Large Language Models for Validating Network Protocol Parsers

Mingwei Zheng
Department of Computer Science
Purdue University
West Lafayette, USA
zheng618@purdue.edu

Danning Xie
Department of Computer Science
Purdue University
West Lafayette, USA
xie342@purdue.edu

Xiangyu Zhang
Department of Computer Science
Purdue University
West Lafayette, USA
xyzhang@purdue.edu

***Abstract*—Network protocol parsers are essential for enabling correct and secure communication between devices. Bugs in these parsers can introduce critical vulnerabilities, including memory corruption, information leakage, and denial-of-service attacks. An intuitive way to assess parser correctness is to compare the implementation with its official protocol standard. However, this comparison is challenging because protocol standards are typically written in natural language, whereas implementations are in source code. Existing methods like model checking, fuzzing, and differential testing have been used to find parsing bugs, but they either require significant manual effort or ignore the protocol standards, limiting their ability to detect semantic violations. To enable more automated validation of parser implementations against protocol standards, we propose PARVAL, a multi-agent framework built on large language models (LLMs). PARVAL leverages the capabilities of LLMs to understand both natural language and code. It transforms both protocol standards and their implementations into a unified intermediate representation, referred to as format specifications, and performs a differential comparison to uncover inconsistencies. We evaluate PARVAL on the Bidirectional Forwarding Detection (BFD) protocol. Our experiments demonstrate that PARVAL successfully identifies inconsistencies between the implementation and its RFC standard, achieving a low false positive rate of 5.6%. PARVAL uncovers seven unique bugs, including five previously unknown issues.**

***Index Terms*—Network protocol parsing, format specifications, large language models**

1. Introduction

Network protocols specify the rules that control communication between devices, for example, TLS [1] for secure communication, FTP [2] for file transfers, and IP [3] for routing data. Network protocol parsing is a critical component of protocol implementations, responsible for analyzing incoming packets, validating their correctness, and discarding malformed ones. This process protects data integrity and enhances system security by preventing the processing of potentially harmful packets.

Despite their critical role, network protocol parsers are error-prone, which can lead to severe security vulnerabilities, including memory corruption, information leakage, and denial-of-service attacks. For instance, the Heartbleed [4] vulnerability in OpenSSL [5]'s TLS parser allowed attackers to access sensitive user information due to a missing bounds check. Similarly, CVE-2021-41773 [6] in Apache's HTTP server exposed confidential files through a path traversal flaw in the request parser.

Validating the correctness of network protocol parsers is inherently challenging. A natural approach is to compare the parser implementation against its official protocol standard to identify inconsistencies. However, this process is complicated by a fundamental representational gap: protocol standards are typically written in natural language (e.g., RFCs), while implementations are written in source code. Although both aim to specify the same behavior, their different representations hinder direct comparison, making systematic validation a significant challenge.

Various techniques have been proposed to address this challenge. Model checking [7]–[9] verifies parser behavior against formal protocol models to detect logical errors. While effective, it often requires manual construction of formal models, making it impractical for applying to large and frequently evolving protocols. Fuzzing [10], [11] automatically produces large volumes of test inputs to expose memory-related issues, such as crashes. However, it primarily targets low-level bugs and fails to assess whether the parser conforms to the protocol's intended semantics, leaving many semantic bugs [12]–[15] undetected. Differential analysis [12], [16] identifies inconsistencies by comparing how different implementations handle the same input. This approach assumes that at least one implementation behaves correctly and may miss bugs shared across all versions. While these methods are effective in certain situations, they fall short of providing an efficient way to validate parser implementations against official protocol standards written in natural language.

Our Approach. We propose PARVAL, an LLM-based multi-agent framework for validating parser implementations against their official protocol standards. PARVAL uses LLMs to automatically abstract both the natural language documentation and the parser implementation into a unified intermediate representation, called format specifications. These specifications capture the expected packet structure

and parsing logic, enabling direct comparison between the protocol documentation and its implementation.

LLMs have demonstrated strong capabilities in extracting format specifications from natural language [15], [17], but applying them to source code remains challenging. In real-world codebases, parsing logic is often scattered across multiple files and functions [11], [12], making it difficult to gather the full context needed for accurate format extraction. Moreover, LLMs are prone to generating incorrect or incomplete outputs (hallucinations), so validating and refining the extracted formats is essential. This typically relies on unit tests to check whether the extracted format matches the implementation behavior. However, without a well-isolated parsing module, generating and executing such tests is difficult. Consequently, errors in the extracted format specifications may go unnoticed, undermining the reliability of comparing the format specifications extracted from the parser implementation and the protocol standard.

