Generating Precise Format Specification for Network Protocols Through Adversarial LLM Interactions

Hengdi Ye, Bing Shui, Jielun Wu, Yufan Zhou, Baowen Xu, Qingkai Shi*
State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing 210023, China

Abstract

This paper aims to address a significant gap in inferring precise format specifications for network protocols, particularly regarding the exact constraints on network packet fields. That is, while current methods can often derive accurate syntactic structures that divide a network packet into multiple fields, they typically struggle to capture the semantic dependencies or constraints among those fields. To address this issue, our central insight is that large language models (LLMs) have proven effective across various tasks, and adversarial interactions among these tasks can significantly alleviate the hallucination problem. Based on the insight, we propose a novel approach to generating precise protocol formats by adversarially combining LLM-based specification inference and code generation. By inputting the structured RFC documents of network protocols into LLMs, we generate both packet formats and reference packet parsers that iteratively refine one another to reduce hallucinations: the packet formats allow us to create a variety of network packets for parser testing, while runtime checks in the parsers help identify format errors. Our evaluation of 8 protocols shows a significant increase in precision and recall, achieving a 326% improvement over the state of the art in inferring field constraints. Furthermore, the improved format specifications enable effective fuzzing, leading to the discovery of 24 zero-day vulnerabilities in widely used protocol implementations.

1 Introduction

Network protocols serve as the language for communication across a wide range of systems. The format specification of a network protocol outlines the structure of network messages (a.k.a. packets) in the language, detailing (1) the *syntactic structure*, i.e., how a message is composed of multiple fields, and (2) the *semantic dependencies*, i.e., constraints among the fields. Such format specifications are essential for many security tasks, including network fuzzing [21,39,51,80], intrusion detection [17,23,24,40], and verification [2–5], etc. As such, a thorough understanding of protocol formats is crucial for maintaining robust network security.

Existing Work. Despite decades of research, deriving precise format specifications remains a significant challenge. One line of existing work takes protocol implementations, either source code [53,55] or compiled binaries [6,7,13,27,33,34,66], as the primary input and applies dynamic or static program analysis to infer format specifications. These techniques, although they leverage rich implementation logic to aid format inference, inevitably inherit flaws from imperfect protocol implementations. As illustrated in §2, if an implementation itself omits a critical semantic constraint among fields, the specification inferred from the implementation will also lack that constraint and thus be blind to the very bug it was meant to uncover. To mitigate this implementation bias, differential analysis that cross-validates multiple independent implementations of the same protocol can reveal inconsistent constraints [56,77]. However, it requires multiple independent protocol implementations, which may be difficult to obtain in practice, and, like other code-based techniques, must be reengineered for different programming languages, incurring substantial practical challenges and human effort.

To mitigate the adverse effects of buggy implementations, a second line of work incorporates modern large language models (LLMs) to directly analyze official RFC documents, which, despite being written in natural language, constitute the authoritative reference for protocol implementations [19,52,63,78]. These approaches either solely depend on the RFC documents or combine RFC analysis with the aforementioned code-based analysis. We observe that while such techniques are generally effective at inferring syntactic structure (e.g., identifying packet fields), they remain inadequate at capturing semantic constraints among fields due to LLM hallucinations (*Note: While strict hallucinations refer to fabrications, for brevity, we use "hallucination" to refer to all possible mistakes made by LLMs, such as omitting critical information*). For instance, ParCleanse achieves nearly 100% precision and recall in inferring syntax but fails to infer any

*Corresponding Author. Email: qingkaishi@nju.edu.cn

cross-field constraints for many protocols [78]. This limitation substantially reduces the effectiveness of downstream security analyses and may permit deeply hidden vulnerabilities to propagate into deployed products.

Our Approach. This paper presents a novel LLM-driven approach, SPAR, that automatically generates format specifications that are precise in both syntax and semantics from RFC documents alone. Our key insight is twofold. First, LLMs have demonstrated strong capabilities across a wide range of tasks, including specification inference [31, 36, 48, 79] and code generation [16, 38, 42, 62]. Second, as evidenced by our experiments, it is unlikely that two distinct LLM-driven tasks will independently produce the same hallucinated errors (*Note: we do not assume one of them must yield a correct result, but argue that hallucinations often occur when LLM lacks sufficient or reliable input information, making it more likely to produce diverse errors across different tasks even with the same inputs*). Consequently, adversarial interactions between these tasks can gradually improve their results and, eventually, mitigate hallucination errors.

Specifically, SPAR feeds RFC documents into an LLM and guides the LLM to produce both a format specification and a reference packet parser. These two artifacts then iteratively refine each other to reduce hallucinations: the inferred format specification enables the generation of diverse network packets for parser testing, while runtime checks in the parser expose inconsistencies and errors in the specification. This adversarial refinement process continues until reaching a fixed point, where the format specification and the reference parser are mutually consistent and compatible. At this stage, we consider the resulting specification to be of high quality, as no further refinement is possible.

During the adversarial interactions, we design three optimizations: (1) path-cover-guided packet generation, which synthesizes a minimal set of packets tailored to traverse a specific set of path segments within a graph structure such as the control-flow graph, (2) linear-negation-guided packet generation, which generates negative packets by negating a single of many constraints that a packet should satisfy, reducing exponential constraint combinations to linear, and (3) slicing-guided RFC documents identification, which utilizes program slicing to find relevant RFC sections such that we can supply minimal yet sufficient RFC sections to the LLM. The first two enable us to generate a linear (rather than exponential) number of network packets, thereby improving the efficiency of both packet generation and LLM interactions. The third further reduces LLM hallucinations by identifying the necessary and sufficient RFC sections to refine the inferred formats or parsers. In short, our approach offers the following advantages, compared to the state of the art:

- SPAR decouples the format inference procedure from protocol implementations, thereby avoiding the risk of inheriting errors from buggy implementations.

Packet $p \in$ Struct Type

Struct Type StructType := { field$^+$ }

Field field ::= Type field_identifier { f(field_identifier+) };
(Type field_identifier: Syntactic Structure; { f(field_identifier+) }: Semantic Dependencies)

Type Type ::= PrimitiveType | ArrayType | CaseType | StructType | ParametricType

Primitive Type PrimitiveType ::= Uint8 | Uint16 | Uint32 | Uint64 | ⋯

Array Type ArrayType ::= Type[const] | Type[h(field_identifier$^+$)]

Case Type CaseType ::= switch(field_identifier) {(case const: Type)$^+$}

Parametric Type ParametricType ::= type_identifier(field_identifier)

Type Def TypeDef ::= typedef Type type_identifier

Identifier field_identifier, type_identifier ∈ String

Constant const ∈ Uint64

Function f ∈ Predicate, h ∈ ArithmeticFunction

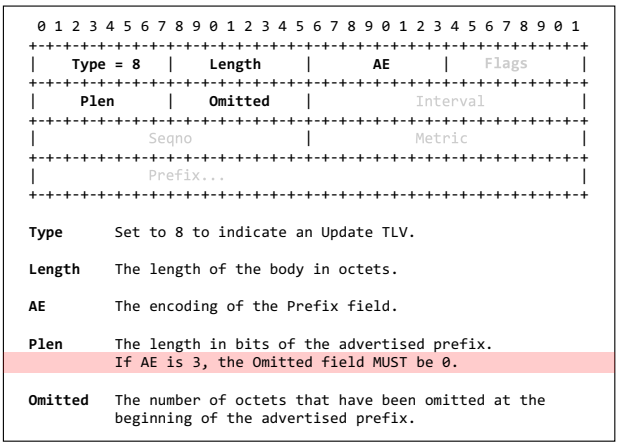
Figure 1: The format specification of a protocol packet

- SPAR relies solely on the RFC documents and, unlike code analysis, does not require reimplementation for protocols implemented in different programming languages.
- SPAR leverages adversarial refinement to produce format specifications that are precise in both syntax and semantics, substantially enhancing security analyses.

Contributions. We make the following contributions:

- We introduce adversarial LLM interactions, a framework that reduces LLM hallucinations by enabling mutual refinement between complementary LLM-based tasks.
- We instantiate this framework to generate protocol format specifications that are precise in both syntax and semantics, along with a reference parser.
- We implement our approach as a tool, namely SPAR, and generate precise format specifications for eight network protocols. Experimental results demonstrate that the specifications we generate achieve close-to-100% precision and recall, significantly outperforming the state of the art. Furthermore, we apply these specifications to protocol fuzzing, uncovering 24 zero-day vulnerabilities in existing protocol implementations, some of which have remained latent for up to 9 years. SPAR has been made publicly available [71].

Organization. The remainder of this paper is organized as follows. §2 provides background knowledge of protocol format and motivates our approach. §3 shows the basic workflow of our approach and discusses the technical challenges. §4 details our approach, and §5 discusses the evaluation results. §6 discusses the limitation and future directions of this work, §7 surveys related work, and §8 concludes this paper.



```
 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 = 8   |    Length     |       AE      |     Flags     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Plen      |    Omitted    |           Interval            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Seqno             |            Metric             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Prefix...                                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Type      Set to 8 to indicate an Update TLV.

Length    The length of the body in octets.

AE        The encoding of the Prefix field.

Plen      The length in bits of the advertised prefix.
          If AE is 3, the Omitted field MUST be 0.

Omitted   The number of octets that have been omitted at the
          beginning of the advertised prefix.
```

Figure 2: A section from RFC 8966.

2 Background and Motivation

This section first defines the format specification of network packets (§2.1) and then presents a real-world protocol bug to illustrate the limitation of existing methods (§2.2).

2.1 Protocol Format Specification

Network communication relies on protocols to govern the data, known as packets, exchanged over networks. The format specification defines the syntactical structure of a network packet and the semantic dependencies among packet fields. Conventionally, such specifications are written in natural language, as in RFC documents. While RFCs provide human-readable context for developers, they often lack mathematical rigor, leading to potential ambiguities in edge cases. To ensure zero-error implementation and many automated security tasks, we often have to translate RFCs' textual descriptions into a "formal" format specification in a domain-specific language, such as the 3D language [50] defined in Figure 1.

In particular, a network packet p is a structure composed of distinct units known as the fields. Each field is defined by a triple, including the field type, the field identifier (or name), and the constraint a field should satisfy. The constraint specifies the semantic dependencies among fields as a predicate (a function returning a Boolean value) over one or more fields. The type of a field could be a primitive type, an array type, a case type, a structure type, or a parametric type. A primitive type UintN means an N-bit unsigned integer. A case type returns a specific type depending on the value of a particular field. A parametric type takes a field as a parameter, meaning that the type's layout relies on the parameter.

