Validating Network Protocol Parsers with Traceable RFC Document Interpretation

MINGWEI ZHENG, Purdue University, USA
DANNING XIE, Purdue University, USA
QINGKAI SHI, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHENGPENG WANG, Purdue University, USA
XIANGYU ZHANG, Purdue University, USA

Validating the correctness of network protocol implementations is highly challenging due to the oracle and traceability problems. The former determines when a protocol implementation can be considered buggy, especially when the bugs do not cause any observable symptoms. The latter allows developers to understand how an implementation violates the protocol specification, thereby facilitating bug fixes. Unlike existing works that rarely take both problems into account, this work considers both and provides an effective solution using recent advances in large language models (LLMs). Our key observation is that network protocols are often released with structured specification documents, a.k.a. RFC documents, which can be systematically translated to formal protocol message specifications via LLMs. Such specifications, which may contain errors due to the hallucination of LLMs, are used as a quasi-oracle to validate protocol parsers, while the validation results in return gradually refine the oracle. Since the oracle is derived from the document, any bugs we find in a protocol implementation can be traced back to the document, thus addressing the traceability problem. We have extensively evaluated our approach using nine network protocols and their implementations written in C, Python, and Go. The results show that our approach outperforms the state-of-the-art and has detected 69 bugs, with 36 confirmed. The project also demonstrates the potential for fully automating software validation based on natural language specifications, a process previously considered predominantly manual due to the need to understand specification documents and derive expected outputs for test inputs.

CCS Concepts: • **Networks** → **Protocol testing and verification**; • **Software and its engineering** → **Correctness**; • **Computing methodologies** → **Natural language processing**.

Additional Key Words and Phrases: Network protocol parsers, Traceability, Large language model

ACM Reference Format:
Mingwei Zheng, Danning Xie, Qingkai Shi, Chengpeng Wang, and Xiangyu Zhang. 2025. Validating Network Protocol Parsers with Traceable RFC Document Interpretation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA078 (July 2025), 23 pages. https://doi.org/10.1145/3728955

1 Introduction

Network protocols play a key role in the Internet of Things era as they define how devices worldwide connect to and communicate with each other. As essential components in network protocol implementations, network protocol parsers parse and validate network messages, which ensures network messages follow specific syntactic and semantic rules, thus preventing invalid or malicious data from disrupting system operations or compromising security. Despite their importance,

Authors' Contact Information: Mingwei Zheng, Purdue University, West Lafayette, USA, zheng618@purdue.edu; Danning Xie, Purdue University, West Lafayette, USA, xie342@purdue.edu; Qingkai Shi, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, qingkaishi@nju.edu.cn; Chengpeng Wang, Purdue University, West Lafayette, USA, wang6590@purdue.edu; Xiangyu Zhang, Purdue University, West Lafayette, USA, xyzhang@cs.purdue.edu.

ACM 2994-970X/2025/7-ARTISSTA078
https://doi.org/10.1145/3728955

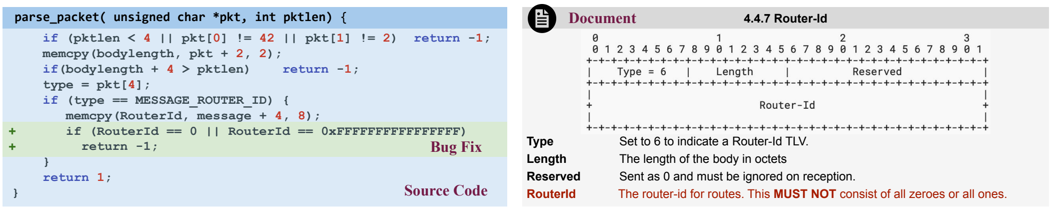
building high-quality network protocol parsers is challenging and error-prone [37, 45]. According to MITRE, input (e.g., network messages) validation issues [33] are among the top four in the CWE Top 25 Most Dangerous Software Weaknesses [32], emphasizing the critical risks that improperly handled inputs pose to system security and reliability.

Existing Works. Our work targets input validation bugs in network protocol parsers, where the parser incorrectly accepts invalid packets or rejects valid packets. Unfortunately, existing test oracles are insufficient to thoroughly detect such bugs because many of them are silent, not causing obvious runtime symptoms (e.g., crashes) or violating other well-established properties (e.g., memory-safety and data-privacy). As a result, conventional fuzzing [19, 36] and static analyzers [9, 43] that rely on non-protocol-specific oracles can hardly detect them. Model checking [15, 34] constructs protocol-specific oracles to detect these bugs. However, formal specifications are often missing. Manually constructing them is time-consuming since network protocols are often described in natural language (e.g., in RFC documents). To address these oracle issues, differential analysis techniques, including both static differential analysis [11, 58] and dynamic differential analysis [6, 38], identify bugs by comparing multiple implementations of the same protocol. While they are effective in many cases, they fail to detect bugs shared by multiple implementations. For instance, the bug in Figure 1 actually exists in multiple Babel implementations, including FRRouting Protocol Suite [12] and Jech/Babel [8], where differential analysis techniques become ineffective.

Our Approach. A popular software validation method is to extract testable properties from specification documents and then construct inputs to test these properties. The key challenge is deriving the expected outputs for given inputs based solely on the documents, which often requires substantial manual effort. Our paper presents ParCleanse, which uses Large Language Models (LLMs) to automate this process. In particular, it extracts formal protocol specifications from RFC documents and derives inputs together with the expected outputs from the specifications to validate parser implementations. ParCleanse generates a set of valid packets (conforming to the format) and invalid packets (violating the format) based on the LLM-extracted protocol formats to test whether target parsers accept valid packets and reject invalid ones. If a parser deviates from this expected behavior, it may indicate a potential inconsistency between the specification and its implementation. Since RFC documents are widely accepted as network protocol standards, ParCleanse overcomes the limitations of differential analysis, which cannot detect bugs shared across multiple implementations. Furthermore, the LLM-based format extraction process is highly automated, reducing the human effort to generate protocol-specific oracles.

However, many RFC documents are lengthy and complex, causing substantial LLM hallucinations. As such, extracted protocol formats may contain errors, leading to misinterpretation of parser behavior. For example, an incorrect field constraint in the extracted format could cause us to mistakenly flag a parser bug if the parser accepts a packet that violates the constraint. To mitigate LLM hallucinations, our approach incorporates two key designs:

- First, our approach uses a divide-and-conquer strategy to systematically decompose an RFC document into smaller and manageable sections while preserving their structural relationships within a knowledge graph called the DocTree. This decomposition mitigates LLM hallucinations across large contexts and enables precise extraction of subformats, which are then combined into a complete protocol format with the hierarchical guidance of the DocTree.
- Second, our approach features traceable inconsistency identification: when an inconsistency between the LLM-extracted format and parser behavior is detected, it is traced back to the relevant sections of the RFC document. This traceability supports an additional validation step to determine whether the inconsistency stems from the LLM's hallucination or a parser implementation bug.



```
parse_packet( unsigned char *pkt, int pktlen) {
    if (pktlen < 4 || pkt[0] != 42 || pkt[1] != 2)  return -1;
    memcpy(bodylength, pkt + 2, 2);
    if(bodylength + 4 > pktlen)    return -1;
    type = pkt[4];
    if (type == MESSAGE_ROUTER_ID) {
       memcpy(RouterId, message + 4, 8);
+      if (RouterId == 0 || RouterId == 0xFFFFFFFFFFFFFFFF)
+        return -1;
    }
    return 1;
}
```

4.4.7 Router-Id

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 6   |    Length     |           Reserved            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                           Router-Id                           +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Type	Set to 6 to indicate a Router-Id TLV.
Length	The length of the body in octets
Reserved	Sent as 0 and must be ignored on reception.
RouterId	The router-id for routes. This **MUST NOT** consist of all zeroes or all ones.

Fig. 1. A bug detected by ParCleanse, its fix, and the corresponding documentation.

To thoroughly validate parser implementations, ParCleanse performs both field-level and structure-level mutations (referred to as **property-level mutations**) to generate comprehensive test cases. It generates both positive and negative inputs for each property, allowing thorough validation of parser implementations by verifying each protocol property against its specification.

Contribution. In summary, we make the following contributions.

- We propose a novel validation approach to detect bugs in network protocol parsers by "comparing" them with protocol formats extracted from RFC documents.
  - It conducts a divide-and-conquer format extraction to interpret the official RFC documents of network protocols to precise and complete network protocol format specifications.
  - It features fine-grained property-level input mutations to thoroughly test parser implementations guided by the extracted specifications.
  - It leverages a traceable inconsistency identification technique, allowing any identified inconsistencies to be traced back to the original specification for a more accurate diagnosis.
- We implement our approach as a prototype tool, ParCleanse,[1] and evaluate it on nine network protocols implemented in C, Python, and Go. The experimental results show that ParCleanse effectively extracts protocol formats from RFCs, achieving 100% precision and recall for message types and 99% precision and 95% recall for field names, outperforming the state-of-the-art LLM-based method, ChatAFL. ParCleanse identifies 69 bugs, with 36 confirmed, outperforming the state-of-the-art protocol parser testing tool, ParDiff.

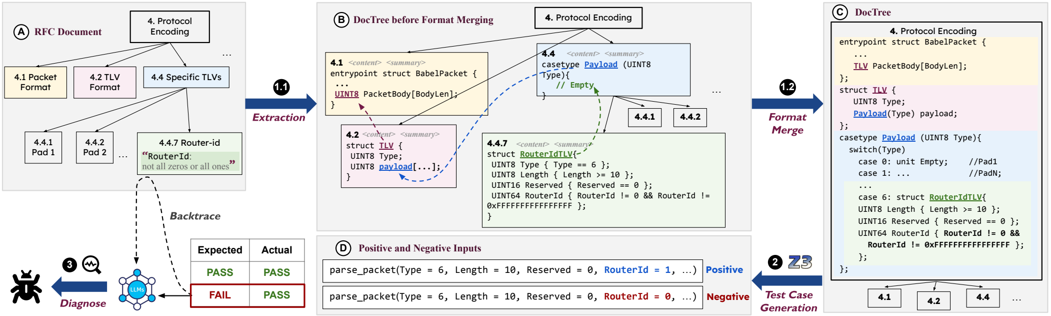
2 Motivation

In this section, we first present a real-world bug detected by ParCleanse (Section 2.1) and illustrate the limitations of existing methods (Section 2.2). We then discuss the inherent challenges of the problem and introduce the design of our technique (Section 2.3).

