ProtocolGuard: Detecting Protocol Non-compliance Bugs via LLM-guided Static Analysis and Dynamic Verification

Xiangpu Song[1], Longjia Pei[1], Jianliang Wu[2*], Yingpei Zeng[3], Gaoshuo He[1], Chaoshun Zuo[4], Xiaofeng Liu[1*], Qingchuan Zhao[5] and Shanqing Guo[167*]
[1]School of Cyber Science and Technology, Shandong University
[2]Simon Fraser University [3]Hangzhou Dianzi University [4]Independent Researcher
[5]City University of Hong Kong [6]Shandong Key Laboratory of Artificial Intelligence Security
[7]State Key Laboratory of Cryptography and Digital Economy Security, Shandong University

***Abstract*—Network protocol implementations are expected to strictly comply with their specifications to ensure reliable and secure communications. However, the inherent ambiguity of natural-language specifications often leads to developers' misinterpretations, causing protocol implementations to deviate from standard behaviors. These deviations result in subtle non-compliance bugs that can cause interoperability issues and critical security vulnerabilities. Unlike memory corruption bugs, these bugs typically do not exhibit explicit error behaviors, resulting in existing bug oracles being insufficient to thoroughly detect them. Moreover, existing works require heavy manual effort to verify findings and analyze root causes, severely limiting their scalability in practice.**

In this paper, we present ProtocolGuard, a novel framework that systematically detects non-compliance bugs by combining LLM-guided static analysis with fuzzing-based dynamic verification. ProtocolGuard first extracts normative rules from protocol specifications using a hybrid method, and performs LLM-guided program slicing to extract code slices relevant to each rule. It then leverages LLMs to detect semantic inconsistencies between these rules and code logic, and dynamically verify whether these bugs can be triggered. To facilitate bug verification, ProtocolGuard first uses LLMs to automatically generate assertion statements and instrument the code to turn silent inconsistencies into observable assertion failures. Then, it produces initial test cases that are more likely to trigger the bug with the help of LLMs for dynamic verification. Lastly, ProtocolGuard dynamically tests the instrumented code to confirm bug identification and generate proof-of-concept test cases. We implemented a prototype of ProtocolGuard and evaluated it on 11 widely-used protocol implementations. ProtocolGuard successfully discovered 158 non-compliance bugs with high accuracy, 70 of which have been confirmed, and the majority of which can be converted into assertions and dynamically verified. The comparison with existing state-of-the-art tools demonstrates that ProtocolGuard outperforms them in both precision and recall rates in bug detection capabilities.

*Corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240521
www.ndss-symposium.org

I. Introduction

Network protocols are the backbone of digital communication, enabling billions of devices to exchange data reliably. Given their widespread deployment, even minor bugs in protocol implementations can cause serious security and interoperability issues. These implementations are expected to strictly conform to specifications (e.g., RFCs), which define precise behavioral rules such as state transitions and message formats. However, natural-language specifications are often lengthy and complex, which can lead to misunderstandings and cause implementations to silently deviate from standard behaviors, thereby introducing non-compliance bugs.

Protocol non-compliance bugs are widespread in real-world implementations, causing incorrect behaviors, interoperability failures, and severe security vulnerabilities [31, 32, 67]. For example, a high-risk vulnerability in MatrixSSL (CVE-2022-46505 [8]) stemmed from flawed validation logic in its session resumption handling, resulting from an incorrect implementation of RFC [20]. This seemingly minor implementation flaw allowed an attacker to use a malformed Session ID to force the reuse of an empty master secret, leading to complete decryption of secure communications on thousands of devices worldwide [23]. However, unlike memory corruption bugs, non-compliance bugs typically manifest as silent logic errors without obvious signals, such as crashes, making them difficult to detect using conventional methods like fuzzing [34, 49].

Recent research has proposed various approaches for detecting protocol non-compliance bugs, including static analysis and differential testing, which have achieved significant success. Unfortunately, existing bug oracles are insufficient for the comprehensive detection of these bugs. Traditional static approaches [31, 32] rely on heuristics that match rule-like conditions to specific code patterns. However, their effectiveness is severely hindered by the diversity of implementation styles and the ambiguity inherent in natural language specifications. Differential testing [54, 61, 65, 67] relies on mutual reference oracles across multiple implementations to detect inconsistencies, but this approach becomes ineffective when all compared implementations exhibit the same incorrect behavior [64]. Moreover, existing approaches require substantial manual effort to verify findings and analyze root causes, severely limiting their scalability in practice.

To address these challenges, we propose ProtocolGuard, a novel approach that integrates static analysis with dynamic validation to detect non-compliance bugs. To accurately identify these bugs, we first extract normative requirements from protocol specifications and transform them into rule sets that serve as detection standards. Leveraging the semantic understanding capabilities of Large Language Models (LLMs), we integrate LLMs with program slicing to extract code slices relevant to each rule, and analyze the inconsistencies between the rules and code implementations, thereby identifying potential non-compliance bugs and their root causes. To efficiently validate these potential bugs, we first leverage LLMs to generate assertion statements for each bug and instrument them into the program, transforming silent logical errors into observable assertion failures. Then, we design a test case generation method that produces high-quality initial test cases and employ directed fuzzing to dynamically verify potential bugs and generate proof-of-concept (PoC) test cases.

We implemented a prototype of ProtocolGuard and conducted a comprehensive evaluation across 11 widely-used implementations of 6 network protocols. ProtocolGuard discovered 158 non-compliance bugs across these implementations, including 156 previously unknown bugs, and achieved an overall precision of 90.6%. We compared ProtocolGuard against state-of-the-art AI-powered code editors, including Cursor [7] integrated with Claude 3.7 Sonnet and DeepSeek R1. ProtocolGuard significantly outperformed these baselines, achieving 86.3% precision and 81.3% recall, compared to the best baseline performance of 71.7% precision and 76.8% recall. Furthermore, our evaluation demonstrates the effectiveness of ProtocolGuard's key components: the assertion generation successfully transforms a majority of silent logic errors into assertion failures, while the test case generation significantly improves fuzzer performance in confirming inconsistency assertions.

Overall, we make the following contributions in this paper:

- We propose a novel hybrid approach that combines LLM-guided static analysis with fuzzing-based dynamic verification to systematically detect non-compliance bugs.
- We design an automated assertion generation mechanism that transforms silent logic errors into observable assertion failures, enabling conventional fuzzers to detect non-compliance bugs.
- We implement a prototype ProtocolGuard, and evaluate it across 11 real-world protocol implementations, successfully revealing 158 non-compliance bugs.

II. Motivation and Challenges

In this section, we first present a real-world example to illustrate the motivation behind our work, then discuss the limitations of existing research as well as the challenges we address.

```c
1  struct client {
2    ...
3    char client_id[MQTT_CLIENT_ID_LEN];
4  }
5  static int connect_handler(struct io_event *e) {
6    struct mqtt_connect *c = &e->data.connect;
7    struct client     *cc = e->client;
8
9    // BUG: Client ID truncation without validation
10   snprintf(cc->client_id, MQTT_CLIENT_ID_LEN, "%s
        ", c->payload.client_id);
11   ...
12 }
```

Listing 1: Simplified code of motivation example.