2.2 Motivating Example

Figure 2 presents a segment from RFC 8966 [9], the official specification for BABEL, a routing protocol. This segment

```
struct {
    UINT8 Type { Type == 8 }
    UINT8 Length;
    UINT8 AE { AE >= 0 && AE <= 3 };
    ...
    UINT8 Plen { Plen <= 248 };
    UINT8 Omitted {
        (AE == 3 && Omitted == 0) ||
        (AE != 3 && Omitted <= ((Plen + 7) / 8))
    };
    ...
    UINT8[(Plen+7)/8-Omitted] Prefix;
};
```

Figure 3: A rigorous formal format specification.

```
1.  void parse_packet(char *packet) {
2.    int type = packet[0];
3.    ......
4.    if (type == MESSAGE_UPDATE /*8*/) {
5.      ......
6.
7.  +   // Check the semantic dependency: AE == 3 && Omitted == 0
8.  +   if (packet[2] == 3 && packet[5] != 0) {
9.  +       debugf("Received IPv6 update with non-zero Omitted.");
10. +       goto fail;
11. +   }
12.
13.     ......
```

Figure 4: A buggy implementation of BABEL from the FR-Routing Protocol Suite and our fix in green.

defines both the syntax and semantics of BABEL's "Update" packet. Specifically, the syntax part specifies that a packet consists of a list of fields, including 8-bit fields such as `Type`, `Length`, and `AE` (Address Encoding), as well as 16-bit fields such as `Interval` and `Seqno`, among many others. The semantic part specifies the constraints these fields must satisfy. For instance, the value of `Type` must be 8 for an "Update" packet, and if `AE` equals 3, `Omitted` must be zero.

Although it is easy for humans to read, reliance on natural language often introduces ambiguity and implicit context. For instance: (1) the valid value range of `AE` and `Plen` is omitted in this segment but refers to earlier text; and (2) a dependency between `AE` and `Omitted`, highlighted in red, is not specified in the description of the two fields, but in that of `Plen`. In contrast, Figure 3 illustrates the formal format written in the language defined before. In the formal specification, the inter-dependencies among fields are formally specified as logical formulas, such as $\texttt{AE} = 3$ `&&` $\texttt{Omitted} = 0$.

The example above shows how a precise format specification can make hidden constraints explicit. Real implementations, however, do not always honor such constraints. Figure 4 shows a buggy implementation of BABEL from FR-Routing [10], a popular open source routing protocol suite for Linux and Unix platforms. The buggy implementation misses the check for the inter-dependency between `AE` and `Omitted`, i.e., $\texttt{AE} = 3$ `&&` $\texttt{Omitted} = 0$, which are added in lines 7-11 by our fix. Because BABEL uses these two fields, together with the `Prefix` field, to compute valid network addresses, this check is critical for routing networks that rely on BABEL to construct correct routing tables. Omitting this check

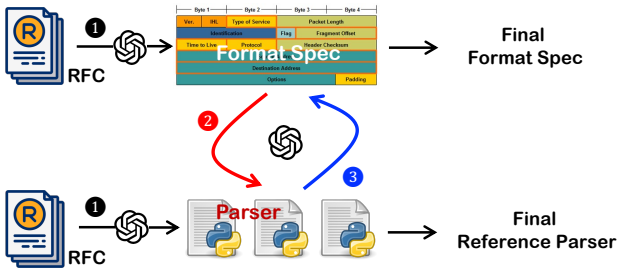


Figure 5: The workflow of SPAR.

can result in incorrect address computation, causing routers to forward packets along suboptimal or congested paths and even enabling traffic-based denial-of-service attacks to critical network services.

Detecting such bugs is difficult using traditional static analysis [54, 60] or fuzzing [20, 74], as they do not involve memory corruption and rarely manifest apparent runtime symptoms such as crashes. Consequently, many existing approaches first try to establish a format specification and then validate it against the implementation. In this context, a format specification, especially one that accurately captures field interdependencies, is essential for identifying such bugs. However, as discussed in §1, prior work remains inadequate at producing format specifications with high-quality semantic dependencies, such as the one between `AE` and `Omitted` in the motivating example.

3 SPAR in a Nutshell

As shown in Figure 5, starting from (1) an initial format specification and an initial parser generated from RFC documents by LLM, we alternately (2) refine the parser using the format specification as an oracle and (3) refine the format specification using the parser as an oracle. Before diving into the details, let us recall two key insights that make this adversarial workflow effective:

- **Insight 1.** LLMs have demonstrated strong capabilities across many tasks, including specification inference, e.g., [31, 36, 48, 79] and code generation, e.g., [38, 42, 62];
- **Insight 2.** It is unlikely that two distinct LLM tasks independently produce identical hallucinated errors, since hallucinations typically arise from ambiguous or underspecified inputs.

The first insight enables us to generate a relatively high-quality initial format specification and parser; the second substantially mitigates hallucinations, allowing us to produce format specifications with precise semantic constraints on packet fields, where the state of the art falls short.

Step 1: Initializing the Format Specification and Parser. Our approach starts with an initial format specification and

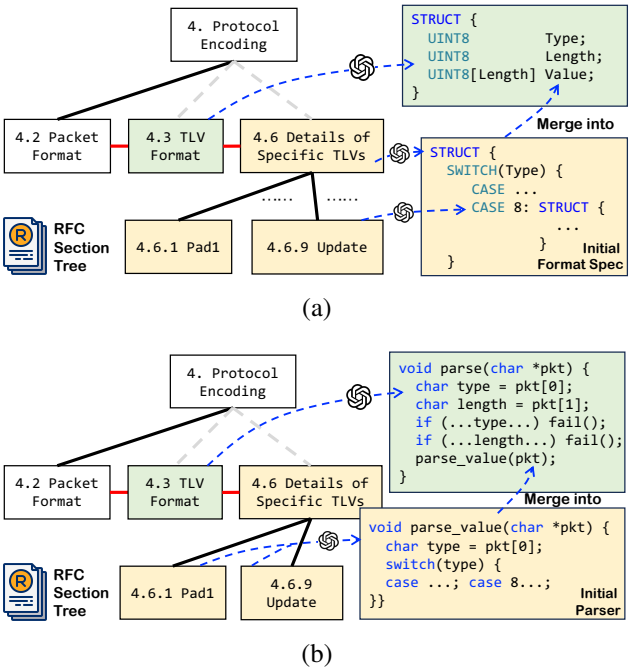


Figure 6: Step 1: Building the initial (a) format and (b) parser.

a parser generated independently by LLM-based interpretation of RFC documents. Basically, it follows the previous work [78] to leverage a divide-and-conquer strategy that (1) divides an RFC document into multiple small sections, (2) interprets each small section to build part of the format specification and part of the parser, and (3) eventually merges these small parts into a complete format and parser. The divide-and-conquer strategy allows the LLM to focus on relevant RFC sections and avoid attentional distraction, thereby reducing hallucinations and producing the initial format and parser.

For example, Figure 6 shows a brief workflow that leverages LLM to generate the initial format specification from RFC 8966, the BABEL RFC. First, the RFC is organized as a tree, with each node corresponding to a section in the table of contents. For example, the tree nodes correspond to the RFC sections 4, 4.2, 4.3, 4.6, 4.6.1, and 4.6.9, respectively. More RFC sections are omitted to keep the figure clear. Second, the LLM interprets each section into a sub-format and adjusts the hierarchy among these sub-formats based on its understanding of the RFC. For example, in Figure 6, the tree node for RFC section 4.6 becomes the child of the tree node for 4.3, and 4.3 becomes the child of 4.2. Third, the sub-formats are merged in bottom-up order, as per the tree, to form a complete format. The procedure of generating an initial parser follows a similar workflow.

While this step initiates our approach, we do not regard it as our key contribution, as it builds on existing work [78]. Thus, we will not detail it in the next section due to the space limit. Notably, the initial format can miss the semantic constraint between `AE` and `Omitted`, i.e., `AE` = 3 `&&` `Omitted` = 0, which

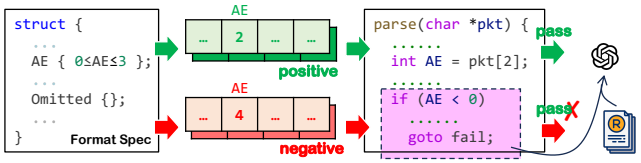


Figure 7: Step 2: Refining parser using format as the oracle.

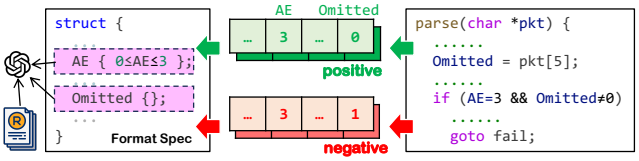


Figure 8: Step 3: Refining format using parser as the oracle.

will be added back via the adversarial LLM interactions in Steps 2 and 3, as the parser contains this constraint.

Steps 2 & 3: Adversarial Refinement between Format & Parser. The adversarial refinement comprises two directions: one refines the parser according to the format specification, and the other refines the format based on the parser. Figure 7 shows how we refine the parser using the format specification as the oracle. Specifically, we extract packet constraints from the format and use the Z3 constraint solver [14] to generate positive and negative packets: positive packets satisfy all specified constraints, while negative packets violate at least one constraint. These packets are used to test the parser. If the parser accepts (rejects) a negative (positive) packet, we identify the relevant code fragments and corresponding RFC sections and provide them to the LLM. LLM is expected to recheck whether the parser contains errors and refine the parser as needed.

In the opposite direction, as illustrated in Figure 8, we refine the format specification using the parser as an oracle. Generating packets from a parser is more challenging than generating them from a format specification because the parser's code structure is more complex. We apply hybrid fuzzing [59, 73, 75], which combines gray-box fuzzing with symbolic execution, to generate packets that the parser accepts or rejects. If an accepted (rejected) packet violates (satisfies) the format specification, we identify the relevant specification fragments and corresponding RFC sections and provide them to the LLM. LLM is expected to recheck whether the format specification is incorrect and, if necessary, refine it.

For the motivating example, as illustrated in Figure 8, the hybrid fuzzer generates a negative packet (where `AE` = 3 `&&` `Omitted` = 1) rejected by the parser. Checking the rejected packet against the format specification, we find that it unexpectedly satisfies all constraints in the format specification, indicating that the format specification does not consider it invalid and thus misses constraints. Consequently, the fields `AE` and `Omitted` are identified in the format specification, which, together with the relevant RFC sections, are sent to LLM for refinement. As a result, we get the correct format specification after the LLM-based refinement.

During the LLM-based refinement, we need to address two challenges to improve our approach's performance: one related to efficiency and the other to LLM hallucinations.

(1) Challenge 1: Exponential Number of Packets. Generating positive and negative packets is central to our approach, but the number of possible packets grows exponentially with the number of constraints in a format specification. For example, for a packet format with n fields, each with m possible types or constraints, enumerating all possible combinations of the fields yields m^n packets. Enumerating all such packets is computationally infeasible. Thus, we propose a practical solution based on *path cover* and *linear negation* that enables efficient yet effective packet generation.

(2) Challenge 2: Attentional Distraction. When prompting an LLM to refine a format specification or parser, our goal is to modify only the components suspected to be faulty. Accordingly, the LLM input should include *all and only* the relevant fragments of the corresponding RFC sections. Providing either an incomplete or the entire RFC would dilute the LLM's focus, increase the risk of hallucinations, and potentially introduce unintended changes to correct components of the format specification or parser. In the following section, we present a practical solution that maps RFC sections to format and parser elements and leverages *program slicing* to precisely isolate the portions requiring refinement.

Step 4: Uncovering the Bug in Motivating Example. As discussed above, the adversarial refinement lets us generate a format specification that includes a correct constraint between the two fields: `AE` = 3 `&&` `Omitted` = 0. With this correct format specification, we have chances to uncover the bug in Figure 4 via fuzzing. Specifically, similar to Step 2, we generate both positive and negative packets to fuzz the buggy BABEL implementation. For example, a negative packet that violates the semantic constraint is nonetheless accepted by the buggy implementation (since it does not have any check on this constraint), thereby exposing the bug in Figure 4.

