Enhancing Protocol Fuzzing via Diverse Seed Corpus Generation

Zhengxiong Luo*, Qingpeng Du†, Yujue Wang‡, Abhik Roychoudhury*, and Yu Jiang‡

*National University of Singapore †Beijing University of Posts and Telecommunications ‡Tsinghua University

***Abstract*—Protocol fuzzing is an effective technique for discovering vulnerabilities in protocol implementations. Although much progress has been made in optimizing input mutation, the initial seed inputs, which serve as the starting point for fuzzing, are still a critical factor in determining the effectiveness of subsequent fuzzing. Existing methods for seed corpus preparation mainly rely on captured network traffic, which suffers from limited diversity due to the biased message distributions present in real-world traffic. Protocol specifications encompass detailed information on diverse messages and thus provide a more comprehensive way for seed corpus preparation. However, these specifications are voluminous and not directly machine-readable.**

To address this challenge, we introduce PSG, which enhances protocol fuzzing by leveraging large language models (LLMs) to analyze protocol specifications for generating a high-quality seed corpus. First, PSG systematically reorganizes the protocol specification metadata into a structured knowledge base for effective LLM augmentation. Then, PSG employs a grammar-free method to generate target protocol messages and incorporates an iterative refinement process for better accuracy and efficiency. Our evaluation on 7 widely-used protocols and 13 implementations demonstrates that PSG can effectively generate diverse, protocol-compliant message inputs. Moreover, the generated seed corpus significantly improves the performance of state-of-the-art black-box and grey-box protocol fuzzers, achieving higher branch coverage and discovering more zero-day bugs.

***Index Terms*—Protocol Fuzzing, Protocol Grammar, Code Generation**

I. Introduction

Protocol implementations are crucial to network infrastructures and, by their nature, are typically exposed directly to the network. As such, they must accurately process any malformed or malicious traffic to prevent security breaches. Undetected vulnerabilities in these implementations can render devices vulnerable to adversaries. For example, the infamous Heartbleed vulnerability [1] in OpenSSL could expose sensitive data and was widely exploited, affecting a significant portion of the Internet infrastructure. Therefore, ensuring the reliability of protocol implementations is crucial.

Fuzzing is an effective testing technique for uncovering bugs in real-world software [2]. In protocol fuzzing, a fuzzer continuously generates input messages and sends them to the target protocol program to uncover potential anomalies. Based on the input-generation method, protocol fuzzers are classified into generation-based and mutation-based. Generation-based fuzzers [3], [4], [5], [6], like BooFuzz [7], generate messages from scratch by adhering strictly to a user-provided protocol model. While this approach can effectively cover the logic within the model, it cannot exercise logic beyond the confines of the model, as pointed out by [8]. Thus, their effectiveness heavily relies on the quality of this user-defined model. However, crafting a comprehensive model is non-trivial due to varied message formats and complex interaction logic.

Mutation-based approaches [9], [10], [11], [12], such as AFLNet [13], produce new inputs by mutating existing ones from a corpus. This corpus is typically initialized with seed inputs extracted from real-world traffic captured during utility exchanges [14], where each input corresponds to a concatenated message sequence. Given that the input space typically maps to the program state space, triggering different program processing logic requires diverse inputs reflecting the enormous variations in protocol formats. Despite much research in optimizing the input mutation [10], [13], [12], the efficacy of mutation-based fuzzers remains highly dependent on the quality and diversity of the initial seed corpus, as it shapes the initial state space for fuzzing and impacts the direction of further exploration. This is particularly true for protocol fuzzing due to its complex logic, where a valid input requires correct message format, sequence, and parameter dependencies—making it challenging to create valid inputs covering new formats through relatively random mutation. However, traditional seed corpus preparation methods lack diversity due to message distribution biases present in network traffic and evolving protocol features. To overcome this, we can utilize protocol specifications instead of relying on the captured traffic for a diverse seed corpus. Protocol specifications, i.e., RFCs (Request for Comments), standardize protocols to ensure interoperability and provide comprehensive details on different formats. However, these RFCs are voluminous with complex interrelationships and are written in natural language rather than machine-readable formal language. Meanwhile, they describe message formats abstractly without specific instantiation details, which need to be contextualized within specific interaction scenarios. Thus, we need an automated tool to accurately interpret RFCs and effectively apply them to generate input messages.

Recent advancements in large language models (LLMs) have shown significant potential in handling such tasks owing to their powerful natural language processing capabilities. Motivated by this, we propose PSG, which leverages an LLM to analyze protocol RFCs and generate a high-quality initial seed corpus for enhancing protocol fuzzing. To achieve this approach, we need to address two main challenges: (i) How to process extensive RFCs for effective LLM augmentation? While the LLM possesses partial knowledge of standardized protocols from its training data, this knowledge remains in-

complete. It is, therefore, necessary to augment the LLM with protocol-specific knowledge. However, each protocol has numerous RFCs with complex interrelationships and substantial content, challenging the LLM's ability to comprehend and utilize them for input generation. (ii) How to effectively generate valid input for various protocols? Generating valid input for protocols is challenging due to the complex message formats and dependencies. Besides, given that different protocols exhibit customized formats, a versatile method that is applicable across diverse protocols while ensuring generation accuracy and efficiency is essential.

For the first challenge, we collect the specification metadata of the target protocol and systematically reorganize it by utilizing protocol parameter registry information and handling their complex interrelationships. This approach constructs a comprehensive overview that outlines various formats within a structured knowledge base, explicitly illustrating their affiliation and dependency relationships. Meanwhile, for an effective LLM augmentation, we enrich this knowledge base by retrieving pertinent RFC content related to specific message generation, ensuring that the LLM is provided with essential yet concise information for better performance.

To address the second challenge, we leverage the established knowledge base to augment the LLM and implement optimization for efficient input generation. First, we plan generation tasks based on the recovered overview of diverse formats while considering their dependencies. Subsequently, directly generating the target messages is challenging due to complexities inherent in formats, which involve field values, relationships, octets, and encoding. A common approach in this domain is to develop specialized grammars coupled with associated generators, which facilitate clear delineation of message formatting logic while avoiding the pitfalls of low-level details involving field computation or message assembly. However, crafting a universal grammar capable of expressing all possible formats is challenging, especially when it comes to varied field relationships. Motivated by this concept yet seeking more flexibility and efficiency, we exploit the LLM's coding abilities to generate Python code—a flexible high-level programming language familiar to the LLM—as an intermediate step in the generation of target messages. This yields a grammar-free approach that is more adaptable. Furthermore, given the complexity of protocol formats and versatile dependencies, generating the correct input in one step remains a challenge. To improve accuracy and efficiency, we incorporate an iterative process that incrementally refines the message by addressing issues identified within the currently erroneous message under dual guidance: (i) *server feedback*, which provides validation or error information; and (ii) *analysis of historical successes*, which aids in pinpointing error-prone fields requiring attention. Specifically, regarding learning from success, we employ message alignment to identify modified bytes that correct original flaws during refinements and conduct lightweight program analysis on the generation code to trace back relevant message fields requiring attention. This strategy not only provides insights useful for subsequent generations to avoid recurring pitfalls, achieving a self-improving generation pipeline, but also improves method generalization when applied across various protocol implementations—especially when server feedback lacks detailed, actionable information.

We evaluate PSG on 7 widely-used protocols: BGP, HTTP, DNS, CoAP, RTSP, SSH, and FTP. The results show that PSG can generate syntactically and semantically correct inputs for all desired formats. It covers $2.9\times$ more unique formats than a method relying solely on the internal knowledge of the LLM without augmentation, and $1.7\times$ more than a method employing the knowledge base to directly generate messages without optimization and guidance. To further demonstrate the effectiveness in enhancing fuzzing, we utilize the generated seed corpus to initialize state-of-the-art protocol fuzzers, including grey-box fuzzers AFLNet [13] and ChatAFL [12] as well as black-box fuzzer Snipuzz [10], and evaluate their performance on 13 protocol implementations. The enhanced seed corpus enables these fuzzers to cover 11.2%, 9.5%, and 10.3% more code branches , respectively, on average over 24 hours, leading to the discovery of 10 new security-critical bugs in these well-tested, widely-used implementations. Most of them had remained undetected for years by existing techniques. As of the submission, 8 of these bugs have been assigned CVEs, all with CVSS scores rated as high or critical, highlighting their security impact. Our main contributions are as follows:

- We propose to enhance protocol fuzzing by harnessing the LLM to analyze protocol RFCs and generate a high-quality seed corpus.
- We design a method to process extensive protocol RFCs for effective LLM augmentation and propose a versatile method applicable to generate valid and diverse seed inputs for various protocols.
- We implement and evaluate PSG on 7 widely-used protocols. The results demonstrate that PSG effectively generates valid and diverse seed inputs, enhancing state-of-the-art protocol fuzzers by covering more code branches and discovering many security-critical new bugs in well-maintained protocol implementations. Given that the protocols are standardized, the generated corpus can be reused to fuzz different implementations of the same protocol.

II. Motivation

This section uses the Border Gateway Protocol (BGP) to highlight the need for a diverse seed corpus for effective protocol fuzzing and shows the potential and challenges of using the LLM for this task.

BGP Overview. Figure 1 illustrates the BGP state machine from RFC 4271 [15], featuring four states: IDLE, OPENSENT, OPENCONFIRM, and ESTABLISHED. It uses five message types: `OPEN`, `KEEPALIVE`, `UPDATE`, `NOTIFICATION`, and `ROUTE REFRESH`. The process starts in the IDLE state, awaiting a connection. Upon establishing a TCP connection, it moves to OPENSENT, where it awaits another peer's `KEEPALIVE` message to confirm connectivity. If successful, the session progresses to the ESTABLISHED state, where both BGP peers can exchange routing information using either `UPDATE` or possibly `ROUTE REFRESH` messages while maintaining connectivity through periodic `KEEPALIVE` messages. Additionally, any issues signaled by a `NOTIFICATION` message can cause the session to revert to the IDLE state.

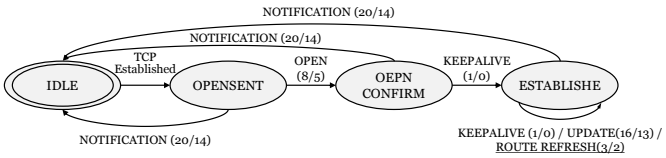


Figure 1: BGP state machine. The ROUTE REFRESH is underlined to show it is used only when specifically enabled during the OPEN exchange. The two numbers in brackets represent the count of formats that are (recognized/unrecognized) by GPT-4o for each message type, respectively.

BGP has five primary message types (or clusters), each of which can encompass various subtypes with different formats for extensibility, as shown in the brackets next to each message type in Figure 1. New features are typically introduced using new *type* codes coupled with specific *value* structures in the Type-Length-Value (TLV) framework. For example, Figure 2 shows an example OPEN message with subtype Multiprotocol Extensions Capability, where *type*, *length*, and *value* fields are shown in Lines 14, 15, and 16-18, respectively. Empowered by this, the OPEN does more than convey basic identifying information (e.g., the *Autonomous System Number* Line 5), but also negotiates capabilities that significantly affect session behavior, including handling of subsequent UPDATE or ROUTE REFRESH messages. These capabilities determine which features are enabled during the session. For example, the Multiprotocol Extensions Capability (RFC 4760 [16], Lines 13-18 in Figure 2) allows multiple address families like IPv6 during routing information exchange. If it is disabled, related routing information expressed in these address families in UPDATE messages will be rejected. Similarly, the Route Refresh Capability (RFC 2918 [17]) allows peers to send ROUTE REFRESH messages in the ESTABLISHED state. Without it, any sent ROUTE REFRESH messages will be discarded by the receiver. Similarly, other message types each also have various subtypes with distinct formats and semantics, affecting the receiver's behavior differently. This diversity in message format is also present in other broad protocols like DNS and HTTP, as shown in our statistics in Section IV-C.

