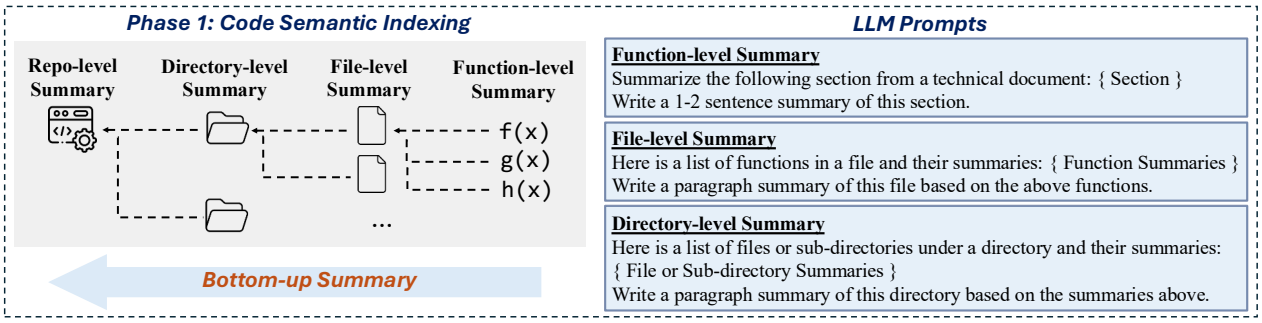


Fig. 4: The workflow of code semantic indexing and prompt templates

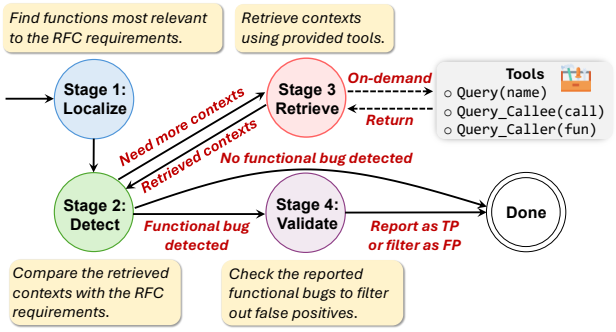


Fig. 5: The state machine of the detection agent

of the directory. Finally, summaries from all directories are aggregated into a repository-level summary, which is a top-level directory summary, reflecting the system's overall structure and responsibilities. This bottom-up indexing process produces a structured semantic index aligned with the code hierarchy, enabling precise and efficient retrieval during detection. The corresponding prompt templates for each indexing level are shown on the right of Figure 4. Notably, the code semantic indexing is one-time effort. When the codebase changes, we can incrementally update the indexes. In our evaluation, we choose the Claude 3.5 Sonnet model and the average cost for a full repository is $1.89, shown in Table V. We can also consider using cheaper models for indexing as the task is relatively simple and does not rely on strong reasoning ability.

Example 1. Consider the repository structure shown on the left of Figure 3(a). We first ask the LLMs to summarize each function. For example, `route_lost` handles route loss, `find_best_route` searches for the best acceptable route, and `send_update_resend` sends and records update messages. Next, the function-level summaries under the same file are combined to produce a file-level summary that captures the main responsibility of each file. In this case, we can discover that `route.c` focuses on route maintenance, and `message.c` on inter-node communication. Finally, all file summaries are merged into a directory-level summary for the `babeld` module. Lastly, we can obtain the hierarchical semantic indexes shown on the right of Figure 3(b).

B. Phase 2: Retrieval-guided Detection

As demonstrated in Section III-A, the hierarchical semantic indexes allow us to effectively bridge RFC documents with their corresponding implementation functions. Similar to how human developers audit code, the analysis often requires collecting additional program constructs, such as caller/callee functions and data structure definitions, to augment the context until a reliable conclusion can be drawn regarding the conformance or violation of a given semantic property. Building on this insight, we introduce the detection agent, which performs retrieval-guided detection to identify inconsistencies between the RFC specification and the code as functional bugs.

Technically, the retrieval-guided detection begins by preprocessing the RFC document and segmenting it into sections and subsections based on its structural headings. From each subsection, the LLMs extract mandatory semantic properties, which serve as guides for the detection process. A single subsection may yield multiple properties depending on its content. For each semantic property, the detection agent scans the implementation by following a workflow similar to a finite state machine, as illustrated in Figure 5. Specifically, it first localizes relevant functions for inspection (***Localization***), determines whether the implementation satisfies the RFC requirement (***Detection***), and, if necessary, retrieves additional context (***Retrieval***). When a potential functional bug is found, the detection agent examines the reasoning chain that produces the reported bug via self-criticism (***Validation***). In the following subsections, we provide further details on each component of the detection workflow shown in Figure 5.

Localization. Guided by the pre-built hierarchical semantic indexes, the detection agent first identifies functions that are likely responsible for implementing the behavior described in a given RFC section. This localization is performed in a top-down manner. Starting from the root directory of the protocol implementation, it navigates through different levels of directories and files by instantiating the prompt template shown in Figure 6(a). At each level, it reviews the semantic summaries of directories or files, selects those most relevant to the RFC section, and descends into the selected entries. Upon reaching a file, it examines the summaries of its functions and identifies those that align with the described behavior.

Localization Prompt
According to a document section, identify which entries are most relevant to the behavior described in the document section.
{ Document Section } { Entry Summaries }
Return a list of names: ["file.c", "subdir"] or ["fun1", "fun2"].

(a) Prompt used for Localization

Detection Prompt
Given a document section and the source code, determine whether the code satisfies the described behavior. { Document Section } { Code }
Return "Yes" if matches, or "No" with a brief explanation if not, or "Unknown " if need extra contexts.

(b) Prompt used for Detection

Retrieval Prompt
Based on the previous LLM reasoning chain, retrieves additional context using the following tools: `Query(name)`, `Query_Callee(call)`, `Query_Caller(fun)`.
{ LLM Reasoning Chain }

(c) Prompt used for Retrieval

Validation Prompt
Review the functional bug and confirm whether it is a true violation of the specification or a false positive.
{ Document Section } { Code } { Functional Bug }
If the functional bug is valid, confirm it. If not, explain why.

(d) Prompt used for Validation

Fig. 6: The prompt templates used in the detection agent

Because this search process is recursive, the final set of relevant functions may span multiple files across the codebase.