To tackle the challenge of retrieving parsing-relevant context from large codebases, PARVAL introduces a *Retrieval-Augmented Program Analysis Agent* to automatically extract code segments directly related to protocol parsing while excluding irrelevant ones. This agent uses a *dependency-aware retrieval mechanism*, beginning from the entry parsing function and recursively using LLMs to analyze data dependencies, control flow, and function calls. It incrementally collects parsing relevant segments until the complete parsing logic is isolated.

To reduce LLM hallucinations in LLM-generated format specifications, PARVAL isolates the extracted parsing logic into a standalone, executable module. This module takes a buffer and its length as input and returns a boolean value indicating whether parsing succeeds. PARVAL then extracts a format specification from this module and generates unit tests to check whether the module's behavior matches the extracted format specification. Any discrepancies identified through testing are used as feedback to iteratively refine the format, improving its accuracy over time.

Contributions. This paper explores the potential of LLMs to validate parser implementations against natural language protocol standards. While our approach does not guarantee soundness or completeness, our evaluation on a real-world protocol demonstrates its practical effectiveness, providing a strong starting point for further investigation. In summary, our main contributions are:

- We introduce PARVAL, a multi-agent framework that uses LLMs to extract parsing behaviors from both source code and protocol standards. These behaviors are represented in a unified intermediate representation, called format specifications, to enable direct comparison.
- PARVAL introduces a *dependency-aware retrieval mechanism* that automatically identifies and extracts relevant parsing logic by analyzing data flow, control flow, and function dependencies throughout the codebase.
- PARVAL isolates parsing logic into a standalone, executable module, enabling test case generation to validate and refine the extracted format specifications. This helps improve the accuracy of format specification extraction and mitigates the impact of LLM hallucination. We compare the extracted format specification against a manually annotated ground truth and find that PARVAL successfully extract the full parsing logic.
- We implement the proposed approach in PARVAL[1]. We have evaluated PARVAL on the Bidirectional Forwarding Detection (BFD) protocol to validate its parser implementation against the corresponding RFC standard. PARVAL successfully identifies seven unique bugs, including five previously undiscovered.

```
1  void bfd_recv_cb (struct thread *t)
2  {
3          …
4        if (sd == bvrf->bg_shop || sd == bvrf->bg_mhop) {
5           mlen = bfd_recv_ipv4(sd, msgbuf, sizeof(msgbuf), …);
6        } else if (sd == bvrf->bg_shop6 || sd == bvrf->bg_mhop6) {
7           mlen = bfd_recv_ipv6(sd, msgbuf, sizeof(msgbuf), …);
8        }
9
10       if (mlen < BFD_PKT_LEN) {
11          cp_debug("too small (%ld bytes)", mlen);
12          return;
13       }
14       /* Parse the control header for inconsistencies:
15       struct bfd_pkt *cp = (struct bfd_pkt *)(msgbuf);
16       if (BFD_GETVER(cp->diag) != BFD_VERSION) {
17          cp_debug("bad version %d", BFD_GETVER(cp->diag));
18          return;
19       }
20          …
21 }
```

Figure 1: Entry Parsing Function in BFD Implementation

Organization. Section 2 presents a motivation example to illustrate the challenges and our solutions. Section 3 details our design. Section 4 evaluates the effectiveness of our tool. Section 5 discusses limitations and future work. Section 6 reviews related work. Section 7 concludes the paper.

2. Motivating Example

Figure 1 shows a simplified version of the entry parsing function in the BFD implementation. The function `bfd_recv_cb` handles incoming BFD packets. It begins by determining the socket type (`sd`) and invokes either `bfd_recv_ipv4` or `bfd_recv_ipv6` to read the message into `msgbuf` (lines 4 - 8). It then validates the message length `mlen` (lines 10 - 13) and proceeds to parse the control header (lines 14 - 20).

To validate the parser implementation, we leverage large language models (LLMs) to extract format specifications from the source code and the RFC document. By translating both into a unified representation, we enable direct comparison and systematic validation. However, accurately extracting protocol formats from source code presents two key challenges.

Challenge 1: Retrieving Parsing-Related Context. LLMs require complete parsing contexts to accurately extract

1. PARVAL is available at: https://github.com/zmw12306/PARVAL

protocol formats. However, in real-world codebases, parsing-relevant logic is often spread across multiple functions, files, and directories, making it challenging to automatically identify and extract all relevant code. For example, in Figure 1, `struct bfd_pkt` (line 15) and `BFD_GETVER` (line 16) are defined in separate files. Without retrieving these dependencies, LLMs cannot fully interpret the parsing logic. Since feeding the entire codebase exceeds the context window of LLMs, a selective context retrieval strategy is necessary.