2.1 A Real-World Example

Figure 1 shows a buggy code snippet of the Babel network routing protocol (from the FRRouting Protocol Suit [12]) and its RFC document. The document specifies the Babel protocol format, which includes four fields: `Type`, `Length`, `Reserved`, and `RouterId`. In RFC documents, the width of each field in the structure table indicates its byte length. In this example, `Type` is one byte, while `RouterId` is eight bytes. This document also restricts `RouterId` from being all zeroes (`0`) or all ones (`0xFFFFFFFFFFFFFFFF` for eight bytes), as these could lead to ambiguities in Babel's routing strategies. The check on `RouterId` helps Babel maintain stability and prevent routing loops, especially in dynamic or frequently changing network topologies. However, this validation is missing in the buggy implementation. Malicious attackers could exploit this oversight to introduce packets with an invalid `RouterId` into the network and disrupt the network, causing instability in path calculation and failure to properly update routing tables.

[1] ParCleanse is publicly available at https://github.com/zmw12306/ParCleanse.



```
entrypoint struct BabelPacket {
 ...
 UINT8 PacketBody[BodyLen];
}
```

```
struct TLV {
 UINT8 Type;
 UINT8 payload[...];
}
```

```
casetype Payload (UINT8
Type){
    // Empty
}
```

```
struct RouterIdTLV{
 UINT8 Type { Type == 6 };
 UINT8 Length { Length >= 10 };
 UINT16 Reserved { Reserved == 0 };
 UINT64 RouterId { RouterId != 0 && RouterId !=
0xFFFFFFFFFFFFFFFF };
}
```

```
entrypoint struct BabelPacket {
   ...
   TLV PacketBody[BodyLen];
};
struct TLV {
   UINT8 Type;
   Payload(Type) payload;
};
casetype Payload (UINT8 Type){
  switch(Type)
    case 0: unit Empty;    //Pad1
    case 1: ...            //PadN;
    ...
    case 6: struct RouterIdTLV{
    UINT8 Length { Length >= 10 };
    UINT16 Reserved { Reserved == 0 };
    UINT64 RouterId { RouterId != 0 &&
      RouterId != 0xFFFFFFFFFFFFFFFF };
    };
};
```

```
parse_packet(Type = 6, Length = 10, Reserved = 0, RouterId = 1, ...)
parse_packet(Type = 6, Length = 10, Reserved = 0, RouterId = 0, ...)
```

Expected	Actual
PASS	PASS
FAIL	PASS

Fig. 2. ParCleanse **extracts specifications from documents to build DocTree. Dashed arrows in Ⓑ indicate hierarchical relationships for forming the protocol format in Ⓒ.** ParCleanse **then generates test cases to detect inconsistencies and backtraces the relevant document section for LLM diagnosis.**

2.2 Limitations of Existing Work

Despite the severity of this issue, existing techniques struggle to detect the bug. A key limitation of existing work is the lack of high-quality oracles. Traditional static analyzers (e.g., KLEE [9], Pinpoint [43]) and conventional fuzzing techniques (e.g., SAGE [19], BooFuzz [36]) rely on general oracles, such as the violations of safety properties and the abnormal behaviors in the runtime, to detect and trigger bugs, respectively. Existing LLM-based testing approaches like ChatAFL [31] and Fuzz4All [53] also rely on crash-based oracles. However, the bug in Figure 1 does not violate such general properties, but rather protocol-specific properties causing silent system state corruptions.

Differential analysis tools like ParDiff [58] and DPIFuzz [38] partially address the need for protocol-specific oracles by comparing multiple implementations of the same protocol. However, these approaches require at least one correct implementation to flag inconsistencies. For the bug in Figure 1, alternative implementations (e.g., Jech/Babel [8]) also miss checking the specific condition. As such, differential analysis cannot detect the bug. Additionally, although model checking could theoretically verify protocol-specific properties, it requires a formal protocol specification, which is missing for Babel. Unfortunately, constructing a formal model from Babel's natural language RFC document would require substantial manual effort.

2.3 Our Approach in a Nutshell

We propose ParCleanse, which leverages LLMs to automatically generate oracles (i.e., network protocol format specifications) from protocol documents and create fine-grained test cases to effectively discover bugs with a low false positive rate. The extracted oracles have two main benefits: (1) covering all formats described in the document, allowing *comprehensive* testing of protocol parsers; and (2) providing *traceability*, which enhances bug understanding, reduces false positives, and facilitates fixes. We will walk through the motivating example, introduce the three phases of ParCleanse, and outline the challenges and solutions at each step.

Phase 1: Protocol Format Extraction (Section 4.1). An intuitive approach is to extract formal specifications directly from RFC documents, as they define expected input and output behaviors with high quality. However, RFC documents are often long and complex, ranging from tens to hundreds of pages. Feeding the entire document into an LLM to automatically generate protocol format specifications (or formats for simplicity) often leads to incomplete or incorrect formats due to LLMs' inherent hallucinations. A more effective approach is to divide the document into smaller pieces and prompt the LLM to generate the format for each piece. However, a challenge arises:

Challenge 1: While dividing a lengthy RFC document into smaller pieces can reduce errors from LLM outputs, these pieces are often interdependent, making it difficult to directly combine inferred sub-protocol formats.

We implement a divide-and-conquer strategy using a data structure, DocTree, to capture hierarchical relationships within RFC documents. As shown in Figure 2, we begin by dividing the RFC 8966 for the Babel protocol into sections (Ⓐ). In step 1.1, the LLM summarizes each section and generates the corresponding protocol format sub-specification. Based on the segmentation, we construct an initial DocTree (Ⓑ) that mirrors the document's table of contents, treating each section as a node. However, the hierarchical structure in the table of contents doesn't align with the protocol's message structure. For example, while *"section 4.1"* (packet format) and *"section 4.2"* (TLV format) appear at the same level in the table of contents, they actually have a nested relationship. Specifically, *"4.1"* should be the parent of *"4.2"*, as *"4.2"* represents a subset of the network packet outlined by *"4.1"*. To correct this misalignment, we prompt the LLM in step 1.2 to refine the DocTree hierarchy based on the relationships between sections as shown with the dashed arrows, producing a structure that accurately reflects the protocol's format. This refined DocTree enables us to generate a coherent, hierarchical protocol format as shown in Ⓒ.

Phase 2: Test Case Generation (Section 4.2). Given the protocol formats with field constraints in Ⓒ, we generate test cases based on the constraints. While solvers like Z3 can efficiently produce both valid and invalid inputs that satisfy or violate these constraints, using the full set of constraints alone does not sufficiently explore the input space, nor does it guarantee diversity in bug discovery.

This limitation arises because the generated tests do not isolate specific fields or individual constraints. As a result, although direct test generation based on the whole specification may trigger some exceptions, it tends to uncover only a small set of bugs, leaving large portions of the input space untested and essential format properties (field-level properties and structure-level properties) unchecked (see the ablation studies in Section 5.5.4 for further discussions).

Challenge 2: Naively generating inputs based on the whole protocol specification is insufficient to validate each format property and exhaustively explore parser behaviors.

ParCleanse addresses this challenge with a fine-grained test-generation strategy. In step 2, we produce both positive and negative inputs that conform and violate the individual constraints (Ⓓ), respectively. A negative input violates only a single format property. If the target parser accepts both types, it indicates a lack of enforcement of the property, highlighting inconsistencies between the specification and the implementation. This systematic, property-level mutation provides a comprehensive protocol validation.

As shown in Ⓓ, to validate the format property for the field `RouterId`, we generate two network messages: one with `RouterId = 1` conforming the constraint, and the other with `RouterId = 0` violating it. In the negative test case, all other fields are identical to the positive case. However, both test cases are accepted by the parser, indicating an inconsistency where the parser fails to enforce the `RouterId` constraint specified by the protocol format.

Phase 3: Inconsistency Identification (Section 4.3). Even when an inconsistency is detected, inaccuracies in the extracted specification may cause false positives, posing this challenge:

Challenge 3: How can we distinguish real inconsistencies from false positives caused by incorrect specifications?

Thus, when an inconsistency arises between the extracted specification and the parser implementation, it is crucial to determine whether it stems from an implementation bug or an error in the LLM-extracted format. ParCleanse enables this by providing complete traceability from the triggered inconsistency back to the exact RFC section where the relevant property is defined. By linking each format property with its source RFC section during format extraction (Phase 1), we establish traceability, allowing an accurate diagnosis of potential format violations in parsers.

In the motivating example, we generate the negative input by intentionally violating the constraint for `RouterId` as specified in *"section 4.4.7"*. To diagnose this inconsistency, we retrieve *"section 4.4.7"* through "backtrace" (indicated by the dashed arrow) from the RFC document and prompt the LLM to analyze the discrepancy. To enhance accuracy, we use chain-of-thought prompting, which requires the LLM to explain its reasoning steps, thereby reducing errors and improving decision reliability. In this case, the LLM concludes that it is due to a bug in the parser implementation. We hence report the bug (i.e., missing checks for RouterId not being all zeros or all ones). On the other hand, if the LLM concludes that the inconsistency is due to the incorrect extracted format, it will refine the format accordingly to improve the testing effectiveness.

3 Problem Formulation

This section introduces key preliminaries, including the formal definitions of network packets and their format syntax, followed by a formal statement of the problem addressed in this paper.

3.1 Protocol Packet and Its Format Syntax

Network protocols establish the standards of data transmission and interpretation for the communication between devices. Typically, data is encoded and transmitted as a sequence of bytes, referred to as a packet. Similar to object fields in memory, the bytes located in a consecutive segment of a protocol packet can indicate a specific unit of information, such as message types and message contents. Concretely, a packet should consist of the following five key elements:

- **Message Types:** The different formats a protocol use for various kinds of messages.
- **Field Names:** The names of individual data fields within a specific type of message.
- **Field Types:** The data types of fields (e.g., byte, bit, struct, and array) and their sizes, which can be either fixed or variable (i.e., depending on some other field).
- **Independent Constraints:** Restrictions on individual fields, such as numeric ranges or fixed values, that do not rely on other fields.
- **Dependent Constraints:** Constraints that involve relationships across multiple fields.