Detection. Based on the localized relevant functions, the detection agent attempts to identify inconsistencies between the implementation and the RFC. Using the template in Figure 6(b), it queries the LLM to make one of three decisions, as shown in Figure 5. Specifically, if the LLM detects a potential functional bug, the agent proceeds to validate it (see the *Validation* stage below). If the LLM concludes that the implementation conforms to the RFC requirement, the agent terminates the analysis for the current property. Otherwise, the current context is insufficient, the agent initiates an additional retrieval to expand the context (see the *Retrieval* stage below). Notably, the detection agent resembles the reasoning pattern of human developers, who iteratively gather evidence from the program constructs to identify potential bugs or to justify the correctness of an implementation.

Retrieval. To assist the LLM in identifying functional bugs or justifying correctness, the detection agent enables demand-driven retrieval of additional program constructs. This is achieved by equipping the LLM with a set of predefined tools, which it can invoke through function calls as needed. Specifically, the agent provides three types of tools. First, `Query(name)` retrieves the definition of a data structure and a macro. This is useful when the current context references symbols whose definitions are not yet available. Second, `Query_Callee(call)` returns the definition of the callee function invoked at the specified call site. This is used when the current function doesn't provide enough information to determine whether the behavior matches the RFC, and the LLM suspects that the called function contains important logic. This selective strategy enables the LLM to focus on semantically meaningful functions while avoiding unnecessary expansion of trivial or unrelated calls. Third, `Query_Caller(fun)` retrieves all callers of the function `fun`, allowing the LLM to examine how the function is used, particularly whether its preconditions are satisfied by the callers. These retrieval operations are triggered on demand, guided by the LLM's reasoning. Benefiting from model's planning ability, the LLMs can choose the proper tools for retrieval. The retrieved results further augment the analysis context in the *Detection* stage for continued analysis. The LLM prompt is shown in Figure 6(c).

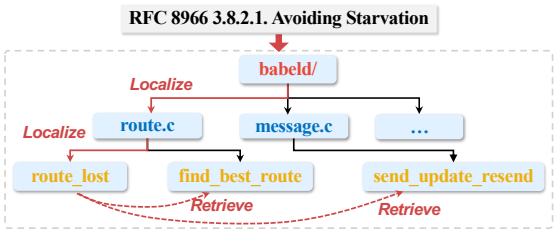


Fig. 7: An illustrative example of Phase 2

Validation. Due to the potential lengthy detection context, the LLMs may hallucinate and report a false positive. To mitigate hallucination, we employ self-critics [33] in the detection agent. Specifically, the detection agent reviews all available information, including the RFC section, the retrieved context, and earlier reasoning steps, to make a final judgment. If the violation is confirmed, the functional bug is reported as a true positive. Otherwise, it is discarded as a false positive. It may also uncover bugs missed during previous detection. The prompt is instantiated using the template in Figure 6(d). Our results in Section IV-E show that the self-critics can reduce the false positives by 71.7% (from 63.9% to 18.1%).

Example 2. Given the RFC section in Figure 1(a) and the semantic indexes in Figure 3(a), the detection agent identifies the relevant function `route_lost` in `route.c` (Figure 7). Finding its context insufficient, the agent retrieves `find_best_route` and `send_update_resend`, which select candidate routes and issue reactive updates, respectively. Notably, `send_update_resend` resides in a different file. With this broader context, the agent observes that only feasible routes are searched (`flag = 1`) and no sequence number request is made in the fallback logic, violating the starvation prevention rule. Retrieval stops here, and the bug is finally identified as shown in Figure 1. The whole process is detailed in Figure 3(b).

IV. Implementation and Evaluation

RFCAudit is implemented atop AutoGen [34], a multi-agent framework designed for building LLM applications. To extract functional specifications, we collect RFC documents

TABLE I: The statistics of evaluation subjects.

Name	Description	LoC	Version (#Page)
Babel	Distance-vector Routing Protocol	9.6K	RFC 8966 (54)
BFD	Bidirectional Forwarding Detection Protocol	17.3K	RFC 5880 (49)
NHRP	NBMA Next Hop Resolution Protocol	9.2K	RFC 2332 (52)
RIPng	Routing Information Protocol Next Generation	9.3K	RFC 2080 (19)
DHCP	Dynamic Host Configuration Protocol	8.6K	RFC 2131 (45)
IGMP	Internet Group Management Protocol	5.2K	RFC 2236 (24)

in plain text format and derive informal functional properties through structured segmentation of the textual content. We employ Claude 3.5 Sonnet [35] as the underlying LLM for RFCAUDIT, configured with a temperature of 0.0 to enforce greedy decoding. This setting reduces randomness in model responses, thereby enhancing the consistency and reproducibility of RFCAUDIT. To facilitate the indexing and detection agents, we implement analysis tools using Tree-sitter [36], a parsing library for different programming languages. In particular, we construct call graphs by leveraging function names and the numbers of their associated parameters or arguments. The output generated by the indexing agent is stored in a JSON file, enabling incremental analysis as the protocol implementation evolves. This design avoids redundant processing by skipping the analysis of unchanged functions, files, and modules.

To assess the performance of RFCAUDIT, we conduct experiments to address the following research questions:

- **RQ1**: How effective does RFCAUDIT identify functional bugs in real-world network protocol implementations?
- **RQ2**: How does RFCAUDIT compare to existing approaches?
- **RQ3**: What are the runtime and token costs of RFCAUDIT?
- **RQ4**: How do the two agents contribute to performance?

A. Dataset

We evaluate RFCAUDIT on six protocol implementations from two widely-used, open-source network stacks: FRRouting [37] and lwIP [38]. FRRouting (FRR) is an Internet routing protocol suite for Linux and Unix platforms, with over 3.6K stars on GitHub. It is deployed in production by major infrastructure providers such as NVIDIA and Orange, and is distributed through mainstream Linux repositories including Debian and Fedora. lwIP is a lightweight TCP/IP stack designed for resource-constrained environments, with over 1.3K stars on GitHub. It is widely integrated into embedded systems, IoT platforms, and real-time applications, and has been adopted by vendors like Intel, Xilinx, and Freescale. Overall, both FRRouting and lwIP are actively maintained and widely used in practice, making them ideal subjects for assessing RFCAUDIT across diverse deployment contexts. For each protocol, we collect the corresponding RFC that the implementation is based on as the reference. The protocol implementation sizes range from 5.2 KLoC to 17.3 KLoC, whereas the corresponding RFC documents span between 19 and 54 pages. The detailed information is shown in Table I.