4 Approach in Detail

Our key contribution lies in the adversarial LLM interactions, which iteratively refine the format specification and the parser using the counterpart as the oracle. This process continues until a fixed point is reached, at which the format specification and the reference parser are mutually consistent. At this stage, we consider the resulting specification to be of high quality, as no further refinement is possible. As shown in Figures 7 and 8, no matter in which direction, the procedure consists of two basic components, one for generating positive and negative

packets (§4.1) and the other for LLM-based refinement with the RFC documents (§4.2).

4.1 Packet Generation

In adversarial LLM interactions, we respectively use the format and the parser as oracles to generate positive and negative protocol packets. We discuss the basic steps (§4.1.1 & §4.1.2) and their optimizations (§4.1.3 & §4.1.4) below.

4.1.1 Generating Packets as per the Format

To ease the discussion of the packet generation algorithm and the follow-up optimizations, we first transform a format specification (defined by the language in Figure 1) into a graph structure, referred to as the packet format graph (PFG).

Definition 1 (Packet Format Graph (PFG))**.** A packet format graph $G = (V, E)$ is a directed acyclic graph, where:

- Each vertex $v \in V$ denotes a packet field, which is a triple (τ, ω, ϕ) consisting of (1) the field type τ, (2) the field identifier (or name) ω, and (3) the field constraint ϕ;
- Each edge $(v_1, v_2) \in E \subseteq V \times V$ represents the sequential order of fields in a packet.

As the 2nd line in Figure 1, a format specification can be regarded as an unconstrained and anonymous struct type, i.e., $(\tau \in \mathsf{StructType}, \_, \mathsf{true})$. Thus, we can use $\mathsf{PFG}(\tau, \omega, \phi)$ to denote the PFG of both a field and the whole specification. Specifically, we use $G_1 \bowtie G_2$ to mean connecting each exit vertex of G_1 to each entry vertex of G_2, where an entry vertex is a vertex without incoming edges, and an exit vertex does not have any outgoing edges. Meanwhile, we use $G_1 \uplus G_2$ to mean a simple union of two graphs. As such, the following inductive rules define how we recursively build a PFG:

1. $\mathsf{PFG}(\tau, \omega, \phi) :=$ a vertex containing (τ, ω, ϕ)
   where $\tau \in \mathsf{PrimitiveType} \wedge \phi$ is an atomic constraint
2. $\mathsf{PFG}(\tau, \omega, \phi_1 \vee \phi_2) := \mathsf{PFG}(\tau, \omega, \phi_1) \uplus \mathsf{PFG}(\tau, \omega, \phi_2)$
3. $\mathsf{PFG}(\tau, \omega, \phi_1 \wedge \phi_2) := \mathsf{PFG}(\tau, \omega, \phi_1) \bowtie \mathsf{PFG}(\tau, \omega, \phi_2)$
4. $\mathsf{PFG}(\tau, \omega, \phi) = \mathsf{PFG}(\tau_0, \omega{::}\omega_0, \phi \wedge \phi_0) \bowtie \mathsf{PFG}(\tau_1, \omega{::}\omega_1, \phi_1) \bowtie \ldots$, **where** $\mathsf{StructType}\ \tau := (\tau_i, \omega_i, \phi_i)_{i=0}^{n}$
5. $\mathsf{PFG}(\tau, \omega, \phi) = \mathsf{PFG}(\tau', \omega{::}0, \phi) \bowtie \mathsf{PFG}(\tau', \omega{::}1, \mathsf{true}) \bowtie \ldots$
   where $\mathsf{ArrayType}\ \tau := \tau'[\mathsf{const}]$
6. $\mathsf{PFG}(\tau, \omega, \phi) = \biguplus_i \mathsf{PFG}(\tau_i, \omega{::}i, \phi \wedge \varpi = c_i)$,
   where $\mathsf{CaseType}\ \tau := \mathsf{switch}(\varpi)\{\mathsf{case}\ c_1 : \tau_1; \ldots\}$

The first is the base rule: if a field is of a primitive type and an atomic constraint (which does not contain any connectives $\wedge$ or $\vee$), we create a single vertex containing the field information. The second and third rules create a union and a connection of two PFGs for the disjunctive and conjunctive constraints, respectively. The fourth and fifth rules state that for a field of a struct or array type, we create a PFG for each element and connect these PFGs in order. In the two rules, the constraint ϕ for the whole struct or array type is retained in the first sub-PFG. Other sub-PFGs have only their own constraints for the field they represent. We consider only constant-length array types here; variable-length array types are discussed later. Finally, the sixth rule deals with the case type: a field can have different types; for each type, we create a PFG and union them. The constraint of each sub-PFG contains the constraints for each case, i.e., $\varpi = c_i$. We omit the rule for parametric types because we can always eliminate them by inlining. In the final graph, each path specifies a possible layout of an actual packet.

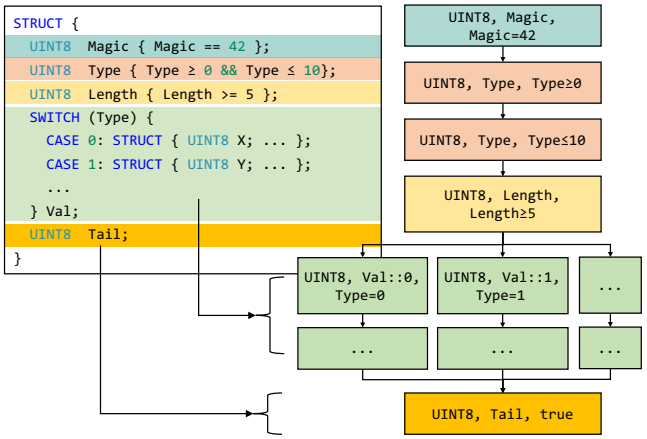


Figure 9: From format specification into PFG.

Example 1. Figure 9 shows the process of transforming a simple format specification to an equivalent PFG. The format specification comprises four fields, each in a different color, corresponding to the four sub-PFGs. As per Rule (4), the sub-PFGs are connected sequentially. Particularly, the Type field is constrained by a conjunctive constraint, thus having two vertices in the same path according to Rule (3): one vertex for the constraint $\texttt{Type} \geq 0$ and the other for the constraint $\texttt{Type} \leq 10$. The `Val` field is of a case type, thus corresponding to a few parallel sub-PFGs as per Rule (6), each with a case-constraint, e.g., $\texttt{Type} = 0$ and $\texttt{Type} = 1$. □

SPAR then enumerates all PFG paths, each representing a possible layout of an actual network packet. During the path traversal, SPAR collects the constraint ϕ_i from each vertex and computes their conjunction $\phi = \wedge_i \phi_i$. To generate positive packets, SPAR directly applies the constraint ϕ to an SMT solver, e.g., Z3 [14], which will generate valid values for each field. To generate negative packets, SPAR negates some ϕ_i and then sends the constraints to the SMT solver.

Example 2. Consider the PFG in Figure 9. The left-most path of the PFG denotes a packet satisfying the constraint $\texttt{Magic} = 42 \wedge \texttt{Type} \geq 0 \wedge \texttt{Type} \leq 10 \wedge \texttt{Length} \geq 5 \wedge \texttt{Type} = 0$ where each field is an 8-bit unsigned integer. Sending the

whole constraint to the SMT solver yields a positive packet where, for example, `Magic` $= 42$, `Length` $= 5$, `Type` $= 0$.

To generate a negative packet, we negate a constraint, e.g., the constraint for the `Magic` field: **Magic** $\neq 42 \wedge$ `Type` $\geq 0 \wedge$ `Type` $\leq 10 \wedge$ `Length` $\geq 5 \wedge$ `Type` $= 0$. In result, a negative packet, whose `Magic` field may be 43, will be generated. □

Variable-Length Array Type. The previous discussion (e.g., Rule (5) for building PFG) does not consider variable-length array types and assumes array lengths are constant. Nonetheless, this issue can often be addressed by a simple extension of the algorithm discussed above, because the length, although not constant, can often be precomputed based on constraints collected from preceding vertices. Since protocol formats are typically designed to be parser-friendly for efficiency, the length can usually be determined from the values of preceding vertices.

Example 3. Figure 3 shows the format specification of the BABEL protocol, which includes a variable-length field, namely `UINT8[(Plen+7)/8-Omitted] Prefix`. The length of this field is constrained by the values of `Plen` and `Omitted`. When generating packets by traversing a PFG path, we invoke an SMT solver over the accumulated constraints upon reaching this field to compute possible values of `Plen` and `Omitted`. As a result, we can compute a possible constant length $c =$ `(Plen+7)/8-Omitted`. We then unfold the array c times, treating it as a constant-length array, as specified in Rule (5). □

4.1.2 Generating Packets as per the Parser

Parser-based packet generation is similar to format-based generation discussed above, but differs in that the underlying structure is the parser's control-flow graph rather than a PFG. This shift introduces additional challenges, as the parser code may contain complex constructs such as pointers, loops, and global variables. To generate packets corresponding to different paths in the control-flow graph, a traditional solution is symbolic execution [8, 28]. To improve scalability, we exploit the fact that, unlike declarative format specifications, parsers are executable. We therefore adopt hybrid fuzzing [25, 59, 73, 75], which combines symbolic execution with fuzzing, to systematically explore feasible parser paths and generate packets for each path. Under this approach, positive packets are those accepted by the parser, while negative packets are rejected during parsing.

4.1.3 Optimization I: Positive Packets via Path Cover

As discussed above, we traverse the paths in a packet format graph (PFG, when using the format specification as the oracle) or a control flow graph (CFG, when using the parser as the oracle) to collect constraints and generate positive packets. It is well known that the number of paths in a directed graph is often exponential, known as the path-explosion problem, which causes significant performance issues:

- The enumeration of an exponential number of paths is computationally prohibitive;
- Our evaluation shows that even when a graph contains only a few thousand paths, the performance overhead is prohibitive due to repeated LLM interactions. As illustrated in Figures 7 and 8, each packet generated from a path may expose an inconsistency between the format specification and the parser, which must be resolved using an LLM. Consequently, thousands of paths can result in thousands of LLM invocations, leading to substantial latency and making the process impractical, often requiring many hours to complete.

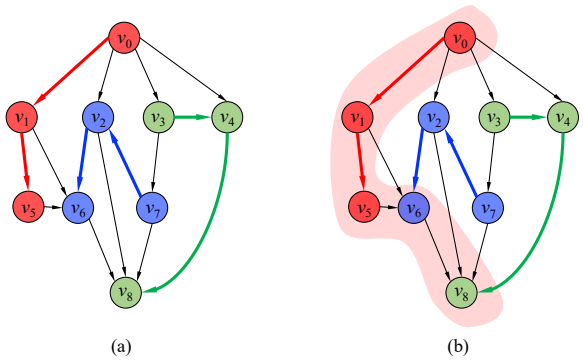


Figure 10: (a) A path cover example. (b) Extending a path segment in the path cover to a full path.

To mitigate path explosion, SPAR adopts a path-cover strategy that ensures every vertex in the PFG or CFG is covered by at least one positive packet, thereby reducing the exponential complexity to linear. In graph theory, given a directed graph $G = (V, E)$, a path cover is a set of vertex-disjoint paths such that every vertex $v \in V$ belongs to at least one path.