Our Solution: We design a *dependency-aware retrieval mechanism* that guides LLMs to selectively collect parsing-relevant code. Starting from the entry parsing function, it traces input buffer usage through data and control dependencies, as well as function calls, to collect essential context such as structs, macros, and helper functions. This dependency analysis guides LLMs to retrieve only the necessary context to accurately extract protocol formats.

Challenge 2: LLM Hallucinations in Extracted Formats. LLMs may generate inaccurate format specifications that deviate from actual parser behavior, such as involving incorrect field types, lengths, or ordering. To correct these hallucinations, it is essential to test the consistency between the extracted format specification and the original parser, using the test results to guide iterative refinement. However, this feedback loop becomes unreliable when the parsing logic is tightly coupled with other parsing unrelated operations. For instance, in Figure 1, the function `bfd_recv_cb` expects a `thread` structure containing socket state and runtime configuration. Before any parsing occurs, the function performs socket-related checks. In contrast, test inputs generated from the extracted format consist solely of raw packet data and cannot be passed directly to `bfd_recv_cb`. To run a test, each packet data must be wrapped in a correctly initialized `thread` structure. This additional setup introduces complexity and obscures whether test failures stem from the format itself, incorrect `thread` initialization, or buggy socket handling. This undermines the reliability of test-based feedback, making it difficult to detect and correct LLM hallucinations in the extracted format.

Our Solution: To reduce LLM hallucinations, PARVAL extracts the parsing logic into a standalone, executable module that focuses only on packet parsing. This module takes a raw buffer and its length as input and returns a boolean indicating whether parsing succeeds. Valid and invalid test packets generated from the extracted format specification are directly passed to this module. If the module rejects valid packets or accepts invalid ones, this indicates a mismatch between the LLM extracted format specification and the actual parsing behavior. Such mismatches are used to iteratively refine the extracted format, improving its accuracy.

3. Approach

Figure 2 presents an overview of our approach, which consists of four stages. In **Stage 1 Parser Isolation** (Section 3.1), the *Retrieval-Augmented Program Analysis Agent*

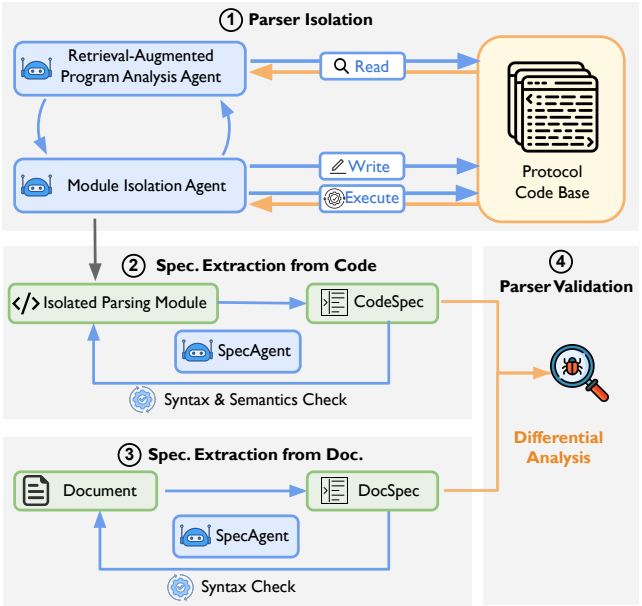

Figure 2: The Overview of PARVAL

Task Description: Refactor the given protocol implementation to isolate input parsing logic into a new parser function. This function should:
- Take only two parameters: the input buffer and its length.
- Return True if the input is valid, False otherwise.
- Exclude checks unrelated to the input buffer or its length.

Iteratively call 'query_name' to retrieve definition from the codebase. Please output complete function content for each newly added or modified function.

Example: [example]
Entry parsing function: [function]

Figure 3: LLM Task Description for Parser Isolation

and the *Module Isolation Agent* collaborate to extract an *Isolated Parsing Module* from the protocol codebase. In **Stage 2 Spec. Extraction from Code** (Section 3.2) and **Stage 3 Spec. Extraction from Doc.** (Section 3.3), the *Spec Agent* extracts format specifications from the *Isolated Parsing Module* and protocol standards, producing *CodeSpec* and *DocSpec*, respectively. In **Stage 4 Parser Validation** (Section 3.4), the two format specifications are compared to identify inconsistencies, which may point to bugs in the implementation or missing details in the protocol standard. The following sections describe each stage in detail.

3.1. Stage 1: Parser Isolation

In this stage, we isolate the parsing logic from the protocol codebase using an LLM-based multi-agent framework that integrates *dependency-aware retrieval mechanism* and *parser module construction*. As shown in Figure 2, the framework consists of two LLM agents: the *Retrieval-augmented Program Analysis Agent* and the *Module Isolation Agent*, augmented by tools to interact (i.e., read, write, or

execute) with the protocol codebase. The task description for this multi-agent system is shown in Figure 3.