Observation-1: Protocol typically features numerous message formats. Generating diverse inputs reflecting these variations is crucial for activating the corresponding processing logic in implementations.

Impact of Seed Corpus on Fuzzing Exploration. A fuzzer's input generator is responsible for producing diverse inputs reflecting the enormous variations in the protocol message formats. How effectively does a fuzzer achieve this goal?

Figure 2 shows the mutation trace from an OPEN message with Multiprotocol Extensions Capability (with value $\langle AFI, Reserved, SAFI \rangle$) to one with Route Refresh Capability (with no capability value and capability length set to 0). This involves altering five fields and detecting three, as highlighted in Figure 2. This is a complex task necessitating multiple mutation steps due to low success rate in one attempt. However, partial mutations may lead to rejection;

```
01 Marker = 0xffffffffffffffffffffffffffffffff
02 Length = 37                                    => 33
03 Type   = OPEN (1)
04 Version = 4
05 My AS          = 64512
06 Hold Time      = 180
07 BGP Identifier = 192.168.10.5
08 Opt Param Len  = 8                             => 4
09 \Opt Params\
10 | Param Type   = Capability (2)
11 | Param Length = 6                             => 2
12 | \Param Value\
13 | |--[Multiprotocol extensions capability]--
14 | | Code     = Multiprotocol extensions (1)    => Route refresh (2)
15 | | Length   = 4                               => 0
16 | | AFI      = IPv6 (2)                        => Delete
17 | | Reserved = 0                               => Delete
18 | | SAFI     = Unicast (1)                     => Delete
```

Figure 2: A BGP OPEN message with Multiprotocol Extension Capability, along with its mutation trace to Route Refresh message. For bracketed fields, bracketed values are actual values, and unbracketed text shows semantics.

for example, changing only the *Length* field in Line 2 may fail sanity checks. Such rejected inputs would be discarded by the fuzzer as they do not improve code coverage. For further demonstration, we conducted a statistical analysis of fuzzing BGP implementations using the well-known fuzzer AFLNet. Starting with an initial seed corpus covering ten different message formats, after 48 hours of fuzzing—even with optimized mutation strategies—AFLNet explored only five more message formats out of 82 total. This highlights the importance of high-quality seed corpus for effective fuzzing.

Observation-2: Protocol fuzzers have introduced various optimizations in the mutation strategy for input generation, though, the quality of the initial seed corpus remains a critical factor limiting effectiveness.

The seed corpus is typically crafted based on captured messages exchanged between protocol utilities, but this lacks diversity due to the simplicity of project-provided utilities and biases in real-world message distribution [18]. Protocol implementations are developed based on the RFCs for interoperability, making them a comprehensive and accurate source for diverse message formats. However, natural language-expressed RFCs are not directly usable by computers. Recent advances in LLM offer the potential to bridge this gap.

LLM's Awareness of Different Message Formats. Recent research ChatAFL [12] shows the LLM's effectiveness in mitigating seed corpus diversity issue for *textual protocols* by leveraging their internal knowledge. The insight is that the LLM is pre-trained on vast internet data that already includes publicly available protocol RFCs. However, we question if the LLM *accurately* and *comprehensively* captured this knowledge for directly generating diverse inputs for protocol fuzzing. To explore this, we empirically study GPT-4o's awareness of different BGP message formats.

First, we evaluated GPT-4o's awareness of five primary message clusters and found it can provide comprehensive and accurate answers, aligning with ChatAFL's fundamental insights.

Then, we further investigated its grasp of different formats within each cluster by querying format details three times per format name, considering at least one correct answer as

successful recognition. Figure 1 shows that, out of all 82 message formats, GPT-4o correctly answered 48 of them (59%), indicating the LLM's incomplete or inaccurate knowledge capture. Despite all related RFCs being available before 2022 and likely included in its training data [19], the model exhibits a narrower expert knowledge boundary compared to the broader scope of the training data due to the long-tail effect [20]. Notably, accurately answering these questions is just one step towards comprehensive message generation, which also requires correct field values and message orders to satisfy protocol constraints—a challenging task demanding strong reasoning skills. This is further supported by the results in Section IV-D.

Observation-3: GPT-4o, as a representative LLM, has partial but not comprehensive awareness of different message formats for specific public protocols. Thus, relying solely on its encoded knowledge is insufficient for generating diverse inputs covering varied formats needed for protocol fuzzing.

Basic Idea and Challenges. As discussed above, a diverse seed corpus covering various message formats is crucial for effective fuzzing, with RFCs serving as a comprehensive source for various message formats. While the LLM may only capture partial protocol knowledge, its excellent NLP abilities can help bridge the gap between natural language-expressed RFCs and machine-readable inputs by augmenting additional RFC information. This approach offers a general solution beyond relying solely on LLM-encoded knowledge and thus can be applied to similar tasks. To achieve this approach, we must tackle the following two challenges:

Challenge#1*: Handling complex and extensive RFCs for effective LLM augmentation.* This involves two aspects: (i) As shown above, the LLM lacks an overarching view of various message formats, necessitating reconstructing this view for generation task planning. (ii) RFCs are complex and extensive, requiring careful processing. For instance, BGP has 188 RFCs totaling 3455 pages [21], but not all content is relevant to message formats. Meanwhile, RFC interrelations can be intricate—some may update or obsolete others. Therefore, simply feeding all related RFCs into the LLM may exceed text window limits, and irrelevant content could hinder the learning effect [22], [23].

Challenge#2*: Designing a generation strategy that can effectively generate valid inputs for various protocols with varying grammars.* Generating valid protocol inputs is non-trivial, requiring messages to be structurally and semantically correct while adhering to constraints on message order and fields. This challenge intensifies with binary protocols, where field names are implicitly referred to by their positions within the message, and the semantic meanings of field values are not directly interpretable (e.g., value 2 indicates "IPv6" in Figure 2). Textual protocols, however, use clear key-value pairs where both field names and values are explicitly stated. Meanwhile, automatically verifying generated inputs is also challenging. While the LLM exhibits some self-reflection [24], this is unreliable for validation. Besides, diverse grammar across various protocols further complicates developing a universal input generation strategy.

III. System Design

In this section, we first introduce the workflow of PSG and then elaborate on the design details.

Overview. Figure 3 shows PSG's overview. The grey boxes are the inputs and the blue box is the output. Inputs include specification metadata from the Internet Engineering Task Force (IETF) website [25], where official protocol specifications are published, and the under-test protocol server. This server access aids in generating inputs by providing feedback. After analysis, PSG outputs a high-quality seed corpus covering diverse message formats for further fuzzing. At a high level, PSG comprises two main components addressing the two challenges discussed in Section II.

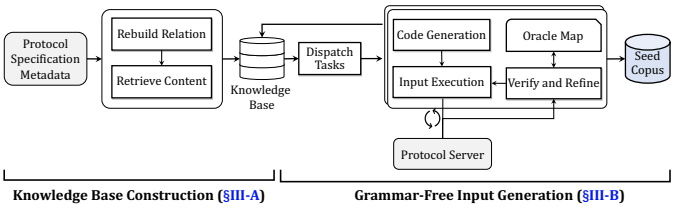


Figure 3: PSG Overview. (a) The knowledge base construction module organizes message formats, rebuilds their relationships, and retrieves related RFC content. (b) The grammar-free input generation module generates the desired messages via an intermediate code generation process and iteratively refines the code for accuracy and efficiency.

(1) Knowledge Base Construction. This module builds a detailed knowledge base from the target protocol's specification metadata to augment the LLM. The knowledge base outlines various protocol formats, their affiliations, and dependencies, serving as a guide for generating inputs in subsequent tasks. Meanwhile, for each message format, we extract key information from extensive RFC documents to ensure essential details are readily available for generation.

Figure 4 shows an example snippet of the knowledge base for BGP. The knowledge base is structured hierarchically: primary message types or clusters are at the top level, followed by extensible fields at the second level. Extensible fields are those that (i) have multiple predefined values, each corresponding to a specific format, or (ii) can be optional in messages (e.g, *Accept* header name in HTTP). These extensible fields can also be nested, creating a tree-like structure represented with additional levels in our format. Leaf nodes provide details about the corresponding formats, including field code (or field name for the optimal extensible fields, i.e., the class (ii) above), dependency, reference RFCs, and retrieved content from these RFCs. Take the `ROUTE REFRESH` message in Figure 4 as an example. The `Subtype` field is an extensible field with registered values including `Route Refresh` and `BoRR`. The `Route Refresh` is denoted using code 1. This format requires the presence of `[OPEN].[Capability].[Route Refresh]` message, since the `Route Refresh` message is only valid when the `Route Refresh Capability` is enabled in the `OPEN` message, as illustrated in Section II. The relevant specifications can be found in Sections 3 of RFC 2918 and Section 4 of RFC 5291. This hierarchical structure provides a clear and

organized view of the message formats and their details, which is essential for the subsequent input generation tasks.

```
OPEN:
  Capability:
    Multiprotocol Extensions:
      Code: 1
      Dependency: null
      Reference: RFC 2858
      Retrieved Content:
        - [RFC 2858 Section 7]
      Generated Message: null
    Route Refresh:
      ...
...
```

```
ROUTE REFRESH:
  Subtype:
    Route Refresh:
      Code: 0
      Reference: RFC 2918, RFC 5291
      Dependency: [OPEN].[Capability].[Route Refresh]
      Retrieved Content:
        - [RFC 2918 Section 3]
        - [RFC 5291 Section 4]
      Generated Message: null
    BoRR:
      ...
```

Figure 4: Example knowledge base for BGP in YAML.

(2) Grammar-Free Input Generation. Based on the established knowledge base, this module first plans the generation tasks based on the dependency requirements and then generates them sequentially. For each task, such as constructing an input containing `OPEN` message with `Multiprotocol Extension Capability`, it extracts the retrieved content from the knowledge base as the background knowledge to support LLM augmentation. Instead of directly generating messages, we instruct the LLM to produce a code that constructs the target messages, enabling a grammar-free generation. Then, this module sends the generated input to the protocol server and receives the response as feedback, which serves two purposes: (i) validation—successful inputs are added to the seed corpus. (ii) refinement guidance—when errors occur, this feedback would be analyzed to identify and correct the error iteratively. During the generation, we also maintain an *Oracle Map* that records historical refinement experiences. It is built from successful refinements and uses lightweight code analysis to pinpoint error causes, providing guidelines for addressing similar errors in future refinement. After completing all generation tasks, this module outputs a high-quality seed corpus with diverse message formats, which is used for further effective fuzzing.

A. Knowledge Base Construction

(1) Rebuild Relation. First, we rebuild the overarching view of diverse protocol formats, including their affiliation and dependencies.

(i) Relation Extraction. In this step, we re-establish different message formats and their affiliated types. We leverage the information from the Internet Assigned Numbers Authority (IANA) registry [26]. IANA coordinates the allocation of values for newly supported protocol parameters, such as message types, options, and extensions. This provides comprehensive information about the assigned values of each parameter. So, our main objective is to classify these protocol parameters into their attributive parent message types. We achieve this by prompting the LLM to identify each parameter's corresponding message type and then grouping related parameters together. Based on this affiliated relationship, we further fill in all the listed possible values for each parameter.