Example 4. Figure 10(a) shows a graph with a path cover. The path cover contains three path segments: (v_0, v_1, v_5), (v_7, v_2, v_6), and (v_3, v_4, v_8), covering all vertices. □

To improve efficiency, it is desirable to generate a minimum path cover, as this minimizes the number of packets that need to be generated and exercised. However, computing a minimum path cover is NP-hard for general directed graphs and not linear even for directed acyclic graphs. Thus, it is often not practical to find the exact minimum path cover. Instead, we adopt a greedy algorithm that runs in linear time with respect to the graph size and reduces the exponential number of paths to a linear one. The linear, greedy algorithm, shown in Algorithm 1, works as follows to approximate a minimum path cover by iteratively building long paths:

- **Initialize** (lines 2-3): Set all vertices in the graph as "uncovered" and initialize an empty path-cover set.

Algorithm 1: Approximate the minimum path cover.

```
1  procedure find_path_cover(G = (V, E))
2      Uncovered ← V;
3      PathCover ← ∅;
4      while Uncovered ≠ ∅ do
5          v ← Uncovered.Pop();
6          π ← v;          /* a path with a single vertex */
7          while v has uncovered neighbors do
8              v ← a neighbor of v with fewest uncovered neighbors;
9              π ← π ∘ v;          /* append v to the path π */
10         PathCover ← PathCover ∪ {π};
11         Uncovered ← Uncovered \ {vertices in π};
12     return PathCover;
```

- **Select Seed** (lines 5-6): Pick an uncovered vertex v as the starting point of a new path.
- **Extend Greedily** (lines 8-9): From the current vertex, move to an adjacent uncovered neighbor. To maximize the path length, a common heuristic is to choose the neighbor having the fewest uncovered neighbors. Choosing the neighbor with the fewest uncovered neighbors (the neighbor with the minimum remaining degree) is a heuristic designed to preserve the graph's future connectivity and avoid "stranding" vertices.
- **Terminate Path** (line 7): Stop when the current vertex has no uncovered neighbors.
- **Repeat** (lines 4, 11): Mark vertices in the path as "covered"; repeat previous steps until all vertices are covered.

As shown in the preceding example, a path segment in a path cover may not include all fields (i.e., not from an entry vertex to an exit vertex). Thus, each path segment must be extended to a complete path, after which constraints are collected from the full path for packet generation.

Example 5. Assume the graph in Figure 10 is a PFG. A path segment in the path cover does not include all vertices and therefore lacks constraints for certain fields. As shown in Figure 10(b), to generate a positive packet, we extend each path segment, e.g., (v_0, v_1, v_5), into a complete path, e.g., $(v_0, v_1, v_5, v_6, v_8)$, enabling packet generation based on the constraints along the complete path. □

4.1.4 Optimization II: Negative Packets via Linear Negat.

Recall that we collect constraints from either a PFG path (when using the format specification as the oracle) or a CFG path (when using the parser as the oracle) to generate packets. Given a positive packet p_0 that satisfies the collected constraints $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$ (where each ϕ_i corresponds to the constraint of the i-th field in order), we can generate negative packets by negating one or multiple sub-constraints ϕ_i, leading to a total of 2^n possible negative packets. Apparently, this is not computationally feasible in practice.

To reduce the computational overhead, we ensure that the constraint of each field is negated only once, yielding n negative packets $p_1, p_2, \ldots, p_n$ with respect to (1) $\neg\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$, (2) $\phi_1 \wedge \neg\phi_2 \wedge \cdots \wedge \phi_n$, ..., and ($n$) $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \neg\phi_n$. This linear negation ensures that each field is validated by both a positive and a negative packet, and the packet p_i is intentionally designed to check the constraint of the i-th field.

While this optimization applies to both Steps 2 & 3 (see Figures 7 & 8), in Step 2, we should test the parser using the packets in a sequential order from p_0 to p_n. Consider the following scenario to illustrate the insight. If the parser has not first been exercised with packets p_0 and p_1, the parser generated from LLM may not be correct with respect to ϕ_1. In other words, the parser may encode a wrong constraint for the first field, rejecting p_0 or accepting p_1. Consequently, when testing a later negative packet such as $p_{n>1}$, the parser may reject the packet due to ϕ_1 instead of the intended violated constraint $\neg\phi_n$ (note that parsers typically validate field constraints in a fixed sequential order). This undermines the purpose of p_n, which is meant to specifically check the parser's handling of the n-th field constraint ϕ_n.

4.2 LLM-Based Refinement

Recall the workflow in Figures 7 and 8. After generating positive and negative packets, we detect inconsistencies between the format and the parser. The inconsistent components, either a code snippet in the parser or a fragment of the format, are then provided to the LLM along with the "relevant RFC sections". LLM is expected to recheck the inconsistent components and refine them if necessary.

It is challenging to define what constitutes the "relevant RFC sections." The previous work [78] leverages the divide-and-conquer strategy of format/parser generation shown in Figure 6, which enables a precise tracing of parser code snippets or specification fragments back to their corresponding RFC sections. Consequently, once a mismatch is identified, we only need to provide the relevant RFC sections to the LLM, rather than the entire RFC document. For instance, in Figure 6, if we identify that the constraint for the `Length` field may be incorrect, we can easily trace it back to RFC section 4.3, since the code snippets and format fragments are generated based on that RFC section. To ease the explanation, we define a mapping $\mathcal{M}$ that maps a field f to the RFC sections from which it is generated, denoted $\mathcal{M}(f)$.

However, as shown in the example below, defining the "relevant RFC sections" solely as $\mathcal{M}(f)$ is often insufficient, as RFC sections frequently cross-reference one another for clarification. This interdependence makes it challenging to identify a minimal set of RFC sections that is both sufficient and necessary for LLM-based refinement, a challenge our optimization aims to address.

Example 6. Figure 11 illustrates the motivation for this optimization. Part (a) shows the initial format specification, which

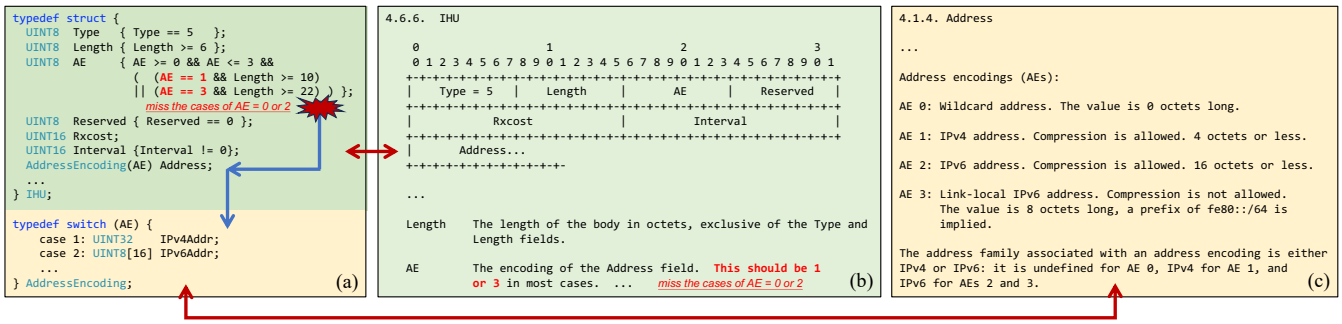


Figure 11: Motivation of Optimization III. (a) A specification where the constraint of the field AE is not correct due to the omission of two cases, AE = 0 and AE = 2. (b) RFC section 4.6.6, which derives the struct type, IHU. (c) RFC section 4.1.4, which derives the parametric case type, AddressEncoding.

consists of two components. The first is a struct type, `IHU`, generated by the LLM from the RFC section 4.6.6 in (b). The second is a parametric case type, `AddressEncoding`, generated by the LLM from the RFC section 4.1.4 in (c). The struct type `IHU` contains several fields that define the format. Its final field, `Address`, is a case type whose structure depends on the value of `AE` (short for Address Encoding). As a result, the overall length of an `IHU` packet also depends on `AE`.

Since the RFC section 4.6.6 discusses only the cases where `AE` = 1 or 3 (highlighted in red), the initial format specification generated from this RFC section does not produce a correct constraint for the `AE` field. As shown in the figure, it captures the interdependencies between `AE` and `Length` for `AE` = 1 and `AE` = 3, but misses the cases where `AE` = 0 or `AE` = 2.

Suppose we identify this flaw during adversarial interactions with the LLM. A common approach is to find $\mathcal{M}(\texttt{AE})$, i.e., the RFC section 4.6.6, and resubmit this RFC section to the LLM, expecting it to correct the flaw. However, the RFC section 4.6.6 does not provide sufficient information about `AE` — which is defined in a different section (the RFC section 4.1.4) — and thus cannot resolve the issue. □

4.2.1 Optimization III: Relevant RFC Sections

This optimization is conceptually related to program slicing, which was originally proposed to identify program statements that a given statement depends on or that may be affected by it [67]. In our setting, given a field f, slicing enables us to track data dependency and identify all related packet fields (e.g., $\{f_1, f_2, \ldots\}$), (1) on which f depends or (2) that f may influence. Consequently, we can collect all necessary and relevant RFC sections as $\mathcal{M}(f) \cup \bigcup_i \mathcal{M}(f_i)$. In the following, we present a concrete example.

Example 7. Consider the last example again: it misses the RFC section 4.1.4 during LLM-based refinement. Our optimization tracks data dependencies involving the `AE` field and identifies that the field, `Address`, whose type is `AddressEncoding(AE)`, depends on `AE`. Thus, we will take into consideration the RFC section that defines the type `AddressEncoding`. Since `AddressEncoding` is generated from the RFC section 4.1.4, we incorporate this RFC section during LLM-based refinement. Since the RFC section 4.1.4 defines the relationship between `AE` and the length of `Address`, we can successfully recover the relationship between `AE` and `Length` and include the previously missing cases where `AE` = 0 or 2 into the field `AE`'s constraint. □

Table 1: Ground Truth of Eight Protocols (RFC Pages, Field, # 1-Constraint, and # n-Constraint Count) .

Protocol	# Pages	# Fields	# 1-Constraints	# n-Constraints
BFD	49	49	15	2
IGMPv3	53	24	6	2
BGPv4	104	77	16	17
BABEL	54	112	40	14
HTTP2	96	111	11	8
DNS	55	88	8	8
NTP	110	56	2	32
SCTP	152	148	42	16

While the procedure above uses Step 3 (refining the format specification via the parser) as an example, it is applicable to Step 2 (refining the parser based on the format specification). This is because both Step 2 and Step 3 seek mismatches between the format specification and the parser; once a mismatch is found, we can follow the same process as in the example above to identify the relevant RFC sections. The difference between Step 2 and Step 3 is that, in Step 2, we ask the LLM to refine the code snippet in the parser, whereas in Step 3, as in the example above, we ask the LLM to refine a fragment of the format specification.

5 Evaluation

We implement SPAR on top of AutoGen [49], an agentic programming framework for LLM interactions, and the Z3 SMT solver [14] for automated network packet generation

Table 2: Precision (%) and Recall (%) of Format Specifications Generated by SPAR, ParCleanse, and ChatAFL across Eight Protocols on *Field Type*, *Field Name*, *1-Constraint*, and *n-Constraint*.