The ***Retrieval-augmented Program Analysis Agent*** uses an LLM to iteratively analyze code and retrieve relevant context through a ***dependency-aware retrieval mechanism***. Starting from the entry parsing function, it traces input buffer usage via data and control dependencies, as well as function calls, to identify macros, types, and helper functions critical to parsing. It then employs an AST-based retrieval tool, built on Tree-sitter [18], to extract the corresponding definitions from the source code (**read**). In parallel, the ***Module Isolation Agent*** utilizes the retrieved context to construct an *isolated parsing module*. It iteratively **writes** identified parsing logic into a standalone, executable module and **executes** it, continuously refining the module based on build results and error feedback. When additional definitions or clarifications are needed, it collaborates with the *Retrieval-augmented Program Analysis Agent* to retrieve more contexts. The final output is a fully ***isolated parsing module*** that takes an input buffer and its length as input and returns a boolean indicating parsing success. Unlike the original codebase, which often mixes parsing with other processing logic, this isolated module contains only the parsing logic and is independently testable.

Example. For the entry parsing function shown in Figure 1, isolating the parsing logic requires understanding the behavior of its callee functions, as additional parsing logic may reside within them. To achieve this, LLMs must retrieve relevant context, just as a human would when analyzing the code. Hence, the *Retrieval-augmented Program Analysis Agent* retrieves the definition of `bfd_recv_ipv4`, `bfd_recv_ipv6`, and `BFD_GETVER` to determine whether they contain parsing-related operations. The agent identifies `msgbuf` and `mlen` as the received BFD packet and its length. Therefore, determining whether a piece of code contributes to BFD parsing involves analyzing its dependencies on `msgbuf` and `mlen`. For example, at line 15 in Figure 1, `msgbuf` is cast into a `bfd_pkt`, making `cp` directly data-dependent on `msgbuf`. The `if` condition at line 16 operates on a field of `cp` and performs input validation, making it directly relevant to BFD packet parsing. In contrast, the `if` conditions at lines 4 and 6 check the socket types rather than BFD packet fields, hence irrelevant to BFD packet parsing. The resulting *Isolated Parsing Module* generated by the *Module Isolation Agent* is shown in Figure 4.

3.2. Stage 2: Specification Extraction from Code

In this stage, we employ ***SpecAgent*** to transform the *isolated parsing module* into a formal representation of the input format. This representation, called ***CodeSpec***, is written in a domain-specific language (DSL) for describing protocol formats. *SpecAgent* performs this translation automatically by analyzing the parsing logic and iteratively refining the resulting specification based on feedback from both syntax and semantic checks.

```
1  bool parse_bfd_packet (uint8_t *msgbuf, size_t mlen)
2  {
3        if (mlen < BFD_PKT_LEN) {
4           cp_debug("too small (%ld bytes)", mlen);
5           return false;
6        }
7        /* Parse the control header for inconsistencies:
8        struct bfd_pkt *cp = (struct bfd_pkt *)(msgbuf);
9        if (BFD_GETVER(cp->diag) != BFD_VERSION) {
10          cp_debug("bad version %d", BFD_GETVER(cp->diag));
11          return false;
12       }
13         … // other parsing logic
14       return true;
15 }
```

Figure 4: *Isolated Parsing Module* Example

Syntax Check. To ensure syntactic correctness, *SpecAgent* uses the DSL's grammar to generate an initial *CodeSpec* from the *isolated parsing module*. It then iteratively validates and refines the specification until it conforms to the DSL rules.

Semantic Check. Beyond syntactic correctness, *CodeSpec* must match the semantic behavior of the *isolated parsing module*. We verify this by generating symbolic test cases from *CodeSpec*—positive cases that conform to the format and negative ones that violate it. These are run against the *isolated parsing module* to check whether valid inputs are accepted and malformed ones are rejected. If a positive case fails or a negative case passes, it indicates a semantic mismatch. To resolve such discrepancies, *SpecAgent* iteratively refines *CodeSpec* based on the test outcomes. Specifically, when a positive case fails, we instrument the parser to capture its execution trace, which is then provided to *SpecAgent* to guide the correction. In contrast, if a negative case is incorrectly accepted, *SpecAgent* analyzes how the input violates *CodeSpec* and adjusts the format to enforce the intended parsing behavior.

3.3. Stage 3: Specification Extraction from Doc.

In this stage, ***SpecAgent*** analyzes protocol standards (e.g., RFCs) to extract a structured format specification, ***DocSpec***, using the same DSL as in Stage 2. This transforms the natural language description of the message format—such as field definitions, length constraints, and value ranges—into a formal representation. Using a unified DSL enables direct, systematic comparison between *DocSpec* and the code-derived *CodeSpec*.