A. Motivating Example

Listing 1 presents a simplified code snippet from Sol [14], an MQTTv3.1.1 protocol implementation, that contains a critical non-compliance bug. This vulnerability enables attackers to impersonate legitimate clients by exploiting the silent truncation of oversized client identifiers, potentially leading to denial-of-service attacks against clients with identical identifiers [1]. Specifically, when the broker receives a CONNECT packet containing a client identifier longer than `MQTT_CLIENT_ID_LEN`, the `connect_handler` function processes the connection request normally. At line 10, the code copies the client ID from the network payload `c->payload.client_id` into a fixed-size buffer `cc->client_id` using `snprintf`. However, due to the absence of prior length validation, the server silently truncates characters beyond the limit without generating any error or warning. The root cause of this bug stems from an incomplete implementation of the specification rule. While the implementation superficially adheres to the explicit rule that states: *'The Server MUST allow ClientIds which are between 1 and 23 UTF-8 encoded bytes in length, and MAY allow ClientIds that contain more than 23 encoded bytes'* [17], it fails to preserve the complete client identifier when processing oversized IDs. This silent truncation can result in multiple clients sharing the same truncated identifier, violating the protocol's implicit requirement that each client must have a distinct identifier for proper session management and message routing.

Existing protocol bug detection approaches face significant challenges in identifying such bugs. Traditional fuzzers that rely on memory sanitizers fail to detect such bugs, as these bugs typically manifest as silent logic errors without causing program crashes [34, 49]. Similarly, differential fuzzing approaches [54, 61, 67] prove ineffective for this issue, as both vulnerable and correct MQTT protocol implementations return identical CONNACK success responses, rendering cross-reference oracles unable to distinguish the underlying behavioral differences. Furthermore, conventional static analysis approaches [31, 32] typically rely on predefined heuristic rules derived from specifications. However, these approaches cannot assess whether the implementation logic is semantically correct and compliant with the specification, particularly when specifications lack detailed guidance on field processing requirements such as client identifier handling. These limitations highlight the need for a novel detection approach capable of accurately identifying protocol non-compliance bugs.

B. Challenges and Our Solutions

Recent advancements in LLMs have demonstrated remarkable capabilities in understanding both source code and natural language [33], presenting an intuitive solution to analyzing whether protocol implementations comply with their specifications from a semantic perspective. However, directly applying LLMs to the entire source code is impractical, as LLMs' reasoning ability is inversely proportional to input context length [41]. This raises several key challenges.

C1: How to provide LLMs with appropriate rule-relevant code implementations? A natural solution is to apply program slicing to extract only rule-relevant code statements for LLM analysis, as large code context would reduce LLM's performance [41]. However, traditional slicing approaches require manual specification of slicing criteria (i.e., the variables and program points of interest from which slicing begins), limiting their application [56, 63]. Moreover, even with correct criteria, conventional slicing typically contains semantically unrelated code fragments, leading to noisy inputs for LLMs.

Solution: To address this challenge, we propose an LLM-guided program slicing approach that combines semantic inference from LLMs with LLVM-based static analysis. We first use LLMs to automatically identify rule-relevant variables as slicing criteria. Based on this, we perform rule-oriented forward slicing to extract relevant code, followed by a hybrid pruning strategy that removes semantically unrelated code.

C2: How to effectively verify non-compliance bugs without explicit error signals? Once bugs are identified through static analysis, existing methods typically require manual construction of corresponding input messages for validation [32, 65], which is prohibitively time-consuming and does not scale to large codebases. A promising approach is hybrid testing, which combines static analysis to detect vulnerabilities followed by directed fuzzing for validation [53]. However, non-compliance bugs often do not cause program crashes [54, 67], rendering current memory sanitizer-based directed fuzzing ineffective [34, 49].

Solution: To overcome this challenge, we propose using LLM agents to automatically generate assertion statements that serve as bug oracles for detecting non-compliance bugs. These assertions are instrumented to convert silent inconsistencies into assertion failures. When programs fail to handle malformed requests as expected, these assertions actively abort program execution, enabling existing fuzzing strategies to detect such bugs and generate PoC test cases.

C3: How to provide high-quality initial test cases for fuzzing-based dynamic verification? The effectiveness of fuzzing campaigns heavily depends on the quality of initial test cases [44]. Reusing existing test cases from the community is ineffective because they typically focus on general functionality testing rather than targeting the specific malformed inputs necessary to expose non-compliance bugs. While LLMs show great potential in message generation [47], directly prompting them to generate binary protocol messages yields poor results, as this requires them to perform precise calculations and adhere to strict binary format specifications [40].

Solution: To address this challenge, we first utilize LLMs to generate natural-language descriptions of counterexamples, specifying input message sequences and field values that would violate each identified rule. Then, based on these descriptions, we employ LLM agents to synthesize Python scripts that programmatically construct the required inputs as initial test cases, rather than directly generating raw message bytes.

III. Design

In this section, we present the design of ProtocolGuard, a hybrid framework that systematically detects protocol non-compliance bugs. Figure 1 illustrates the overall framework of ProtocolGuard, which comprises four components: protocol rule extraction, LLM-guided program slicing, LLM-based inconsistency detection, and fuzzing-based dynamic verification.

A. Protocol Rule Extraction

To determine whether a protocol implementation adheres to its specification, we first extract normative rules from specification documents (e.g., RFCs). This step is necessary because original documents typically contain substantial content unrelated to implementation constraints, such as background information and optional recommendations, which can reduce the accuracy of LLM analysis [52]. Previous research [31, 32] indicates that RFCs employ a prescriptive tone to describe protocol behavior, typically achieved by combining modal keywords from RFC 2119 (e.g., MUST, SHOULD), comparative keywords (e.g., greater or less than), and protocol-specific keywords (e.g., message and field names) to strictly constrain how implementations process messages. Therefore, we define sentences as potential protocol rules if they contain at least one protocol-specific keyword and one modal or comparative keyword. Although LLMs demonstrate strong semantic understanding capabilities [57], we found that directly using them to extract rules usually produces inaccurate results [42]. To address this issue, we design a hybrid rule extraction method that leverages keyword matching to identify candidate rules with high precision and employs LLMs to refine these rules. This method consists of three main steps.