Protocol	Field Type			Field Name			1-Constraints			n-Constraints		
	SPAR	ParCleanse	ChatAFL	SPAR	ParCleanse	ChatAFL	SPAR	ParCleanse	ChatAFL	SPAR	ParCleanse	ChatAFL
BFD	**100/100**	100/100	100/39	**100/100**	100/100	100/39	**100/100**	93/93	100/20	**100/100**	-/0	-/0
IGMPv3	**100/100**	54/54	100/50	**100/100**	96/96	100/50	**100/100**	100/67	67/67	**100/100**	-/0	-/0
BGPv4	**100/100**	96/100	100/35	**100/100**	96/100	100/35	**100/100**	93/87	80/31	**100/100**	100/41	-/0
BABEL	**100/100**	100/91	85/25	**100/100**	100/91	85/25	**100/100**	100/85	81/32	**100/100**	100/64	-/0
HTTP2	**97/97**	96/96	100/14	**100/100**	96/96	100/14	**100/100**	100/40	100/45	**100/100**	100/38	100/13
DNS	**100/100**	92/64	100/18	**100/100**	92/64	100/18	**100/100**	100/75	100/38	**100/88**	-/0	-/0
NTP	**100/100**	92/88	100/80	**100/100**	92/88	100/80	**100/100**	100/50	100/100	**100/97**	75/9	-/0
SCTP	**100/100**	97/95	100/30	**100/100**	100/98	100/30	**100/100**	98/93	100/24	**100/100**	100/31	-/0
Avg	**99/99**	91/86	98/36	**100/100**	96/91	98/36	**100/100**	98/74	91/45	**100/98**	95/23	100/2

based on the formal format specifications. We set the model temperature to 0 to minimize randomness and improve the reproducibility. To show the efficacy of our approach, we conduct experiments with SPAR to address the following four research questions:

- **RQ1 (Effectiveness).** How effective is SPAR in generating formats (particularly, the semantic constraints), compared to the state of the art?
- **RQ2 (Efficiency).** What is the end-to-end efficiency of SPAR, measured by total runtime, solver overhead, number of refinement rounds, and monetary cost?
- **RQ3 (Ablation Study).** How does each component of SPAR affect its overall effectiveness and efficiency?
- **RQ4 (Usefulness).** When applied to security applications, are the format specifications helpful in uncovering vulnerabilities in protocol implementations?

Protocols. As shown in Table 1, we include three protocols from ParCleanse [78] (BFD, BGPv4, BABEL) and select the other five (IGMPv3, HTTP2, DNS, NTP, SCTP) as they contain complex semantic constraints over the packet fields and, thus, are appropriate for demonstrating the strengths of SPAR, i.e., its strong capability to infer inter-dependencies among packet fields, and the weaknesses of existing work. All selected protocols feature tree-based, non-recursive packet formats, which align with the current processing scope of SPAR and prior work such as ParCleanse. This limitation is further discussed in §6.

Table 1 lists the ground truth of each protocol, including the pages of the RFC documents downloaded from the IETF DataTracker [26], the number of packet fields each protocol comprises, and the number of constraints on those fields. In particular, the constraints are divided into two classes: those that constrain a single field (the *# 1-Constraints* column) and those that constrain multiple fields (the *# n-Constraints* column). Due to the lack of ground truth, following common practice and ParCleanse [78], two authors independently manually extracted ground truth from RFCs, resolving discrepancies via consensus. As shown in the table, none of the protocols is trivial, each having an RFC document more than 49 pages (84 on average, up to 152), containing more than 24 fields (83 on average, up to 148), 2 single-field (18 on average, up to 42) and 2 cross-field (12 on average, up to 32) constraints. To mitigate the threats brought by the manually constructed ground truth and contribute to the community's future research, we make it publicly available [71].

Baselines. To show the effectiveness of SPAR, particularly the adversarial LLM interactions, in §5.1 and §5.4, we compare SPAR against ParCleanse [78] and ChatAFL [39], both of which are recent techniques that acquire formal format specification from RFC documents via LLM but without adversarial LLM interactions. The evaluation results show that SPAR, with adversarial LLM interactions, can infer constraints on network packets with much higher precision and recall (close to 100%). As a result, protocol fuzzers with our format specification can uncover more hidden vulnerabilities. 3DGEN [19] is another recent work that leverages LLM for format inference. In general, it differs from SPAR in that it does not leverage adversarial LLM interactions either. We do not compare SPAR with it because it is not publicly available.

LLM Selection. By default, we use GPT-4.1 [45], a widely used LLM, for evaluation. In §5.3, we show that the effectiveness of SPAR is not compromised when using other popular LLMs, such as Gemini-2.0-Flash [22], Claude-3.5-Sonnet [46], and Qwen3-Coder [69].

Environment. All experiments are conducted on a server with the following configurations: 64 cores, 128 threads, 3.4 GHz CPU, and 256 GB of memory, running Ubuntu 22.04.

5.1 RQ1: Effectiveness

As shown in Figure 1, each field comprises a syntactic component (name and type) and a semantic component (constraints). Thus, the evaluation results in Table 2 are categorized into these two parts: the *Field Name* and *Field Type* columns reflect

Table 3: Ablation Study 1, 2, 3: Precision (%) and Recall (%) of Protocol Formats Generated by SPAR and Three Variants.

Approach	Field Type	Field Name	1-Constraints	n-Constraints
SPAR	**99/99**	**100/100**	**100/100**	**100/98**
W/O RFC	94/47	96/35	87/33	-/0
W/O Refine.	95/95	100/98	88/67	91/22
W/O Opt.III	98/98	100/100	97/95	93/74

Table 4: Ablation Study 4: Precision (%) and Recall (%) of Protocol Formats Generated by SPAR with Different LLMs.

LLMs	Field Type	Field Name	1-Constraints	n-Constraints
Gemini-2.0-Flash	98/98	98/98	100/98	100/92
Claude-3.5-Sonnet	100/100	100/100	100/100	100/99
Qwen3-Coder	100/100	100/100	100/100	100/96

syntactic accuracy, while the *1-Constraints* and *n-Constraints* columns represent semantic dependencies. In cases where a tool fails to infer any constraints, precision is marked as "-" (not applicable). To be counted as a true positive, a *Field Name* must be manually verified as semantically equivalent to the ground truth. For a *Field Type*, primitive types require an exact match, while complex types (such as structs) require a recursive match. Finally, both *1-* and *n-Constraints* are evaluated based on their logical equivalence via an SMT solver.

Syntactically, all three tools perform well, with precision often near 100%. SPAR leads with perfect scores (close to 100%) across all protocols. While ChatAFL maintains high precision, its recall is significantly lower because it only generates specifications for a subset of message types, e.g., missing many packet types in the case of BABEL.

In terms of the semantic constraints across packet fields (which our approach emphasizes), all three approaches correctly infer most single-field constraints, as these single-field constraints are relatively straightforward. Nonetheless, the advantage of our approach is clear as SPAR achieves 100% precision and recall, whereas ParCleanse and ChatAFL achieve a 74% and 45% recall, respectively. The benefits of our design, i.e., the adversarial LLM interactions, become apparent when we look at the semantic constraints over multiple fields, i.e., the *n-Constraints* column. On average, SPAR achieves 326% (98% vs. 23%) and 4800% (98% vs. 2%) improvement in recall, when compared to ParCleanse and ChatAFL.

5.2 RQ2: Efficiency

To assess the practical viability of SPAR, we evaluate its efficiency across the eight protocols. Table 5 presents the total runtime in seconds, the computational overhead of the SMT solver, the number of refinement rounds, and the approximate monetary cost of LLM API usage obtained from 100 averaged runs of SPAR, each starting at RFC input and ending when the final specification is produced.

Table 5: Efficiency of SPAR Measured by Total Runtime, Number of Refinement Rounds, Solver Overhead, and Cost.

Protocol	Runtime (s)	Solver Overhead	Rounds	LLM Cost
BFD	452.15	3.12%	3.51	$0.72
IGMPv3	378.42	1.15%	5.65	$0.88
BGPv4	2580.33	4.71%	10.32	$6.25
BABEL	5412.90	5.38%	18.05	$10.45
HTTP2	895.60	3.05%	6.28	$4.51
DNS	2735.14	4.60%	9.76	$7.15
NTP	872.45	1.34%	4.20	$1.56
SCTP	1285.67	3.81%	7.82	$14.05

Runtime and Solver Overhead. As shown in Table 5, the total runtime required to generate a format specification ranges from 378.42 seconds (IGMPv3) to 5,412.90 seconds (BABEL), while the computational overhead introduced by the SMT solver remains consistently low, consuming only 1.15% to 5.38% of the total execution time. The results show the practical efficiency of our approach. First, we believe a maximum runtime of around 1.5 hours is reasonable for an automated pipeline that generates precise packet format specifications, enabling us to detect many vulnerabilities that have been hidden for years (see §5.4). Second, the light solver overhead demonstrates the effectiveness of our packet-generation optimizations. By keeping constraint solving bounded, the path-cover strategy and linear negation mitigate path explosion, ensuring that the SMT solver serves as a lightweight validation engine rather than a computational bottleneck.

Refinement Rounds and LLM Cost. To reach a precise, converged specification, SPAR executes between 3.51 (BFD) and 18.05 (BABEL) rounds of adversarial refinement, incurring a total LLM API cost ranging from $0.72 (BFD) to $14.05 (SCTP). All reported LLM API costs in Table 5 are based on GPT-4.1 [45] using OpenAI's official pricing ($3.00 per million input tokens and $12.00 per million output tokens). The bounded refinement rounds indicate that the adversarial interaction is stable: the framework converges on a fixed point rather than falling into infinite refinement loops, even when processing complex protocols. Also, the monetary cost is moderate, with a maximum of $14.05, which is a one-time investment but could enable many downstream security applications relying on format specifications.

5.3 RQ3: Ablation Study

To validate the necessity and robustness of each component in SPAR, we conduct a detailed ablation study to assess their impacts on both effectiveness and efficiency. The evaluated variants of SPAR include:

1. **SPAR w/o RFC**: We remove the RFC from the input and instead instruct the LLM to generate the parser and format directly. This variant aims to confirm that LLM

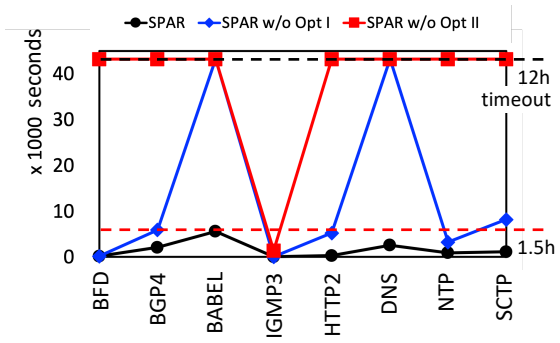


Figure 12: SPAR vs. SPAR w/o Optimizations I & II.

cannot directly generate high-quality format specifications without understanding the RFC documents.

2. **SPAR w/o Adversarial LLM Interactions**: We remove the refinement process (Steps 2 & 3 in Figure 5) to show that the adversarial LLM interaction is significant.
3. **SPAR w/o Optimizations I, II, or III**: We remove each optimization SPAR to show their contributions to effectiveness and efficiency.
4. **SPAR with Different LLMs**: We conduct comparative experiments across different LLMs, showing that the effectiveness of SPAR is independent of the specific LLM.