B. RQ1: Effectiveness of RFCAUDIT

1) The Effectiveness of Bug Detection: To answer RQ1, we apply RFCAUDIT to each protocol implementation and its corresponding RFC to detect inconsistencies. For every reported issue, we manually verify whether it represents a real violation, and record the counts of true and false positives to compute precision. Since a single underlying issue may result in multiple reported bugs (e.g., when the same property is described across multiple sections of the RFC), we group relevant reports into unique bugs. For each unique bug, we check the project's issue tracker and pull requests to determine whether it is a previously unreported bug (i.e., new) or a known issue (i.e., old) that has not yet been fixed in the latest version.

TABLE II: The statistics of main results.

Protocols	#Incons.	TP	Precision	Unique Bug		#Total Bug
				#New	#Old	
Babel	16	12	75.0%	8	2	10
BFD	30	28	93.3%	9	5	14
NHRP	12	10	83.3%	10	0	10
RIPng	3	2	66.7%	2	0	2
DHCP	12	8	66.7%	4	0	4
IGMP	10	8	80.0%	7	0	7
Total	83	68	81.9%	40	7	47

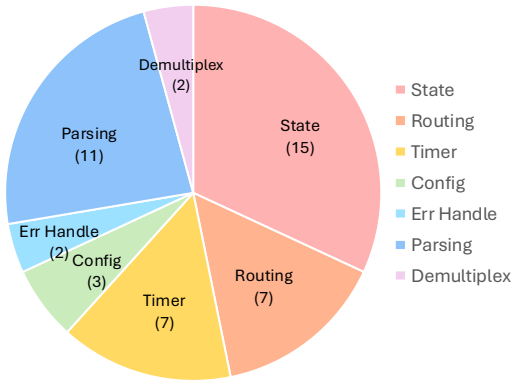


Fig. 8: Distribution of functional bug categories.

Results. As shown in Table II, RFCAUDIT generates a total of 83 bug reports across six protocols, of which 68 are true positives, yielding an overall precision of 81.9%. Grouping related bug reports, we uncovered 47 unique bugs, including 40 new bugs and 7 already reported ones. To date, 20 out of 47 unique bugs have been confirmed or their patches have been approved or merged by the developers. Overall, these results demonstrate that RFCAUDIT is effective in identifying bugs across diverse protocols. To better understand the diversity of functional bugs detected by RFCAUDIT, we categorize the 47 unique bugs into seven categories, namely *Parsing*, *Config*, *Routing*, *Err Handle*, *State*, *Timer*, and *Demultiplex*. As shown in Figure 8, most bugs fall under State (15/47) and Parsing (11/47), reflecting common pitfalls in protocol state management and input validation. RFCAUDIT also uncovered Timer misconfiguration (e.g., incorrect IGMP delays), Demultiplexing issues (e.g., incorrect BFD session lookup), and Err Handle gaps (e.g., miss error-on-error prevention). This variety highlights the ability of RFCAUDIT to detect diverse semantic violations specified by the RFCs. Full bug details are available in our artifact [23].

TABLE III: The precision, recall and F1 score of context retrieval. NA indicates that the divisor is 0.

Protocol	Properties	After *Localization*			After *Retrieval*			#Miss
		Function (%)	Type (%)	Macro (%)	Function (%)	Type (%)	Macro (%)	
Babel	15	50.0 / 91.7 / 64.7	NA / 0.0 / NA	NA / 0.0 / NA	50.0 / 91.7 / 64.7	100.0 / 100.0 / 100.0	100.0 / 100.0 / 100.0	1
BFD	22	57.1 / 66.7 / 61.5	NA / 0.0 / NA	NA / 0.0 / NA	72.7 / 100.0 / 84.2	66.7 / 100.0 / 80.0	100.0 / 100.0 / 100.0	0
NHRP	15	71.4 / 85.7 / 77.9	NA / 0.0 / NA	NA / 0.0 / NA	76.9 / 95.2 / 85.1	100.0 / 100.0 / 100.0	100.0 / 100.0 / 100.0	0
RIPng	24	60.7 / 70.8 / 65.4	NA / 0.0 / NA	NA / 0.0 / NA	37.3 / 100.0 / 54.3	100.0 / 33.3 / 50.0	NA / 0.0 / NA	0
DHCP	34	60.7 / 68.0 / 64.1	NA / NA / NA	NA / NA / NA	65.3 / 94.0 / 77.1	NA / NA / NA	NA / NA / NA	0
IGMP	21	52.9 / 75.0 / 62.0	NA / NA / NA	NA / 0.0 /	/ 100.0 /	NA / NA / NA	100.0 / 100.0 / 100.0	0
Total	131	59.3 / 71.3 / 65.6	NA / 0.0 / NA	NA / 0.0 / NA	56.0 / 95.1 / 70.5	87.5 / 77.8 / 82.4	100.0 / 75.0 / 100.0	1

2) The Effectiveness of Context Retrieval: To assess context retrieval accuracy, we manually construct ground-truth mappings for each RFC property. Each property is annotated with the minimal set of functions, types, and macros required to capture its semantics. Since this annotation is labor-intensive, we sample 30 RFC subsections (5 per protocol) from our datasets, covering a total of 131 properties across six protocols. For each property, we compare the elements retrieved by RFCAUDIT against the ground truth and measure precision, recall, and F1 score after two stages: (i) *localization* and (ii) *retrieval*. We then compute average recall across all sampled properties to assess how effectively RFCAUDIT collects the necessary context for inconsistency detection. Since both the retrieved element count and ground truth element count can be zero for types and macros, making recall undefined, we use "NA" to indicate these cases. Finally, we examine missed bugs and identify those caused by incomplete retrieval.