Candidate Rule Identification. We first strip irrelevant elements (e.g., directories and HTML tags) from the specification documents, segment the text into sentences, and then group them according to section hierarchy. To identify potential rules, we automatically construct three categories of keyword lists for each specification using LLMs. We did not directly apply existing work [32] because they rely on manual construction. For protocol-specific keywords, we employ a background-augmented prompting [57], incorporating the corresponding Wireshark dissector code [13] as context. These dissectors are designed to parse protocol messages and contain detailed information about message structure, enabling LLMs to extract

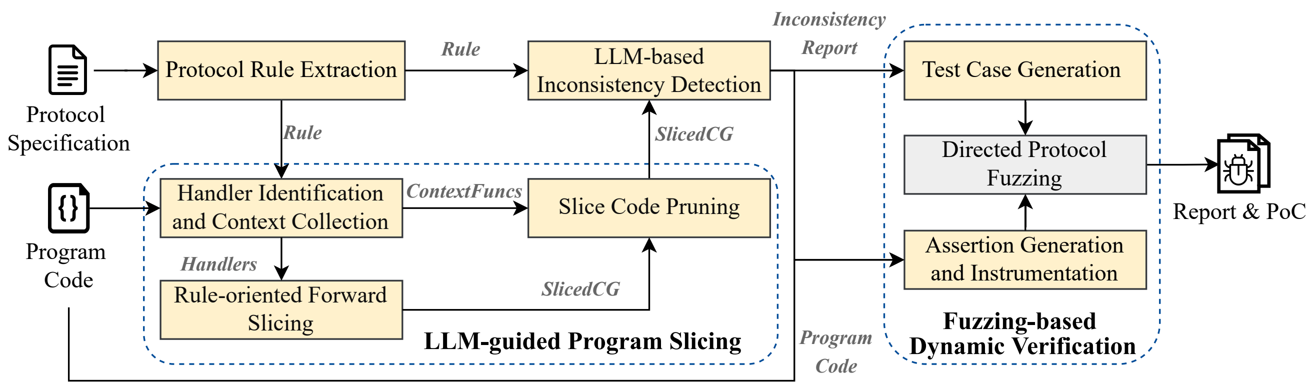

Protocol Specification
Program Code
Protocol Rule Extraction
Rule
LLM-based Inconsistency Detection
Inconsistency Report
Rule
SlicedCG
Handler Identification and Context Collection
ContextFuncs
Slice Code Pruning
Handlers
Rule-oriented Forward Slicing
SlicedCG
LLM-guided Program Slicing
Program Code
Test Case Generation
Directed Protocol Fuzzing
Assertion Generation and Instrumentation
Fuzzing-based Dynamic Verification
Report & PoC

Fig. 1: Overview of ProtocolGuard.

all message and field names comprehensively. For modal keywords, we directly use keywords defined in RFC 2119, excluding optional ones (e.g., MAY, OPTIONAL) as they lack strong obligatory force [51]. For numerical comparison keywords, we reuse existing keyword lists in [32]. To better align with the specific terminology and phrasing conventions used in different specifications, we then expand these initial keywords by leveraging LLMs to identify synonyms and alternative expressions that occur in the document. Finally, we perform heuristic keyword matching on each sentence, retaining only sentences containing at least one protocol-specific keyword and one modal or numerical comparison keyword as candidate protocol rules.

Rule Contextualization. The rule sentences extracted through heuristic matching often suffer from incomplete semantics and unresolved referential dependencies, as individual sentences typically express only partial aspects of the complete rule. For example, RFC 8446 [21] states that *'If this extension is present in the ClientHello, servers MUST NOT use...'*, where *this extension* implicitly refers to a specific field (i.e., *Supported Version*) mentioned in the surrounding context. Such incomplete contexts can lead to misinterpretations during inconsistency analysis. To address this, we employ LLMs to analyze the surrounding section of each candidate rule, merging logically connected clauses (e.g., those linked by causality or coreference) to construct semantically complete and self-contained rules.

Structured Rule Representation. To facilitate efficient processing by downstream components, we employ LLMs to convert rule descriptions into a structured JSON format. Each structured object contains the original textual rule, the constrained request message type, its related internal fields, and the corresponding response message type and fields when applicable. Listing 2 presents an example corresponding to the motivating example discussed in Section II-A. In this structure, `req_type` specifies the request message type constrained by the rule, and `req_fields` lists the relevant fields mentioned in the rule, including both the target field and its parent fields in the message hierarchy (`ClientId` nested within `Payload`). `res_type` indicates the expected response type that the protocol implementation should return, while `res_fields` includes any referenced fields.

```
{
  "rule": "The Server MUST allow ClientIds which
    are between 1 and 23 UTF-8 encoded bytes in
    length, and MAY allow ClientIds that contain
    more than 23 encoded bytes",
  "req_type": "CONNECT",
  "req_fields": ["Payload", "ClientId"],
  "res_type": "CONNACK",
  "res_fields": []
}
```

Listing 2: Example of structured rule representation.

B. LLM-guided Program Slicing

With the structured rules extracted, we employ a hybrid program slicing method that combines LLM-based source code analysis with LLVM-based program analysis to extract code slice relevant to each rule. Our key insight is that protocol implementations typically follow an event-driven architecture, dispatching various message handler functions based on incoming message types [28, 50]. This pattern implies that the key processing logic related to a specific message is usually encapsulated within the call subgraph of its handler function. To extract precise code slices, we use LLMs to analyze source code and automatically locate the handler functions and field variables described in the rule, which serve as slicing targets. We then perform forward slicing based on LLVM, leveraging its powerful infrastructure for precise static analysis [63]. Since LLVM-based slicing tends to include code that is data-dependent but semantically irrelevant, we further integrate LLMs and heuristic strategies to remove unrelated logic. Algorithm 1 outlines the overall workflow of this approach, which consists of three phases.

Phase 1: Handler Identification and Context Collection. To identify the message handler functions that serve as starting points for program slicing, we construct a specialized call graph focused on message processing, termed *MessageCG*

Algorithm 1 Workflow of LLM-guided Program Slicing

```
Input: M (LLVM Module), Rule, LLM
Output: CodeSlice
    // Phase 1: Handler Identification and Context Collection
 1: MessageCG ← ExtractMessageProcessingSubgraph(M)
 2: Handlers ← IdentifyMessageHandlers(MessageCG, Rule, LLM)
 3: ContextFuncs ← IdentifyContextFunctions(MessageCG, Handlers, Rule, LLM)
    // Phase 2: Rule-oriented Forward Slicing
 4: SlicingCriteria ← IdentifySliceCriterion(Handlers, MessageCG, Rule, LLM)
 5: if SlicingCriteria $\neq \emptyset$ then
 6:     InitialSlice ← ForwardSlice(MessageCG, SlicingCriteria)
 7:     SlicedCG ← CompleteSliceCode(Handlers, InitialSlice, LLM)
 8: else
 9:     SlicedCG ← MessageCG
10: end if
    // Phase 3: Slice Code Pruning
11: PrunedCG ← PruneIrrelevantCode(Handlers, SlicedCG, Rule, LLM)
12: CodeSlice ← GenerateCodeSlice(M, PrunedCG, ContextFuncs)
```

(line 1 of Algorithm 1). This approach is necessary because complete program call graphs include numerous functions irrelevant to message processing, and performing identification on the entire graph would significantly expand the search space and increase the likelihood of false positives for LLMs. We begin by locating system calls used to receive network data, such as `recv` and `recvmsg`, as these serve as the network interfaces for receiving messages [50]. From these points, we perform an inter-procedural backward data flow analysis on the message buffer parameters of these system calls, tracking their propagation across functions until they reach the functions where the buffer parameter is first defined or initialized. These originating functions are designated as the root nodes for *MessageCG*. Finally, starting from the root nodes, we conduct a forward call graph traversal, including any function that subsequently receives the message buffer or its derived variables as an argument. The resulting *MessageCG* thereby captures the complete function tree involved in message processing, providing a relevant subgraph for identifying the handler functions.