3.4. Stage 4: Parser Validation

In the final stage, we validate the parser implementation by comparing the protocol specifications extracted from code (*CodeSpec*, from Stage 2) and protocol standards (*DocSpec*, from Stage 3). Using *Differential Analysis*, we identify discrepancies such as mismatched fields or constraints. Each inconsistency is manually examined to determine whether it is a false positive or a true semantic mismatch. Confirmed

mismatches are then classified as either parser bugs or issues in the protocol standard.

4. Evaluation

Our tool is built on top of AutoGen [19], a multi-agent framework for developing LLM applications. We use Claude-3.5 Sonnet [20] as the LLM API with a temperature of 0 to reduce randomness and enhance reproducibility. For repository interaction, we build our tool using Tree-sitter [18], an AST-based parser to analyze and manipulate source code. For protocol format extraction (Section 3.2 and Section 3.3), we use 3D language [21] as the DSL and EverParse [22] as the syntax checker. We evaluate the effectiveness of our approach by addressing the following research questions:

- **RQ1**: How effective are the four stages of PARVAL?
- **RQ2**: How well does the LLM extract format specifications from code compared to a baseline?
- **RQ3**: What are the root causes of the discrepancies between the implementation and the RFC?
- **RQ4**: How much manual effort does PARVAL require?

4.1. Dataset

We evaluate PARVAL on the BFD protocol using the C-based implementation from FRRouting [23] (stable/8.4), and RFC 5880 [24] as the official protocol standard. PARVAL is then applied to validate the implementation against the RFC.

4.2. RQ1: Effectiveness of Each Stage in PARVAL

4.2.1. Stage 1: Parser Isolation. We evaluate parser isolation by measuring the precision and recall of the resulting *isolated parsing module*. Precision reflects the correctness of extracted parsing logic, while recall indicates how much of the original logic is preserved. Ground truth is established by manually annotating the original parser, aided by NetLifter [11], which uses LLVM-based static analysis to identify parsing-relevant IR instructions.

Results. The *isolated parsing module* achieves 100% precision and recall, showing that all validation logic is correctly extracted and preserved. This demonstrates PARVAL 's ability to isolate complete and accurate parsing logic. We also assess whether isolation alters parser behavior. In the BFD case, packet parsing does not depend on runtime context (e.g., socket type or protocol version), so the isolated module maintains the intended input structure. However, for protocols influenced by configuration or negotiation state, isolation may unintentionally broaden or restrict accepted inputs—requiring additional control flow modeling to preserve these semantics.

4.2.2. Stage 2: Specification Extraction from Code. We evaluate PARVAL 's accuracy in generating *CodeSpec*, the format specification extracted from the *isolated parsing module*, consisting of field names, types, and constraints. A ground truth is manually constructed from the module's logic, and precision and recall are used to measure the correctness and completeness of the extracted specification.

TABLE 1: Precision and Recall for Format Specification Extraction from the *Isolated Parsing Module*

Metric	PARVAL	Precision	Recall
Field Name	11	100.0%	100.0%
Field Type	11	100.0%	100.0%
Field Constraint	4	100.0%	100.0%

Results. As shown in Table 1, PARVAL correctly extracts all 11 field names, 11 field types, and 4 field constraints with perfect precision and recall. Since the isolated parser fully encapsulates the original parsing logic, the extracted *CodeSpec* is both accurate and complete for this target.

4.2.3. Stage 3: Specification Extraction from Doc. We evaluate *DocSpec*, the format extracted from the RFC, by comparing it against a manually constructed ground truth, using precision and recall to assess accuracy and completeness.

TABLE 2: Precision and Recall for Format Specification Extraction from RFC Document

Metric	PARVAL	Precision	Recall
Field Name	37	100.0%	100.0%
Field Type	37	97.3%	97.7%
Field Constraint	15	100.0%	58.3%

Results. As shown in Table 2, PARVAL extracts 37 field names, 37 field types, and 15 field constraints from the RFC document. Field names and types are extracted with near-perfect precision and recall, as they are explicitly defined. Constraint extraction is more difficult. While the precision remains 100%, recall drops to 58.3%, this is because many constraints are implied rather than explicitly stated in the RFC documents. Despite some missing constraints, the high precision indicates that those extracted constraints are reliable and useful for validation. In particular, if an implementation fails to enforce a constraint specified in *DocSpec*, it is highly likely to indicate a true implementation bug.

4.2.4. Stage 4: Parser Validation. We validate the parser by comparing *CodeSpec* and *DocSpec*, focusing on field type and constraint differences, as field names do not affect behavior. Discrepancies are identified by mismatches between the two specifications, and then traced back to source code and the RFC to determine whether they represent true inconsistencies.