Results. Table III reports the context retrieval accuracy across protocols. In the *localization* stage, RFCAUDIT retrieves 172 functions, of which 102 are true positives, yielding 59.3% precision and 71.3% recall (102 of 143 ground-truth functions). This stage is limited to function retrieval and, by design, does not include types or macros. After the *retrieval* stage, precision is 56.0% for functions (136 of 243), 87.5% for types (7 of 8), and 100.0% for macros (7 of 7). More importantly, recall improves significantly—95.1% (136 of 143) for functions, 77.8% (7 of 9) for types, and 75.0% (6 of 8) for macros—with only one bug missed overall, caused by code misinterpretation rather than missing context. In this task, recall is more critical than precision. Low precision only means that unnecessary elements are retrieved, causing additional overhead, while low recall means the needed context is completely missed, preventing the detection of true inconsistencies. Note that achieving high precision is challenging for two reasons: (1) some functions are semantically close to the target property but not truly relevant, making them difficult to filter out, and (2) a single RFC property is often distributed across multiple functions or files, so aggressive filtering risks discarding essential information. For these reasons, our retrieval design explicitly prioritizes recall, even at the cost of including borderline or loosely related items. Precision-oriented refinements are left for future work.

3) False Positive Analysis: We analyze the 15 false positives reported by RFCAUDIT to understand their root causes. Based on this analysis, we classify them into four categories: *(i) incomplete context retrieval* (3 cases), where necessary function, type or macro definitions are not retrieved; *(ii) RFC misinterpretation* (6 cases), where RFCAUDIT mistakenly treats optional or incorrect properties as mandatory requirements; *(iii) code misinterpretation* (5 cases), where RFCAUDIT retrieves the correct context but fails to correctly interpret the code's control flow or semantics; and *(iv) intentional implementation choice* (1 case), where the observed behavior is a deliberate deviation from the RFC. The most common root cause is RFC misinterpretation, including 4 cases where the LLM extracts incorrect properties and 2 cases where the LLM treats optional clauses as mandatory. Such errors may be mitigated by fine-tuning the LLM on RFC texts to improve its ability to extract correct mandatory requirements.

C. RQ2: Baseline Comparison

We compare RFCAUDIT with GitHub Copilot and LTL-Fuzzer [4]. We do not compare with other network protocol bug detection tools, as they either cannot detect functional bugs (e.g., ChatAFL [15], NetLifter [26]), or target narrow scope like parsing (e.g., ParCleanse [30], ParVAL [39]).

1) Comparison with GitHub Copilot: We compare RFCAUDIT with GitHub Copilot using three LLMs: Claude 3.7 Sonnet (Thinking Mode), Claude 3.5 Sonnet, and GPT-4o. All are accessed via Copilot's Workspace Chat interface in VS Code, which supports repository-level queries. To match RFCAUDIT's design, we provide Copilot with one RFC section per prompt and ask it to identify deviations from implementations. For Claude 3.7 Sonnet, we manually examine all generated bug reports. However, Claude 3.5 Sonnet and GPT-4o produce hundreds of candidates, making full manual validation impractical. To avoid over-reporting and minimize false positives, we instruct each LLM baseline to only report bugs with 100% confidence. To enable a fair and scalable evaluation, we adopt a sampling-based evaluation. For each protocol, we randomly sample up to twice the number of bug reports identified by RFCAUDIT (e.g., since RFCAUDIT generates 16 bug reports for Babel, we sample 32). If the model produces fewer than that, we evaluate all of them. Each sampled case is manually reviewed by a domain expert.

Results. Table IV shows the performance of GitHub Copilot under three model configurations. With Claude 3.7 Sonnet (no sampling), it generated 215 bug reports, of which 78 were true positives (36.3% precision), uncovering 37 unique bugs (29 new, 8 old). Claude 3.5 Sonnet achieved 19.9% precision over 538 inconsistencies and found 26 bugs. GPT-4o returned the most bug reports (1,315) but the lowest precision

TABLE IV: Baseline comparison across three model configurations.

Protocols	Copilot + Claude 3.7 Sonnet					Copilot + Claude 3.5 Sonnet (With Sample)					Copilot + GPT-4o (With Sample)				
	#Incons.	#TP	Precision	#New	#Old	#Incons.	#Sample	Precision	#New	#Old	#Incons.	#Sample	Precision	#New	#Old
Babel	76	16	21.1%	7	3	173	32	12.5%	3	1	493	32	3.1%	1	0
BFD	45	37	82.2%	5	5	115	60	23.3%	7	4	244	60	20.0%	6	3
NHRP	27	10	37.0%	9	0	89	24	20.8%	4	0	204	24	16.7%	4	0
RIPng	6	1	16.7%	1	0	45	6	16.7%	1	0	84	6	0.0%	0	0
DHCP	32	9	28.1%	4	0	64	24	25.0%	4	0	188	24	42.0%	1	0
IGMP	29	5	17.2%	3	0	52	20	15.0%	2	0	102	20	5.0%	1	0
Total	215	78	**36.3%**	29	8	538	166	**19.9%**	21	5	1315	166	**11.4%**	13	3

TABLE V: Token usage (In: input tokens, Out: output tokens), financial cost, and execution time per protocol.

Protocol	Phase 1: Code Semantic Indexing			Phase 2: Retrieval-guided Detection			Total		
	Tokens (In/Out)	Cost ($)	Time (min)	Tokens (In/Out)	Cost ($)	Time (min)	Tokens (In/Out)	Cost ($)	Time (min)
Babel	261K / 71K	1.85	44	1340K / 41K	4.63	39	1061K / 112K	6.48	83
BFD	449K / 119K	3.13	71	1086K / 30K	3.71	52	1535K / 149K	6.84	123
NHRP	319K / 81K	2.17	49	960K / 25K	3.26	56	1279K / 106K	5.43	105
RIPng	197K / 52K	1.37	31	429K / 9K	1.42	23	626K / 61K	2.79	54
DHCP	248K /44K	1.40	28	1353K / 24K	4.42	47	1601K / 68K	5.82	75
IGMP	248K / 28K	1.40	28	390K / 16K	1.41	20	638K / 44K	2.81	48
Average	287K / 66K	1.89	42	926K / 24K	3.14	39	1123K / 90K	5.03	81

(11.4%) and the fewest bugs (16). Overall, all three baselines suffer from low precision and high validation overhead. In contrast, RFCAUDIT achieves 81.9% precision and discovers 47 bugs in total (40 new) with far less manual effort (Table II). RFCAUDIT consistently reports the lowest false-positive rates and the highest bug counts across all protocols. Notably, RFCAUDIT 's use of Claude 3.5 (a non-reasoning model) still outperforms Claude 3.7 (a reasoning model).