(ii) Update Reference RFCs. The RFCs recorded in the IANA registry are not always up-to-date because protocol specifications evolve continuously, while the registry typically records the original version. To ensure accuracy, we need to update the reference RFCs for each entry. We achieve this by analyzing the RFC header to determine its status and relation with other RFCs (e.g., whether it is obsoleted or updated), following guidelines from the *RFC Style Guide* [27]. Let $S = \{R_1, R_2, \ldots, R_n\}$ represent the reference RFCs for an entry, we analyze each R_i's header individually: if R_i is obsoleted by another RFC, we replace it in S; if R_i is updated by others, we add those to S. This process is repeated until no more changes can be made to S.

(2) Retrieve Content. Based on the recovered overarching view of different message formats established above, we further retrieve the relevant content from RFCs for each entry in the knowledge base (see Figure 4). This helps to filter out irrelevant information and retain necessary yet concise information, ensuring that the LLM has essential knowledge.

(i) RFC Splitting. First, for each message format, we analyze its reference RFCs individually. Instead of directly splitting one RFC document into fixed-length segments—which can disrupt content integrity and context—we leverage the structured nature of the RFC, i.e., each section is self-contained and less reliant on others. Therefore, we use sections as our primary unit for splitting. This approach produces multiple blocks per RFC document, each with a manageable size and self-contained content.

(ii) Block Selection and Dependency Identification. Next, we analyze each block individually to select blocks relevant to the corresponding format and identify the dependencies between different formats. We leverage the LLM's text comprehension capabilities to determine the relevance using the prompt shown in Figure 5. Meanwhile, to address potential bias in the LLM's responses, we adopt a voting mechanism by asking the same question multiple times to obtain a consensus. These selected blocks and identified dependencies are then stored in the "Retrieved Content" and "Dependency" fields of the corresponding entry in the knowledge base.

```
You are analyzing a section from [RFC number]. Your task is to determine:
1. if this section provides format specifications directly related to [Message Format].
   Specifically, look for field values, lengths, structures, and their relationships as
   they pertain to [Message Format]. Respond with "yes" only if it includes related
   specifications. Otherwise, respond with "no".
2. if this section indicates a dependency between [Message Format] and [Other Message
   Format List]. Respond the dependency if it exists. Otherwise, respond with "null"
```

Figure 5: Prompt template for knowledge base construction with placeholders, where [RFC number], [Message Format], and [Other Message Format List] denote the actual RFC number, target message format, and the list of other message formats, respectively.

B. Grammar-Free Input Generation

After constructing the knowledge base, the next step is to generate valid inputs for the target protocol. This involves planning generation tasks to ensure message dependencies using the protocol overview provided by the knowledge base and utilizing stored necessary knowledge to augment the LLM to generate diverse yet protocol-compliant inputs. Algorithm 1 outlines the key steps.

(1) Dispatch Tasks. PSG first plans the generation tasks and then conducts them sequentially (Line 1). This planning process should ensure that (i) all message formats are covered,

Algorithm 1: Grammar-Free Input Generation

```
Input : S - Information of the protocol server
        B_k - Knowledge base for the target protocol
Output: Corpus - Generated seed corpus
1  for F in TaskPlanning(B_k) do
2      I ← null
3      D_F, Π_F = SearchKnowledgeBase(B_k, F) // Prepare
         format details D_F and prefix message sequence
         Π_F for subsequent generation
4      for (attempts from 1 to MaxGenerationAttemps) and (I is null)
         do
           // Generate an initial input
5          LLM.init()
6          C°_F = LLM(P_gen[S, D_F])
7          M°_F = Execute(C°_F)
8          R_{M°_F} = Q^S⟨Π_F, M°_F⟩
           // Validate the input and refine it if necessary
9          R_LLM =
             LLM(P_val_ref[M°_F, R_{M°_F}, OracleMap{R_{M°_F}}])
10         if Validate(R_LLM) then
11             UpdateKnowledgeBase(B_k, F, M°_F)
12             I = Concatenate(Π_F, M°_F)
13             Corpus.append(I)
14         else
15             for (i from 1 to MaxRefineDepth) and (I is null) do
16                 C^i_F = Extract(R_LLM)
17                 M^i_F = Execute(C^i_F)
18                 R_{M^i_F} = Q^S⟨Π_F, M^i_F⟩
19                 R_LLM =
                     LLM(P_val_ref[M^i_F, R_{M^i_F}, OracleMap{R_{M^i_F}}])
20                 if Validate(R_LLM) then
21                     UpdateKnowledgeBase(B_k, F, M^i_F)
22                     I = Concatenate(Π_F, M^i_F)
23                     Corpus.append(I)
24                     Δ_B(M^i_F) = AlignDiff(M°_F, M^i_F)
25                     NewAttenFields =
                         Probe(C^i, Δ_B(M^i_F))
26                     UpdateOracleMap(R_{M°_F}, NewAttenFields)
27 return Corpus
```

and (ii) dependencies between different message types should be considered to determine generation order. For example, for Figure 1, a KEEPALIVE message should follow an OPEN message. Therefore, the generation of the KEEPALIVE message should be scheduled after the OPEN message. This dependency is crucial as it enables the protocol server to transition into an appropriate state, thereby facilitating more precise feedback during the evaluation of the generated KEEPALIVE messages. To achieve this sequencing, PSG leverages these dependencies to establish a macro-level order for generating messages. It starts with OPEN messages, followed by KEEPALIVE messages, then UPDATE messages, etc. Within each message type, PSG processes all associated message formats before moving on to the next type. For BGP protocol messages specifically, it completes all formats of the OPEN message before proceeding with those of the subsequent types like the KEEPALIVE.

For each generation task, e.g., generating an input containing format F of message type T, the algorithm retrieves details D_F from the knowledge base using F as entry and computes the prefix message sequence Π_F (Line 3). To prepare this prefix message sequence, it first identifies the necessary message type prefixes $\Pi_T = \langle \Pi_T^0, \Pi_T^1, \ldots, \Pi_T^n \rangle$ to drive the protocol server to a state that accepts message type T. Here, aim for brevity by computing the shortest sequence. Since modern protocol fuzzers already incorporate sequence-level mutations, our focus is on generating concise sequences as seed inputs. Then, the algorithm instantiates each message type $\Pi_T^i (i \in [0, n])$ as Π_F^i by selecting a concrete message from message type T. Here, the concrete message is guaranteed to be already generated and stored in the knowledge base since the generation order computed in the planning ensures this. The selection is based on the dependency records in entry F within the knowledge base. If specific dependencies exist for Π_T^i, it selects the corresponding concrete message accordingly; otherwise, it randomly chooses a message from type T. This process is repeated until all the message types in Π_T are instantiated, forming the final prefix message $\Pi_F = \langle \Pi_F^0, \Pi_F^1, \ldots, \Pi_F^n \rangle$.

(2) Code Generation. For a target format F, the above process prepares the necessary information D_F for augmenting the LLM and prefix message sequence Π_F to drive the protocol server to the appropriate state. The algorithm then enters the generation phase (Lines 4-26).

During this generation phase, the algorithm attempts multiple times (up to MaxGenerationAttempts) to produce a valid input for F (Line 4). Each attempt involves iteratively generating an initial input (Lines 5-9) and refining it until either a valid input is achieved or the maximum refinement depth is reached (Lines 10-26). This design is based on the fact that the initial generation significantly influences the final outcome. If the initial attempt deviates greatly from the correct format, refinement becomes more challenging and time-consuming. Thus, a "restart" mechanism is necessary to ensure efficiency.

In each generation attempt, the LLM is reset (Line 5) and operates in conversational mode to generate the target message gradually. We provide the LLM with server information (S) and format details (D_F) for augmentation (Line 6). D_F contains specifications from RFCs related to F, helping compensate for the LLM's limited protocol format knowledge (see Section II). Server information S, including IP address, port number, and some configuration details, is crucial as some protocol fields may depend on this data for generation.

Providing the above necessary information, we instruct the LLM to generate the desired message. There are several choices for the generation: (i) *Direct Message Generation*. Since protocols can be binary, messages may contain non-ASCII binary data. In this case, instructing the LLM to provide the hex stream for the message is a feasible solution. However, this approach is challenging due to complex field dependencies and hierarchical structures. Our experience shows that while the LLM can assign correct values for most fields, it struggles with correctly generating complete hex streams. It tends to omit some fields during assembly and has difficulty accurately computing relational fields like length and checksum, even when it knows how to calculate them. The inefficiency of this approach is further demonstrated in our evaluation in Section IV-D. (ii) *Grammar-Based Generation*. Designing grammar helps address the above problem of complex field relationships by explicitly defining their logic. The LLM only needs to correctly express the format logic using the grammar without performing error-prone low-level computations. A

generator then uses this logic to create fields and assemble messages, ensuring consistency between expressed logic and computed results. However, this approach is also challenging. First, each protocol has its own format, necessitating a versatile grammar that can represent all possible protocol formats. This demands significant expertise and manual effort and may be infeasible for some complex protocols. Second, implementing both a parser for interpreting this grammar and a generator for message creation is non-trivial. Third, teaching the LLM to use this grammar correctly poses additional challenges.

To address these challenges, we propose an approach that bridges the gap between natural language and protocol language by leveraging the LLM's coding capabilities [28]. Specifically, we instruct the LLM to generate a Python code, C_F°, for constructing the target message (Line 6). Conceptually, Python can be considered a form of grammar. The LLM has acquired an extensive understanding of Python due to their significant exposure to this language during training. This approach is similar to the grammar-based generation but is more flexible and adaptable to various formats because Python code is more expressive than traditional grammar. Given that protocols are typically implemented using programming languages, including their format handling logic, any protocol that can be realized in one programming language can also be represented in Python. Thus, this approach effectively scales across various protocol formats. Python's operational capabilities allow accurate management of complex field dependencies and logic—challenges that the LLM struggles with when generating messages directly. Meanwhile, providing a code compels the LLM to deliver concrete results for each field, avoiding the common issue of producing abstract text or descriptions when asked to generate messages directly. Python was chosen as the generation language over C/C++ or others because it is high-level and easy to read and write, enabling the LLM to concentrate on format logic without falling into low-level issues. Also, Python code can be executed directly without compilation, making it ideal for subsequent interactive refinement.