TABLE 3: Differential Analysis Results: *CodeSpec* vs. *DocSpec*

Discrepancy Type	Total Inconsis.	True Inconsis.	Extraction Error
Field Type	21	21	0
Field Constraint	15	13	2
Total	36	34	2

Results. As shown in Table 3, PARVAL reports 36 discrepancies between *CodeSpec* and *DocSpec*, including 21 type

and 15 constraint mismatches. Among the 36 discrepancies, 34 are confirmed as true inconsistencies, and only two are false positives caused by LLM hallucination, yielding a low 5.6% false positive rate. This demonstrates PARVAL's effectiveness in detecting real mismatches between parser implementation and its protocol standard.

4.3. RQ2: Baseline Comparison

Setup and Metrics. To assess PARVAL's end-to-end effectiveness in extracting format specification from source code (including both parser isolation (Stage 1) and specification extraction (Stage 2)), we compare *CodeSpec* against a manually constructed ground truth based on the full BFD implementation, not just the *isolated parsing module*. Precision and recall are used to measure how well PARVAL captures field names, types, and constraints.

Results. As shown in Table 4, PARVAL achieves 100% precision and recall, confirming its accuracy and completeness against the full implementation.

TABLE 4: Precision and Recall for Format Specification Extraction from the Full BFD Implementation

Metric	PARVAL	Precision	Recall
Field Name	11	100.0%	100.0%
Field Type	11	100.0%	100.0%
Field Constraint	4	100.0%	100.0%

4.4. RQ3: Root Cause of Identified Discrepancies

Setup and Metrics. To identify the root causes of true discrepancies, we manually analyze each true inconsistency and classify it as either an implementation bug or an RFC issue such as unclear, inaccurate, or missing descriptions. For implementation bugs, we check existing bug reports to determine whether they are newly discovered. For RFC issues, we carefully examine the RFC descriptions to assess their root causes.

TABLE 5: Root Causes of Identified Inconsistencies

Discrepancy Type	Implementation Bug	RFC Issue
Field Type Mismatch	21	0
Constraint Mismatch	11	2
Total	32	2

Results. As shown in Table 5, 32 of 34 true inconsistencies are implementation bugs, while two stem from RFC issues. To avoid redundancy in reporting, we group inconsistencies with the same root cause and identify seven unique bugs, five of which are newly detected by PARVAL, detailed in Table 6. The two RFC-related issues are also previously undocumented and detailed in Table 7.

TABLE 6: The Detected Implementation Bugs

No.	Bug Description	New
1	Flag `M` should always be 0.	✗
2	Miss check Authentication Present (A) to handle optional Authentication Section.	✗
3	Miss validation for Simple Password Authentication Section format.	✓
4	Miss validation for Keyed MD5 Authentication Section format.	✓
5	Miss validation for Meticulous Keyed MD5 Authentication Section format.	✓
6	Miss validation for Keyed SHA1 Authentication Section Format.	✓
7	Miss validation for Meticulous Keyed SHA1 Authentication Section format.	✓

TABLE 7: The Detected RFC Document Issues

No.	RFC Document Issues	New
1	Miss explicitly mention that `Detect Mult` should not be 0.	✓
2	Miss explicitly mention that `Length` should be at least 24.	✓

4.5. RQ4: Estimation of Manual Effort

We assess the manual effort involved in each stage of PARVAL. Stage 1 requires manually identifying the entry parsing function, which takes approximately 10 minutes. Stages 2 and 3 are fully automated and require no human intervention. Stage 4, which involves comparing CodeSpec and DocSpec and analyzing inconsistencies, is currently manual and takes about 20 minutes.

5. Discussion

This work explores the potential of LLMs to validate network protocol parsers by extracting and comparing protocol format specifications from source code and natural language documents. Rather than providing a fully verified or complete solution, PARVAL demonstrates how LLMs can assist in a domain traditionally dominated by static analysis and formal methods. While promising, PARVAL does not provide formal soundness or completeness guarantees. Both *CodeSpec* and *DocSpec* are LLM-generated and may contain inaccuracies, requiring manual inspection to distinguish true inconsistencies from extraction errors.

Given these limitations, PARVAL is best viewed as a practical assistant that complements formal techniques. Its effectiveness must be evaluated empirically. Although we demonstrate its utility on BFD, broader validation across diverse protocols and implementations is needed to assess generality and robustness. Future work includes extending support to parsers written in other programming languages, further automating the validation process, and integrating PARVAL into existing protocol testing frameworks.

6. Related Work

6.1. Techniques for Protocol Parser Validation

Model Checking. Model checking-based methods [7]–[9] formally verifies protocol implementations against their specifications. However, constructing formal models often requires significant manual effort, limiting its applicability to practical validation tasks.