Impacts on Effectiveness. Table 3 shows the results of SPAR and its variants SPAR w/o RFC, SPAR w/o Refinement, and SPAR w/o Optimization III. As shown, the three variants exhibit significant performance degradation, particularly in the inference of 1-constraints and n-constraints. For 1-constraint inference, the degradation ranges from 5% to 67%. For n-constraint inference, the degradation ranges from 25% to 100%. This confirms that both the RFC input, the refinement process, and the third optimization are effective components of SPAR. The table does not include SPAR w/o Optimization I and SPAR w/o Optimization II, because, as discussed below, the two optimizations are designed for improving efficiency and cannot finish format inference in 12 hours for many protocols.

Impacts on Efficiency. Optimizations I and II aim to reduce the number of network packets generated during execution, thereby significantly lowering the time overhead of format inference. As shown in Figure 12, variants that disable either Optimization I or Optimization II may fail to complete format inference within 12 hours. In contrast, our optimized approach consistently completes the inference process in at most 1.5 hours, demonstrating the substantial efficiency gains enabled by these optimizations.

Impacts by LLMs. Table 4 shows the impact of different underlying LLMs on the performance of SPAR. The results indicate that the variations in both precision and recall across different LLM choices are minimal, with fluctuations remaining within 8%. This observation suggests that SPAR does not rely on any particular LLM for effectiveness and remains highly robust across a range of model backends.

```
1.  void bfd_recv_cb(...) {
2.    ......
3.    struct bfd_pkt *cp = (struct bfd_pkt *)(msgbuf); // BFD packet
4.
5.  - if ((cp->len < BFD_PKT_LEN)
6.  + if ((cp->len < (GETABIT(cp->flags) ? BFD_PKT_LEN+2 : BFD_PKT_LEN))
7.          || (cp->len > mlen)) {
8.      return
9.    }
10.   ......
11. + if (!cp->discrs.remote_discr && GETSTATE(cp->flags) != PTM_BFD_DOWN &&
12. +                                GETSTATE(cp->flags) != PTM_BFD_ADM_DOWN) {
13. +   return;
14. + }
15.   ......
```

Figure 13: Mis- and missing checks of cross-field constraints in an implementation of the BFD protocol.

```
1.  void parse_packet(char *packet) {
2.      int type = packet[0];
3.      char router_id[8]={0};
4.      ......
5.      if (type == MESSAGE_UPDATE /*8*/) {
6.          char prefix[16];
7.          ......
8.          if (packet[3] & 0x40) { // Check Router-Id flag
9.              if (packet[2] == 1)
10.                 memcpy(router_id + 4, prefix + 12, 4);
11.             else
12.                 memcpy(router_id, prefix + 8, 8);
13.             have_router_id = 1;
14.         }
15.
16. +       if (have_router_id && all_zero_or_one(router_id)) {
17. +           debugf("Received prefix with invalid router id.");
18. +           goto fail;
19. +       }
20.         ......
```

Figure 14: A missing check of cross-field constraints in an implementation of the BABEL protocol.

5.4 RQ4: Usefulness

To evaluate the usefulness of SPAR for security applications, we use the format specifications generated by SPAR to audit existing protocol implementations and identify vulnerabilities. Our evaluation primarily focused on two target systems: Holo [11] and FRRouting [10]. Holo is a Rust-based protocol suite designed for high-scale, automation-driven networks, while FRRouting is widely deployed in Linux systems. In particular, given the format generated by our approach, we use the Z3 solver [14] to automatically generate positive and negative packets to test the packet parsers in FRRouting and Holo (as we do when testing the reference parser during iterative refinement). When a positive packet fails a parser or a negative packet passes the test, we report an issue in the parser implementations. In total, as detailed in Table 6, we identified 24 zero-day vulnerabilities, all depending on the correctly inferred constraints. These vulnerabilities have been confirmed or fixed by the developers.

Beyond our motivating example, we discuss two additional vulnerabilities that underscore the importance of identifying cross-field constraints, a core driver of our technical design. Both cases have been hidden in the code for over 6 years.

Table 6: Zero-Day Vulnerabilities Discovered by SPAR.

ID	Protocol	Source	Root Causes
1	BABEL	FRR 10.5	Not enforcing "ae=3, omitted=0" (Update)
2	BABEL	FRR 10.5	Missing Router-ID check (Router-ID)
3	BABEL	FRR 10.5	Missing Router-ID check (Update)
4	BABEL	FRR 10.5	Rejection of reserved bits (Ack Req)
5	BABEL	FRR 10.5	Rejection of reserved bits (IHU)
6	BABEL	FRR 10.5	Rejection of reserved bits (MH Req)
7	BABEL	FRR 10.5	Missing zero-interval check (Ack Req)
8	BABEL	FRR 10.5	Consume TLV on mandatory bit (Hello)
9	BABEL	FRR 10.5	Consume TLV on mandatory bit (IHU)
10	BABEL	FRR 10.5	Missing sub-TLV handling (Ack Req)
11	BABEL	FRR 10.5	Missing sub-TLV handling (Ack)
12	BABEL	FRR 10.5	Missing sub-TLV handling (Router-ID)
13	BABEL	FRR 10.5	Missing sub-TLV handling (Next Hop)
14	BFD	FRR 10.5	Discard "YourDiscr=0" (Local UP/INIT)
15	BFD	FRR 10.5	Accept "YourDiscr=0" (Pkt UP/INIT)
16	BFD	FRR 10.5	Early session state update
17	BFD	FRR 10.5	Incorrect length validation (<24 vs 26)
18	BFD	Holo 0.8	Incorrect packet length check logic
19	BFD	Holo 0.8	Version check performed post-lookup
20	BFD	Holo 0.8	Detect Mult check post-lookup
21	BFD	Holo 0.8	State check post-lookup
22	BFD	Holo 0.8	Discriminator check post-lookup
23	BFD	Holo 0.8	Multihop bit check post-lookup
24	BFD	Holo 0.8	Missing Auth section length check

Case 1. Figure 13 shows a vulnerable implementation of the BFD protocol, which misses checks for a couple of cross-field constraints. In the implementation, the variable `cp` denotes a BFD packet and `cp->xxx` accesses a field `xxx` in the packet.

First, the original mis-check at line 5 checks a single-field constraint on the `cp->len` field, which, however, should be a cross-field constraint as our fix in line 6: there is an inter-dependency between the value of the `cp->len` field and the `cp->flags` field. Second, lines 11-14 miss a check between the session ID (mapped to the packet field `cp->discrs.remote_discr`) and the protocol state (mapped to the packet field `cp->flags`). Specifically, if the session ID is zero, the protocol state must be `DOWN` or `ADMIN_DOWN`. Since the BFD protocol uses the session ID to locate a remote session and treats a session ID of zero as a wildcard, this check is critical for preventing attackers from hijacking a valid session. In other words, omitting this check allows attackers to spoof a network session without guessing the random 32-bit session ID, potentially leading to blind packet injection, denial-of-service attacks, and corruption of the protocol state.

As shown in §5.1, ParCleanse and ChatAFL struggle to infer these cross-field constraints and thus cannot detect such missing or incorrect checks in BFD's implementation.

Case 2. Figure 14 shows a vulnerable implementation of the BABEL protocol, which misses a check for a cross-field constraint. In the implementation, the packet field Prefix (mapped to the variable `prefix`) contains the address prefix being advertised in an Update message. Furthermore, when the packet field Router ID flag (`packet[3] & 0x40`) is set, a local variable `router_id` is derived from `prefix`. Another packet field, AE (`packet[2]`), controls which segment of `prefix` will be used to construct `router_id`.

As shown by our fix in lines 16-19, `router_id` must not be all zeros or all ones, which are reserved values representing unspecified or broadcast addresses and cannot uniquely identify a router. Consequently, a cross-field constraint exists between the Router ID flag, the AE, and the Prefix: when the flag is set, the specific segment of the Prefix dictated by the AE cannot be all zeros or all ones to avoid constructing an invalid `router_id`. Violating it can allow an attacker to bypass internal security filters with a reserved, all-zeros identifier, enabling them to inject fake routes into the system's routing table and leading to data interception, blackholing, or denial-of-service attacks.

As in Case 1, ParCleanse and ChatAFL struggle to infer the cross-field constraints and thus cannot detect this missing check in BABEL's implementation.

Beyond Network Fuzzing. Beyond the network fuzzing application demonstrated above, SPAR could also be applied to other scenarios. For example, applying automated protocol format inference to network traffic auditing and intrusion detection represents a paradigm shift from lexical, signature-based analysis to semantic, structure-aware inspection. In network traffic auditing engines like Wireshark [68], developers traditionally rely on manually drafted packet dissectors to interpret online network traffic. SPAR automates this procedure by generating format specifications that map raw byte-streams into named, human-readable fields and structures.

Similarly, applying SPAR's inferred formats to Intrusion Detection Systems (IDS) like Snort [58] addresses the fundamental blind spots of traditional payload inspection. Historically, Snort has allowed defenders to define rules (relying on rigid byte sequences or regular expression matching) to block malicious network traffic. Without a format specification, this approach scans the payload blindly; thus, it is highly susceptible to evasion techniques such as packet fragmentation, obfuscation, or deliberate alignment shifts. SPAR enables structure-aware detection rules: Rather than relying on rigid byte-sequence matching, defenders can define rules that evaluate semantic constraints between specific fields, such as validating a length header against the actual payload size, thereby enabling semantics-aware network intrusion detection.

6 Limitations

In this section, we discuss the limitations of this work and possible future directions.

Packet Generation. To avoid generating an explosive number of packets, SPAR employs a path-cover strategy (Optimization I) and linear negation (Optimization II, i.e., negating one constraint at a time). Although effective as demonstrated

in our evaluation, this approach does not, in theory, guarantee thorough coverage; it is an intentional trade-off to allow computational feasibility. A natural next step toward relaxing this trade-off is to investigate improved coverage strategies to generate more effective positive or negative packets. For example, we may adopt combinatorial testing [44] to systematically generate a sufficient number of negative test cases by negating multiple constraints at a time.

Protocols Defined by Multiple RFCs. The protocols evaluated in our study are each defined by a single RFC document. In practice, to handle protocols spanning multiple RFCs, we can merge RFCs at Step 1, which currently leverages the existing capabilities of ParCleanse [78]. Since our key contribution lies in subsequent adversarial refinement rather than in the initial RFC merging, we do not evaluate ParCleanse's capability to merge RFCs in this work. However, we believe that addressing multiple RFCs is a promising direction worth further study.

Expressiveness of Format Specification. SPAR relies on the 3D language to specify format specification and ParCleanse to build the initial format specification, thus inheriting their limitations. Specifically, SPAR currently assumes that a protocol format follows a tree-structured hierarchy, which precludes support for the more complex, recursive, and non-linear formats found in certain multi-layered protocols. While SPAR currently handles intra-structure constraints effectively, it does not enforce global semantic constraints or dependencies between fields located across different structures. Enhancing the expressiveness of the underlying formal specifications to accommodate these sophisticated, state-dependent protocol architectures remains a valuable direction for future work.

RFC Ambiguities. Since RFCs are written in natural language, they suffer from inherent ambiguities, which we can categorize into two types: *linguistic ambiguity* and *undefined behavior*. First, linguistic ambiguity arises from permissive terminology (e.g., "typically", "should", or "may"). Our approach attempts to address this kind of ambiguity through Optimization III (see Examples 6–7), which leverages program slicing to enforce that the LLM cross-checks different RFC sections to extract sufficient information. While our evaluation shows promising results, we recognize that resolving textual ambiguities remains an open challenge that cannot be solved universally [41, 61].