To ensure correct interpretation, these elements must be organized in a specific form defined in Figure 3. Each packet is a Struct Type object, aggregating multiple fields, each defined by a type and identifier, with optional constraints. Apart from the Struct Type, a type in the protocol format may be a Primitive Type, Array Type, or Case Type. Primitive Type (e.g., UINT8, UINT16) defines fixed-size fields. Array Type represents a sequence of elements of the same type. The length of an Array-Typed object can be constant

Packet	$p \in \text{Struct Type}$
Type	$\text{Type} := \text{pType} \mid \text{ArrayType} \mid \text{CaseType} \mid \text{StructType}$
Identifier	$\mathit{identifier} \in \text{String}$
Primitive Type	$\text{pType} := \text{UINT8} \mid \text{UINT16} \mid \text{UINT32} \mid \text{UINT64} \mid \cdots$
Array Type	$\text{ArrayType} := \text{Type}[\mathit{const}] \mid \text{Type}[f(\mathit{identifier}^{+})]$
Case Type	$\text{CaseType} := \text{switch}(\mathit{identifier})\ \{\text{case}\ \mathit{const} : \text{Type}\}^{+}$
Struct Type	$\text{StructType} := \text{struct}\ \mathit{identifier}\ \{\mathit{field}^{+}\}\ (f(\mathit{field}^{+}) \odot 0)^{*}$
Field	$\mathit{field} := \text{Type}\ \mathit{identifier}$
Constant	$\mathit{const} \in \text{UINT64}$
Function	$f \in \text{ArithFunction}$
Cmp Operator	$\odot := \geq \mid \leq \mid > \mid < \mid = \mid \neq$

Fig. 3. The format syntax of protocol packets

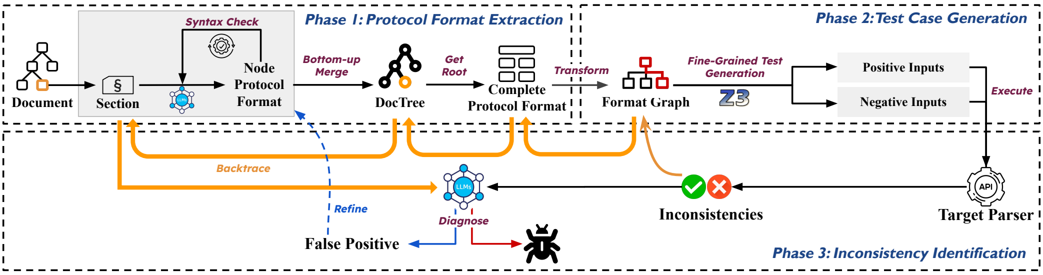


Fig. 4. The pipeline of ParCleanse

or determined by an arithmetic expression relating to some other Primitive-Typed fields. Case Type defines different possible layouts for a field or group of fields, depending on the value of a control field. This structure enables different formats for various message types, assigning a unique structure to each type based on its specific functionality. Without loss of generality, we suppose that the constraints upon the fields in a Struct-Typed object are arithmetic constraints, which are in the form of $f(\mathit{field}^+) \odot 0$. Here f is an arithmetic function and $\odot$ is a comparison operator.

Example. In Figure 2, Ⓒ: "UINT8" is a Primitive Type. TLV is a Struct Type with two fields: Type and Payload, arranged sequentially in memory. 'TLV PacketBody[BodyLength]' defines PacketBody as a sequence of TLVs, occupying a total of BodyLength bytes. Payload is a Case Type controlled by Type. For Type = 0, Payload is an empty struct, representing Pad1; for Type = 6, Payload represents RouterIdTLV.

3.2 Problem Statement

Definition 1 (Network Protocol Parser)**.** A network protocol parser is a function f that maps a network packet p to an element in the set $\{\text{pass}, \text{fail}, \text{crash}\}$. Specifically, $f(p) = \text{pass}$ or fail indicates whether the packet conforms to the protocol format, while $f(p) = \text{crash}$ indicates a failure of the parser when processing p.

Definition 2 (Network Protocol Document and Approximate Format)**.** An RFC document specifies the valid format of protocol packets in natural language. Ideally, we would derive a precise protocol format $\mathcal{F}$ from the RFC document, a function where $\mathcal{F}(p) = \text{pass}$ if a packet p is valid, and fail otherwise. However, due to inaccuracies in automated format extraction, we obtain an approximate format $\mathcal{F}'$ instead of $\mathcal{F}$. This approximation $\mathcal{F}'$ may not fully align with $\mathcal{F}$, and therefore $\mathcal{F}'(p)$ may or may not match $\mathcal{F}(p)$ for any given packet p. Consequently, observing $f(p) \neq \mathcal{F}'(p)$ does not directly imply $f(p) \neq \mathcal{F}(p)$, as the discrepancy could be due to inaccuracies in $\mathcal{F}'$.

Our objective is to identify packets p for which $f(p) \neq \mathcal{F}'(p)$ and to determine if these discrepancies are due to (1) errors in the parser f, or (2) errors in the extracted approximation $\mathcal{F}'$. For case (2), we iteratively refine $\mathcal{F}'$ to more closely align it with $\mathcal{F}$.

4 Design

We present ParCleanse, which automatically extracts complete protocol formats from RFC documents and validates network protocol parsers with traceability-assisted inconsistency identification. As shown in Figure 4, ParCleanse consists of three phases. Specifically, Phase 1 (Section 4.1) extracts the complete protocol format from the RFC document via divide-and-conquer. This is achieved by extracting the protocol format from each section, followed by a bottom-up merge to get the complete protocol format. The DocTree maintains a mapping between document content and extracted format to enable traceability. Phase 2 (Section 4.2) generates fine-grained test cases for the target parser, including both positive and negative inputs created via property-level mutation, which ensures that each protocol property is thoroughly tested against the parser. If an inconsistency is detected between parser executables and the extracted formats, Phase 3 (Section 4.3) identifies the root cause by tracing back to the relevant document section and either reports a bug or diagnoses it as a false positive and refines the format.

4.1 Phase 1: Protocol Format Extraction

As shown in Figure 4, Phase 1 extracts complete protocol formats from RFC documents using a divide-and-conquer approach with a hierarchical DocTree structure. Each document section is represented as a node in the DocTree, with edges capturing hierarchical relationships. Protocol formats are extracted from each node (i.e., RFC section) individually, with syntax checks conducted on generated formats. If invalid, the error is fed back to the LLM for regeneration. These formats are then merged bottom-up within the DocTree, with the root node ultimately representing the complete protocol format. In what follows, we first introduce the DocTree structure and its initial generation in Section 4.1.1. Then, we discuss how DocTree supports the divide-and-conquer strategy for format extraction in Section 4.1.2. Finally, we discuss how traceability is maintained throughout this phase in Section 4.1.3.

4.1.1 DocTree Initialization. DocTree is a hierarchical representation of an RFC document, designed to reflect its structure. It preserves relationships between different sections, essential for combining extracted formats into a complete protocol format.

Definition 3. A DocTree is formally defined as a tuple $\mathcal{T}_{\text{DocTree}} = (N, E)$, where:

- N is a set of nodes, with each node n_i defined as $n_i = (\texttt{content}, \texttt{summary}, \texttt{format})$, representing a specific RFC document section. Each node includes (1) the section content and summary and (2) the protocol format extracted from that section.
- $E \subseteq N \times N$ is a set of directed edges. Each edge $(n_1, n_2) \in E$ represents a hierarchical relationship where n_2 is a subcomponent or dependent section of n_1. These edges preserve the document's structure, supporting the accurate merging of section formats into a complete protocol format.

ParCleanse begins by constructing the initial DocTree using the RFC document's table of contents. However, sections at the same level often have implicit dependencies that are not reflected in the table of contents, which are critical for accurately merging extracted formats. To capture these hidden dependencies, ParCleanse employs a rule-based prompting method. First, ParCleanse prompts LLMs to summarize each section. Then, it re-prompts the LLMs with the summaries of sections at the same hierarchical level, asking them to identify any dependencies between them. The prompts are as follows:

Prompt to generate section summaries

Task: Please summarize a given RFC section: {Section}

Prompt to identify hierarchy dependencies among sections

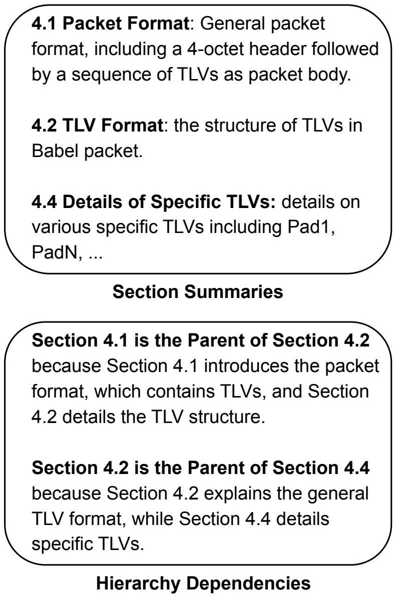
Task: Analyze the hierarchical structure of the following sections in an RFC document: {Section Summaries}
Instructions: Identify Parent-Child relationships where one section provides a detailed breakdown of another.

This method allows ParCleanse to uncover hidden dependencies, ensuring a detailed and accurate DocTree representation for later format merge.

Example 1. For the RFC example in Figure 2 Ⓐ, document sections *"4.1"*, *"4.2"*, and *"4.4"* are listed at the same hierarchy level in the table of contents. *"section 4.1"* introduces the general packet format, *"section 4.2"* details the TLV format (a component of the format specified by *"section 4.1"*), and *"section 4.4"* elaborates on specific TLVs. Therefore, *"section 4.2"* should be a subcomponent of *"section 4.1"*, and *"4.4"* should expand on *"4.2"*. The LLM outputs for section summaries and hierarchy dependencies are listed below in Figure 5. After parsing the LLM's responses, ParCleanse adjusts the DocTree by setting the node representing "*section 4.1*" as the parent node for "*section 4.2*", and "*section 4.2*" as the parent node for "*section 4.4*". By repeating this process across all sections, ParCleanse constructs the DocTree without format.

Algorithm 1: Protocol Formats Extraction using DocTree

```
Input: $\mathcal{T}_{\text{DocTree}}$: DocTree, $DSL$: Protocol format syntax
Output: $F_{\text{complete}}$: Complete Protocol Format
1  Function ProtocolFormatExtraction($\mathcal{T}_{DocTree}$, $DSL$):
2      foreach $n_i \in \mathcal{T}_{DocTree}$ do
3          $F_i \leftarrow$ LLMGenerateFormat($n_i$.content, $DSL$);
4          ($v$, errorMsg) $\leftarrow$ SyntaxChecker($F_i$);
5          while $v == False$ do
6              $F_i \leftarrow$
                 LLMSyntaxRefine($n_i$.content, errorMsg, $DSL$);
7              ($v$, errorMsg) $\leftarrow$ SyntaxChecker($F_i$);
8          $n_i.F \leftarrow F_i$;
9      $F_{\text{complete}} \leftarrow$ MergeFormats($\mathcal{T}_{DocTree}$);
10     return $F_{complete}$;
11 Function MergeFormats($\mathcal{T}_{DocTree}$):
12     foreach $n_i \in \mathcal{T}_{DocTree}$ in bottom-up order do
13         if $n_i$.hasChildren() then
14             $C_{\text{children}} \leftarrow \{(n_c.\text{summary}, n_c.F) \mid$
                 for each child $n_c \in$ children of $n_i\}$;
15             $n_i.F, n_i.\text{summary} \leftarrow$
                 LLMMergeFormats($n_i$.content, $n_i.F$, $C_{\text{children}}$);
16     return $\mathcal{T}_{DocTree}.root.F$;
```



4.1 Packet Format: General packet format, including a 4-octet header followed by a sequence of TLVs as packet body.

4.2 TLV Format: the structure of TLVs in Babel packet.

4.4 Details of Specific TLVs: details on various specific TLVs including Pad1, PadN, ...

Section Summaries

Section 4.1 is the Parent of Section 4.2 because Section 4.1 introduces the packet format, which contains TLVs, and Section 4.2 details the TLV structure.