Figure 6 shows the prompt template for generation (i.e., $\mathbb{P}_{gen}$ in Line 6). Here, we use $\mathbb{P}_{gen}[\mathsf{S}, D_F]$ to represent filling the placeholders in the prompt template with the protocol server information S and format details D_F. Specifically, we instruct the LLM to generate a Python function named *solution()* that returns the required message in bytes format. First, we present the function prototype with essential requirements like avoiding external libraries and ensuring dynamic length calculation through simple example implementation. Avoiding external libraries makes the generated code self-contained and executable without additional dependencies, simplifying execution. This is based on the fact that internal Python functions are sufficient for protocol field computation and encoding. Dynamic calculation is crucial for managing complex field dependencies by delegating these computations to Python, ensuring precise value generation while allowing the LLM to focus on message logic. Then, using one-shot learning [29], we provide an example of generating an MQTT Publish message. This method aids the LLM in understanding the task and formatting its output correctly. Notably, this MQTT example applies universally across the generation of different protocols without modification. It serves primarily as a structural template that demonstrates systematic message construction methodology. Specifically, it shows the LLM how to construct protocol messages through structured field construction, proper encoding patterns (e.g., using `to_bytes()` rather than direct value assignment), dynamic length calculation patterns (e.g., `len()`), and modular assembly approaches. The protocol-specific semantics D_F are systematically prepared by the knowledge base construction process and delivered on demand via the `[FORMAT INFORMATION]` placeholder in the prompt template. Finally, we use the same "QA" form to guide the LLM to output the *solution* function based on the provided detailed format information.

```
Provide a Python function named solution():
def solution():
    """
    This function returns the required message in bytes format with requirements:
    1. No External Libraries: Do not use any external library.
    2. Dynamic Length Calculation: Use `len()` to calculate length fields.
    """
    # Example implementation (update according to specific requirements):
    message_data = b"Example message"
    message_length = len(message_data)
    # Return the message in bytes format
    return message_length.to_bytes(2, 'big') + message_data
```
Instruction and Function Prototype

```
# Q: Construct a MQTT Publish message.
# Solution using Python:
def solution():
    message_type = 3 # Publish
    qos_level = 0
    retain = 0
    flags = (message_type << 4) | (qos_level << 1) | retain
    ## Define variable header
    # Define the topic name
    topic_name = "sensors/temperature"
    topic_name_encoded = topic_name.encode()
    topic_name_length = len(topic_name_encoded) # length calculation expression
    # Define message content
    message_content = "Temperature is 25C"
    message_content_encoded = message_content.encode()
    # Calculate remaining_length using expression
    remaining_length = 2 + topic_name_length + len(message_content_encoded)

    def encode_remaining_length(length):
        encoded_bytes = b""
        while True:
                encoded_byte = length % 128
                length //= 128
                # If there is more data to encode, set the top bit of this byte
            if length > 0:
                encoded_byte |= 128
                encoded_bytes += bytes([encoded_byte])
            if length == 0:
                break
        return encoded_bytes

    # Encode remaining_length using MQTT variable length encoding
    remaining_length_encoded = encode_remaining_length(remaining_length)
    # Build the variable header
    variable_header = topic_name_length.to_bytes(2, 'big') + topic_name_encoded
    # Build the fixed header
    fixed_header = bytes([flags]) + remaining_length_encoded
    # Combine all parts to form the final packet
    result = fixed_header + variable_header + message_content_encoded

    return result
```
One-Shot Example

```
# Q: For a [SERVER INFORMATION], construct a request with
 [FORMAT INFORMATION]
# Solution using Python:
```
Task Details

Figure 6: Prompt template for message generation with placeholders. It instructs the LLM to generate a message using code, showcasing an MQTT message generation example for one-shot learning. This MQTT example is universally applicable across different protocols without modification.

(3) Input Execution. The generated code C_F° is then executed to produce the concrete message M_F° for F (Line 7). Due to complex protocol formats, this message might be invalid, and we lack an automatic validation method. To address this challenge, we use the protocol server, which understands the

protocol, to provide feedback on the generated message. This is based on the fact that a valid message should be accepted by the server with appropriate responses, while an invalid one would trigger an error response including explanatory details or cause disconnection. To obtain in-depth semantic feedback on M_F°, we should ensure that the server is in a state ready to accept F, as the protocol server is stateful and a state mismatch could lead to request rejection before thorough analysis. Therefore, we leverage the prefix message sequence Π_F to drive the server to the correct state before sending M_F°. Further, for a more precise analysis, we extract only response $\mathcal{R}_{M_F^\circ}$ directly related to M_F°, instead of all the responses for a concentration of Π_F and M_F° (Line 8). To illustrate this process, we define the function $Q^{\mathsf{S}}\langle \Pi_F, M_F \rangle$ as: $Q^{\mathsf{S}}\langle \Pi_F, M_F \rangle = \textsc{Extract}(\textsc{SendToServer}(\mathsf{S}, \Pi_F \cdot M_F), M_F)$. Here, SENDTOSERVER sends the combined messages, consisting of the prefix message Π_F and the target message M_F, to server S and receives the responses, while EXTRACT retrieves the portion of the responses that directly relates to M_F.

(4) Verify and Refine. Given the server's response $\mathcal{R}_{M_F^\circ}$, the LLM validates the response and determines whether the generated message is valid (Line 9). If valid, the message M_F° is recorded and used to construct the final input I (Lines 10-13). Otherwise, the algorithm iteratively refines the message M_F^i (i indicates refinement steps) to address the issues identified by $\mathcal{R}_{M_F^\circ}$ (Lines 15-26).

```
Executing the above code generates the request: [GENERATED MESSAGE]. After sending it
to the server, the server [SERVER RESPONSE]. Analyze following these steps:
1. Check the response:
  - If the response is as expected, answer 'yes'.
  - If not, answer 'no' and then analyze as follows.
2. Identify the issue:
  - Analyze the response. Identify which request fields might be causing issues.
  - If response lacks actionable information, review each field to confirm correctness.
  - Note: Pay special attention [ATTENTION FIELDS].
3. Modify the Python code to correct any issue found during the review.
```

Figure 7: Prompt template for message validation and refinement using Chain-of-Thought with placeholders.

In this phase, we employ the LLM in a conversational mode to iteratively refine messages by building on previous interactions and providing continuous feedback. This approach contrasts with most existing LLM-assisted testing research that typically uses the LLM in a one-shot manner. During each iteration, validation and refinement are conducted using three inputs: the previously generated message M_F^i, the server's response $\mathcal{R}_{M_F^i}$, and its corresponding entry in the Oracle Map (Lines 9, 19). The server response typically contains error codes indicating specific meanings and details about the error, such as missing values. The Oracle Map is a dynamically constructed data structure that is initialized as empty at the beginning of each protocol's message generation process and progressively expanded as different message formats are generated. Upon successful refinements, it records the corresponding historical refinement experiences, which subsequently inform the correction of analogous errors in future iterations. It maps server responses to the fields requiring special attention during refinement, defined as $OracleMap : \mathcal{R} \rightarrow \{F_1, F_2, \ldots, F_n\}$, where $\mathcal{R}$ represents the server response, and $\{F_1, F_2, \ldots, F_n\}$ are the fields requiring special attention. The map updates upon successful refinements (Lines 24-26), and the details are depicted later in the refinement process. Figure 7 illustrates the prompt template $\mathbb{P}_{val\_ref}$ used for validation and refinement by filling placeholders with these three parameters (Lines 9, 19). It employs a Chain-of-Thought (CoT) strategy to guide LLM reasoning by breaking tasks into logical steps [30]. This provides a clear and structured path for the LLM to follow: first, check if the response is as expected; if not, identify the unexpected elements and modify the code accordingly. When the feedback lacks actionable information, review each field in the original code and pay special attention to relationship fields and fields requiring attention. The third step is essential since the server response may lack clarity or simply reject without detailed reasons using a reserved error code. In this case, prior experiences stored in the Oracle Map can provide valuable insights.

If the generated message is valid (Line 9), the algorithm records this message M_F° in the corresponding format entry in the knowledge base, providing a concrete message for further prefix message sequence preparation in future tasks (Line 11). It then constructs the final input I by concatenating the prefix message Π_F and the generated message M_F° (Line 12). This concatenation forms a complete and valid protocol message sequence, ready for further fuzzing (Lines 13-13).

If the generated message is invalid (Line 14), the algorithm enters the refinement phase with multiple iterations (Lines 15-26). Each iteration mirrors the initial generation phase by extracting the new code C_F^i (Line 16), executing it to obtain a refined message M_F^i (Line 17), and validating its response $\mathcal{R}_{M_F^i}$ (Line 19). This iterative process repeats until a valid message is generated or the maximum refinement depth is reached. Upon generating a valid message, the knowledge base updates with this refined message M_F^i (Line 21) and constructs the final input I for further fuzzing (Line 22-23). These steps are similar to those in the initial phase but include an additional operation of comparing original and refined messages (Line 24). This comparison helps identify effective code modifications, determining which fields were critical in addressing error responses $\mathcal{R}_{M_F^\circ}$ and making the message valid (Line 25). This information is updated to the Oracle Map, and the details of this process are outlined below.

00	05	00	03	FD	--	--	1A	--	--	09
00	09	00	07	FD	0F	00	1A	00	62	09

Figure 8: Alignment Example.

First, we identify the differences between the original message M_F° and the refined message M_F^i. Since these messages can differ in length, direct comparison is challenging. Prefix or suffix-priority comparisons are also ineffective due to possible changes in the middle of the message. To address this, we employ a technique called message alignment from protocol reverse engineering [31], [32], [18], [33], [34]. This method aligns message bytes by adding padding bytes as needed to uncover nuances while maximizing commonalities, as shown in Figure 8. After alignment, the two messages are the same length, allowing for a byte-by-byte comparison. We further reflect this difference to the refine message M_F^i, represented

as $\Delta_B(M_F^i)$ (Line 24). Its corresponding calculation function is defined as: $\text{ALIGNDIFF}(M_F^\circ, M_F^i) = \{i \mid \mathbb{M}_F^\circ[i \overset{M_F^i}{\mapsto} j] \neq \mathbb{M}_F^i[i \overset{M_F^i}{\mapsto} j], 0 \leq i < \text{len}(M_F^i)\}$. Here, $\mathbb{M}_F^\circ$ and $\mathbb{M}_F^i$ are the aligned versions of M_F° and M_F^i, respectively, and $i \overset{M_F^i}{\mapsto} j$ denotes a mapping function that maps the index i in the original message M_F^i to the index j in the aligned message $\mathbb{M}_F^i$. This mapping is determined by the alignment process. The result $\Delta_B(M_F^i)$ contains the indices of the bytes of M_F^i showing differences. This helps identify which message bytes were modified during refinement, offering insights into the subsequent back-stepping of modified message fields.

Second, based on the identified byte differences $\Delta_B(M_F^i)$, we backtrack their corresponding fields. Since the message M_F^i is generated by the code C_F^i, we analyze this code to understand the involved field.

```
Var = Const | Expression
## Instrumentation code below
if type(Var) == int:
    Var = PredefineIntValues
elif type(Var) == bool:
    Var = not Var
elif type(Var) == str:
    Var = StrMutator(Var)
elif type(Var) == bytes:
    Var = BytesMutator(Var)
```

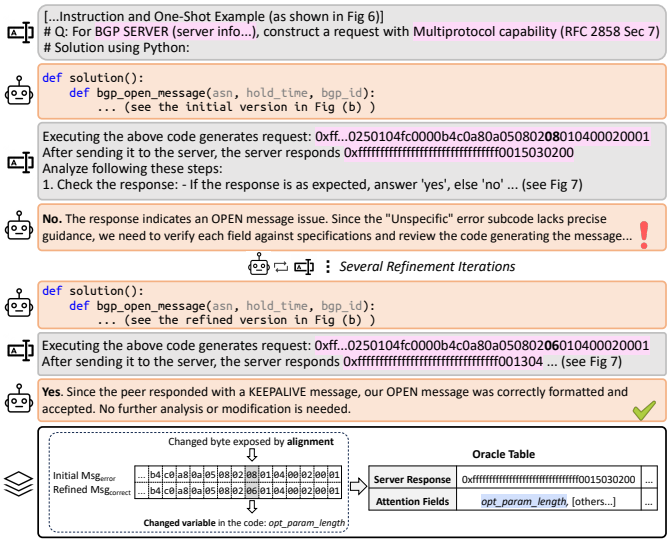
Figure 9: Instrumentation code for probe analysis.