We next perform a breadth-first search on the *MessageCG* starting from the entry function, using an LLM to identify the handler functions (line 2 of Algorithm 1). Specifically, at each step of the traversal, we prompt the LLM to analyze whether the current function contains handling logic for the input message type specified in the rule. When the LLM returns a positive response, we consider the function as the message handler for the corresponding message type and save this mapping relationship to avoid redundant identification in future iterations. To ensure completeness, the traversal does not terminate upon finding a handler function, but continues exploring all remaining functions at the current depth level of the breadth-first search. This is because the processing logic for a single message type may be distributed across multiple sibling functions in the call graph, such as in libcoap [12].

After identifying handler functions, we further perform contextual analysis by tracing their reverse call paths and using LLMs to analyze whether the upstream functions contain connection and resource management logic (line 3 of Algorithm 1). This contextual analysis is essential because message processing logic encompasses not only handler functions but also broader contextual operations that typically occur upstream in the call chain before handler invocation. The omission of these functions frequently leads to incomplete semantic understanding and incorrect LLM analysis results. Therefore, we collect such upstream contextual functions *ContextFuncs* and incorporate them into the final code slice after forward slicing (line 12), ensuring that the extracted code slice captures the complete message processing workflow.

Phase 2: Rule-oriented Forward Slicing. Given the handler functions identified for a specific rule, we perform slicing in three steps: (1) identifying the rule-relevant variables in the source code and map them to their LLVM instructions; (2) conducting forward data-flow analysis to extract all instructions that are transitively dependent on these target variables; and (3) refining the resulting code slice to ensure both syntactic correctness and interpretable.

We begin by prompting LLMs to analyze the source code of handler functions and identify variables representing the fields mentioned in the rule (line 4 of Algorithm 1). We choose to analyze source code rather than LLVM intermediate representation (IR) in this step, as its higher-level abstraction and human-readable structure allow LLMs to better understand program semantics [38]. Once the relevant variables are identified, we utilize debug information preserved during compilation to accurately map them to their LLVM instructions, particularly their definitions and first uses. In addition, we prompt LLMs to identify auxiliary variables involved in message processing logic. These variables are not explicitly mentioned in the rules and may have no data dependencies with field variables, but they are essential for preserving the contextual completeness of message processing logic. For example, variables used for message integrity check in Mosquitto [10] (e.g., `pos` for tracking the parsing position and `remaining_length` for the bytes left to process) are critical for the slice, as their absence would prevent LLMs from correctly interpreting the program logic and result in false positives. If LLMs fail to identify field variables, including the rules that do not involve any fields, we default to a conservative approach: the subgraph of *MessageCG* originating from the handler function is treated as the slice code *SlicedCG* (line 9 of Algorithm 1).

With the target instructions identified, we generate the slice using a worklist-based forward data-flow analysis (line 6 of Algorithm 1). We begin by initializing a worklist with the identified instruction set. At each iteration, we dequeue an instruction from the worklist, analyze its def-use chains [15], and add any newly discovered dependent instructions back into the worklist. This process continues until the worklist is empty. To support inter-procedural slicing, we track parameter passing by adding the corresponding formal parameters in callees to the worklist when variables are used as function arguments. We also handle both direct assignments, such as the `store` instruction, and indirect assignments introduced by function calls with pointer or reference arguments to ensure continuous tracking of variable assignments. In these cases, we add the target variables that depend on already tracked instructions to the worklist. Once the traversal completes, the collection of all visited instructions constitutes the instruction-level code slice *InitialSlice*. For downstream LLM analysis, we maintain a mapping from each instruction in the slice to its corresponding source code line number through debug information, enabling reconstruction of a human-readable code slice.

However, the initial slice *InitialSlice* derived from instruction-level data dependencies is insufficient for effective LLM analysis due to two major limitations. Firstly, it is often syntactically incorrect because the control flow dependencies of the code are ignored in the previous data-flow analysis process. For example, instructions within branch statement bodies (e.g., `if` or `switch`) are included if they have data dependencies with the slicing targets. However, the conditional expressions of these statements are often excluded from the slice when they lack direct data flow dependencies with those instructions. As a result, the slice includes body statements without the condition that governs them, breaking the syntactic and control flow structure of the original code. Similarly, essential control-flow changing statements are frequently excluded, such as `return`, `break`, and `goto` statements. Secondly, the slice may lack sufficient semantic context. Crucial statements for program understanding, such as logging calls or error-code assignments, are usually pruned for not being data-dependent on the target instructions. However, these statements contain vital clues about the program's logic and intent [39, 59], and their presence can enhance LLM analysis.

To address these limitations, we perform slice completion combining abstract syntax tree (AST)-based structure reconstruction with semantic enrichment (line 7 of Algorithm 1). Firstly, we use ASTs to recover missing control flow structures. We analyze each function in the source code, mapping its LLVM instructions to their corresponding control structures, including line numbers and scope boundaries for branch statements. For each instruction in the slice, we check whether it resides within the body of a control flow statement. If so, we include the associated condition and any required control-flow changing statements to preserve the completeness of control flow dependencies. Secondly, to enrich the semantic context, we prompt LLMs to analyze the original function body and identify non-data-dependent but semantically necessary informative statements that were pruned during slicing. These include logging calls and error-code assignments, which offer valuable clues about control logic and program intent. We recover such statements to improve the interpretability of the final code slice.

Phase 3: Slice Code Pruning. Although the generated slice *SlicedCG* is significantly smaller than the original *MessageCG*, we apply a pruning strategy that combines heuristic filtering with semantic analysis to further reduce the length of the slice (line 11 of Algorithm 1), which is beneficial for improving the performance of LLM analysis [41].

Firstly, we apply coarse-grained heuristics to prune irrelevant response sending functions. For rules that specify expected response types, we remove the internal implementations of functions that send irrelevant responses, along with their callees. Notably, we retain the call sites to these response sending functions to preserve the visibility of all response behaviors for analysis. Only their internal implementations are removed to reduce slice length. This pruning is effective because most protocols send only a few types of responses during single message processing, making the detailed implementations of functions sending other response types irrelevant to the current rule analysis. To support this pruning, we trace backward from network send system calls (e.g., `send`, `sendto`) and use LLMs to map each sending function to its corresponding response type. This mapping enables us to discard functions that generate responses not required by the rule.

Secondly, we perform fine-grained semantic pruning to address limitations of LLVM-based slicing. Traditional slicing approaches tend to indiscriminately retain all code that accesses the same field variables, regardless of whether the logic is actually relevant to the rule. For instance, in Mosquitto, the variable `topic_filter` is used both for topic parsing and for permission checking. For rules targeting only topic parsing, functions involved solely in access control are irrelevant and should be removed. We address this issue by leveraging LLMs to evaluate the semantic relevance of each function to the rule context and exclude rule-irrelevant functions from the slice. To preserve the structured integrity of the slice, we only prune a function if it does not serve as a critical intermediary in the call paths.

Finally, we combine the pruned code slice *PrunedCG* with the collected contextual functions *ContextFuncs* that reside along the upstream call paths before the handler function to construct the final code slice *CodeSlice* (line 12 of Algorithm 1).