2) Comparison with LTL-Fuzzer: We compare RFCAUDIT with LTL-Fuzzer [4], which checks protocol implementations against LTL properties via instrumentation and fuzzing. Setting up LTL-Fuzzer requires substantial manual efforts: translating informal specifications into LTL formulas and mapping each atomic predicate to code locations. Consequently, it mainly captures event-ordering temporal properties but cannot naturally express data-format checks, arithmetic constraints, or timing requirements. We applied RFCAUDIT to the 15 zero-day bugs reported in LTL-Fuzzer's evaluation and also examined whether the 47 violated properties from our dataset could be encoded for LTL-Fuzzer properties.

Results. RFCAUDIT successfully detected 12 of the 15 zero-days originally reported by LTL-Fuzzer, without requiring manual property construction or predicate-to-location mapping. These included 7 in TinyDTLS and 5 in Contiki-Telnet. In contrast, when we attempted to encode the 47 violated properties from our dataset as LTL-Fuzzer properties, only 11 could be expressed; the remaining 36 required constraints beyond plain LTL or lacked precise predicate locations. These results indicate that RFCAUDIT is effective in automatically detecting temporal property violations while extending to a broader range of functional inconsistencies.

D. RQ3: Efficiency of RFCAUDIT

Setup and Metrics. To quantify the efficiency of RFCAUDIT, we measure input/output token usage, financial cost, and execution time for analyzing each repository. The results are reported separately for Phase 1 (Code Semantic Indexing) and Phase 2 (Retrieval-guided Detection). Although Claude 3.5 is used for both phases in our evaluation, Phase 1 primarily involves lightweight summarization. Since it does not require complex reasoning, it could be replaced by smaller or less expensive LLMs with minimal impact on performance.

Results. As shown in Table V, RFCAUDIT uses an average of 1123K input and 90K output tokens, costs $5.03, and completes each protocol analysis in 81 minutes. Since the analysis has a complete summarization of the whole code base and compares code against the full RFC, input token usage and cost largely depend on the code base size and RFC length. Overall, RFCAUDIT is efficient for functional bug detection.

E. RQ4: Ablation Studies

Ablation Study 1: Without Code Semantic Indexing. To evaluate the impact of code semantic indexing, we disable semantic summarization in Phase 1. The agent still follows the directory structure and explores the codebase hierarchically but no longer uses LLM-generated summaries. Instead, it relies solely on directory and file names plus function signatures. This setup follows the same design of AGENTLESS [32] and LocAgent [31], which uses code structures without semantic summaries. As shown in Table VI, removing semantic indexing drops precision from 81.9% to 51.4% and reduces detected bugs from 47 to 26. To explain this drop, we analyze each false positive and classify its root cause according to the four categories in Section IV-B3. We observe that false positives due to *(i) incomplete context retrieval* rise significantly (from 3 to 14), making it the main source of false positives in this setting. Without semantic summaries, the agent confuses similarly named helper functions across modules—for example, retrieving `parse_update_subtlv` instead of the correct `parse_packet` when validating Babel update TLVs. This highlights the importance of semantic summarization for accurate context retrieval and reduced false positives.

TABLE VI: Ablation Studies.

Protocol	Ablation Study 1			Ablation Study 2			Ablation Study 3		
	Precision	#New	#Old	Precision	#New	#Old	Precision	#New	#Old
Babel	52.6%	5	3	48.0%	9	3	35.2%	11	3
BFD	68.8%	8	5	70.6%	5	5	67.9%	10	5
NHRP	20.0%	2	0	45.0%	6	2	20.4%	7	0
RIPng	NA	0	0	25.0%	1	0	5.9%	1	0
DHCP	0%	0	0	50.0%	3	0	30.0%	4	0
IGMP	57.1%	3	0	18.2%	2	0	29.6%	4	0
Total	**51.4%**	18	8	**47.1%**	26	10	**36.1%**	37	8

TABLE VII: Impact of Removing Single Retrieval Tool.

w/o `Query`			w/o `Query_Callee`			w/o `Query_Caller`		
Precision	#New	#Old	Precision	#New	#Old	Precision	#New	#Old
50.0%	31	10	**47.7%**	37	8	**47.3%**	31	8

Ablation Study 2: Detection Without Retrieval. To evaluate the impact of the *Retrieval* stage, we disable all code retrieval tools (`Query`, `Query_Callee`, `Query_Caller`). The detection agent must make decisions using only the initially retrieved function(s), without fetching additional context. As shown in Table VI, this results in a sharp drop in precision (from 81.9% to 47.1%) and fewer detected bugs (from 47 to 36). This shows that retrieval is important for reducing false positives and improving detection quality. To assess the impact of each retrieval tool, we disable each tool individually. As shown in Table VII, removing any single tool sharply reduces precision from 81.9% with full retrieval to about 47–50%, with little difference across tools. However, the number of detected bugs varies, with the largest drop occurring when `Query_Caller` is removed (from 47 to 39), suggesting it contributes the most to overall bug coverage.

Ablation Study 3: Detection Without Self-critics. To evaluate the impact of self-critics, we disable it during detection. This leads to a higher false positive rate (18.1% to 63.9%) and fewer bugs (47 to 45), emphasizing its crucial role in enhancing detection precision and finding overlooked bugs.

F. Case Studies

We present two representative case studies:

Bug #1: Incorrect request forwarding. RFC 8966 specifies an ordered forwarding rule: when hop count is 2 or more, a node must first attempt to forward via a feasible route (excluding the requester); only if no such route exists should it fall back to a non-feasible one. As shown in Figure 9(a), FRRouting ignored this ordering and always chose a non-feasible route. This misbehavior could cause forwarding loops or prevent requests from reaching valid next hops, undermining routing convergence. We submitted a fix to prioritize feasible routes, which has been merged by the developers.