Conventional Fuzzing. Fuzzing is a widely used approach for testing protocol implementations by generating test cases

and executing them to trigger unexpected behavior, particularly crashes. Mutation-based fuzzing, such as AFLNET [10], creates test inputs by mutating existing valid or semi-valid inputs. While it requires little knowledge of the protocol format, its effectiveness relies heavily on the quality of seed inputs and often struggles to reach deep execution paths when parsing conditions are complex. In contrast, generation-based fuzzing, like BooFuzz [25], generates inputs using predefined grammars, improving coverage but requiring additional effort to define these grammars. NetLifter [11] automates grammar construction through static analysis to extract path conditions, but it still focuses only on low level bugs such as crashes. However, many protocol implementation bugs are semantic bugs that do not cause crashes but violate protocol specifications, which conventional fuzzing fails to detect.

Differential Analysis. Differential analysis identifies inconsistencies by comparing multiple implementations of the same protocol. Static tools like ParDiff [12] extract state machines from different protocol implementations to detect deviations. Dynamic differential testing tools, like DPI-Fuzz [16] and Prognosis [26], generate test cases to execute across multiple implementations and compare their runtime behaviors. While these approaches can uncover semantic bugs (i.e., silent violations of protocol rules), they rely on the availability of multiple independent implementations, which is not always feasible. They also fail to detect bugs shared across all implementations, limiting their effectiveness in certain scenarios.

6.2. Protocol Format Lifting

Previous work has focused on extracting protocol formats from either source code or protocol standards to support validation and testing.

Format Extraction from Source Code. Tools like Netlifter [11] and ParDiff [12] use static analysis to extract protocol formats from source code. These methods often struggle with loops, either relying on imprecise heuristics for loop summarization, or bounded loop unrolling, leading to incomplete or inaccurate format extraction. Additionally, these tools are typically language-specific and require significant effort to support new languages. PARVAL overcomes these limitations by leveraging LLMs for flexible, language-agnostic format extraction. It summarizes loop behavior and validates the extracted logic through test case execution.

Format Extraction from Protocol Standards. Recent works [15], [17], [27] highlight the potential of LLMs for extracting format specifications from protocol standards. 3DGen [17] and ParCleanse [15] apply LLMs to transform unstructured protocol descriptions into structured formats. In contrast, ChatAFL [27] avoids feeding RFCs to the model and instead relies on pre-trained models like GPT-3.5, which have been trained on RFCs and are able to answer format-related queries directly.

PARVAL combines both source code and document-based format extraction, enabling cross-validation to identify inconsistencies between implementations and their protocol standards. By leveraging LLMs, PARVAL offers a flexible, language-agnostic approach to protocol format lifting.

6.3. Large Language Models for Coding Tasks

Large Language Models (LLMs) have been widely used for various coding tasks, such as code generation [28]–[31], program testing [32]–[35], static bug detection [36]–[41], reverse engineering [42]–[44], and automated repair [45]–[47]. These models leverage extensive code repositories, natural language data, and domain-specific knowledge to tackle tasks that once demanded substantial human expertise [31], [48]–[51]. However, LLM-based approaches also face challenges such as hallucination [52], [53] and incapabilities in reasoning about complex program behaviors [54], [55]. To overcome these challenges, recent research has explored techniques such as chain-of-thought prompting [56], retrieval-augmented generation [57], and the incorporation of symbolic reasoning [37]–[39], aiming to enhance both the reliability and transparency of model outputs. PARVAL builds on these advancements by leveraging retrieval-augmented-LLMs for protocol format extraction and using symbolic test case generation for refinement, thereby reducing hallucinations and enhancing correctness.

7. Conclusion

This paper presents PARVAL, a multi-agent framework that leverages LLMs to validate network protocol parsers against protocol standards. By extracting format specifications from both source code and natural language documentation, PARVAL enables direct comparison to uncover inconsistencies between implementations and standards. We evaluate PARVAL on the BFD protocol and demonstrate its effectiveness in identifying mismatches with high precision, uncovering seven unique implementations bugs, five of which were previously unknown. While PARVAL does not offer formal guarantees, it illustrates the practical utility of LLMs in improving protocol correctness. Our results underscore the potential of LLM-assisted validation as a complement to traditional static analysis and formal methods, offering a more scalable approach to protocol verification.

Acknowledgment

We thank all the anonymous reviewers and our shepherd, Nathan Dautenhahn, for the insightful feedback and guidance. 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, and ONR N00014-23-1-2081. 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] "The transport layer security (tls) protocol," https://datatracker.ietf.org/doc/html/rfc5246.

[2] "File transfer protocol (ftp)," https://datatracker.ietf.org/doc/html/rfc959.