C. LLM-based Inconsistency Detection

After extracting the code slice for each rule, we employ LLMs to analyze inconsistencies between the rules and their code implementations. Unlike heuristic-based approaches [31, 32], LLMs can analyze the underlying semantics of code and identify deeper semantic inconsistencies between the implemented code and the prescribed rule.

We design a structured prompt that instructs the LLM to act as a protocol compliance auditor, providing both the rule description and the corresponding code slice as input. The prompt guides the LLM through a systematic analysis process: firstly, understanding the specific requirements of the rule; secondly, examining the control flow and data processing logic of the code slice; and finally, determining whether the implementation satisfies the rule's requirements or exhibits inconsistencies. To ensure structured and machine-parsable output, we require the LLM to respond in JSON format with results indicating whether an inconsistency was detected, and the specific location (function name and line numbers) of any identified issues. Due to space constraints, the complete prompt template is provided in the Appendix A. Listing 3 presents an example of the LLM-based inconsistency detection results, corresponding to the motivating example described in Section II-A. To enhance the reliability of the analysis, we query LLMs three times for each rule-slice pair and apply a self-consistency strategy [58], where the LLM analyzes its previous outputs to converge on a conclusion.

```
{
 "result": "violation found!",
 "reason": "..., the critical violation occurs in
     `connect_handler` where externally provided
    client IDs are truncated.",
 "violations": [
  {"function_name": "connect_handler", "filename
    ": "/path/handlers.c", "code_lines": [394]}
 ]
}
```

Listing 3: Example of inconsistency analysis result for the motivating example.

D. *Fuzzing-based Dynamic Verification*

We employ fuzzing to automatically confirm the inconsistencies identified by LLM-based static analysis and to generate corresponding PoCs. To achieve effective dynamical verification, we design two complementary approaches: *assertion generation and instrumentation*, and *test case generation*.

Assertion Generation and Instrumentation. We automatically generate assertion statements from inconsistency reports to serve as bug oracles for fuzzing-based dynamic verification. Our key insight is that non-compliance bugs typically stem from implementations failing to properly validate external inputs, leading to unexpected program behaviors that deviate from specification requirements [30, 32, 65]. Based on this observation, we can proactively synthesize the expected validation logic by analyzing both the protocol rules and their corresponding implementation, and instrument the synthesized logic as assertions at points of inconsistency. When fuzzing inputs trigger unexpected behavior, these assertions detect the deviation from expected protocol semantics and intentionally abort program execution, which enables detection of such bugs by mainstream fuzzers [34, 49].

We employ an LLM agent to perform this complex generation and instrumentation task through an iterative process. We design a structured prompt based on multi-step instructions to guide the agent, which is primarily divided into four steps: (1) analyzing the inconsistency report and the protocol rule to extract the expected validation constraints and conditions, (2) examining the code implementation to identify where the validation is missing or incorrectly implemented, (3) generating assertion statements with appropriate help functions and instrumenting them at the identified locations, and (4) invoking compilation tools to verify the correctness of the generated code. If compilation errors are detected, the agent collects the error messages and performs iterative refinement until syntactically correct code is produced. We employ LLM agents for this task because they can autonomously utilize

```
1  static int connect_handler(struct io_event *e) {
2    // ...
3    // Generated assertion to validate client ID
4    assert(client_id_length_validation(c));
5    snprintf(cc->client_id, MQTT_CLIENT_ID_LEN, "%s
         ", c->payload.client_id);
6  }
7  // Generated helper function
8  static int client_id_length_validation(struct
      mqtt_connect *c) {
9    size_t original_len = strlen((char *)c->payload
         .client_id);
10   return (original_len < MQTT_CLIENT_ID_LEN);
11 }
```

Listing 4: Instrumented assertion for motivating example.

various programming tools, such as code search tools to gather surrounding context and compilation tools to ensure code correctness, thereby enhancing both the accuracy of assertion logic and the reliability of code generation [60].

Figure 4 shows an example of assertion generation corresponding to the motivating example shown in Section II-A. The LLM agent generates a validation helper function `client_id_length_validation` that checks whether the incoming client ID exceeds the buffer capacity. The assertion statement is then instrumented at line 4 before the problematic `snprintf` operation. When fuzzers supply a client ID longer than `MQTT_CLIENT_ID_LEN`, the assertion triggers a program crash, transforming the silent bug into assertion failures with explicit crash signals.

Test Case Generation. To improve the effectiveness of fuzzing-based dynamic verification, we use LLMs to generate initial test cases that are both syntactically valid and intentionally violate specific protocol rules. These test cases are designed to drive execution toward the code paths associated with inconsistencies, thereby increasing the likelihood of triggering assertion failures.

We design two structured prompts to guide LLMs in synthesizing executable scripts rather than raw message bytes. This approach ensures that the generated inputs conform to protocol syntax, without requiring LLMs to reason about low-level packet encodings or boundary constraints. Instead, LLMs only need to understand high-level APIs, which significantly improves the accuracy of test case generation [40]. Specifically, we first provide an LLM with rule descriptions and inconsistency analysis reports, prompting it to generate natural-language descriptions of structured message sequences that are likely to trigger the identified inconsistencies as counterexamples. These counterexamples specify the message types, their ordering, and the key field values that intentionally violate the rule constraints. Subsequently, we use the counterexamples as input to our second prompt, which guides an LLM agent to synthesize executable Python scripts capable of generating the corresponding packets in PCAP format. Additionally, the second prompt instructs the agent to automatically execute the generated script and invoke traffic analysis tools to verify that the packet trace matches the expected messages. If the verification fails, the agent collects the feedback and iteratively refines the script until it produces a correct test case.

Directed Protocol Fuzzing. With the generated assertions and initial test cases, we perform directed protocol fuzzing to confirm the identified inconsistencies and generate PoCs. We collect the code locations of all instrumented assertions as directed targets for fuzzers. To enhance efficiency, we employ the path pruning strategy [37, 46], focusing the fuzzer's exploration on code paths related to inconsistencies while avoiding unnecessary exploration of unrelated program regions.

IV. Implementation

We implemented a prototype of ProtocolGuard comprising approximately 9k lines of C++ and 2.9k lines of Python code. The implementation consists of two primary components: a static analysis module for identifying protocol inconsistencies and a dynamic verification module for confirming them through directed fuzzing.

Static Analysis Component. The static analysis module encompasses protocol rule extraction, LLM-guided program slicing, and LLM-based inconsistency detection. We implemented the rule extraction component in Python using the lxml library [16] for document parsing. The program slicing component is built on LLVM passes using C++. We used GLLVM [27] to generate whole-program LLVM IR and Clang's AST library [3] to preserve control flow information for slice completion. Additionally, we used SVF [55] to address the challenge of indirect function calls in static analysis. We employed DeepSeek series models for their state-of-the-art performance and cost efficiency [18]. To optimize for different task requirements, we employed the powerful DeepSeek R1 [36] for inconsistency detection, while the more efficient DeepSeek V3 [45] was used for program slicing, where low latency is critical for an iterative workflow. Due to space limitations, all complete prompt templates used in ProtocolGuard are included in our repository.