Bug #35: Miss loop detection. RFC 2332 specifies that if an NHS forwards an NHRP Resolution Reply that lists its own protocol address in the Responder Address Extension, it must discard the packet and generate an NHRP Error Indication of type "Loop Detected". As shown in Figure 9(b), FRRouting omitted this check, allowing persistent forwarding loops that consume resources and delay address resolution. The bug was confirmed by the developers.

```
void handle_request(struct neighbour *neigh, const unsigned char
*prefix, unsigned char plen, unsigned char hop_count, ..) {
   ..
   if (hop_count <= 1)  return;
-  other_route = find_best_route(prefix, plen, 0, neigh);
+  other_route = find_best_route(prefix, plen, 1, neigh);
+  if (!other_route || route_metric(other_route) >= INFINITY) {
+      /* If no feasible route found, try non-feasible routes */
+      other_route = find_best_route(prefix, plen, 0, neigh);
   }
```

(a) Bug #1: Incorrect Request Forwarding

```
static void nhrp_peer_forward(struct nhrp_peer *p,
                              struct nhrp_packet_parser *pp){
   type = htons(ext->type) & ~NHRP_EXTENSION_FLAG_COMPULSORY;
   switch (type) {
   case NHRP_EXTENSION_RESPONDER_ADDRESS:
+   if (hdr->type == NHRP_PACKET_RESOLUTION_REPLY) {
+     cie = nhrp_cie_pull(.., &cie_protocol);
+     if(cie && sockunion_same(&cie_protocol, &pp->if_ad->addr)){
+       nhrp_packet_send_error(pp, NHRP_ERROR_LOOP_DETECTED, 0);
+       goto err;
+     }
+   }
     ..
```

(b) Bug #35: Miss Loop Detection for Responder Address Extension

Fig. 9: Case Studies.

G. Threats to Validity.

Several threats may affect the validity of our findings. First, we adopt Claude 3.5 Sonnet as our language model with a temperature setting of 0.0 for stable analysis results, thereby enhancing the reproducibility of RFCAUDIT. Nevertheless, alternative models or configurations may yield variations in performance [40]–[42]. So we ran controlled experiments using two additional general-purpose API models: GPT-4o and DeepSeek-V3, to analyze the BFD protocol in FRRouting, where RFCAUDIT discovered the most bugs. RFCAUDIT detected 11 and 13 true bugs with the precision of 71.4% and 70.3%, respectively, demonstrating the effectiveness across models. Second, our manual classification of true positives and false positives may introduce subjective bias. To mitigate this risk, two authors independently reviewed each reported functional bug and resolved any disagreements through discussion [41], [43]. We further validated our results by reporting newly discovered bugs to protocol developers, where all bug reports that received a response have been confirmed as true positives. Third, our method relies on high-quality and well-structured documentation such as RFCs. Ambiguous or poorly maintained documentation might hinder semantic understanding, potentially threatening the effectiveness of our technique.

H. Evaluation on Industry-Scale Protocol Stacks

We further applied RFCAUDIT to two large protocol stacks: **aws-c-http**, a 74K LoC HTTP library in the AWS Common Runtime, and the **Linux TCP/IP stack**, about 190K LoC in the kernel implementing core transport and network protocols. We evaluated aws-c-http against RFC 7230 (HTTP/1.1), and Linux against RFC 793 (TCP), RFC 768 (UDP), and RFC 791 (IP). On aws-c-http, RFCAUDIT detected 18 inconsistencies, of which 12 were confirmed as true positives, yielding 12 unique bugs. On the Linux TCP/IP stack, it flagged 3 incon-

sistencies, with 1 true positive. These results demonstrate that RFCAUDIT scales to industry-scale codebases and can expose real specification violations with high precision.

V. DISCUSSION AND FUTURE WORK

While our current evaluation targets network protocols, RFCAudit is not limited to this domain. The key requirement is the availability of specification documents that define expected functionality and constraints. In domains such as system libraries, cloud APIs, or security-critical frameworks, specifications are often available as API references, design guidelines, or standards documents. RFCAUDIT can be adapted to align such specifications with code and flag inconsistencies as potential bugs. We choose network protocols for evaluation because RFCs are detailed and publicly accessible, but extending RFCAudit to other domains is a promising next step to further demonstrate its generality and effectiveness.

VI. RELATED WORK

A. Bug Detection in Network Protocol Implementation

The correctness of network protocol implementations is critical for ensuring secure and reliable digital communication. Existing bug detection techniques for protocol implementations can be broadly categorized into three approaches. First, fuzzing-based techniques—such as BooFuzz [44] and SAGE [45]—primarily identify bugs by triggering crashes during execution. Netlifter [26] and ChatAFL [15] improve fuzzing coverage by using static analysis or LLMs to infer packet formats and generate valid inputs, but still rely on crashes, overlooking subtle semantic issues. LLMIF [46] and mGPTFuzz [16] extract formats or FSMs from specifications using LLMs to uncover semantic bugs, but focus narrowly on state transition or input validation violations and are limited to specific IoT devices, hard to generalize to broader protocol testing. LTL-Fuzzer [4] uses LTL formulas to guide fuzzing and detect event-ordering violations, but demands significant manual effort for formula crafting and predicate-to-code mapping, and is limited to temporal property violations. Second, differential analysis techniques [18]–[20] detect bugs by comparing the behaviors of multiple independent implementations of the same protocol. These approaches can uncover semantic discrepancies that fuzzing may miss. However, they are inherently limited to scenarios where multiple protocol implementations are available for comparison. Third, formal verification-based techniques rigorously check protocol implementations against formally specified properties. While powerful in reasoning about semantic correctness, they require substantial manual effort to construct, maintain, and validate formal models, posing significant barriers to practical deployment. Beyond the three categories, EBugDec [47] targets inconsistencies between RFC evolution and protocol implementations. However, its scope is limited to RFC-evolutionary bugs, primarily focusing on packet parsing issues. In contrast to these prior efforts, our proposed technique, RFCAUDIT, addresses functional correctness by bridging the informal specifications in RFC documents with the program semantics of the protocol implementation, facilitating the functional bug detection with high precision and efficiency.