Section 4.2 is the Parent of Section 4.4 because Section 4.2 explains the general TLV format, while Section 4.4 details specific TLVs.

Hierarchy Dependencies

Fig. 5. LLM Output for Generating Section Summaries and Identifying Hierarchy Dependences

4.1.2 Protocol Format Extraction via Divide-and-Conquer. Algorithm 1 outlines the overall process of protocol format generation, with inputs DocTree ($\mathcal{T}_{\text{DocTree}}$) and protocol format syntax (DSL). Here, DSL is a detailed description for format syntax defined in Section 3.1, guiding LLMs to generate outputs satisfying this syntax.

Node-level Protocol Format Generation (Divide). The protocol format for each DocTree node is generated as described by lines 3–8 in Algorithm 1. First, the content of each node and DSL are passed to the LLM to produce an initial protocol format (line 3). A syntax checker then validates the format (line 4). If syntax errors are found, they are returned to the LLM for refinement (lines 5–7). This cycle of validation and refinement repeats until the format is syntax-correct. Once validated, the final format is saved in the corresponding DocTree node (line 8). This process is repeated for every node in the DocTree (lines 2–8).

Merge Format (Conquer). After generating formats for all nodes, the algorithm merges them into a single protocol format (lines 11–16). The merging process starts from the leaf nodes and moves upwards, following the hierarchy defined by the DocTree (line 12). For each node with child nodes, the algorithm gathers the summaries and formats of its children as pairs (line 14). These pairs, represented by $C_{Children}$ (line 14), along with the content and current format of the node itself, are provided to the LLM to produce a unified format and an updated summary for that node (line 15). Nodes without children retain their original format and summary. This merging process continues until it reaches the root node, whose final format represents the protocol format for the entire document, resulting in a complete DocTree. The prompt used for merging formats is as follows:

> Prompt to merge formats (LLMGenerateFormats)
>
> Task: Merge multiple protocol formats into a single comprehensive format. Current section: {section}; Current format: {format}; Summaries and formats of child nodes: {children}.

Example 2. In the DocTree shown in Figure 2 Ⓑ, "*section 4.4*" serves as the parent to "*section 4.4.1*" through "*section 4.4.7*", each defining a specific `TLV` (Type-Length-Value) format. Each `TLV` begins with a `Type` field, followed by its unique structure. Since the original format for "*section 4.4*" is empty, merging its child formats creates a unified `TLV` structure, shown in Figure 6 (a). The `TLV` struct contains a `Type` field and a `Payload` field whose structure depends on the value of `Type`. By applying this merging strategy iteratively, we obtain a complete DocTree shown in Figure 2 Ⓒ, with the root node containing the complete protocol format.

4.1.3 Traceability between RFC Documents and Extracted Protocol Formats. In this phase, traceability is established by linking each part of the generated protocol format to its corresponding section in the original RFC document. The DocTree structure maintains these links by storing both the section content and its generated format in each node. Even when multiple formats are combined into a single structure (like a Case Type switch), this traceability remains intact. For instance, the `RouterIdTLV` format can be traced directly back to "*section 4.4.7*".

4.2 Phase 2: Fine-Grained Testing by Property-Level Mutation

As shown in Figure 4, Phase 2 transforms the complete protocol format into a Format Graph, where each path represents a valid protocol format. By iterating over each path in the Format Graph, both positive and negative test cases are generated. These test cases are then used to validate the parser executables. In what follows, we first describe how the format is transformed into a Format Graph in Section 4.2.1, and then illustrate how test inputs are generated in Section 4.2.2.

4.2.1 Format Graph Construction. A Format Graph is a Directed Acyclic Graph (DAG) that represents the protocol format. Each Primitive Type (defined in Section 3.1) is represented by a node in the Format Graph. Complex types such as Struct Type and Array Type are represented as subgraphs, which are connected to form a complete Format Graph.

Definition 4. A Format Graph is a tuple $G = (N, S, E)$, where:

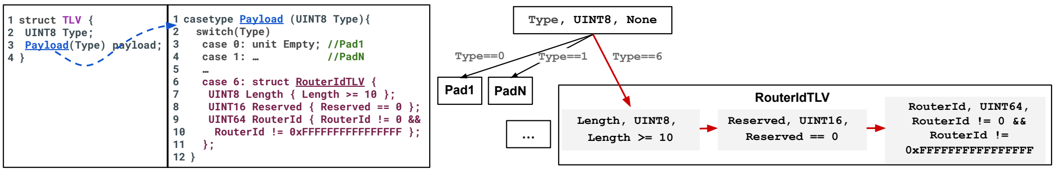
- N is the set of nodes, with each node n_i as a tuple $n_i = (\texttt{name}, \texttt{type}, \texttt{constraint})$, representing a field with a Primitive Type. `constraint` defines the field-level constraint.
- S is the set of subgraphs, where each subgraph $s_i = (N_i, S_i, E_i)$ is a Format Graph and represents a subgraph of the complete Format Graph. Subgraphs are used to represent complex types such as Struct Type, Array Type, and Case Type in protocol formats.
- $E \subseteq (N \cup S) \times (N \cup S)$ is the set of directed edges, allowing connections between nodes, subgraphs, or both. Each edge is a triplet $(x, \texttt{constraint}, y)$, indicating that y can only follow x if certain conditions, defined by `constraint`, are met. If no such condition exists, `constraint` is set to `None`. This `constraint` is a structural constraint that applies primarily to Array Type (to define variable lengths within arrays) and Case Type (to map specific case values to corresponding case formats). It is distinct from the field-level constraints in each node, which apply directly to individual fields.

Each complete path in the Format Graph is an ordered sequence of fields and constraints, representing a valid protocol format. The combinations of these paths define all valid protocol formats.

```
Algorithm 2: Format Graph Generation
   Input: $F_{\text{complete}}$: Complete Protocol Format
   Output: $G = (N, S, E)$: Format Graph
1  Function GenerateFormatGraph($F_{complete}$):
2      return GenerateStructGraph($F_{complete}$.EntryStruct)
3  Function GenerateStructGraph(struct):
4      $N \leftarrow \emptyset, S \leftarrow \emptyset, E \leftarrow \emptyset$, prevnodes $\leftarrow \emptyset$;
5      foreach field ∈ struct.fields do
6          if field is PrimitiveType then
7              $n \leftarrow$ (field.name, field.type, field.constraint);
8              Connect(prevnodes, $n$), update $N$, $E$;
9              prevnodes $\leftarrow \{n\}$;
10         else
11             $s \leftarrow$ GenerateSubgraph(field);
12             Connect(prevnodes, $s$'s start node), update $S$, $E$;
13             prevnodes $\leftarrow \{s$'s end nodes$\}$;
14     return $(N, S, E)$;
15 Function GenerateSubgraph(field):
16     if field is StructType then return GenerateStructGraph(field);
17     else if field is ArrayType then
18         if array length is fixed then
19             return a single subgraph with sequential instances of field.type;
20         else
21             return multiple subgraphs for different possible lengths;
22     else if field is CaseType then
23         return a graph where each case has a subgraph, connected to the control field based on its value;
```

Algorithm 2 constructs the Format Graph from the complete protocol format F_{complete} generated in Phase 1. It begins by calling `GenerateStructGraph` on the entry struct of F_{complete} (lines 1–2). `GenerateStructGraph` constructs the graph for a Struct Type by first initializing empty sets for nodes N, subgraphs S, and edges E (line 4). It then iterates through each field in the struct and builds the graph incrementally (lines 5–13): (1) **Primitive Type**: Each field of Primitive Type is represented as an individual node (lines 6–9). `Connect` adds edges between the current node and all previously processed nodes (line 8), ensuring connectivity within the Format Graph. (2) **Complex types (Struct, Array, or Case types)**: subgraphs are created recursively for each field (lines 11–13). For Struct Type, a subgraph is constructed by recursively calling `GenerateStructGraph`(line 16). For Array Type, if the array length is fixed, a single sequential subgraph is created (lines 18–19). If the length is dynamic (i.e., dependent on other fields), multiple subgraphs are generated to represent different possible lengths (lines 20–21). These subgraphs are connected with the previous nodes via edges that enforce length constraints. For Case Type, each case is represented by a subgraph connected to the node representing the control field (lines 22–23). The edge between the control field node and each case subgraph enforces the constraint that the control field's value must match the corresponding case value. By following the field order in the protocol format, the final Format Graph accurately represents the protocol structure and is ready for test case generation (line 14).

Example 3. The format in Figure 6 (a) is transformed into the Format Graph shown in Figure 6 (b). First, the algorithm creates a node for the `Type` field in the `TLV` struct, including its type (`UINT8`) and constraint (`None`). Since the `Payload` field is a Case Type, it then generates a subgraph for each possible value of the control field `Type`. For example, when `Type` equals 0, an empty subgraph



```
struct TLV {
 UINT8 Type;
 Payload(Type) payload;
}
```

```
casetype Payload (UINT8 Type){
  switch(Type)
   case 0: unit Empty; //Pad1
   case 1: …           //PadN
   …
   case 6: struct RouterIdTLV {
    UINT8 Length { Length >= 10 };
    UINT16 Reserved { Reserved == 0 };
    UINT64 RouterId { RouterId != 0 &&
     RouterId != 0xFFFFFFFFFFFFFFFF };
   };
}
```

Type, UINT8, None
Type==0
Type==1
Type==6
Pad1
PadN
...
RouterIdTLV
Length, UINT8, Length >= 10
Reserved, UINT16, Reserved == 0
RouterId, UINT64, RouterId != 0 && RouterId != 0xFFFFFFFFFFFFFFFF

(a) Protocol Format

(b) Format Graph

Fig. 6. Transform Protocol Format to Format Graph for Test Generation

representing Pad1 is created, as Pad1 has no additional fields. This subgraph is connected to the Type node with the constraint Type == 0. Similarly, when Type is 6, a subgraph for RouterIdTLV is created, containing sequential nodes for length, reserved, and RouterId. This subgraph is connected to the Type node with the constraint Type == 6.