Dynamic Verification Component. The dynamic verification module integrates assertion generation and instrumentation, test case generation, and directed protocol fuzzing. For assertion and test case generation, we used Cursor [7] as an LLM agent platform, automated through cursorkleos [5] to minimize manual intervention and enable batch processing. We selected Claude 3.7 Sonnet provided by Cursor for its superior performance in the code generation task compared to DeepSeek series models. For test case generation, we employed Scapy [22], a widely-used Python library for packet manipulation and generation, to synthesize executable test scripts. The generated packets are saved in PCAP format and validated using tshark [26] to verify message sequence correctness. We built our fuzzing framework upon AFLNet [49], the state-of-the-art gray-box fuzzer for network protocols. We also integrated SelectFuzz's [46] selective instrumentation strategy to optimize directed fuzzing performance.

V. Evaluation

To evaluate the effectiveness of ProtocolGuard, we conducted comprehensive experiments on real-world protocol implementations and aim to answer the following questions:

- **RQ1.** Can ProtocolGuard effectively detect non-compliance bugs in real-world protocol implementations? (Section V-A)
- **RQ2.** How does ProtocolGuard compare to existing state-of-the-art tools in detecting protocol inconsistencies? (Section V-B)
- **RQ3.** Can the generated assertions effectively serve as oracles for fuzzing to verify non-compliance bugs? (Section V-C)
- **RQ4.** Can the generated test cases improve the efficiency of dynamic verification? (Section V-D)

Dataset. We selected 11 open-source protocol implementations written in the C language, covering six widely adopted network protocols, as shown in Table I. Our selection criteria were: (1) Protocol significance: We chose protocols critical to modern network infrastructure, covering IoT communication (MQTT, CoAP), secure transport (TLS 1.3), file transfer (FTP), and network services (DHCPv6); (2) Implementation diversity: We selected implementations with varying scales (4.4K-1456.3K LoC), architectural approaches, and target deployment environments; (3) Community activity: All projects and their developers show active development with recent community engagement. This diverse benchmark enables comprehensive evaluation of ProtocolGuard's effectiveness across different protocol complexities and implementation styles.

Subject	Version	LoC	Protocol	Specification
Sol	373d8	4.4K	MQTT 3.1.1	OASIS MQTT 3.1.1
TinyMQTT	6226ad	11.5K	MQTT 3.1.1	OASIS MQTT 3.1.1
Mosquitto	849e0f	46.2K	MQTT 5.0	OASIS MQTT 5.0
libcoap	17c3fe	45.3K	CoAP	RFC 7252
FreeCoAP	3adc2e	26.6K	CoAP	RFC 7252
pure-ftpd	381857	22.2K	FTP	RFC 959 et al.*
uFTP	646404	6.7K	FTP	RFC 959 et al.
TLSE	1af154	41.8K	TLS 1.3	RFC 8446
wolfSSL	7fb750	1456.3K	TLS 1.3	RFC 8446
Dnsmasq	2.91	33.4K	DHCPv6	RFC 8415
NDHS	4b2728	5.6K	DHCPv6	RFC 8415

TABLE I: Protocol implementations used for evaluation.

Environment. We conducted all experiments in Docker images on a local machine with one Intel(R) Xeon(R) Gold 6226R CPU and 256 GB RAM, and a Ubuntu 22.04 LTS system. We used default parameters for the DeepSeek model for our analysis.

A. *Discovered Real-world Non-compliance Bugs*

Table II shows the bug discovery effectiveness of ProtocolGuard in 11 open-source protocol implementations. Overall, ProtocolGuard extracted 420 rules from the official specifications and systematically analyzed these implementations, detecting 181 inconsistencies with an overall precision rate of 90.6%. After verification, we confirmed 158 unique non-compliance bugs, of which 156 were previously undiscovered

*RFC 959, 2228, 2389, 2428, 3659

new bugs, and 2 were known but not yet fixed bugs. At the time of writing, we reported all 158 bugs to the relevant vendors, with 70 confirmed and 17 fixed.

The discrepancy between detected inconsistencies (181) and bugs (158) stems from three factors. Firstly, some implementations intentionally deviate from specifications to provide extended functionality (e.g., Mosquitto plugins), which we classify as functional features rather than bugs. Secondly, multiple inconsistencies often point to the same underlying issue in the code. Thus, we merge these related findings into one single bug. Lastly, certain behaviors violate the RFC specifications used in our evaluation but have been permitted in updated standard drafts, so we do not include them in the final bug statistics after discussing with the vendors.

Subject	**Rules**	**Inconsistencies**			**Non-compliance**
		TP	**FP**	**Precision**	**Bugs**
Sol	83	39	2	95.1%	39
TinyMQTT	83	29	3	90.6%	27
Mosquitto	118	15	2	88.2%	4
libcoap	30	4	1	80.0%	2
FreeCoAP	30	2	0	100.0%	2
pure-ftpd	54	17	2	89.5%	13
uFTP	54	18	1	94.7%	15
TLSE	58	25	2	92.6%	25
wolfSSL	58	7	1	87.5%	7
Dnsmasq	77	12	2	85.7%	11
NDHS	77	13	1	92.9%	13
Total	**420**	**181**	**17**	**90.6%**	**158**

TABLE II: Bug discovery results of ProtocolGuard. The *Rules* indicates the total number of valid rules identified by ProtocolGuard, *Inconsistencies* indicates the total number of inconsistencies found between rules and codes, *Non-compliance Bugs* indicates the number of bugs after verification.

To better understand the diversity of these bugs, we classified the 158 discovered non-compliance bugs by their root causes, as shown in Table V in Appendix B. The most common bug category is related to message parsing (labeled as *Parsing*), which accounts for approximately 37% of the total bugs. These bugs primarily result from the inappropriate validation of input messages. Protocol state violations (*State*) are the next most common and account for 22%, typically associated with improper maintenance of the state machine. The remaining bugs are distributed across error handling (*Error*, 16%), session management (*Session*, 13%), and security mechanisms (*Security*, 12%). This distribution reveals an important insight: while network message parsers have been shown to be prone to errors [65, 66], there are more bugs occurring in the deeper protocol processing logic, demonstrating the necessity of comprehensive testing of protocol implementation compliance beyond parser-level validation.

These non-compliance bugs not only compromise program robustness but also pose severe security risks, enabling attackers to bypass access controls, launch denial-of-service attacks, or compromise communication integrity and confidentiality. Together, these findings highlight the critical importance of protocol compliance testing for software security [54]. To further illustrate the security impact of these discoveries, we discuss three representative bugs as case studies.