To tackle this problem, we devise a lightweight probe analysis on the refined code C_F^i (Lines 25-26). This method identifies field value assignments in C_F^i and modifies them individually to observe their effects on the *solution()* function's outcome. Specifically, we analyze the Abstract Syntax Tree (AST) of C_F^i to extract the field assignments with form $s : \texttt{Var} = Const \mid Expression$, where we mainly focus on two statement types: (i) *Const*: simple assignment statements using constant values; (ii) *Expression*: mathematical expression-based assignments that may include functions like *len()*. The former applies to standard fields, while the latter pertains to relational fields. Other statements are ignored as they typically involve message assembly (as illustrated in Figure 2 in supplementary material). Let $s_1, s_2, \ldots, s_n$ be the recognized statements in C_F^i. For each statement s_j ($j \in [1, n]$), we insert an instrumentation snippet after s_j to reassign the variable `Var` based on its type, as shown in Figure 9. For example, if `Var` is an integer, we assign it several predefined values, including 0, $2^8 - 1$, etc., to reveal its effect on the output as much as possible. This is similar to a boolean. If `Var` is a string or bytes, we mutate its value while ensuring the length remains the same. This prevents implicit changes to the length field and provides a more precise root cause, addressing the over-tainting problem faced by traditional taint analysis tools. Notably, these cases already cover both *Const* and *Expression* assignment types as the *Expression* result is also in an integer. Let $C_F^i \| \hat{s_j} \|$ donate the code with instrumentation inserted after statement s_j. We check if this modification is effective using the condition underlined: $\text{PROBE}(C_F^i, \Delta_B(M_F^i)) = \{\text{Var}(s_j) \mid \underline{((\text{EXECUTE}(C_F^i \| \hat{s_j} \|) \oplus M_F^i) \cap \Delta_B(M_F^i)) \neq \emptyset}, s_j \in C_F^i\}$. If true, it indicates that modifying s_j affects bytes in $\Delta_B(M_F^i)$, showing that the field assigned by s_j is related to the refinements from M_F° to M_F^i. Then, the involved variable name $\text{Var}(s_j)$ is added to the set of fields requiring special attention. This process is repeated for all statements $s_j \in C_F^i$.

Example Workflow. To illustrate the generation process, we present a concrete example using the BGP knowledge base constructed in Figure 4. Since the `OPEN` message is typically the initial message in BGP communication, PSG begins by generating an `OPEN` message with `Multiprotocol Extension Capability` (with concrete example shown in Figure 2). Figure 10a demonstrates the generation workflow. PSG populates the prompt template from Figure 6 with BGP server information and relevant RFC content (i.e., RFC 2858 Section 7 as recorded in the knowledge base). After obtaining the initial generation code, PSG executes it to produce the concrete message. Given that the `OPEN` message serves as the first communication message and `Multiprotocol Extension Capability` has no dependencies (as documented in the knowledge base), PSG sends the generated message directly to the server without requiring a prefix message sequence. After receiving the server response, PSG incorporates this feedback into the prompt template shown in Figure 7 and instructs the LLM to validate the message. When the response indicates message invalidity, the LLM initiates refinement. However, since the server response only indicates an error without specific details, the LLM systematically reviews the generated code to verify consistency with the specification. Through iterative refinement, the LLM eventually produces a valid message. Figure 10b illustrates both initial and refined versions of the generation code. The initial error stemmed from incorrect processing of the `opt_param_length` field, where the LLM erroneously added 2 bytes for the `capability` code and `length` fields that were already included in the `capability` value calculation. After several refinement iterations, the LLM identifies and corrects this error, achieving server acceptance.

Upon successful generation, PSG updates the knowledge base by storing the generated message in the "Generated Message" field of the corresponding format entry in Figure 4. More importantly, PSG enriches the Oracle Map with insights from this successful refinement. Given the server's limited error feedback, the LLM required multiple iterations to identify the root cause of this subtle error. Therefore, capturing the refinement experience becomes crucial for guiding future generation tasks. PSG employs message alignment to identify differences between original and refined messages, discovering only a single byte change from `0x08` to `0x06`. To determine which field corresponds to this modified byte, PSG applies probe analysis to trace the field variable responsible for this byte value and records this information in the Oracle Map entry associated with the initial server response. This indicates that when encountering similar errors in future generations, the LLM can pay special attention to `opt_param_length`. Regarding locating the root cause in the code, another choice is directly comparing the initial and successfully refined code. However, this approach may introduce noise, as the LLM tends to reconstruct code during refinement, potentially modifying the implementation of multiple fields even when their

corresponding values remain unchanged. Consequently, PSG employs a more precise approach based on field-level probe analysis to accurately identify root cause fields.



(a) The generation workflow, including the conversation with the LLM and update of the Oracle Map after successful refinement.

```
def solution():
  def bgp_open_message(asn, hold_time, bgp_id):
    # BGP OPEN Message Body                          ① Construct lines 4-7
    version = b'\x04' # Version: 4
    my_as = asn.to_bytes(2, 'big')
    hold_time_bytes = hold_time.to_bytes(2, 'big')
    bgp_identifier = bytes(map(int, bgp_id.split('.')))

    # Optional Parameters for MP-BGP          ② Construct and assemble lines 13-18
    capability_code = b'\x01'
    capability_length = b'\x04'
    afi = b'\x00\x02' # AFI for IPv6
    reserved = b'\x00'
    safi = b'\x01' # SAFI for unicast
    capability_value = afi + reserved + safi
    capability = capability_code + capability_length + capability_value

    # Optional Parameter                      ③ Construct and assemble lines 9-12
-   opt_param_length = len(capability) + 2 # 2 extra bytes for Cap. Code and Len<= Initial
+   opt_param_length = len(capability)                                          <= After Refinement
    opt_param_type = b'\x02' # Type 2 for Capabilities
    opt_param = opt_param_type + opt_param_length.to_bytes(1, 'big') + capability

    # Total Optional Parameters Length④ Construct line 8 and assemble lines 4-18
    total_opt_params_length = len(opt_param)
    total_opt_params = total_opt_params_length.to_bytes(1, 'big') + opt_param
    # Construct the OPEN message
    open_message = version + my_as + hold_time_bytes + bgp_identifier + total_opt_params

    # BGP Header          ⑤ Construct lines 1-3 and assemble the whole message
    marker = b'\xff' * 16 # Marker: All ones (16 bytes)
    message_length = len(open_message) + 19 # 19 = 16 (Marker) + 1 (Type) + 2 (Length)
    bgp_header = marker + message_length.to_bytes(2, 'big') + b'\x01' # Type 1 for OPEN

    # Full BGP OPEN message
    bgp_open_packet = bgp_header + open_message
    return bgp_open_packet

  # Configuration for r1 (sender)
  asn = 64512
  hold_time = 180
  bgp_id = "192.168.10.5"
  # Generate the BGP OPEN message to be sent to r2
  return bgp_open_message(asn, hold_time, bgp_id)
```

(b) The LLM-generated code with our explanatory annotation (refer to Figure 2) to better understand the generation code. The code lines of the initial version and the refined version are highlighted.

Figure 10: Generation workflow for BGP `OPEN` message with `Multiprotocol Extension Capability` shown in Figure 2.

IV. Evaluation

In this section, we evaluate PSG from two aspects: (**A1**) the effectiveness of the proposed seed corpus generation method, and (**A2**) how the generated seed corpus enhances the performance of state-of-the-art protocol fuzzers, which is PSG's ultimate goal. From these two aspects, we conduct a series of experiments aiming to answer *four research questions*:

RQ1 How effective is PSG in generating more diverse seed corpus? (A1, Section IV-C)

RQ2 How does each module contribute to the effectiveness of PSG? (A1, Section IV-D)

RQ3 Can PSG help state-of-the-art protocol fuzzers explore more code regions? (A2, Section IV-E)

RQ4 How effective are protocol fuzzers, when augmented with PSG, in exposing new bugs in real-world protocol implementations? (A2, Section IV-F)

A. Implementation

We implement a prototype of PSG in Python 3. The well-known GPT-4o [19] is selected as the large language model. The LLM usually accepts one hyperparameter *temperature* $\in [0, 1]$, which regulates the randomness and creativity. We use temperature 0.5, a common practice to balance the diversity and quality of the generated responses [35]. Besides, this temperature setting of 0.5 is also used by recent related work ChatAFL's initial seed enrichment module [12], which also facilitates a better comparison in the evaluation. PSG comprises two main components: (1) *The knowledge base construction module* includes an RFC content crawler and parser for processing and section splitting. It utilizes the LLM to extract the message formats and the corresponding retrieved content, organizing them in a YAML format shown in Figure 4. In particular, the retrieval RFC content is stored separately in text files due to its large size. (2) *For the grammar-free input generation module*, we implement the message generator and publisher. (i) The message generator follows Algorithm 1. Specifically, we configure *MaxRefineDepth* (Line 15) to two since using the LLM in conversational mode with excessive refinement steps may result in context loss. We evaluate another parameter *MaxGenerationAttempts* (Line 4) in Section IV-C. During program analysis in the refinement, we use Python's *ast* library to extract the abstract syntax trees for identifying the desired assignment statements and their locations for subsequent code instrumentation. (ii) The message publisher interacts with the server by sending generated messages and receiving feedback, including the responses or server behaviors like connection shutdowns.

B. Evaluation Setup

Subjects. We select 7 widely-used public protocols as targets for evaluation, including BGP, CoAP, SSH, DNS, HTTP, RTSP, and FTP. We select these protocols based on various characteristics. (i) Regarding format complexity, they cover different types: HTTP, RTSP, and FTP are textual protocols; DNS is a mixed binary and textual protocol; whereas the remaining are binary protocols. (ii) For their application domain, they are typical protocols in different scenarios: BGP is a routing protocol and is the backbone of the internet; HTTP, DNS, and SSH are essential protocols for daily use; CoAP is a widely-used IoT messaging protocol tailored for constrained devices; RTSP is a streaming protocol used in multimedia applications; FTP is a widely-used file transfer protocol. Their

popularity and varied complexities help evaluate PSG's effectiveness and generalization capabilities. Furthermore, since we use MQTT as the one-shot learning example in generation (see Figure 6), we exclude it from the subjects to avoid bias and to show that this example's generalization when PSG is applied to other protocols.

To determine if the generated seed corpus can enhance performance of existing protocol fuzzers when testing protocol implementations, we further include representative implementations for these selected protocols (see Table I), as referenced in recent works [12], [8], [13], [36], [37]. We select at least one implementation for each protocol. During seed generation for each protocol, we use only one implementation as the target for feedback acquisition. The first-row subjects in Table I for each protocol are used here; e.g., *cpp-httplib* is the interaction target for HTTP seed generation. In the following fuzzing phase, all corresponding implementations are tested, e.g., *cpp-httplib* and *Lighttpd* are tested based on the generated seed corpus. This helps assess the generalization ability of the generated seed corpus across various implementations of the same protocol. Besides, for SSH implementation, we disabled its randomness to ensure the reproducibility of the generated seeds, a commonly used method in protocol fuzzing community [38].

Table I: Detailed information about the selected subjects

Protocol	Subject	#Stars	Description
HTTP	cpp-httplib	13,416	C++ header-only HTTP/HTTPS library.
	Lighttpd	623	Web-server for high-performance environments.
DNS	SmartDNS*	8,615	DNS server to obtain the fastest website IP.
	Dnsmasq	-	Lightweight, easy-to-configure DNS forwarder.
BGP	FRR*	3,507	IP routing protocol suite.
	OpenBGPD*	58	A lightweight and free BGP implementation.
	GoBGP*	3,709	BGP implementation for modern environments.
CoAP	libCoAP	812	CoAP implementation for IoT devices.
	FreeCoAP*	135	CoAP client/server and HTTP/CoAP proxy.
RTSP	GStreamer*	218	Media-handling components.
	LIVE555	781	Multimedia streaming using open protocols.
FTP	ProFTPD	547	Highly configurable FTP daemon for Unix
SSH	OpenSSH	3,392	Implementation of the SSH protocol (version 2)

- * means we have uncovered new bugs in that project (see Table V).
#Stars: Number of stars on GitHub.