4.2.2 Fine-Grained Test Generation. To thoroughly validate protocol parsers, we first encode each path in the Format Graph into a Z3 formula. Nodes (representing fields) and edges (representing additional constraints) along the path are converted into Z3 expressions, which are combined into an entire formula representing a valid format. Then, for each path formula, both positive and negative inputs are generated:

Positive Inputs. For each path formula, the Z3 SMT solver generates a satisfiable assignment, providing concrete values for the fields. These values are used to construct a binary packet, which is tested against the target parser. If the parser accepts the packet, it is considered temporarily consistent. If the parser rejects it, an inconsistency between the extracted format and the parser implementation is detected. Typically, protocol parsers indicate whether a packet is successfully parsed or rejected using status codes (e.g., returning 0 for success and -1 for failure) or error messages logged through specific functions. Otherwise, we annotate the expected endpoint of successful parsing. If the parser reaches this point, the packet is considered successfully parsed; otherwise, an early exit indicates rejection.

Negative Inputs. Negative inputs are generated by **property-level mutations** on positive inputs. **Format properties** can be divided into two categories: field-level properties, which define constraints on individual fields, and structure-level properties, which address the overall format and structure of the packet. Hence, we apply two types of mutations: (1) Field-level mutation: This mutation negates one field value at a time against its constraint, keeping other field values unchanged. Constraints on edges (from Case Type or Array Type) are not negated in this case, as this would lead to a different path in the Format Graph. (2) Structural mutation: This mutation adds or removes specific fields or bytes to violate the structural properties of the packet (e.g., adding or removing bytes to violate the length constraint in Array Type, or adding/removing a field from the packet). Mutations are applied by negating the constraint corresponding to the selected property while adding additional constraints to enforce that all other field values remain unchanged. The modified Z3 formula is then solved again using the Z3 SMT solver to generate a mutated packet. If the target parser rejects the mutated packet as expected, the extracted format aligns with the parser implementation. Otherwise, if the parser accepts the mutated packet, this reveals an inconsistency between the LLM-extracted format and the parser implementation.

Example 4. For the path highlighted in red in Figure 6 (b), the Z3 formula is:

$$0 \leq \mathtt{Type} \leq 2^8 - 1 \land \mathtt{Type} = 6 \land 0 \leq \mathtt{Length} \leq 2^8 - 1 \land \mathtt{Length} \geq 10 \land 0 \leq \mathtt{Reserved} \leq 2^{16} - 1$$

$$\land\, \mathtt{Reserved} = 0 \land 0 \leq \mathtt{RouterId} \leq 2^{64} - 1 \land \mathtt{RouterId} \neq 0 \land \mathtt{RouterId} \neq 0xFFFFFFFFFFFFFFFF$$

In this formula, Type is constrained between 0 and $2^8 - 1$ because it is of type UINT8. Similar for other variables. Solving this Z3 formula produces a positive input: Type = 6, Length = 10, Reserved = 0, `RouterId` = 1. The parser accepts the binary packet constructed from this assignment, indicating temporary consistency. Next, negative test cases are generated by mutating the positive test case. When we negate the constraint `Length` ≥ 10, the Z3 formula is:

$$
\begin{aligned}
&0 \le \texttt{Type} \le 2^8 - 1 \land \texttt{Type} = 6 \land 0 \le \texttt{Length} \le 2^8 - 1 \land \neg(\texttt{Length} \ge 10) \land 0 \le \texttt{Reserved} \le 2^{16} - 1\\
&\land \texttt{Reserved} = 0 \land 0 \le \texttt{RouterId} \le 2^{64} - 1 \land \texttt{RouterId} \ne 0 \land \texttt{RouterId} \ne 0xFFFFFFFFFFFFFFFF\\
&\land \texttt{Type} = 6 \land \texttt{Reserved} = 0 \land \texttt{RouterId} = 1
\end{aligned}
$$

Solving this Z3 formula produces a negative input: `Type` = 6, `Length` = 0, `Reserved` = 0, `RouterId` = 1. This negative input changed the value of `Length` to 0, while keeping other field values unchanged. The parser correctly rejects it, confirming its adherence to the specification on this field-level property. However, when we set `RouterId` to 0 — violating the format constraint: `RouterId` ≠ 0 — the parser still accepts the packet. This indicates an inconsistency, as the parser does not enforce this constraint on `RouterId` required by the LLM-extracted protocol format.

4.3 Phase 3: Traceability-Assisted Inconsistency Identification

As shown in Figure 4, Phase 3 identifies each inconsistency. When an inconsistency is detected during testing, there are two potential causes: an error in the LLM-extracted protocol format (as the input is generated based on this format) or a bug in the implementation. Therefore, inconsistencies should not be immediately treated as implementation bugs. An additional validation step is needed to identify the source of the inconsistency by cross-referencing relevant sections of the RFC document. To achieve this, we backtrace the inconsistency to its corresponding RFC section in the document, retrieve that section, and prompt the LLM to diagnose it with the following prompt:

> Prompt to identify inconsistencies
>
> Task: {Constraint} is allowed by [myformat/parser] but not by [myformat/parser]. According to the RFC section: {Section}, identify whether myformat or parser is correct, and provide evidence from the RFC section.

This process helps determine whether the inconsistencies are caused by mistakes in the extracted format or bugs in the implementation. If the LLM validator identifies an inconsistency as an implementation error, we report it as a bug. On the other hand, if it is identified as a format extraction mistake, we use this feedback to further refine the extracted format.

Traceability between Inconsistencies and RFC Documents. Each inconsistency is directly tied to a specific constraint or set of constraints. Since traceability between the format and the corresponding RFC section is maintained, any inconsistency can be traced back to the exact section of the RFC where the relevant constraint is defined. This traceability-based approach ensures that every step, from extracting the format to finding bugs, is linked to the original RFC document.

Example 5. Consider the inconsistency regarding `RouterId` introduced in Example 4. This constraint comes from the subgraph `RouterId` shown in Figure 6 (b), which is derived from `RouterIdTLV` in the LLM-extracted protocol format shown in Figure 6 (a). `RouterIdTLV` corresponds to RFC document "*section 4.4.7*" (Figure 2 Ⓑ). So we backtrace and retrieve the content of RFC "*section 4.4.7*" and provide it to the LLM validator to identify the root cause of the inconsistency. The LLM validator determines that it is a bug in the implementation instead of the extracted format. So we report the bug to the developers. It is confirmed and now fixed as shown in Figure 1.

5 Evaluation

We implement our tool using OpenAI's GPT-4o API [35] and the Z3 SMT solver (version 4.11.2) [13]. We choose GPT-4o for its strong natural language understanding and DSL grammar extraction capabilities [28], which are essential for interpreting protocol specifications from RFC documents. Additionally, GPT-4o has shown strong performance in software testing [7, 17], including protocol

Table 1. Protocol Dataset and Ground Truth Format.

Protocol	Dataset				Ground Truth Format					
	RFC (Pages)	Repo.	Lang.	Description	Msg Type	Field	Indep. Constr.	Dep. Constr.	LoC	Time
BABEL	8966 (54)	FRR	C	Distance-vector routing protocol	11	97	66	8	171	5h
BFD	5880 (49)	FRR	C	Bidirectional forwarding detection	6	43	22	2	69	2h
BGP-4	4271 (104)	FRR	C	Border Gateway Protocol 4	4	62	46	6	164	6h
IPv4	791 (45)	Go Net	Go	Internet protocol v4	1	14	3	1	16	0.5h
ICMPv4	791, 792 (66)	Go Net	Go	Internet Control Message Protocol for IPv4	11	73	27	3	106	2h
ICMPv6	8200, 4443 (66)	Go Net	Go	Internet Control Message Protocol for IPv6	6	29	14	0	66	2h
IPv6	8200 (42)	Impacket	Python	Internet protocol v6	1	8	1	0	11	0.5h
DHCP	2131 (45)	Impacket	Python	Dynamic host configuration protocol	1	16	4	1	21	0.5h
TCP	793 (85)	Impacket	Python	Extensible Authentication Protocol	1	23	6	4	43	2h

validation [56]. We set the temperature to 0 for minimal randomness and better reproducibility. For protocol format extraction (Section 4.1), we use everparse [39] as the syntax checker. To evaluate the effectiveness of our approach, we conduct experiments to address the following research questions.

- **RQ1**: How accurate are the message formats extracted from the RFC documents?
- **RQ2**: How effective is ParCleanse in inconsistency identification and bug detection?
- **RQ3**: How effective is ParCleanse compared to existing approaches?
- **RQ4**: How effective is each component of ParCleanse?

5.1 Dataset

To evaluate ParCleanse's capability to support multiple programming languages, we construct a new protocol dataset, as existing datasets [31, 58] only include protocol implementations in C or C++. We filter GitHub repositories related to network protocol implementations with over 2,000 stars and actively maintained. To increase diversity in implementations, we select three repositories, each using a different programming language: C, Go, and Python. Based on these criteria, we choose FRR [12], Go Networking [2], and Impacket [4]. For each repository, we select three protocols with RFC documents longer than 40 pages. In total, we get nine different protocols with a maximum document length of 104 pages and an average of 62 pages per protocol. For each protocol, we download the corresponding RFC documents from the IETF DataTracker [3] as input for ParCleanse. For each protocol implementation, we built the parsing executables, as our tool does not require access to source code. This makes our method applicable to both source-available and source-unavailable scenarios. The protocol and codebase information is available in Table 1.

5.2 RQ1: Effectiveness of ParCleanse on Message Format Extraction

5.2.1 Setup and Metrics. To evaluate the accuracy of protocol formats generated by ParCleanse, we first establish a ground truth for network protocol formats. Two authors, each with more than three years of expertise in network protocols, independently reviewed the relevant RFC documents and manually wrote the input formats. They tracked the time taken to complete the formats for each protocol, compared their results, discussed inconsistencies, and reached a consensus.