Case Study 1: TLS 1.3 Downgrade via Version Negotiation Flaw in wolfSSL. This bug, identified as ID 61 in Table V, demonstrates a critical flaw in wolfSSL's TLS version negotiation mechanism. Listing 5 presents the vulnerable code segment. RFC 8446 [21] mandates that *'If a Supported Versions extension is present in the ClientHello, servers MUST NOT use the ClientHello.legacy_version value for version negotiation and MUST use only the supported_versions extension to determine client preferences'*. This rule ensures a reliable mechanism for negotiating TLS 1.3, preventing inadvertent or malicious downgrades. However, wolfSSL's implementation violates this requirement by prematurely triggering a downgrade decision based solely on the `legacy_version` field (lines 4-7) before processing the `supported_versions` extension (line 11), ignoring the priority specified in the RFC. Specifically, in scenarios where both `legacy_version` and `supported_versions` are set to 0x0304 (TLS 1.3), wolfSSL erroneously forces a downgrade to TLS 1.2 (line 7) instead of proceeding with the TLS 1.3 handshake. This violation eliminates TLS 1.3's forward secrecy guarantees [24], enabling attackers who later compromise the server's private key to retroactively decrypt all previously captured communications between the affected client and server.

```
1  int DoTls13ClientHello(...) {
2    if (!ssl->options.dtls) {
3      // RFC violation: checking legacy_version
       first
4      if (args->pv.major > SSLv3_MAJOR || (args->pv
       .major == SSLv3_MAJOR &&
5          args->pv.minor >= TLSv1_3_MINOR)) {
6        // Downgrade to TLS 1.2
7        wantDowngrade = 1;
8      }
9    }
10   if (!wantDowngrade)
11     ret = DoTls13SupportedVersions(ssl, ...);
12   if (wantDowngrade)
13     ret = DoClientHello(ssl, ...);
14 }
```

Listing 5: Simplified code of case study 1.

Case Study 2: Missing Initial CONNECT Packet Check in Sol. This bug, identified as ID 93 in Table V, reveals a critical protocol state machine flaw in Sol. The MQTTv3.1.1 specification [17] explicitly requires that *'After a Network Connection is established by a Client to a Server, the first Packet sent from the Client to the Server MUST be a CONNECT Packet'*. This rule is fundamental to the MQTT protocol's security model, as the CONNECT packet contains key information for client authentication and session state management. However, Sol's packet processing logic lacks state verification that would reject non-CONNECT packets in the initial state. This allows an attacker to bypass the server's authorization policy by directly sending a SUBSCRIBE packet with a specific topic filter without completing the required initial authentication

Project	Cursor (Claude 3.7)			Cursor (DeepSeek R1)			ProtocolGuard (DeepSeek R1)		
	TP/FP/FN	Precision	Recall	TP/FP/FN	Precision	Recall	TP/FP/FN	Precision	Recall
Sol	37/3/7	92.5%	84.1%	16/5/28	76.2%	36.4%	**39/2/5**	**95.1%**	**88.6%**
pure-ftpd	16/9/4	64.0%	80.0%	10/14/10	41.7%	50.0%	**17/2/3**	**89.5%**	**85.0%**
libcoap	**5**/3/**2**	62.5%	**71.4%**	4/7/3	36.4%	57.1%	4/**1**/3	**80.0%**	57.1%
TLSE	21/5/7	80.8%	75.0%	18/11/10	62.1%	64.3%	**25/2/3**	**92.6%**	**89.3%**
Average	20/5/5	71.7%	76.8%	12/37/13	49.3%	52.0%	**21/2/4**	**86.3%**	**81.3%**

TABLE III: Comparison of inconsistency discovery results by ProtocolGuard and Cursor.

step, thereby obtaining sensitive messages [54].

Case Study 3: Missing Re-authorization after AUTH in uFTP. This bug, labeled as ID 55 in Table V, shows an authentication state management flaw in uFTP's handling of secure channel negotiation. RFC 2228 [19] requires that *'The AUTH command, if accepted, removes any state associated with prior FTP Security commands. The server must also require that the user reauthorize...'*. This rule defines a critical security control strategy, as the AUTH command is responsible for negotiating and initiating a security mechanism (e.g., TLS) to encrypt the control channel, preventing attackers from leveraging credentials captured over insecure channels to hijack subsequently established secure sessions. However, uFTP lacks logic to clear existing authentication state during AUTH command processing, leading to a session hijacking vulnerability exploitable in Man-in-the-Middle (MitM) attacks. When a MitM attacker observes plaintext authentication (USER/PASS commands), they can subsequently inject an AUTH command into the same TCP session. Due to uFTP's failure to clear authentication state during AUTH processing, the attacker inherits the authenticated session context within the newly established TLS tunnel. Therefore, the attacker can execute arbitrary high-risk commands within the encrypted channel without re-authentication, effectively bypassing network-based intrusion detection systems that rely on plaintext traffic analysis

B. Comparison with Existing Tools

To evaluate the effectiveness of LLM-guided program slicing and the inconsistency detection capability of ProtocolGuard, we conducted a comparative analysis against existing state-of-the-art methods.

We initially considered several related works, but found them unsuitable for a direct comparison. The source code for RIBDetector [32] is unavailable, while tools like Pardiff [65] and ParCleanse [66] are focused on inconsistency bugs within message parsers and are difficult to apply to entire protocol implementations. Similarly, differential fuzzing lacks a general implementation supported for diverse protocols and requires significant manual analysis. Therefore, we selected Cursor [7], a state-of-the-art AI code editor, as the most relevant baseline. We chose Cursor for its advanced agentic capabilities, such as automated context collection, which are more powerful than those in other tools like Github Copilot [2, 11]. For the evaluation, we employed two leading LLMs within Cursor: DeepSeek R1, to maintain consistency with ProtocolGuard, and Claude 3.7 Sonnet, both of which are the top coding models [25]. To ensure fairness, we manually provided Cursor with the same rules and prompts used by ProtocolGuard.

We manually analyzed the results produced by ProtocolGuard and Cursor to determine true positives (TP), false positives (FP), and false negatives (FN), from which we calculated precision and recall rates. To ensure the reliability of the evaluation, we invited two independent researchers with experience in protocol vulnerability analysis to manually analyze each result. We then cross-validated the results from the two researchers, and any discrepancies were resolved by a third researcher to eliminate potential bias. Due to the time-intensive nature of manual analysis, we conducted this comprehensive evaluation on four programs: Sol, pure-ftpd, libcoap, and TLSE.

Table III shows the detailed results of ProtocolGuard and Cursor on these four programs. Overall, ProtocolGuard outperforms both Cursor configurations. On average, ProtocolGuard correctly identifies 21 inconsistencies (TP) with only 2 FP results and 4 FN results, achieving a mean precision of 86.3% and a recall of 81.3%. ProtocolGuard outperforms Cursor with Claude 3.7 (71.7% precision, 76.8% recall) and is significantly better than Cursor with DeepSeek R1 (49.3% precision, 52.0% recall). Notably, the substantial performance gap between ProtocolGuard and Cursor when both use the same DeepSeek R1 model demonstrates the effectiveness of ProtocolGuard, which achieves more accurate inconsistency detection than general-purpose AI code editors.