Basic Seed Corpus. To prepare the basic seed corpus (i.e., the version not enhanced by PSG) for each subject, we utilize off-the-shelf messages encoded in test suites and example utilities provided by the protocol implementations. Specifically, we exclude unit test suites containing incomplete messages (e.g., only headers without bodies, or specific fields for targeted function testing) because reusing these message snippets would require constructing corresponding complete messages by filling missing fields and handling complex field dependencies, which is non-trivial and error-prone. To dump the test messages encoded in the code, we follow a typical practice: executing the off-the-shelf test server and client utilities or examples while capturing their exchanged messages, as outlined in [14]. To better demonstrate the necessity of a more systematic approach in generating a more comprehensive seed corpus (i.e., the goal of PSG), we combine seed corpus derived from multiple implementations into one initial set. For example, when testing HTTP implementations *cpp-httplib* and *lighttpd*, we merge their respective seed corpus together to form a relatively more diverse seed corpus as a starting point for subsequent fuzzing.

Base Protocol Fuzzers. To evaluate the effectiveness of the generated seed corpus in enhancing fuzzing, we compare the protocol fuzzer's performance using both basic and PSG-enhanced seed corpus. We select AFLNet [13], ChatAFL [12], [39], and Snipuzz [10] as our baseline fuzzers, representing state-of-the-art mutation-based approaches with distinct technical innovations and fuzzing paradigms. Specifically, AFLNet incorporates state feedback mechanisms to conduct state-aware fuzzing; ChatAFL leverages LLM's internal knowledge for protocol grammar acquisition and performs grammar-based mutations; and Snipuzz employs on-the-fly message format inference through message probing to identify grammatical roles of message bytes. Moreover, AFLNet and ChatAFL are grey-box fuzzers, while Snipuzz is a black-box fuzzer that operates without internal instrumentation. This paradigmatic diversity allows us to assess the generalizability of our approach across different fuzzing paradigms. For ChatAFL, we ensure fair comparison by using the same LLM (GPT-4o) as PSG. Since ChatAFL primarily targets textual protocols, we adapt it to support binary protocols by presenting message content in hexadecimal format, as many binary bytes are unprintable in standard text representation. Similarly, we adapt Snipuzz to support binary protocols and UDP-based protocols. While other fuzzers exist, our selected tools are already advanced and represent diverse technical approaches to protocol fuzzing optimization, allowing us to demonstrate how high-quality seed corpus generation enhances existing protocol fuzzers across different methodological foundations.

Evaluation Metrics. First, we evaluate PSG's generated seed corpus itself using *Format Coverage*: identifying semantically and syntactically correct seed inputs and calculating unique formats covered compared to all documented formats in RFCs. Then, we evaluate PSG's fuzzing performance enhancement using two widely-accepted metrics: *Code Coverage* and *Bug Detection*. For the code coverage, we use both branch coverage and line coverage, which are the most commonly used metrics in the fuzzing community.

C. Generation Effectiveness (RQ1)

First, we evaluated the knowledge base construction module's ability to extract message formats from RFCs, which is the basis for later generation. By manual inspection against RFCs and IANA registries, we confirmed that PSG accurately extracted the formats for evaluated protocols, with format numbers shown in Table II.

Table II: Format coverage ($\frac{\#Covered\ Formats}{\#Total\ Formats}$) achieved by basic seed corpus and PSG under different generation attempts n (denote as PSG@n)

Protocol (Total)	Basic	PSG@1	PSG@2	PSG@3	PSG@4	PSG@5
BGP (82)	10 (12%)	55 (67%)	66 (80%)	76 (93%)	82 (100%)	82 (100%)
CoAP (32)	21 (66%)	22 (69%)	25 (78%)	29 (91%)	32 (100%)	32 (100%)
DNS (86)	17 (20%)	79 (92%)	84 (98%)	86 (100%)	86 (100%)	86 (100%)
HTTP (171)	32 (19%)	166 (97%)	171 (100%)	171 (100%)	171 (100%)	171 (100%)
RTSP (43)	14 (33%)	42 (98%)	43 (100%)	43 (100%)	43 (100%)	43 (100%)
FTP (63)	30 (48%)	61 (97%)	63 (100%)	63 (100%)	63 (100%)	63 (100%)
SSH (132)	41 (31%)	67 (51%)	87 (66%)	102 (77%)	119 (90%)	132 (100%)

Then, we further evaluated the grammar-free input generation module's ability to produce messages for the desired formats under different *MaxGenerationAttempts* settings, a parameter (Line 4 in Algorithm 1) that limits generation attempts per message format. As shown in Table II, for HTTP and RTSP, with simple key-value pair structures, PSG achieved mostly 100% coverage within 1 attempt. The results on FTP demonstrate comparable performance, with high coverage achieved within 1 attempt owing to its straightforward textual command syntax. In contrast, BGP, CoAP, and SSH require 4, 4, and 5 attempts , respectively, to achieve 100% coverage due to their more intricate formats involving complex field relationships and message assembly requirements. Nevertheless, empowered by the refinement process, PSG still achieved high format coverage in the initial attempt across all protocols. On average, PSG requires only 1.39 attempts per message format, showcasing both efficiency and accuracy.

Additionally, Table II presents the format coverage statistics for the basic seed corpus. These seeds exhibit low format coverage, averaging only 32.7% across all protocols. This limited coverage underscores the importance of PSG's generation capability, which utilizes the protocol specifications as the analysis source and applies an automated method to analyze these specifications to achieve more comprehensive protocol format coverage.

Message Comparison. To better understand the characteristics of PSG-generated messages, we compare them with those from real-world traffic. Our investigation reveals that for message formats that both approaches contain, PSG-generated messages and those from real-world traffic are roughly similar, especially regarding the validity of key fields (e.g., Type field), since invalid values for key fields would be rejected by the server while PSG-generated messages have all been validated. When examining these shared formats more closely, notable differences can be observed in the following aspects:

(1) Minimal data payloads: PSG-generated messages typically contain concise data fields, whereas real-world traffic often carries extensive payload data to fulfill actual communication needs. This characteristic can enhance fuzzing efficiency because data-related fields typically trigger less complex parsing logic than other structural fields. Since most protocol fuzzers employ byte-level mutation without grammar awareness, extensive mutations on lengthy data content yield diminishing returns while consuming computational resources.

(2) Simplified optional field combinations: PSG-generated messages typically contain fewer combinations of optional fields within individual messages. In real-world traffic, a single message may combine multiple optional fields, while PSG typically treats each optional field as a distinct format and generates them individually across different messages. When optional fields are parsed independently in protocol implementations, combining them within one message does not trigger additional code logic, and fuzzing efficiency remains unaffected. Conversely, when different optional fields interact and their combinations activate more complex code paths, fuzzing efficiency may be impacted. However, classical mutation operators employed by fuzzers, such as `crossover`, which splice segments from different test cases, can help generate additional combinations to mitigate this limitation.

Figure 11 illustrates both characteristics using HTTP protocol examples. The PSG-generated message demonstrates minimal payload content and only essential headers, while the real-world traffic exhibits chunked transfer encoding with multiple optional header fields (`Connection`, `Transfer-Encoding`) combined within a single message.

(a) PSG-generated HTTP message

```
POST /upload HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-
       urlencoded

abc
```

(b) HTTP message from real-world traffic

```
POST /cgi.pl?post-len HTTP/1.1
Host: www.example.org
Connection: close
Content-Type: application/x-www-form-
       urlencoded
Transfer-Encoding: chunked

a0123456789[omitted...]
```

Figure 11: Comparison between PSG-generated and real-world HTTP messages

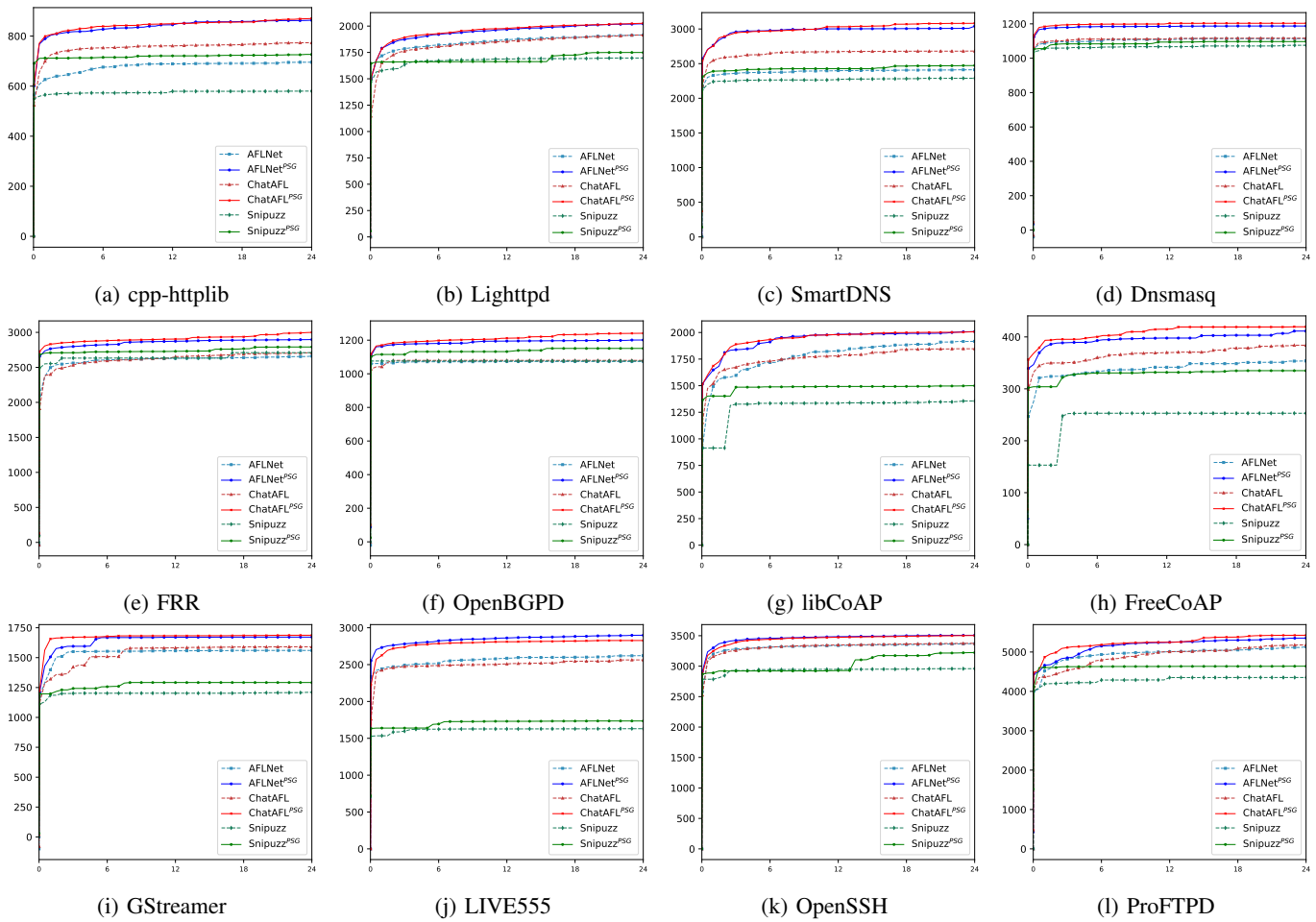
D. Ablation Study (RQ2)

Method. To answer **RQ2**, we conducted an ablation study to evaluate the contribution of PSG's two modules (m_1: knowledge base construction and m_2: grammar-free input generation) towards increasing the format coverage. Since m_2 relies on m_1, we developed such two variants: (i) PSG$^\circ$: Both m_1 and m_2 are disabled. We directly instruct the LLM to generate as many diverse messages as possible without specific planning guidance from RFC knowledge. This highlights the importance of supplementing the LLM with task-specific RFC knowledge despite its potential exposure from training data. (ii) PSG^{m_1}: only enabling m_1. i.e., using m_1's knowledge base for generation planning and knowledge augmentation. This variant directly generates message hex streams without intermediary code generation and refinement process introduced in m_2. (iii) PSG$^{m_{1+2}}$: full version of PSG with both modules enabled. For a fair comparison, PSG$^{m_{1+2}}$ and PSG^{m_1} use generation attempts as five for each message format (following Section IV-C). As for PSG$^\circ$, due to a lack of structured planning, we instruct it to produce an equivalent number of messages as other variants.