To quantify the correctness of extracted formats, we define format metrics across five element types: Message Types, Field Names/Types, Independent Constraints, and Dependent Constraints, as introduced in Section 3.1. Table 1 lists the count of each element type for each protocol's ground truth format (column *"Ground Truth Format"*), the line count for each (column *"LoC"*), and the average manual time spent labeling the ground truth (column *"Time"*). Column *"Field"* in column *"Ground Truth Format"* represents the count for Field Names and Field Types. Since the number of field names and types is identical, we merge them into a single column. The statistics in the table reflect the complexity of protocol formats. For example, BABEL has 11 message types, 97 fields, and 74 constraints (66 independent, 8 dependent), totaling 171 LoC, and takes an average of 5 hours to manually construct the ground truth format, indicating high complexity. Simpler protocols like IPv4 and IPv6 have fewer fields and constraints, and require less time to manually write the ground

Table 2. ParCleanse **Protocol Format Extraction Results: Precision/Recall(%).**

Protocol	Msg Type	Field Name	Field Type	Indep. Constr.	Dep. Constr.
BABEL	100/100	100/91	95/87	98/86	100/75
BFD	100/100	100/100	100/100	100/100	-/0
BGP-4	100/100	95/95	94/94	98/63	100/50
IPv4	100/100	100/100	100/100	100/33	100/100
ICMPv4	100/100	100/100	100/93	100/96	100/33
ICMPv6	100/100	94/100	90/97	100/100	-/-
IPv6	100/100	100/100	100/100	100/100	-/-
DHCP	100/100	100/100	94/94	83/75	0/0
TCP	100/100	100/78	83/65	67/33	0/0
Total	100/100	99/95	94/91	98/82	73/44

truth formats. We then compare ParCleanse 's extracted formats with ground truth, measuring precision (the proportion of correct extracted elements) and recall (the proportion of ground truth elements accurately captured) for each element type. For example, for Message Types, the precision and recall are calculated as follows:

$$\text{Precision} = \frac{\text{Correct Message Types in Extracted Format}}{\text{Total Message Types in Extracted Format}}, \quad \text{Recall} = \frac{\text{Ground Truth Message Types Covered by Extracted Format}}{\text{Total Message Types in Ground Truth Format}}.$$

Since the extracted element count or ground truth element count can be zero for each element type, making precision or recall undefined, we use "-" to indicate these cases. Comparing the ground truth with the formats extracted by ParCleanse takes an estimated fifteen minutes per protocol.

5.2.2 *Results.* The results are shown in Table 2. ParCleanse achieves 100% precision and recall in extracting message types across all protocols. Field names and field types are also accurately identified, achieving overall precision and recall of 99%/95% for field names and 94%/91% for field types. Independent constraints are well-detected across the nine protocols, with a precision of 98%. In terms of recall, the tool identified 155 out of 189 ground truth independent constraints, resulting in an average recall of 82%. Dependent constraints are particularly challenging, as they require understanding how different fields influence one another, often through implicit rules or cross-references within the protocol specification. These complexities make accurate extraction difficult for LLMs. Overall, ParCleanse achieves a high precision of 73%. For the ICMPv6 and IPv6 protocols, since there is no dependent constraint in either the ground truth (Table 1) or the LLM-extracted format, the precision and recall are both denoted with "-". For BFD, since the LLM-extracted format contains no dependent constraints, its precision is denoted with "-". Among these protocols, TCP shows a drop in performance due to the handling of the `DataOffset` field, which involves header alignment and bit-level offsets. These implicit constraints are not straightforward to extract from natural language descriptions, leading to reduced accuracy when interpreting the precise field layout in the TCP header. Further discussion can be found in Section 7.

Conclusion. ParCleanse excels in extracting protocol formats from RFC documents, achieving over 90% precision and recall for most elements, with high precision on challenging element types.

5.3 RQ2: Effectiveness of Inconsistency Identification and Bug Detection

5.3.1 *Setup and Metrics.* We evaluate ParCleanse on both inconsistency identification and bug detection. We create one positive test case per format path, which already achieves great performance. Generating multiple cases per path could enhance bug detection, but would significantly increase the manual effort required to verify inconsistencies. Thus, we use a single representative case per path, with negative test cases determined by property counts along the path. Also, for Array Type with variable length, we generate test cases containing zero and one element.

Table 3. ParCleanse **Results on Inconsistency (Incons.) Identification and Bug Detection**

Protocol	Detected Incons.		Identified Incons.			Correctly Identified Incons.			Identification Acc (%)				# Bugs		
			Impl. Error		Format Error	Impl. Error		Format Error	Impl. Error		Format Error	Overall			
	Logical	Crash	Logical	Crash		Logical	Crash		Logical	Total			Unique	New	Confirmed
BABEL	27	0	23	0	4	23	0	4	100	100	100	100	23	23	23
BFD	12	0	10	0	2	10	0	2	100	100	100	100	10	10	10
BGP-4	3	0	0	0	3	0	0	3	100	100	100	100	0	0	0
IPv4	2	0	2	0	0	2	0	0	100	100	100	100	2	2	0
ICMPv4	16	0	16	0	0	16	0	0	100	100	100	100	16	16	0
ICMPv6	10	0	8	0	2	8	0	2	100	100	100	100	8	8	0
IPv6	1	2	1	2	0	1	2	0	100	100	100	100	2	1	1
DHCP	4	2	4	2	0	3	2	0	75	100	100	83	5	5	2
TCP	11	0	11	0	0	9	0	0	82	100	100	82	3	3	0
Total	86	4	75	4	11	72	4	11	97	100	100	97	69	68	36

Inconsistency Identification. We first record the number of inconsistencies detected by ParCleanse in Phase 2 (Section 4.2), including logical inconsistencies (e.g., the extracted constraint mismatches with the implementation) and crashes. In Phase 3 (Section 4.3), ParCleanse classifies each inconsistency as either an *Implementation Error* or a *Format Extraction Error* (a mistake in the extracted format), with crashes always classified as *Implementation Errors*. We manually check each classification to assess the inconsistency identification accuracy of ParCleanse. It takes an estimated five minutes to check each inconsistency.

Bug Detection. For each inconsistency classified as an *Implementation Error*, we record whether it is a logical or crash issue. We report the number of *unique* and *new* (i.e., previously unknown) bugs detected, as well as the *confirmed* bugs.

5.3.2 Results. The results of inconsistency identification and bug detection are in Table 3.

Inconsistency Identification. As shown in column *"Detected Incons."*, ParCleanse detects a total of 90 (86 logical and 4 crash) inconsistencies across all protocols. In column *"Identified Incons."*, 79 (75 logical and 4 crash) of the 90 inconsistencies are identified as *Implementation Errors*, and 11 as *Format Extraction Errors*. Our manual evaluation confirms that ParCleanse correctly identifies 87 inconsistencies: 76 (72 logical and 4 crash) of 79 *Implementation Errors* and all 11 *Format Extraction Errors*, achieving an overall 97% accuracy and 100% accuracy for seven protocols. Since crashes are always *Implementation Errors*, the identification accuracy for crashes is 100%. Among 86 detected logical inconsistencies, 83 (72 logical *Implementation Errors*, 11 *Format Extraction Errors*) are accurately classified, achieving 97% accuracy, highlighting ParCleanse 's effectiveness in distinguishing between implementation and format extraction errors.

Bug Detection. As shown in column *"Correctly Identified Incons."*, 72 logical and 4 crash issues detected by ParCleanse are true bugs, including 69 unique bugs. This demonstrates ParCleanse's ability to detect diverse bugs through fine-grained test generation, which thoroughly covers the input space and behaviors. Notably, 68 of these bugs are new, underscoring ParCleanse's effectiveness. We reported them to the developers, with 36 confirmed so far and others pending review. ParCleanse generated a PoC (Proof of Concept) for each bug. Of the 36 confirmed bugs, 17 are fixed and merged, 18 have approved pull requests pending merge, and 1 remains unresolved.

Conclusion. ParCleanse demonstrates strong effectiveness in inconsistency identification and bug detection, achieving 97% identification accuracy. It successfully detects 68 new bugs, with 36 confirmed, across nine protocols. Each bug is generated with a PoC.

5.4 RQ3: Comparative ParCleanse with Baseline Methods

5.4.1 Setup and Metrics. We compare ParCleanse with two baseline methods to further evaluate ParCleanse's format extraction performance and bug detection effectiveness.

ChatAFL [31]. We compare ParCleanse's input format extraction capability with a state-of-the-art LLM-based testing tool, ChatAFL, which also uses LLMs to retrieve protocol formats. To ensure a fair comparison, all experiments involving LLMs use GPT-4o as our approach.

Table 4. ParCleanse vs. Baseline on Protocol Format Extraction: Precision/Recall (%).

Approach	Msg Type	Field Name	Field Type	Indep. Constr.	Dep. Constr.
ParCleanse	**100/100**	**99/95**	**94/91**	**98/82**	**73/44**
ChatAFL [31]	89/55	78/35	-/-	81/10	-/0

Table 5. ParCleanse vs. Baseline on Bug Detection for BABEL, BFD, and BGP-4.

Approach	Unique	New	Confirmed
ParCleanse	**33**	**33**	**33**
ParDiff [58]	4	1	1

Table 6. The precision (%) and recall (%) of ablations

Approach	Msg Type	Field Name	Field Type	Indep. Constr.	Dep. Constr.
ParCleanse	**100/100**	**99/95**	**94/91**	**98/82**	73/**44**
ParCleanse - refine	**100/100**	**99**/94	93/89	88/68	**85/44**
ParCleanse - refine - divide and conquer	97/71	91/54	65/37	79/26	0/0

ParDiff [58]. We compare ParCleanse to ParDiff, a state-of-the-art differential testing tool to detect network protocol parsing bugs. ParDiff, built on LLVM, requires access to source code and only supports parsers written in C. In contrast, ParCleanse does not require source code and can work directly with parser executables. Although this makes the comparison unfair, we still compare bug detection results by running ParDiff on the BABEL, BFD, and BGP-4 protocols, as it does not support the Python and Go implementations used in six other protocols. We do not compare bug detection with ChatAFL, as ChatAFL only detects crashes.

5.4.2 Results. Table 4 presents the protocol format extraction comparison between ParCleanse and ChatAFL. ChatAFL struggles to capture accurate protocol formats, partly because it lacks support for specifying individual field types (column *"Field Type"*), a crucial feature in binary protocol formats. Additionally, ChatAFL struggles with complex constraints (e.g., dependencies) and is limited to expressing basic constraints, such as concrete values. These limitations greatly reduce its effectiveness in accurately extracting network protocol formats.

Table 5 shows the comparison bug detection results of ParCleanse and ParDiff. ParCleanse detects 33 unique bugs in the FRR project, all of which are new. In contrast, ParDiff detects only 4 unique bugs in the FRR project, including only 1 new bug. This is due to two main factors: first, many bugs are shared across both tested implementations, which ParDiff cannot detect through differential analysis. Second, ParDiff halts bisimulation on each FSM path after the first state mismatch, missing bugs in further state transitions along that path.

Conclusion. ParCleanse achieves high performance on protocol format extraction and bug detection, outperforming state-of-the-art methods.