The second form of ambiguity is termed "undefined behavior", such as the lack of specifying actions for format violations. For example, Figure 2 illustrates an instance from BABEL's RFC [9]. It specifies that "*if* `AE` *equals 3, the* `Omitted` *field MUST be 0*" but fails to define how a packet should be handled if this constraint is violated. Such ambiguities are quite common not only in format specifications but also in protocol state machines. As such, different vendors may implement their own logic for these undefined behaviors, leading to many problems in recent years [15].

Since the design principle in this work is to decouple the format inference procedure from protocol implementations (as discussed in §1), inferring these implementation-specific or vendor-specific behaviors is outside our scope. SPAR instead extracts format specifications only from RFCs and intentionally avoids hallucinating or guessing such behaviors. Nonetheless, in practice, if we can obtain trusted implementations and their test cases, the code and runtime information are definitely helpful, as demonstrated by many implementation-based format inference techniques, e.g., [6, 7, 13, 27, 33, 34, 53, 55, 66]. As a future direction, extending SPAR to integrate trusted implementations could provide another source of truth for resolving these undefined behaviors and generating vendor-specific specifications.

7 Related Work

Given the critical role of network protocol specifications, numerous surveys have reviewed existing methods for inferring them [18, 32, 43, 57, 76]. Our work differs from prior approaches in two key respects: (1) it does not depend on protocol implementations and therefore avoids inheriting errors from buggy code, and (2) it leverages recent advances in LLMs to infer precise format specifications directly from the official documents. In what follows, we review related work along these two dimensions.

Decoupling from Protocol Implementations. Network trace analysis represents a distinct family of techniques that infers protocol formats without access to source code or binary implementations. Instead, these methods rely on statistical and data-driven analyses of captured network traffic to delineate field boundaries and recover structural patterns. Discoverer [12] infers protocol formats by recursively clustering network packets of the same type and analyzing structural similarities within each cluster. ReverX [1] models network packets as a formal language, constructs a finite-state machine, and derives protocol formats from the inferred automaton. Biprominer [64] mines variable-length patterns and transition probabilities from network traces to identify protocol keywords and infer protocol formats. ProDecoder [65] extends Biprominer by applying n-gram–based data mining to extract semantic information. AutoReEngine [35] uses frequent keyword mining and data mining techniques to classify packets and infer protocol formats. NemeSys [29, 30] represents packets as feature vectors and applies sequence alignment and clustering to infer protocol formats. NetPlier [72] leverages probabilistic analysis to identify keyword fields, cluster packets, and infer protocol formats through multi-sequence alignment. REInPr [47] employs probabilistic models to cluster network packets and infer protocol formats from packet characteristics within each cluster. InSyfer [70] uses a graph neural network–based adaptive packet classification model to cluster network traffic and infer protocol specifications.

These techniques differ from our approach in two key respects. First, they rely on a large corpus of network traffic for accurate inference, which may not be available in many scenarios; in contrast, our approach relies exclusively on publicly available RFC documents. Second, although effective at recovering packet syntax, their ability to infer precise semantic constraints is inherently limited by the observed traffic. By analyzing complete RFC documents, our approach operates with substantially broader capability bounds and can capture richer semantic dependencies.

LLM-Driven Protocol Validation. LLM-driven protocol validation has emerged as a trend that seeks to automatically extract network protocol specifications from RFC documents, a task traditionally regarded as labor-intensive. These approaches capitalize on LLMs' ability to understand structured natural language and have demonstrated promising results. For example, ParCleanse [78] and 3DGen [19] derive protocol format from RFC documents, followed by refinement through a testing procedure against existing parser implementations. ChatAFL [39] constructs a syntactical structure for individual message types and performs message mutation or prediction guided by crash-based runtime oracles. While the techniques above are designed for general protocols, mGPT-Fuzz [37] and LLMIF [63] focus on special protocols, i.e., the Matter and the Zigbee protocols, respectively.

Our approach differs from the aforementioned techniques in two key respects. First, these approaches rely on existing protocol implementations to refine inferred formats. In contrast, SPAR is fully decoupled from protocol implementations, enabling a broader applicability and avoiding the risk of inheriting format errors from imperfect implementations. More importantly, as demonstrated in our evaluation, SPAR shows promising effectiveness in inferring semantic constraints among packet fields, but existing approaches remain limited in capturing such constraints.

8 Conclusion

In this paper, we propose an LLM-driven method to extract precise format specifications covering both syntax and semantics directly from RFCs. By pairing specification inference with reference parser generation in an adversarial loop, our approach iteratively and effectively corrects inherent LLM hallucinations. Answering our research questions on effectiveness and efficiency (§5.1 and §5.2), SPAR advances the current state of the art by decoupling inference from buggy implementations and using adversarial interactions to correct hallucinations. Most notably, it closes a critical gap in identifying complex semantic dependencies, achieving a 326% improvement in recall for cross-field constraints compared to prior tools like ParCleanse and ChatAFL, while converging in at most 1.5 hours and costing under $15. Answering our research questions on robustness and usefulness (§5.3 and §5.4), our ablation study demonstrates that each component of SPAR is necessary for the system to remain effective or efficient, and the performance of SPAR varies by less than 8% across four different LLM backends. SPAR's specifications helped uncover 24 zero-day vulnerabilities in widely deployed routing suites (FRRouting [10] and Holo [11]), some of which were hidden for up to 9 years. Despite these gains, SPAR inherits limitations from a few aspects discussed in §6, which could be further enhanced in the near future.

Acknowledgments

We sincerely thank the anonymous reviewers for their suggestions. This work was partially supported by the National Natural Science Foundation of China (No. 62472215), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025-XDXM118), and the 111 Center (No. B26023).

Ethical Considerations

Stakeholder Analysis. We identified four primary groups of stakeholders impacted by our research: (1) *The Network Community*: General users whose internet connectivity relies on the stability of routing infrastructure. They benefit from more robust protocols but face risks if vulnerabilities are exploited before patches are applied. (2) *Software Maintainers*: The developers responsible for the codebases we analyzed. They face the burden of triaging and patching the reported issues. (3) *Network Operators*: ISPs and organizations running FRRouting or Holo who must deploy updates to mitigate risks. (4) *Malicious Actors*: Entities that might seek to weaponize the capability to generate precise protocol specifications to discover zero-day exploits.

Impacts. The publication of SPAR has both positive and negative impacts on the identified stakeholders.

Positive Impacts. The primary contribution of this work is the automated generation of precise format specifications, which bridges the critical gap between ambiguous, human-readable RFC text and verifiable specification code. We highlight two key benefits. First, infrastructure hardening: the 24 detected bugs merely validate this capability. The broader benefit is the ability to use these precise specifications for formal verification, automated conformance testing, and the generation of safer parsers. This reduces the long-term likelihood of implementation flaws across the entire routing ecosystem, not just in the specific software versions we tested. Second, we remove ambiguity: by formalizing protocol formats, we provide a "ground truth" that helps developers resolve inconsistencies across implementations, thereby fostering better interoperability and stability for the internet at large.

Negative Impacts. We identified two main negative risks. First is the "dual-use" risk: lowering the bar for exploitation. A

precise format specification is effectively a high-fidelity map of the attack surface. While it enables defenders to verify code, it simultaneously enables malicious actors to perform highly efficient, structure-aware fuzzing on critical routing infrastructure that we did not analyze. Second is the burden on maintainers, as reporting a large volume of vulnerabilities simultaneously can overwhelm volunteer-driven open-source communities.

Mitigations. We recognized that reporting 24 vulnerabilities simultaneously could overwhelm the volunteer maintainers of FRRouting and Holo. To mitigate this burden, for each identified vulnerability, we developed and verified the fix ourselves. We grouped related fixes into comprehensive Pull Requests to streamline the review process for maintainers. This approach shifted the remediation labor cost from the open-source community to the research team, ensuring that our work resulted in immediate improvement rather than administrative debt. All 24 detected bugs have been acknowledged and confirmed by the respective development teams. As of this writing, patches for 20 vulnerabilities have been merged into the main branches. The remaining patches have been submitted and are currently under code review. To protect users during this interim period, we have withheld the specific crash inputs and exploit primitives for the unmerged vulnerabilities from the public version of this paper and our artifact repository.

To mitigate the dual-use risk of SPAR, we have deliberately excluded the vulnerability detection and testing harness from both the public repository and the artifacts submitted for review, choosing to only release the source code of SPAR for specification generation. By releasing only the generator code, we enable defensive research while enforcing a skill barrier that prevents low-skill actors from immediately using our work to exploit unpatched systems. We believe this balance prioritizes the safety of the live routing infrastructure over the convenience of fully automated reproducibility during the review process.

Decision. We concluded that the benefits of demystifying protocol ambiguity outweigh the risks. The "security through obscurity" of vague RFCs is not a valid defense since attackers have always been manually reverse-engineering these protocols. Our work levels the playing field by giving defenders and developers the automated tools necessary to systematically secure their implementations.

Open Science

We have open-sourced the SPAR artifact under the MIT license and archived it on Zenodo [71]. The artifact includes all datasets and components necessary to repeat the experiments: the format and parser generation (incorporating components from ParCleanse), the adversarial refinement loop, and the three optimizations evaluated in the paper. It also includes the RFC documents, ground-truth specifications, and pre-computed results for the eight protocols, as well as scripts to set up the environment and run the offline evaluation without API credentials. A detailed README provides setup instructions and guides users through both offline evaluation and, for those with API credentials, the full workflow.

References

[1] Joao Antunes, Nuno Neves, and Paulo Verissimo. Reverse engineering of protocols from network traces. In *Proceedings of the Working Conference on Reverse Engineering*, WCRE'11, pages 169–178. IEEE, 2011.

[2] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS'18, pages 1383–1396. ACM, 2018.

[3] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP'17, pages 483–502, 2017.

[4] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for tcp implementations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'06, pages 55–66. ACM, 2006.

[5] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Found. Trends Priv. Secur.*, 1(1–2):1–135, 2016.

[6] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS'09, pages 621–634. ACM, 2009.

[7] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS'07, pages 317–329. ACM, 2007.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Pro-*

ceedings of the USENIX Conference on Operating Systems Design and Implementation, OSDI'08, pages 209–224. USENIX, 2008.

[9] J. Chroboczek and D. Schinazi. RFC 8966: The babel routing protocol. https://www.rfc-editor.org/rfc/rfc8966, 2025.

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

[11] The Holo Community. Holo routing. https://github.com/holo-routing/holo/, 2025.

[12] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: automatic protocol reverse engineering from network traces. In *Proceedings of the USENIX Conference on Security Symposium*, USENIX Security'07, pages 14:1–14:14. USENIX, 2007.

[13] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: automatic reverse engineering of input formats. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS'08, pages 391–402. ACM, 2008.

[14] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08, pages 337–340. Springer, 2008.

[15] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *Proceedings of the USENIX Conference on Security Symposium*, USENIX Security'15, pages 193–206. USENIX, 2015.

[16] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Trans. Softw. Eng. Methodol.*, 33(7):189:1–189:38, 2024.

[17] Holger Dreger, Anja Feldmann, Michael Mai, Vern Paxson, and Robin Sommer. Dynamic application-layer miscprotocol analysis for network intrusion detection. In *Proceedings of the Conference on USENIX Security Symposium*, USENIX Security'06, pages 18:1–18:16. USENIX, 2006.