[3] "Internet protocol (ip)," https://datatracker.ietf.org/doc/html/rfc791.

[4] Heartbleed, "The heartbleed bug," https://heartbleed.com, 2020.

[5] "Openssl," https://www.openssl.org.

[6] "Cve-2021-41773," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-41773.

[7] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *NSDI*. USENIX, 2004, pp. 155–168.

[8] G. Díaz, F. Cuartero, V. V. Ruiz, and F. L. Pelayo, "Automatic verification of the TLS handshake protocol," in *SAC*. ACM, 2004, pp. 789–794.

[9] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," in *ICSE*. IEEE Computer Society, 2003, pp. 385–395.

[10] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[11] 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*, ser. CCS '23. ACM, 2023, p. 1287–1301.

[12] 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," in *Proc. ACM Program. Lang.*, ser. OOPSLA '24. ACM, 2024, pp. 1208–1234.

[13] 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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 427–441. [Online]. Available: https://doi.org/10.1145/3575693.3575731

[14] M. Zheng, J. Yang, M. Wen, H. Zhu, Y. Liu, and H. Jin, "Why do developers remove lambda expressions in java?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 67–78.

[15] M. Zheng, D. Xie, Q. Shi, C. Wang, and X. Zhang, "Validating network protocol parsers with traceable rfc document interpretation," in *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2025.

[16] G. S. Reen and C. Rossow, "Dpifuzz: a differential fuzzing framework to detect dpi elusion strategies for quic," in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20. ACM, 2020, pp. 332–344.

[17] S. Fakhoury, M. Kuppe, S. K. Lahiri, T. Ramananandro, and N. Swamy, "3dgen: Ai-assisted generation of provably correct binary format parsers," *arXiv preprint arXiv:2404.10362*, 2024.

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

[19] 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

[20] "Claude 3.5 sonnet," https://www.anthropic.com/claude/sonnet.

[21] "3d: Dependent data descriptions for verified validation," https://project-everest.github.io/everparse/3d.html.

[22] "Everparse," https://project-everest.github.io/everparse/3d-lang.html.

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

[24] "Bidirectional forwarding detection (bfd)," https://datatracker.ietf.org/doc/html/rfc5880.

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

[26] T. Ferreira, H. Brewton, L. D'Antoni, and A. Silva, "Prognosis: closed-box analysis of network protocol implementations," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 762–774.

[27] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, ser. NDSS '24. The Internet Society, 2024.

[28] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[29] Y. Ding, M. J. Min, G. Kaiser, and B. Ray, "Cycle: Learning to self-refine the code generation," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 392–418, 2024.

[30] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, "Evaluating language models for efficient code generation," in *First Conference on Language Modeling*, 2024. [Online]. Available: https://openreview.net/forum?id=IBCBMeAhmC

[31] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma *et al.*, "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," *arXiv preprint arXiv:2406.11931*, 2024.

[32] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.

[33] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2312–2323.

[34] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "Whitefox: White-box compiler fuzzing empowered by large language models," *arXiv preprint arXiv:2310.15991*, 2023.

[35] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3643769

[36] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2237–2248.

[37] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[38] 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

[39] 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

[40] J. Guo, C. Wang, X. Xu, Z. Su, and X. Zhang, "Repoaudit: An autonomous llm-agent for repository-level code auditing," *arXiv preprint arXiv:2501.18160*, 2025.

[41] 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

[42] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, "Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4554–4568.

[43] X. Xu, Z. Zhang, Z. Su, Z. Huang, S. Feng, Y. Ye, N. Jiang, D. Xie, S. Cheng, L. Tan *et al.*, "Leveraging generative models to recover variable names from stripped binary," *arXiv preprint arXiv:2306.02546*, 2023.

[44] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," *arXiv preprint arXiv:2403.05286*, 2024.

[45] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1430–1442.

[46] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[47] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1282–1294. [Online]. Available: https://doi.org/10.1145/3597926.3598135

[48] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

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

[50] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[51] 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," *arXiv preprint arXiv:2306.03324*, 2023.

[52] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia *et al.*, "Recode: Robustness evaluation of code generation models," *arXiv preprint arXiv:2212.10264*, 2022.

[53] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[54] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia, "Reasoning runtime behavior of a program with llm: How far are we?" *arXiv preprint cs.SE/2403.16437*, 2024.

[55] C. Liu, S. D. Zhang, A. R. Ibrahimzada, and R. Jabbarvand, "Codemind: A framework to challenge large language models for code reasoning," *arXiv preprint arXiv:2402.09664*, 2024.

[56] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[57] M. Minor and E. Kaucher, "Retrieval augmented generation with llms for explaining business process models," in *International Conference on Case-Based Reasoning*. Springer, 2024, pp. 175–190.