5.5 RQ4: Ablation Studies

To evaluate the effectiveness of each ParCleanse design in mitigating LLM hallucinations, we conduct ablation studies to assess the impact of traceability-assisted format refinement (Section 5.5.1) and divide-and-conquer strategy (Section 5.5.2) on protocol format extraction, as well as the impact of traceability on inconsistency identification (Section 5.5.3) and fine-grained testing on inconsistency and bug detection (Section 5.5.4).

5.5.1 Format Extraction without Traceability-Assisted Format Refinement. We evaluate the impact of traceability-assisted format refinement (Section 4.3) on the extracted protocol format, with results presented in Table 6. Specifically, we compare the protocol format after refinement (Phase 3) in row "ParCleanse" to the format without refinement (after Phase 1) in row *"ParCleanse - refine"*.

Overall, refinement improves the quality of extracted protocol formats, increasing the precision and recall of independent constraints (column *"Indep. Constr."*) by 11% (88% to 98%) and 21% (68% to 82%). This demonstrates that traceability-assisted format refinement effectively mitigates LLM

Table 7. Impact of and Fine-Grained Testing and Traceability-Assisted Inconsistency Identification

Approach	Incons. Identification Accuracy			# Incons.		# Bugs		
	Logical	Format Error	Total	Logical	Crash	Unique	New	Confirmed
ParCleanse	**97%**	**100%**	97%	75	4	**69**	68	36
ParCleanse- traceability	84%	0%	84%	75	4	-	-	-
ParCleanse- fine grained testing - traceability	4%	0%	90%	22	211	4	4	0

hallucinations by correcting inaccurately generated constraints. However, ParCleanse ’s precision of dependent constraints slightly decreases after refinement. This is because, without refinement, fewer dependent constraints are generated with high precision (85%). After refinement, the model identifies two additional missing constraints but produces inaccurate formulas due to the complexity of dependencies, leading to a minor precision drop. Despite this, the refinement effectively enhances the overall accuracy and completeness of the extracted protocol formats.

5.5.2 Format Extraction without Divide-and-Conquer. To evaluate the effectiveness of the divide-and-conquer approach (Section 4.1), we compare the format extraction performance in Phase 1 (Table 6 row "ParCleanse- *refine*") with the performance by directly feeding the entire RFC document into the LLM (with unnecessary lines removed to fit within the model's input window) in row "ParCleanse- *refine - divide and conquer*". The results reveal a consistent and significant performance drop across all five element types when using the full document without divide-and-conquer. For example, field name recall decreases from 94% to 54%, and field type recall drops from 89% to 37%. Extraction of constraints suffers even more, with independent constraint recall falling by 55% (from 68% to 26%) and all dependent constraints being missed. These findings highlight that the divide-and-conquer strategy effectively improves the accuracy of protocol format extraction.

5.5.3 Inconsistency Identification without Traceability-Assisted Inconsistency Identification. We conduct an ablation study to evaluate the impact of traceability-assisted inconsistency identification (Section 4.3). Instead of using the corresponding RFC section, the LLM is provided with the entire RFC document for each inconsistency detected in Phase 2. The results are shown in Table 7 row "ParCleanse- *traceability*". Without traceability, the LLM fails to identify any *Format Extraction Errors*, causing accuracy in this category to drop from 100% to 0%. Additionally, the overall accuracy decreases from 97% to 84%, underscoring the importance of traceability-assisted inconsistency identification in effectively distinguishing implementation errors from format extraction errors.

5.5.4 Inconsistency and Bug Detection without Fine-Grained Testing. We conduct an ablation study to assess the impact of fine-grained testing. In this setup, the full format produced in Phase 1 (Section 4.1) is encoded as a single Z3 formula, used for generating 50 positive and 100 negative test cases (both more than the amount that ParCleanse generates for each protocol). These cases are then executed against the target parser to obtain parsing results. Since inputs are generated based on the entire format, individual fields and document sections are unknown during the diagnosis step (Section 4.3). Therefore, in this setting, during diagnosis, the LLM is provided with binary input and the complete RFC document (instead of only the relevant RFC section supported by traceability) to help locate the source of any inconsistencies. We then compare the performance of inconsistency identification accuracy and bug detection of this approach against ParCleanse.

The comparison results are shown in Table 7 row "ParCleanse- *fine grained testing - traceability*". Without traceability-assisted inconsistency identification, accuracy in identifying real implementation errors drops sharply from 97% to 4%, highlighting that, without traceability, LLMs struggle to accurately detect logical inconsistencies due to hallucinations. By linking inconsistencies to specific

RFC sections, traceability provides essential context to mitigate hallucinations and enhance inconsistency identification accuracy. Column *"#Incons."* lists the number of inconsistencies detected by each tool, while *"#Bugs"* shows the number of detected bugs. Without fine-grained property-level mutation, most detected issues are crashes (211 out of 233), with only 4 unique bugs identified. This indicates that inputs generated based on the entire format fail to *comprehensively* test the input space. In contrast, ParCleanse, using fine-grained testing, detected 75 logical inconsistencies and 4 crashes, with 69 unique bugs detected. This demonstrates that fine-grained test generation enables thorough input space exploration, triggering more logical inconsistencies and unique bugs.

6 Threats to Validity.

An internal threat to validity is the manually established ground truth (Section 5.2). Human error or bias in defining input formats may impact evaluation accuracy. To address this, two experts independently draft the formats, compare results, and resolve discrepancies to reach a consensus [54, 59].

An external threat is the potential for bugs in the RFC documents. Our method assumes the document is of high quality and treats it as the ground truth. In fact, protocol documents are generally regarded as reliable oracles [5, 30]. Additionally, in our experiments, we did not observe any document bugs. Another threat is potential data leakage. Since LLMs are pretrained on vast datasets, the model may have seen the documents we tested, which could undermine the effectiveness of the divide-and-conquer approach for protocol format extraction (Section 4.1). To mitigate this, we compare with a baseline where the entire document is fed directly to GPT-4o (Section 5.5.2). These experiments validate the effectiveness of our divide-and-conquer design.

7 Limitations and Future Work

LLMs in Extracting Complex Protocol Formats. ParCleanse uses LLMs (e.g., GPT-4o) to interpret RFC documents. While LLMs generally perform well when extracting clearly stated protocol formats, they struggle with implicit details like padding in the TCP header. For instance, RFC 793 [1] specifies that "the TCP header padding is used to ensure that the TCP header ends and data begins on a 32-bit boundary", but it does not explicitly describe how padding is applied or calculated. Correctly constructing the padding field requires reasoning about alignment constraints between header size, data offset, and structure layout, which are only implicitly conveyed in the document. As a result, LLMs struggle to accurately infer these relationships, leading to hallucinations. Future work could enhance format extraction by fine-tuning LLMs on domain-specific datasets, developing rule-based post-processing for implicit formats, and incorporating self-verifying mechanisms.

ParCleanse in Handling Non-standardized Documentation. ParCleanse extracts protocol formats from structured RFCs, as most protocols have Standards Track RFCs as official standards. For protocols without RFC documentation, relying on non-standardized or incomplete sources is often unreliable. Future work could integrate alternative sources (e.g., technical manuals, legacy parsers) and human-in-the-loop feedback to improve adaptability.

8 Related Work

Conventional Fuzzing. Conventional network protocol fuzzing (e.g., BooFuzz [36] and SAGE [19]) identifies bugs by exposing crashes. Netlifter [42] combines conventional fuzzing with static analysis. It leverages static symbolic analysis to collect path constraints and generate varied test cases for fuzzing. However, they still rely on crashes as oracles to detect bugs.

Differential Analysis. Differential analysis [26, 40, 58] has been widely used for bug detection. Existing differential analysis finds bugs by comparing multiple implementations. Static differential analysis tools locate semantic differences by comparing models derived from these implementations. For example, ParDiff [58] compares protocol formats derived from different parser implementations to find inconsistencies. Dynamic differential testing techniques like DPIFuzz [38] feed different implementations with the same input and compare their execution behaviors. While these static and dynamic approaches can detect semantic bugs (i.e., silent violations of protocol rules), they cannot identify bugs present in both implementations, a limitation inherent to differential analysis. To address this, ParCleanse compares each implementation directly with its official specification, enabling the detection of such overlooked bugs.

Traditional Static Analyzers and Model Checking. Static analyzers (e.g., Pistachio [46]) detect protocol-specific bugs by checking implementations against data-dependent rules, while model checking [10, 15, 34] verifies behavior using formal models. Both methods require extensive manual effort to define rules or models, as protocols are typically described in natural language. ParCleanse overcomes this by using LLMs to automatically extract protocol formats from RFC documents.

LLM-based Approaches. LLMs have shown effectiveness across various domains in software engineering, including code generation [16, 25, 27, 60], software testing [14, 22, 41, 55], program analysis [20, 23, 24, 48, 49], comment/specification generation [18, 47, 51, 54], and automated repair [21, 44, 52, 57]. For network protocol testing, LLM-based methods construct protocol models automatically to assist fuzzing. ChatAFL [31] leverages LLMs' prior knowledge to generate protocol formats, but only works well on simple and well-known protocols, as it does not directly learn the format details from the document content. mGPTFuzz [29] extracts finite-state machines (FSMs) from RFCs but lacks detailed protocol formats and symbolic constraints. To bridge these gaps, ParCleanse uses a divide-and-conquer approach to extract comprehensive protocol formats from RFCs, covering message types, field names, field types, independent field constraints, and dependent constraints, enabling more accurate protocol modeling and testing. LLMIF [50] also leverages LLMs to extract formats from documents for test case generation. But unlike ParCleanse, which queries LLMs only when an inconsistency is detected between the protocol parser's output and the expected format, LLMIF queries LLMs for every test case. Additionally, LLMIF is tailored to Zigbee protocols and relies heavily on Zigbee-specific structures like clusters, commands, and ZCL frames. This makes it challenging to adapt to general protocols (lack of these structures) without significant modification. In contrast, ParCleanse is designed for general protocol testing.

9 Conclusion

This work proposes ParCleanse to automatically validate network protocol implementations (in various programming languages) using protocol format specifications extracted from RFC documents. ParCleanse accurately extracts formats through a divide-and-conquer approach and thoroughly tests protocol parsers with fine-grained testing. ParCleanse also supports traceable inconsistency identification, allowing each inconsistency to be traced back to the original document section for accurate diagnosis. Our experiments show that ParCleanse extracts protocol formats precisely, outperforming ChatAFL. ParCleanse detects 69 bugs in total, demonstrating the potential for automated software validation from natural language specifications.

Acknowledgment

We thank all the anonymous reviewers for the insightful 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, 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. Chengpeng Wang is the corresponding author.