B. LLM-aided Static Bug Detection

LLM-aided static bug detection has advanced rapidly [48]–[54], typically following two directions. One uses LLMs to supply domain knowledge like function specifications [48]–[50] and bug definitions [51], to assist symbolic approaches in analyzing the program for bug detection. For example, IRIS leverages LLM-inferred sources and sinks to detect taint-style vulnerabilities [50], while KNighter synthesizes static analyzers from bug patches for kernel bug detection [51]. The other direction adopts agent-centric solutions that reason over source code directly without compilation [52]–[54]. Techniques such as LLMSAN [52] and LLMDFA [53] combine LLMs with SMT solvers and parsing-based analyzers, while LLift uses progressive prompting to find kernel vulnerabilities [54]. More recently, RepoAudit extends these efforts to detect multiple types of data-flow bugs [55]. In contrast, our work targets functional bugs, where bug specifications often lack clear formalization. The diverse nature of functional properties also prevents the application of standard static analysis frameworks. Nonetheless, our retrieval-based approach draws inspiration from traditional bug detection methodologies, enabling localizing code segments relevant to targeted functional properties, facilitating effective subsequent detection.

VII. CONCLUSION

This paper presents RFCAUDIT, an LLM agent designed for detecting functional bugs in network protocol implementations by identifying semantic inconsistencies between the code and RFC documents. Specifically, RFCAUDIT comprises two complementary agents: an indexing agent responsible for semantic indexing of source code, and a detection agent performing retrieval-guided inconsistency detection. In our evaluation, RFCAUDIT successfully uncovered 47 functional bugs across six different protocols, of which 20 have been acknowledged or fixed by the original developers. As the first LLM-based approach explicitly targeting functional bug detection in network protocols, our work offers valuable insights into future research on functional bug detection in domain-specific software systems, demonstrating the significant potential of leveraging LLM capabilities in security auditing tasks.

ACKNOWLEDGMENT

We thank all the anonymous reviewers for the insightful and constructive feedback. We are grateful to the Center for AI Safety for providing computational resources. This work was funded in part by the National Science Foundation (NSF) Awards SHF-1901242, SHF-1910300, Proto-OKN 2333736, IIS-2416835, DARPA VSPELLS - HR001120S0058, ONR N00014-23-1-2081, and Amazon. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] MITRE, "Cve-2020-1472," https://nvd.nist.gov/vuln/detail/CVE-2020-1472, 2020.

[2] Microsoft, "Netlogon remote protocol," https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nrpc/ff8f970f-3e37-40f7-bd4b-af7336e4792f, 2024.

[3] A. A. de Amorim, C. Hritcu, and B. C. Pierce, "The meaning of memory safety," *CoRR*, vol. abs/1705.07354, 2017. [Online]. Available: http://arxiv.org/abs/1705.07354

[4] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, "Linear-time temporal logic guided greybox fuzzing," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1343–1355. [Online]. Available: https://doi.org/10.1145/3510003.3510082

[5] S. Feng, Y. Ye, Q. Shi, Z. Cheng, X. Xu, S. Cheng, H. Choi, and X. Zhang, "ROCAS: root cause analysis of autonomous driving accidents via cyber-physical co-mutation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 1620–1632. [Online]. Available: https://doi.org/10.1145/3691620.3695530

[6] S. Feng, X. Xu, X. Chen, K. Zhang, S. Y. Ahmed, Z. Su, M. Zheng, and X. Zhang, "Intentest: Stress testing for intent integrity in api-calling LLM agents," *CoRR*, vol. abs/2506.07524, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2506.07524

[7] S. Kate, Y. Gao, S. Feng, and X. Zhang, "Roscallbax: Statically detecting inconsistencies in callback function setup of robotic systems," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, pp. 668–689, 2025. [Online]. Available: https://doi.org/10.1145/3715748

[8] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '08. USENIX, 2008, pp. 209–224. [Online]. Available: https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems

[9] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[10] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[11] J. Jiang, M. Zheng, Q. Shi, and X. Z. Zhang, "SFA-Miner: Mining path-sensitive api usage patterns via symbolic finite automata," in *The IEEE Symposium on Security and Privacy*, ser. S&P 2026, 2026.

[12] X. Liu, W. You, Z. Zhang, and X. Zhang, "Tensilefuzz: facilitating seed input generation in fuzzing via string constraint solving," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 391–403. [Online]. Available: https://doi.org/10.1145/3533767.3534403

[13] P. Yao, J. Zhou, X. Xiao, Q. Shi, R. Wu, and C. Zhang, "Falcon: A fused approach to path-sensitive sparse data dependence analysis," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3656400

[14] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 327–337. [Online]. Available: https://doi.org/10.1145/3180155.3180178

[15] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/large-language-model-guided-protocol-fuzzing/

[16] X. Ma, L. Luo, and Q. Zeng, "From one thousand pages of specification to unveiling hidden bugs: Large language model assisted fuzzing of matter iot devices," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/ma-xiaoyue

[17] X. Liu, W. You, Y. Ye, Z. Zhang, J. Huang, and X. Zhang, "Fuzzinmem: Fuzzing programs via in-memory structures," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639172

[18] M. Zheng, Q. Shi, X. Liu, X. Xu, L. Yu, C. Liu, G. Wei, and X. Zhang, "Pardiff: Practical static differential analysis of network protocol parsers," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, pp. 1208–1234, 2024. [Online]. Available: https://doi.org/10.1145/3649854

[19] R. Rutledge and A. Orso, "Automating differential testing with overapproximate symbolic execution," in *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*. IEEE, 2022, pp. 256–266. [Online]. Available: https://doi.org/10.1109/ICST53961.2022.00035

[20] G. S. Reen and C. Rossow, "Dpifuzz: A differential fuzzing framework to detect DPI elusion strategies for QUIC," in *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*. ACM, 2020, pp. 332–344. [Online]. Available: https://doi.org/10.1145/3427228.3427662

[21] O. Udrea and C. Lumezanu, "Rule-based static analysis of network protocol implementations," in *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*, A. D. Keromytis, Ed. USENIX Association, 2006. [Online]. Available: https://www.usenix.org/conference/15th-usenix-security-symposium/rule-based-static-analysis-network-protocol

[22] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings*, R. Morris and S. Savage, Eds. USENIX, 2004, pp. 155–168. [Online]. Available: http://www.usenix.org/events/nsdi04/tech/musuvathi.html

[23] "The artifact of RFCAudit," https://github.com/zmw12306/RFCAudit, 2025.