[18] Julien Duchene, Colas Le Guernic, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. State of the art of network protocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques*, 14(1):53–68, 2018.

[19] Sarah Fakhoury, Markus Kuppe, Shuvendu K. Lahiri, Tahina Ramananandro, and Nikhil Swamy. 3dgen: Ai-assisted generation of provably correct binary format parsers. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, ICSE'25, pages 2535–2547. IEEE, 2025.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *Proceedings of the USENIX Workshop on Offensive Technologies*, WOOT'20, pages 1–10. USENIX, 2020.

[21] Matheus E. Garbelini, Chundong Wang, and Sudipta Chattopadhyay. Greyhound: Directed greybox wi-fi fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 19(2):817–834, 2022.

[22] Google DeepMind. Gemini 2.0: A comprehensive multimodal AI model. https://deepmind.google/technologies/gemini/, 2026.

[23] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and End-to-End protocol semantics. In *Proceedings of the USENIX Conference on Security Symposium*, USENIX Security'01. USENIX, 2001.

[24] Xiaoyan Hu, Wenjie Gao, Guang Cheng, Ruidong Li, Yuyang Zhou, and Hua Wu. Toward early and accurate network intrusion detection using graph embedding. *Trans. Info. For. Sec.*, 18:5817–5831, 2023.

[25] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP'20, pages 1613–1627. IEEE, 2020.

[26] IETF. IETF datatracker. https://datatracker.ietf.org/, 2025.

[27] Jiayi Jiang, Xiyuan Zhang, Chengcheng Wan, Haoyi Chen, Haiying Sun, and Ting Su. Binpre: Enhancing field inference in binary analysis based protocol reverse engineering. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS'24, pages 3689–3703. ACM, 2024.

[28] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[29] Stephan Kleber, Henning Kopp, and Frank Kargl. Nemesys: network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In *Proceedings of the USENIX Workshop on Offensive Technologies*, WOOT'18, pages 1–18. USENIX, 2018.

[30] Stephan Kleber, Rens W. van der Heijden, and Frank Kargl. Message type identification of binary network

protocols using continuous segment similarity. In *Proceedings of the IEEE Conference on Computer Communications*, INFOCOM'20, pages 2243–2252. IEEE, 2020.

[31] Hui Li, Zhen Dong, Siao Wang, Hui Zhang, Liwei Shen, Xin Peng, and Dongdong She. Extracting formal specifications from documents using llms for test automation. In *Proceedings of the IEEE/ACM International Conference on Program Comprehension*, ICPC'25, pages 1–12. IEEE, 2025.

[32] XiangDong Li and Li Chen. A survey on methods of automatic protocol reverse engineering. In *Proceedings of International Conference on Computational Intelligence and Security*, CIS '11, pages 685–689. IEEE, 2011.

[33] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'08, pages 1–17. The Internet Society, 2008.

[34] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Reverse engineering input syntactic structure from program execution and its applications. *IEEE Trans. Softw. Eng.*, 36(5):688–703, 2010.

[35] Jian-Zhen Luo and Shun-Zheng Yu. Position-based automatic reverse engineering of network protocols. *Journal of Network and Computer Applications*, 36(3):1070–1077, 2013.

[36] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. Specgen: Automated generation of formal program specifications via large language models. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, ICSE'25, pages 16–28. IEEE, 2025.

[37] Xiaoyue Ma, Lannan Luo, and Qiang Zeng. From one thousand pages of specification to unveiling hidden bugs: large language model assisted fuzzing of matter iot devices. In *Proceedings of the USENIX Conference on Security Symposium*, USENIX Security'24, pages 4783–4800. USENIX, 2024.

[38] Noble Saji Mathews and Meiyappan Nagappan. Test-driven development and llm-based code generation. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE'24, pages 1583–1594. ACM, 2024.

[39] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the Annual Network and Distributed System Security Symposium*, NDSS'24, pages 1–17. The Internet Society, 2024.

[40] Soo-Jin Moon, Milind Srivastava, Yves Bieri, Ruben Martins, and Vyas Sekar. Pryde: A modular generalizable workflow for uncovering evasion attacks against stateful firewall deployments. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP'24, pages 4440–4458. IEEE, 2024.

[41] Richard Moot and Christian Retoré. Natural language semantics and computability. *Journal of Logic, Language and Information*, 28(2):287–307, 2019.

[42] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. ClarifyGPT: A framework for enhancing llm-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering*, 1(FSE):2332–2354, 2024.

[43] John Narayan, Sandeep K. Shukla, and T. Charles Clancy. A survey of automatic protocol reverse engineering tools. *ACM Computing Survery*, 48(3):40:1–40:26, 2015.

[44] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Survery*, 43(2):11:1–11:29, 2011.

[45] OpenAI. GPT-4.1 model card. https://platform.openai.com/docs/models/gpt-4.1, 2025.

[46] Anthropic PBC. Claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-sonnet, 2026.

[47] Zhen Qin, Zeyu Yang, Yangyang Geng, Xin Che, Tianyi Wang, Hengye Zhu, Peng Cheng, and Jiming Chen. Reverse engineering industrial protocols driven by control fields. In *Proceedings of the IEEE Conference on Computer Communications*, INFOCOM'24, pages 2408–2417. IEEE, 2024.

[48] George E. Raptis, Muhammad Taimoor Khan, Christos Koulamas, and Dimitrios Serpanos. Llm-based generation of formal specification for run-time security monitoring of ics. In *Proceedings of the IEEE International Conference on Cyber Security and Resilience*, CSR'25, pages 957–962. IEEE, 2025.

[49] Microsoft Research. Autogen: a programming framework for agentic AI. https://github.com/microsoft/autogen, 2024.

[50] Microsoft Research. 3D: Dependent data descriptions for verified validation. https://project-everest.github.io/everparse/3d.html, 2025.

[51] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. In *Proceedings of European Conference on Computer Systems*, EuroSys'22, page 166–180. ACM, 2022.

[52] Prakhar Sharma and Vinod Yegneswaran. Prosper: Extracting protocol specifications using large language models. In *Proceedings of the ACM Workshop on Hot Topics in Networks*, HotNets'23, pages 41–47. ACM, 2023.

[53] Qingkai Shi, Junyang Shao, Yapeng Ye, Mingwei Zheng, and Xiangyu Zhang. Lifting network protocol implementation to precise format specification with security applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS'23, page 1287–1301. ACM, 2023.

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

[55] Qingkai Shi, Xiangzhe Xu, and Xiangyu Zhang. Extracting protocol format as state machine via controlled static loop analysis. In *Proceedings of the USENIX Conference on Security Symposium*, USENIX Security'23, pages 7019–7036. USENIX, 2023.

[56] Bing Shui, Yufan Zhou, Jielun Wu, Baowen Xu, and Qingkai Shi. Validating interior gateway routing protocols via equivalent topology synthesis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS'25, pages 827–841. ACM, 2025.

[57] Baraka D. Sija, Young-Hoon Goo, Kyu-Seok Shim, Huru Hasanova, and Myung-Sup Kim. A survey of automatic protocol reverse engineering approaches, methods, and tools on the inputs and outputs view. *Security and Communication Networks*, 2018(1):8370341:1–8370341:17, 2018.

[58] Snort. Protect your network with the world's most powerful open source detection software. https://www.snort.org, 2025.

[59] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of Annual Network and Distributed System Security Symposium*, NDSS'16, pages 1–16. The Internet Society, 2016.

[60] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the International Conference on Compiler Construction*, CC '16, pages 265–266. ACM, 2016.

[61] Han van der Aa, Henrik Leopold, and Hajo A. Reijers. Checking process compliance against natural language specifications using behavioral spaces. *Information Systems*, 78:83–95, 2018.

[62] Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. Teaching code llms to use autocompletion tools in repository-level code generation. *ACM Transactions on Software Engineering and Methodology*, 34(7):1–27, 2025.

[63] Jincheng Wang, Le Yu, and Xiapu Luo. Llmif: Augmented large language model for fuzzing iot devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, SP'24, pages 881–896. IEEE, 2024.

[64] Yipeng Wang, Xingjian Li, Jiao Meng, Yong Zhao, Zhibin Zhang, and Li Guo. Biprominer: Automatic mining of binary protocol features. In *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '11, pages 179–184. IEEE, 2011.

[65] Yipeng Wang, Xiaochun Yun, M. Zubair Shafiq, Liyan Wang, Alex X. Liu, Zhibin Zhang, Danfeng Yao, Yongzheng Zhang, and Li Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *Proceedings of the IEEE International Conference on Network Protocols*, ICNP'12, pages 1–10. IEEE, 2012.

[66] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: automatic reverse engineering of encrypted messages. In *Proceedings of European Conference on Research in Computer Security*, ESORICS'09, page 200–215. Springer, 2009.

[67] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[68] Wireshark. The world's leading network protocol analyzer. https://www.wireshark.org/, 2025.

[69] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, pages 1–35, 2025.

[70] Daoqing Yang, Yu Yao, Yao Shan, Yunfeng Wu, Xiaoli Lin, Wei Yang, and Licheng Yang. Insyfer: Industrial control protocols syntax inference via graph representation learning. *IEEE Transactions on Dependable and Secure Computing*, 22(6):7495–7507, 2025.

[71] Hengdi Ye, Bing Shui, Jielun Wu, Yufan Zhou, Baowen Xu, and Qingkai Shi. The SPAR artifact. https://doi.org/10.5281/zenodo.20409471, 2026.

[72] Yapeng Ye, Zhuo Zhang, Fei Wang, Xiangyu Zhang, and Dongyan Xu. Netplier: Probabilistic network protocol reverse engineering from message traces. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'21, pages 1–18. The Internet Society, 2021.

[73] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the USENIX Security Symposium*, USENIX Security'18, pages 745–761. USENIX, 2018.

[74] Michal Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2025.

[75] Tao Zhang, Yu Jiang, Runsheng Guo, Xiaoran Zheng, and Hui Lu. A survey of hybrid fuzzing based on symbolic execution. In *Proceedings of the International Conference on Cyberspace Innovation of Advanced Technologies*, CIAT'2020, pages 192–196. ACM, 2021.

[76] Xiaohan Zhang, Cen Zhang, Xinghua Li, Zhengjie Du, Bing Mao, Yuekang Li, Yaowen Zheng, Yeting Li, Li Pan, Yang Liu, and Robert Deng. A survey of protocol fuzzing. *ACM Computing Survery*, 57(2):35:1–35:36, 2024.

[77] Mingwei Zheng, Qingkai Shi, Xuwei Liu, Xiangzhe Xu, Le Yu, Congyu Liu, Guannan Wei, and Xiangyu Zhang. Pardiff: Practical static differential analysis of network protocol parsers. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):1208–1234, 2024.

[78] Mingwei Zheng, Danning Xie, Qingkai Shi, Chengpeng Wang, and Xiangyu Zhang. Validating network protocol parsers with traceable rfc document interpretation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1772–1794, 2025.

[79] Mounira Nihad Zitouni, Amal Ahmed Anda, Sahil Rajpal, Daniel Amyot, and John Mylopoulos. Towards the llm-based generation of formal specifications from natural-language contracts: Early experiments with symboleo. In *Proceedings of the IEEE/ACM Requirements Engineering for AI-powered SoftwarE*, RAISE'25, pages 1–9. IEEE, 2025.

[80] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'21, pages 489–502. USENIX, 2021.