References

[1] 1981. RFC 793 - Transmission Control Protocol. https://www.rfc-editor.org/rfc/rfc793.html.

[2] 2024. Go Networking. https://github.com/golang/net.

[3] 2024. IETF DataTracker. https://datatracker.ietf.org.

[4] 2024. Impacket. https://github.com/fortra/impacket.

[5] 2024. Internet Standard. https://en.wikipedia.org/wiki/Internet_Standard.

[6] Fernando Arnaboldi. 2023. XDiFF. https://github.com/IOActive/XDiFF.

[7] Cristian Augusto, Jesús Morán, Antonia Bertolino, Claudio de la Riva, and Javier Tuya. 2024. Software System Testing Assisted by Large Language Models: An Exploratory Study. In *Testing Software and Systems (ICTSS '2024)*. Springer-Verlag, 239–255. doi:10.1007/978-3-031-80889-0_17

[8] babeld. 2024. babeld. https://github.com/jech/babeld.

[9] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. 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 (OSDI '08)*. USENIX, 209–224. https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems

[10] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. 2003. Modular Verification of Software Components in C. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. IEEE, 385–395. doi:10.1109/ICSE.2003.1201217

[11] Sze Yiu Chau, Omar Chowdhury, Md. Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. In *IEEE Symposium on Security and Privacy (S&P '17)*. IEEE, 503–520. doi:10.1109/SP.2017.40

[12] FRR community. 2024. The FRRouting protocol suite. https://github.com/FRRouting/frr.

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08, Vol. 4963)*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[14] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. 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 (ISSTA '23)*. ACM, 423–435. doi:10.1145/3597926.3598067

[15] Gregorio Díaz, Fernando Cuartero, Valentín Valero Ruiz, and Fernando L. Pelayo. 2004. Automatic verification of the TLS handshake protocol. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*. ACM, 789–794. doi:10.1145/967900.968063

[16] Yangruibo Ding, Marcus J Min, Gail Kaiser, and Baishakhi Ray. 2024. Cycle: Learning to self-refine the code generation. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 392–418. doi:10.1145/3649825

[17] Fatemeh Erfan, Mohammad Yahyatabar, Martine Bellaiche, and Talal Halabi. 2024. Advanced Smart Contract Vulnerability Detection Using Large Language Models. In *2024 8th Cyber Security in Networking Conference (CSNet '24)*. IEEE, 289–296. doi:10.1109/CSNet64211.2024.10851734

[18] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2023. An Empirical Study on Using Large Language Models for Multi-Intent Comment Generation. *arXiv preprint arXiv:2304.11384* (2023). doi:10.48550/ARXIV.2304.11384

[19] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44. doi:10.1145/2093548.2093564

[20] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. 2025. RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing. *arXiv preprint arXiv:2501.18160* (2025). doi:10.48550/ARXIV.2501.18160

[21] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE '23)*. IEEE, 1430–1442. doi:10.1109/ICSE48619.2023.00125

[22] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE '23)*. IEEE, 2312–2323. doi:10.1109/ICSE48619.2023.00194

[23] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 111 (2024), 26 pages. doi:10.1145/3649828

[24] Haonan Li, Hang Zhang, Kexin Pei, and Zhiyun Qian. 2025. The Hitchhiker's Guide to Program Analysis, Part II: Deep Thoughts by LLMs. *arXiv preprint arXiv:2504.11711* (2025). doi:10.48550/ARXIV.2308.00245

[25] Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Wei Jiang, Hongwei Chen, Chengpeng Wang, and Gang Fan. 2024. RepoGenix: Dual Context-Aided Repository-Level Code Completion with Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 2466–2467. doi:10.1145/3691620.3695331

[26] Congyu Liu, Sishuai Gong, and Pedro Fonseca. 2023. 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 '23)*. ACM, 427–441. doi:10.1145/3575693.3575731

[27] Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. 2024. Evaluating Language Models for Efficient Code Generation. In *First Conference on Language Modeling (COLM '24)*. https://openreview.net/forum?id=IBCBMeAhmC

[28] My M. Mosthaf and Andrzej Wasowski. 2024. From a Natural to a Formal Language with DSL Assistant. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS '24)*. ACM, 541–549. doi:10.1145/3652620.3687811

[29] Xiaoyue Ma, Lannan Luo, and Qiang Zeng. 2024. From One Thousand Pages of Specification to Unveiling Hidden Bugs: Large Language Model Assisted Fuzzing of Matter IoT Devices. In *Proceedings of the 33rd USENIX Conference on Security Symposium (USENIX Security '24)*. USENIX, 4783–4800. https://www.usenix.org/conference/usenixsecurity24/presentation/ma-xiaoyue

[30] Stephen McQuistin, Mladen Karan, Prashant Khare, Colin Perkins, Gareth Tyson, Matthew Purver, Patrick Healey, Waleed Iqbal, Junaid Qadir, and Ignacio Castro. 2021. Characterising the IETF through the lens of RFC deployment. In *Proceedings of the 21st ACM Internet Measurement Conference (IMC '21)*. ACM, 137–149. doi:10.1145/3487552.3487821

[31] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS '24)*. The Internet Society. doi:10.14722/ndss.2024.24556

[32] MITRE. 2022. CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[33] MITRE. 2024. CWE-20: Improper Input Validation. https://cwe.mitre.org/data/definitions/20.html.

[34] Madanlal Musuvathi and Dawson R. Engler. 2004. Model Checking Large Network Protocol Implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI '24)*. USENIX, 155–168. http://www.usenix.org/events/nsdi04/tech/musuvathi.html

[35] OpenAI. 2024. GPT-4o. https://platform.openai.com/docs/models/gpt-4o.

[36] Joshua Pereyda. 2023. BooFuzz. https://github.com/jtpereyda/boofuzz.

[37] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *Proceedings of the 28th USENIX Conference on Security Symposium (USENIX Security '19)*, Nadia Heninger and Patrick Traynor (Eds.). USENIX, 1465–1482. https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud

[38] Gaganjeet Singh Reen and Christian Rossow. 2020. DPIFuzz: a differential fuzzing framework to detect DPI elusion strategies for QUIC. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC '20)*. ACM, 332–344. doi:10.1145/3427228.3427662

[39] Microsoft Research. 2020. everparse. https://project-everest.github.io/everparse/3d-lang.html.

[40] Richard Rutledge and Alessandro Orso. 2022. Automating Differential Testing with Overapproximate Symbolic Execution. In *2022 15th IEEE Conference on Software Testing, Verification and Validation (ICST '22)*. IEEE, 256–266. doi:10.1109/ICST53961.2022.00035

[41] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proc. ACM Softw. Eng.* 1, FSE, Article 43 (2024), 21 pages. doi:10.1145/3643769

[42] Qingkai Shi, Junyang Shao, Yapeng Ye, Mingwei Zheng, and Xiangyu Zhang. 2023. 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 '23)*. ACM, 1287–1301. doi:10.1145/3576915.3616614

[43] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, 693–706. doi:10.1145/3192366.3192418

[44] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An Empirical Study of Deep Learning Models for Vulnerability Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE '23)*. IEEE, 2237–2248. doi:10.1109/ICSE48619.2023.00188

[45] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. 2022. Hardening attack surfaces with formally proven binary format parsers. In *43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 31–45. doi:10.1145/3519939.3523708

[46] Octavian Udrea and Cristian Lumezanu. 2006. Rule-Based Static Analysis of Network Protocol Implementations. In *Proceedings of the 15th Conference on USENIX Security Symposium (USENIX Security '06)*. USENIX. https://www.usenix.

org/conference/15th-usenix-security-symposium/rule-based-static-analysis-network-protocol

[47] Chengpeng Wang, Jipeng Zhang, Rongxin Wu, and Charles Zhang. 2024. DAInfer: Inferring API Aliasing Specifications from Library Documentation via Neurosymbolic Optimization. *Proc. ACM Softw. Eng.* 1, FSE (2024), 2469–2492. doi:10.1145/3660816

[48] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024. LLMDFA: Analyzing Dataflow in Code with Large Language Models. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems (NeurIPS '24)*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/ed9dcde1eb9c597f68c1d375bbecf3fc-Abstract-Conference.html

[49] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, and Xiangyu Zhang. 2024. Sanitizing Large Language Models in Bug Detection with Data-Flow. In *Findings of the Association for Computational Linguistics (EMNLP '24)*. Association for Computational Linguistics, 3790–3805. doi:10.18653/v1/2024.findings-emnlp.217

[50] Jincheng Wang, Le Yu, and Xiapu Luo. 2024. LLMIF: Augmented Large Language Model for Fuzzing IoT Devices. In *2024 IEEE Symposium on Security and Privacy (S&P '24)*. IEEE, 881–896. doi:10.1109/SP54263.2024.00211

[51] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification (CAV '24)*. Springer, 302–328. doi:10.1007/978-3-031-65630-9_16

[52] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. ACM, 1282–1294. doi:10.1145/3597926.3598135

[53] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ACM, Article 126, 13 pages. doi:10.1145/3597503.3639121

[54] Danning Xie, Byungwoo Yoo, Nan Jiang, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S Lee. 2023. Impact of Large Language Models on Generating Software Specifications. *arXiv preprint arXiv:2306.03324* (2023). doi:10.48550/ARXIV.2306.03324

[55] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. 8, OOPSLA2 (2024). doi:10.1145/3689736

[56] Zhe Yang, Hao Peng, Yanling Jiang, Xingwei Li, Haohua Du, Shuhai Wang, and Jianwei Liu. 2025. ChatHTTPFuzz: large language model-assisted IoT HTTP fuzzing. *International Journal of Machine Learning and Cybernetics* (2025), 1–22. doi:10.1007/s13042-024-02527-3

[57] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*. ACM, 1592–1604. doi:10.1145/3650212.3680384

[58] Mingwei Zheng, Qingkai Shi, Xuwei Liu, Xiangzhe Xu, Le Yu, Congyu Liu, Guannan Wei, and Xiangyu Zhang. 2024. ParDiff: Practical Static Differential Analysis of Network Protocol Parsers. In *Proc. ACM Program. Lang. (OOPSLA '24)*. ACM, 1208–1234. doi:10.1145/3649854

[59] Mingwei Zheng, Jun Yang, Ming Wen, Hengcheng Zhu, Yepang Liu, and Hai Jin. 2021. Why Do Developers Remove Lambda Expressions in Java?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE '21)*. IEEE, 67–78. doi:10.1109/ASE51524.2021.9678600

[60] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024). doi:10.48550/ARXIV.2406.11931

Received 2024-10-31; accepted 2025-03-31