[24] C. Liu, S. Gong, and P. Fonseca, "KIT: testing os-level virtualization for functional interference bugs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds. ACM, 2023, pp. 427–441. [Online]. Available: https://doi.org/10.1145/3575693.3575731

[25] V. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 460–465. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00062

[26] Q. Shi, J. Shao, Y. Ye, M. Zheng, and X. Zhang, "Lifting network protocol implementation to precise format specification with security applications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 1287–1301. [Online]. Available: https://doi.org/10.1145/3576915.3616614

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

[28] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, A. Zaks and M. V. Hermenegildo, Eds. ACM, 2016, pp. 265–266. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[29] G. Díaz, F. Cuartero, V. V. Ruiz, and F. L. Pelayo, "Automatic verification of the TLS handshake protocol," in *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus, March 14-17, 2004*, H. Haddad, A. Omicini, R. L. Wainwright, and L. M. Liebrock, Eds. ACM, 2004, pp. 789–794. [Online]. Available: https://doi.org/10.1145/967900.968063

[30] M. Zheng, D. Xie, Q. Shi, C. Wang, and X. Zhang, "Validating network protocol parsers with traceable RFC document interpretation," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, pp. 1772–1794, 2025. [Online]. Available: https://doi.org/10.1145/3728955

[31] Z. Chen, R. Tang, G. Deng, F. Wu, J. Wu, Z. Jiang, V. K. Prasanna, A. Cohan, and X. Wang, "Locagent: Graph-guided LLM agents for code localization," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Association for Computational Linguistics, 2025, pp. 8697–8727. [Online]. Available: https://aclanthology.org/2025.acl-long.426/

[32] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Demystifying llm-based software engineering agents," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, pp. 801–824, 2025. [Online]. Available: https://doi.org/10.1145/3715754

[33] Z. Lin, Z. Gou, T. Liang, R. Luo, H. Liu, and Y. Yang, "CriticBench: Benchmarking LLMs for critique-correct reasoning," in *Findings of the Association for Computational Linguistics: ACL 2024*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 1552–1587. [Online]. Available: https://aclanthology.org/2024.findings-acl.91/

[34] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "Autogen: Enabling next-gen LLM applications via multi-agent conversation," in *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024. [Online]. Available: https://openreview.net/forum?id=uAjxFFing2

[35] Anthropic, "Claude 3.5 sonnet," https://www.anthropic.com/claude/sonnet, 2025.

[36] "Tree-sitter," https://tree-sitter.github.io/tree-sitter/.

[37] F. community, "The frrouting protocol suite," https://github.com/FRRouting/frr, 2024.

[38] "lwip - a lightweight tcp/ip stack," https://github.com/lwip-tcpip/lwip, 2025.

[39] M. Zheng, D. Xie, and X. Zhang, "Large language models for validating network protocol parsers," in *2025 IEEE Security and Privacy, SP 2025 - Workshops, San Francisco, CA, USA, May 15, 2025*, M. Blanton, W. Enck, and C. Nita-Rotaru, Eds. IEEE, 2025, pp. 56–64. [Online]. Available: https://doi.org/10.1109/SPW67851.2025.00009

[40] X. Deng, J. Da, E. Pan, Y. Y. He, C. Ide, K. Garg, N. Lauffer, A. Park, N. Pasari, C. Rane *et al.*, "Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?" *arXiv preprint arXiv:2509.16941*, 2025. [Online]. Available: https://arxiv.org/abs/2509.16941

[41] D. Xie, B. Yoo, N. Jiang, M. Kim, L. Tan, X. Zhang, and J. S. Lee, "Impact of large language models on generating software specifications," *CoRR*, vol. abs/2306.03324, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.03324

[42] D. Xie, M. Zheng, X. Liu, J. Wang, C. Wang, L. Tan, and X. Zhang, "Core: Benchmarking LLMs' code reasoning capabilities through static analysis tasks," in *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. [Online]. Available: https://openreview.net/forum?id=WJIDorHiuZ

[43] M. Zheng, J. Yang, M. Wen, H. Zhu, Y. Liu, and H. Jin, "Why do developers remove lambda expressions in java?" in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 67–78. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678600

[44] J. Pereyda, "Boofuzz," https://github.com/jtpereyda/boofuzz, 2023.

[45] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012. [Online]. Available: https://doi.org/10.1145/2093548.2093564

[46] J. Wang, L. Yu, and X. Luo, "LLMIF: augmented large language model for fuzzing iot devices," in *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2024, pp. 881–896. [Online]. Available: https://doi.org/10.1109/SP54263.2024.00211

[47] J. Chen, F. Li, Q. Chen, P. Li, L. Xu, and W. Huo, "Ebugdec: Detecting inconsistency bugs caused by RFC evolution in protocol implementations," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2023, Hong Kong, China, October 16-18, 2023*. ACM, 2023, pp. 412–425. [Online]. Available: https://doi.org/10.1145/3607199.3607222

[48] C. Wang, J. Zhang, R. Wu, and C. Zhang, "Dainfer: Inferring API aliasing specifications from library documentation via neurosymbolic optimization," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 2469–2492, 2024. [Online]. Available: https://doi.org/10.1145/3660816

[49] C. Ye, Y. Cai, and C. Zhang, "When threads meet interrupts: Effective static detection of interrupt-based deadlocks in linux," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/ye

[50] Z. Li, S. Dutta, and M. Naik, "Llm-assisted static analysis for detecting security vulnerabilities," *CoRR*, vol. abs/2405.17238, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2405.17238

[51] C. Yang, Z. Zhao, Z. Xie, H. Li, and L. Zhang, "Knighter: Transforming static analysis with llm-synthesized checkers," *CoRR*, vol. abs/2503.09002, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2503.09002

[52] C. Wang, W. Zhang, Z. Su, X. Xu, and X. Zhang, "Sanitizing large language models in bug detection with data-flow," in *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, Y. Al-Onaizan, M. Bansal, and Y. Chen, Eds. Association for Computational Linguistics, 2024, pp. 3790–3805. [Online]. Available: https://aclanthology.org/2024.findings-emnlp.217

[53] C. Wang, W. Zhang, Z. Su, X. Xu, X. Xie, and X. Zhang, "LLMDFA: analyzing dataflow in code with large language models," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/ed9dcde1eb9c597f68c1d375bbecf3fc-Abstract-Conference.html

[54] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3649828

[55] J. Guo, C. Wang, X. Xu, Z. Su, and X. Zhang, "Repoaudit: An autonomous llm-agent for repository-level code auditing," *CoRR*, vol. abs/2501.18160, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2501.18160