Table III: Format coverage achieved by each PSG variant for each protocol under identical generation attempts of five.

Protocol (Total)	PSG$^\circ$	PSG^{m_1}	PSG$^{m_{1+2}}$
BGP (82)	19 (23%)	22 (27%)	82 (100%)
CoAP (32)	11 (34%)	15 (47%)	32 (100%)
DNS (86)	18 (21%)	40 (47%)	86 (100%)
HTTP (171)	40 (23%)	157 (92%)	171 (100%)
RTSP (43)	16 (37%)	28 (65%)	43 (100%)
FTP (63)	49 (78%)	61 (97%)	63 (100%)
SSH (132)	28 (21%)	42 (32%)	132 (100%)
Average	**34%**	**58%**	**100%**

Results. Table III shows the format coverage achieved by the three variants. Without knowledge base augmentation, PSG$^\circ$ achieves only 34% average format coverage due to limited protocol awareness and understanding. After providing augmentation, PSG^{m_1}'s result increases to 58%, with particularly notable improvements on textual protocols: HTTP and RTSP. This underscores the importance of comprehensive



(a) cpp-httplib (b) Lighttpd (c) SmartDNS (d) Dnsmasq

(e) FRR (f) OpenBGPD (g) libCoAP (h) FreeCoAP

(i) GStreamer (j) LIVE555 (k) OpenSSH (l) ProFTPD

Figure 12: The average code branches covered (y-axis) by baseline fuzzers (AFLNet, ChatAFL, and Snipuzz) and PSG enhanced fuzzers (AFLNet$^{\text{PSG}}$, ChatAFL$^{\text{PSG}}$, and Snipuzz$^{\text{PSG}}$) over 24 hours (x-axis) on each subject across ten repeated runs.

protocol overviews and task-specific knowledge augmentation. However, even with the planning and RFC augmentation, PSG^{m_1} doesn't reach 100% because the complex message formats make direct message generation error-prone. This is evident from PSG^{m_1}'s limited improvement on the binary protocols BGP, CoAP, and SSH, as well as mixed protocol DNS. PSG addresses this by introducing code generation as an intermediary step along with a refinement process guided by server feedback and prior success, enabling the LLM to focus on format logic over peripheral details and gradually refine towards the desired message.

E. Code Coverage Analysis (RQ3)

Method. To answer **RQ3**, we use PSG-generated seed corpus as input for baseline fuzzers AFLNet, ChatAFL, and Snipuzz, donated as AFLNet$^{\text{PSG}}$, ChatAFL$^{\text{PSG}}$, and Snipuzz$^{\text{PSG}}$, and compare their performance against the versions using the basic seed corpus (see Section IV-B). Besides, since ChatAFL already incorporates a seed corpus enrichment module, we substitute this module with PSG-generated seed corpus in ChatAFL$^{\text{PSG}}$ to foster a comparison.

Results. Figure 12 illustrates the branch coverage trends across all subjects, while Table IV presents the line coverage data. On average, AFLNet$^{\text{PSG}}$, ChatAFL$^{\text{PSG}}$, and Snipuzz$^{\text{PSG}}$ achieve (11.2%, 9.4%), (9.5%, 8.6%), and (10.3%, 8.4%) higher (branch, line) coverage than their respective baselines. All results are statistically significant per the Mann-Whitney U test [40]. We exclude GoBGP results due to its Go language implementation, which is not supported by GCov. Without high-quality diverse seed inputs, baseline fuzzers easily reach coverage plateaus in early stages. In contrast, the diverse seed inputs generated by PSG activate varied program logic, enabling fuzzers to further explore more code branches and continuously increase code coverage. Notably, ChatAFL$^{\text{PSG}}$ outperforms ChatAFL across all subjects, indicating that PSG's seed corpus is more effective than ChatAFL's seed enrichment module, which relies solely on the LLM's internal knowledge without augmentation—resulting in low diversity—and generates messages directly without optimization strategies—leading to low accuracy. While ChatAFL—a state-of-the-art protocol fuzzer with advanced strategies—outperforms AFLNet on most subjects, this advantage diminishes when using PSG-generated seeds,

highlighting the critical role of a diverse seed corpus in fuzzing. Besides, as for Snipuzz, a black-box protocol fuzzer equipped with advanced format inference capabilities, PSG still enables higher code coverage. This occurs because certain message formats differ substantially from one another, and even with format-aware mutation strategies, it remains difficult to transform a message from one existing format to another unseen format, as valid mutations typically require simultaneous changes to multiple fields, as demonstrated in our example in Figure 2.

Table IV: The average line coverage for each tool on each subject after 24 hours.

	AFLNet	AFLNet^{PSG}	ChatAFL	ChatAFL^{PSG}	Snipuzz	Snipuzz^{PSG}
cpp-httplib	803	938	873	946	719	836
Lighttpd	3935	4060	3935	4072	3691	3826
SmartDNS	6656	8004	7179	8274	6432	6936
Dnsmasq	1725	1831	1731	1862	1700	1729
FRR	8206	8982	8287	9089	8536	8783
OpenBGPD	1890	2113	1901	2196	1918	2052
libCoAP	3446	3581	3296	3576	2818	2980
FreeCoAP	1039	1193	1116	1223	820	1000
GStreamer	3555	3825	3632	3850	2952	3149
LIVE555	5232	5831	5108	5675	3281	3447
OpenSSH	7528	7746	7528	7737	6689	7328
ProFTPD	11273	11826	11423	11968	9799	11021
Improv.		**+9.4%**		**+8.6%**		**+8.4%**

F. Bug Detection (RQ4)

AFLNet^{PSG} and ChatAFL^{PSG} found 10 new bugs in the widely-used implementations of evaluated protocols (see Table V). Eight of them have been assigned CVE identifiers, all of which received CVSS scores of *High* or *Critical*, highlighting their severity.

Without being equipped with PSG, AFLNet, ChatAFL, and Snipuzz only found bug#3, bug#5, and bug#6 in Table V. These subjects are well-maintained and have undergone extensive community testing. *FRR*, *Gstreamer*, and *GoBGP* had even been incorporated into OSS-Fuzz [41] and have been continuously fuzzed for a long time. However, PSG still helps fuzzers to expose new bugs in these projects. The column $\boldsymbol{T}_{Exposure}$ indicates each bug's exposure duration from introduction to discovery. Some bugs persisted for up to 9 years due to inaccessible code paths caused by inadequate initial seed corpus. With a diverse seed corpus generated by PSG, these hidden bugs were revealed, demonstrating its efficacy in guiding fuzzers to previously unexplored code regions where hidden bugs may reside.

V. Discussion

Validation of the Generated Seed Corpus. As shown in Section IV-C in the evaluation, the inputs generated by PSG may sometimes be invalid due to protocol complexity and the unpredictable nature of the LLM. For complex protocols, more generation attempts are needed to achieve a high format coverage, which consequently may incur more invalid inputs within the seed corpus. Despite this, such defects are manageable during fuzzing, as many fuzzers employ feedback mechanisms like coverage feedback to prioritize interesting inputs that increase coverage. Thus, invalid inputs are quickly filtered out by these mechanisms. Furthermore, it should be noted that invalid inputs are not inherently uninteresting as they represent slight variations from valid ones and could potentially trigger unexpected behaviors in the target implementation.

Table V: Statistics of 10 previously-unknown bugs discovered by PSG-enhanced fuzzers in extensively tested subjects

Subject	Bug Type	Threats	$T_{Exposure}$	CVE (CVSS Score)
FreeCoAP	Out-of-bounds Read	DoS	61 Months	2024-31029 (High)
FreeCoAP	Null Pointer Dereference	DoS, ID	108 Months	2024-31030 (Critical)
FreeCoAP	Null Pointer Dereference	DoS	108 Months	2024-40493 (Critical)
FreeCoAP	Buffer Overflow	RCE, DoS	111 Months	2024-40494 (Critical)
OpenBGPD	Undefined Behavior	DoS	45 Months	2024-48082 (Critical)
FRR	Null Pointer Dereference	DoS, ID	41 Months	CVE Requested
GoBGP	Buffer Overflow	DoS, ID	72 Months	2023-46565 (High)
GoBGP	Buffer Overflow	DoS, ID	10 Months	CVE Requested
SmartDNS	Integer Behavior	DoS, MC	56 Months	2024-42643 (High)
GStreamer	Assertion Failure	DoS	62 Months	2024-44331 (High)

* DoS: Denial of Service; RCE: Remote Code Execution; ID: Information Disclosure. $T_{Exposure}$: Exposure time.

Dependence on Server Feedback Quality. Our approach's validation and refinement capabilities for generated messages rely on the quality of server responses. When generated messages contain errors, servers that provide rich, detailed error information enable a rapid and effective refinement process. However, the informativeness of these responses varies significantly across different implementations. While some implementations offer comprehensive error details, specifying exactly which fields or values are problematic, others provide only basic error indicators without actionable diagnostic information. This variability in feedback quality might impact our technique's effectiveness, as low-quality feedback that lacks specific error details can potentially impede the refinement process and require more generation attempts.

Dependence on Specification Quality. The effectiveness of our seed generation approach is inherently tied to the completeness and clarity of protocol specifications. Our method fundamentally relies on well-structured, high-quality RFC documentation to extract comprehensive protocol knowledge. For protocols with ambiguous or poorly maintained documentation-particularly proprietary protocols-the LLM's semantic understanding may be compromised, potentially undermining the effectiveness of our approach. Furthermore, protocols that undergo rapid evolution without timely documentation updates also present ongoing challenges. Future research directions could mitigate these documentation limitations by developing hybrid approaches that combine specification analysis with direct inference of message formats from implementation source code, thereby reducing dependence on external documentation quality.

Benefits for Generation-Based Fuzzers. PSG mainly targets enhancing mutation-based fuzzers, with its outcome directly applicable to these fuzzers. The generated seed corpus can also benefit generation-based fuzzers like BooFuzz [7] and Peach [3]. These fuzzers require a user-defined protocol model that includes data models specifying the message formats. Such data models need to define not only the abstract-level format—comprising field names, octets, and types—but also concrete field values [42], [43], which are usually crafted by referring to captured network traffic and thus also suffer from

the same diversity issue. In such cases, the messages generated by PSG can provide more diverse references for creating these data models.

VI. Related Work

Mutation-Based Protocol Fuzzing. Recent research in protocol fuzzing mainly focuses on optimizing the fuzzing stage, such as mutation strategies and test throughput. AFLNet [13] and NSFuzz [44] enhance mutations using state feedback, while Nyx-net [45] improves throughput with snapshot fuzzing to reduce overhead. The seed corpus is crucial for mutation-based fuzzers as it guides fuzzing exploration, especially in complex protocols. Our work complements these fuzzing optimizations by enhancing the seed corpus foundation to maximize their potential.

Generation-Based Protocol Fuzzing. These fuzzers, including BooFuzz [7] and Peach [3], rely on a user-defined protocol model detailing message formats (data models) and valid message sequences (state model). Creating this model requires a deep understanding of the protocol, which is challenging. These fuzzers generate new inputs by strictly following this protocol model, making its quality crucial for their effectiveness,as revealed by [8]. Our work focuses on enhancing mutation-based fuzzers, but the generated seed corpus can also benefit generation-based fuzzers by providing valuable references for constructing the data models.

LLM Assisted Fuzzing. Recent advancements in the LLM have shown their potential in enhancing fuzzing [46], [47], such as test case generation [48], [49], [50], [12] and test harness generation [51], [52], [53]. TitanFuzz [49] utilizes the LLM to generate test cases for deep learning libraries. Fuzz4All [50] uses the LLM to mutate existing test cases for compiler testing. ChatAFL [12] uses the LLM to enhance the seed corpus and mutation strategies. These efforts focus on text-based test cases without involving complex binary formats and rely solely on the internal knowledge of the LLM. However, recent studies indicate they may lack in-depth domain knowledge [54], [55] and thus fall short of providing a diverse seed corpus. Therefore, various methods have been proposed to integrate domain-specific expertise [56], [57], [58]. One related work is LLMIF [59], which augments the LLM with specifications for fuzzing Zigbee protocols. It separates the generation of header and body and represents extracted message formats in JSON. While effective for Zigbee characteristics, extending this approach to diverse protocol families still requires modifications to accommodate varied protocol requirements and formats. In contrast, PSG introduces a protocol-agnostic framework that leverages code generation as an intermediate step, seamlessly handling complex field dependencies and diverse message formats across the full spectrum of protocols from binary to text-based.

VII. Conclusion

This paper presents PSG, which leverages the LLM to generate diverse seed corpus for enhancing protocol fuzzing. Rather than relying exclusively on the intrinsic knowledge of the LLM, PSG systematically organizes and retrieves extensive protocol knowledge for task planning and precise supplementation when generation. Based on this, PSG exploits the LLM's proficiency in code programming to use it as an intermediate step to bridge the gap between natural-language RFCs and obscure protocol-language messages, thereby ensuring adaptability across various protocols. Meanwhile, lightweight program analysis of the generated code and server feedback synergistically guide PSG to perform an iterative refinement process to improve the generation accuracy. Our experiments show that PSG can effectively generate valid seed inputs for diverse protocols and enhance the performance of the state-of-the-art protocol fuzzers.

Acknowledgment

This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore. This research is also supported in part by the National Key Research and Development Project (No. 2022YFB3104000).

References

[1] Heartbleed, "Cve-2014-0160: Heartbleed - a vulnerability in openssl." 2014, http://heartbleed.com.

[2] G. OSS-Fuzz, "OSS-Fuzz Trophies: As of august 2023, OSS-Fuzz has helped identify and fix over 10,000 vulnerabilities and 36,000 bugs across 1,000 projects." 2024, https://google.github.io/oss-fuzz/.

[3] M. Eddington. (2024) Peach fuzzing platform. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

[4] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, "ICS protocol fuzzing: Coverage guided packet crack and generation," *ACM/IEEE Design Automation Conference (DAC)*, 2020.

[5] Z. Luo, J. Yu, Q. Du, Y. Zhao, F. Wu, H. Shi, W. Chang, and Y. Jiang, "Parallel fuzzing of iot messaging protocols through collaborative packet generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3431–3442, 2024.

[6] J. Yu, Z. Luo, F. Xia, Y. Zhao, H. Shi, and Y. Jiang, "Spfuzz: Stateful path based parallel fuzzing for protocols in autonomous vehicles," *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:273909539

[7] Jtpereyda. (2024) BooFuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz.

[8] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: Packet sequence oriented fuzzing for protocol implementations," *32nd USENIX Security Symposium*, 2023.

[9] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, "Polar: Function code aware fuzz testing of ICS protocol," *ACM Trans. Embed. Comput. Syst.*, 2019.

[10] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference," in *ACM SIGSAC CCS*, 2021.

[11] N. Bars, M. Schloegel, N. Schiller, L. Bernhard, and T. Holz, "No peer, no cry: Network application fuzzing via fault injection," *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:272367787

[12] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *NDSS*, 2024.

[13] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[14] AFLNet. (2020) Prepare message sequences as seed inputs. https://github.com/aflnet/aflnet?tab=readme-ov-file#step-1-prepare-message-sequences-as-seed-inputs.

[15] Y. Rekhter, S. Hares, and T. Li. (2006) A Border Gateway Protocol 4 (BGP-4). [Online]. Available: https://www.rfc-editor.org/rfc/rfc4271.html

[16] R. Chandra, T. J. Bates, Y. Rekhter, and D. Katz. (2007) Multiprotocol Extensions for BGP-4. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4760.html

[17] E. Chen. (2000) Route Refresh Capability for BGP-4. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2918.html

[18] Z. Luo, K. Liang, Y. Zhao, F. Wu, J. Yu, H. Shi, and Y. Jiang, "DynPRE: Protocol reverse engineering via dynamic inference," in *NDSS*, 2024.

[19] OpenAI, "GPT-4o." 2024, https://platform.openai.com/docs/models/gpt-4o#gpt-4o.

[20] N. Kandpal, H. Deng, A. Roberts, E. Wallace, and C. Raffel, "Large language models struggle to learn long-tail knowledge," in *International Conference on Machine Learning*. PMLR, 2023, pp. 15 696–15 707.

[21] IETF. (2024) Document search: Bgp rfcs. [Online]. Available: https://datatracker.ietf.org/doc/search?name=bgp&sort=&rfcs=on&by=group&group=

[22] J. Li, M. Wang, Z. Zheng, and M. Zhang, "Loogle: Can long-context language models understand long contexts?" *arXiv preprint arXiv:2311.04939*, 2023.

[23] T. Li, G. Zhang, Q. D. Do, X. Yue, and W. Chen, "Long-context llms struggle with long in-context learning," *arXiv preprint arXiv:2404.02060*, 2024.

[24] Z. Ji, T. Yu, Y. Xu, N. Lee, E. Ishii, and P. Fung, "Towards mitigating llm hallucination via self reflection," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 1827–1843.

[25] Internet-Society. (1986) Internet Engineering Task Force. [Online]. Available: https://www.ietf.org/about/introduction/

[26] ICANN. (2024) The central repository for protocol name and number registries used in many internet protocols. https://www.iana.org.

[27] H. Flanagan and S. Ginoza. (2014) RFC Style Guide. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7322#section-4

[28] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The programmer's assistant: Conversational interaction with a large language model for software development," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*, 2023, pp. 491–514.

[29] T. Gao, A. Fisch, and D. Chen, "Making pre-trained language models better few-shot learners," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2021, pp. 3816–3830.

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

[31] G. Bossert, F. Guihéry, and G. Hiet, "Towards automated protocol reverse engineering using semantic information," *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014.

[32] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu, "Netplier: Probabilistic network protocol reverse engineering from message traces," *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.

[33] K. Liang, Z. Luo, Y. Zhao, W. Zhang, R. Shi, Y. Jiang, H. Shi, and C. Hu, "Mdiplier: Protocol format recovery via hierarchical inference," *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 547–557, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:274468138

[34] Y. Zhao, Z. Luo, K. Liang, F. Wu, W. Zhang, H. Shi, and Y. Jiang, "Protocol syntax recovery via knowledge transfer," *Comput. Networks*, vol. 258, p. 111022, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:275340607

[35] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," in *The Eleventh International Conference on Learning Representations*, 2023.

[36] R. Natella and V.-T. Pham, "Profuzzbench: a benchmark for stateful protocol fuzzing," *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:231592362

[37] F. Wu, Z. Luo, Y. Zhao, Q. Du, J. Yu, R. Peng, H. Shi, and Y. Jiang, "Logos: Log guided fuzzing for protocol implementations," *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:272617576

[38] aflnet, "AFLNet: A greybox fuzzer for network protocols," https://github.com/aflnet/aflnet.

[39] ChatAFL. (2024) Chatafl with gpt-4 and large window support. https://github.com/ChatAFLndss/ChatAFL/tree/large-window.

[40] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. W. Hicks, "Evaluating fuzz testing," *ACM SIGSAC CCS*, 2018.

[41] Google, "OSS-Fuzz," 2024, https://github.com/google/oss-fuzz.

[42] P. Tech., "Peach fuzzer configuration file (Peach Pit)." Website, https://peachtech.gitlab.io/peach-fuzzer-community/v3/PeachPit.html.

[43] jtpereyda. (2024) BooFuzz Quickstart. https://www.iana.org.

[44] S. Qin, F. Hu, Z. Ma, B. Zhao, T. Yin, and C. Zhang, "Nsfuzz: Towards efficient and state-aware network service fuzzing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–26, 2023.

[45] S. Schumilo, C. Aschermann, A. Jemmett, A. R. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," *Seventeenth European Conference on Computer Systems*, 2022.

[46] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

[47] X. Zhu, W. Zhou, Q.-L. Han, W. Ma, S. Wen, and Y. Xiang, "When software security meets large language models: A survey," *IEEE/CAA Journal of Automatica Sinica*, vol. 12, no. 2, pp. 317–334, 2025.

[48] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

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

[50] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[51] Google. (2024) A Framework for Fuzz Target Generation and Evaluation. https://github.com/google/oss-fuzz-gen.

[52] Y. Lyu, Y. Xie, P. Chen, and H. Chen, "Prompt fuzzing for fuzz driver generation," *CCS*, 2024.

[53] C. Zhang, M. Bai, Y. Zheng, Y. Li, W. Ma, X. Xie, Y. Li, L. Sun, and Y. Liu, "Understanding large language model based fuzz driver generation," *arXiv e-prints*, pp. arXiv–2307, 2023.

[54] C. Ling, X. Zhao, J. Lu, C. Deng, C. Zheng, J. Wang, T. Chowdhury, Y. Li, H. Cui, X. Zhang *et al.*, "Domain specialization as the key to make large language models disruptive: A comprehensive survey," *arXiv preprint arXiv:2305.18703*, 2023.

[55] A. T. Mallen, A. Asai, V. Zhong, R. Das, D. Khashabi, and H. Hajishirzi, "When not to trust language models: Investigating effectiveness of parametric and non-parametric memories," in *The 61st Annual Meeting Of The Association For Computational Linguistics*, 2023.

[56] S. Borgeaud, A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, G. B. Van Den Driessche, J.-B. Lespiau, B. Damoc, A. Clark *et al.*, "Improving language models by retrieving from trillions of tokens," in *International conference on machine learning*. PMLR, 2022, pp. 2206–2240.

[57] M. Komeili, "Internet-augmented dialogue generation," *arXiv preprint arXiv:2107.07566*, 2021.

[58] Q. Liu, D. Yogatama, and P. Blunsom, "Relational memory-augmented language models," *Transactions of the Association for Computational Linguistics*, vol. 10, pp. 555–572, 2022.

[59] J. Wang, L. Yu, and X. Luo, "Llmif: Augmented large language model for fuzzing iot devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 881–896.