We conducted an investigation into the root causes of FPs and FNs listed in Table II and Table III produced by ProtocolGuard. The root causes can be categorized into four primary types. Firstly, the program slicing strategy of ProtocolGuard may miss critical processing logic in protocol implementations that use callback-based design patterns. For example, Sol decouples message processing from response transmission and connection handling, resulting in these functions not residing on the same call path as the message handler functions. Consequently, ProtocolGuard fails to capture the concrete implementations of cleanup functions, such as connection termination. For rules that depend on the correct handling of such cleanup logic, this omission may lead LLMs to perceive missing behavior when analyzing inconsistencies, resulting in FPs and FNs. Secondly, certain protocol implementations (e.g., libcoap) support multiple transport layer protocols (e.g., UDP, TCP, WebSocket), which involve numerous distinct processing functions in the message handler functions. This makes ProtocolGuard simultaneously include parsing code from different transport layer protocols within the same slice, resulting in an excessively large context and may even exceed the maximum context limitation of the LLM (DeepSeek R1's 128K token limit), degrading the LLM's reasoning capabilities [41]. Therefore, compared to other subjects, ProtocolGuard exhibits lower precision and recall on libcoap. Thirdly, when rule-relevant variables are encapsulated within data structures and only accessed in deeply called functions, ProtocolGuard cannot trace across intermediate functions in the call chain when performing data dependency analysis, resulting in code slices that lack the concrete implementations of key processing functions. Lastly, some errors stem from the inherent reasoning limitations of LLMs. Even when provided with accurate code slices, LLMs sometimes fail to fully comprehend complex logical relationships and intricate program semantics.

We further analyzed the performance of the baseline tool, Cursor. When using DeepSeek R1, Cursor's performance is significantly inferior to that of ProtocolGuard using the same model. We observed that this disparity primarily stems from DeepSeek R1's unstable function-calling capabilities [9], which prevents Cursor from using tools to retrieve relevant code, resulting in severe hallucinations and errors. In contrast, Cursor using Claude 3.7 demonstrated better performance, which we attribute to this model's superior tool invocation capabilities, combined with high-quality prompts identical to those used in ProtocolGuard, significantly enhancing the completeness of its context collection [6]. Nevertheless, Cursor relies on keyword extraction from user queries and vector similarity search from codebases to grep relevant code snippets, which cannot systematically capture cross-function data dependencies and control flow relationships, resulting in the underperformance of Cursor in certain scenarios.

C. *Effectiveness of Assertion Generation and Instrumentation*

To evaluate whether the generated assertion statements are correct and can effectively guide fuzzing to verify non-compliance bugs, we conducted a comprehensive evaluation of the assertion generation and instrumentation component.

We evaluated the effectiveness from three perspectives. Firstly, we used each project's native compilation toolchain to verify the syntactic correctness of the code instrumented with assertions. Secondly, for all assertions that passed compilation, we conducted a manual review to verify their semantic accuracy, confirming whether each assertion precisely reflected the constraints of the corresponding protocol rules and would abort programs upon receiving non-compliant input. Finally, for semantically correct assertions, we performed directed protocol fuzzing with a 24-hour time budget to determine whether these assertions could be triggered by the fuzzer, leading to program crashes, thereby validating their utility as effective bug oracles.

Table IV presents the detailed evaluation results. All assertions generated by ProtocolGuard successfully passed compilation verification, with an average implementation accuracy of 88.9% and an average crash-triggering rate of 68.4% during fuzzing-based dynamic verification. Overall, the assertion generation module can accurately generate appropriate assertion statements and can, in most scenarios, generate PoCs that violate protocol rules, effectively assisting analysts in confirming inconsistencies detected by static analysis.

Protocol	**Total**	**Syntactic**	**Semantic (Rate)**	**Crash (Rate)**
Sol	41	41	39 (95.1%)	32 (78.0%)
TinyMQTT	32	32	27 (84.4%)	24 (75.0%)
Mosquitto	17	17	15 (88.2%)	12 (70.6%)
libcoap	5	5	4 (80.0%)	2 (40.0%)
FreeCoAP	2	2	2 (100.0%)	2 (100.0%)
pure-ftpd	19	19	19 (100.0%)	17 (89.5%)
uFTP	19	19	18 (94.7%)	15 (78.9%)
TLSE	27	27	22 (81.5%)	15 (55.6%)
wolfSSL	8	8	6 (75.0%)	4 (50.0%)
Dnsmasq	14	14	13 (92.9%)	9 (64.3%)
NDHS	14	14	12 (85.7%)	7 (50.0%)
Average	18	18	16 (88.9%)	13 (68.4%)

TABLE IV: Results of assertion generation and instrumentation analysis. The *Total* indicates the total number of assertions generated, *Syntactic* indicates the number of assertions that were successfully compiled, *Semantic (Rate)* indicates the number and accuracy rate of semantically correct assertions generated, and *Crash (Rate)* indicates the number and rate of unique crashes triggered by the assertion failure.

We conducted a root cause analysis of failed assertion generation and observed three primary failure reasons. Firstly, due to the lack of a dynamic program execution context, LLMs often make incorrect assumptions about variable states and control flows. For example, we observed that some generated assertions use `NULL` checks to detect empty fields, while the actual protocol implementation treats empty strings as indicators of missing values. As a result, such assertions are never triggered during execution. Secondly, when protocol implementations lack explicit validation logic, LLMs must independently understand the implementation context and protocol rules to synthesize missing validation code. However, when multiple data structures contain members with similar names, the LLM agent (i.e., Cursor) sometimes incorrectly selects variables and types for logic generation, as it relies on a keyword-based tool (e.g., *Grep*) and vector similarity search to infer context [4]. This deviation in context understanding can easily lead to hallucinations during code generation, resulting in the synthesis of semantically invalid or even completely incorrect validation logic. Moreover, in cases where the missing logic is inherently complex, such as logic that spans multiple functions or involves implicit control conditions, the LLM's capabilities may be insufficient to accurately understand the detailed necessary context, resulting in hallucinations or the generation of semantically invalid validation logic.

We also investigated the root causes of assertions that fail to trigger during dynamic verification. After excluding incorrect results of assertions, we found that the primary reason was that the fuzzing inputs failed to reach the execution paths of the assertions due to the following reasons. Firstly, many assertions are located in code modules that require specific configurations to be activated. If the configuration is not enabled, these code paths are inaccessible to fuzzers. Secondly, the fuzzer (i.e., AFLNet) lacks awareness of protocol formats, and its mutation strategy is prone to generating invalid inputs that disrupt message structures, causing most test cases to be rejected during early validation. This has a particularly severe impact on highly complex and structured protocols, making it difficult to trigger assertion statements located deep within the protocol state path in DHCP and TLS protocol implementations. Lastly, it is challenging for AFLNet to trigger assertions in multi-party interaction logic because its two-party fuzzing model limits exploration of assertions in MQTT protocol implementations [54].

D. Effectiveness of Test Case Generation

To validate whether the test case generation component improves the efficiency of fuzzing-based dynamic verification, we conducted a comparative experiment. Our hypothesis is that high-quality, rule-specific initial seeds should significantly outperform random seeds in guiding the fuzzer toward assertion-instrumented code paths. In this experiment, we compared the crash discovery performance of AFLNet using two sets of initial seeds. The experimental group used seeds from ProtocolGuard, which produces syntactically valid and rule-violating message sequences. The control group, as a baseline, used randomly generated messages for each protocol message type. To ensure a fair comparison, all other fuzzing configurations remained identical for both groups.

Figure 2 summarizes the results of the comparative experiment evaluating the impact of initial test case generation on dynamic verification effectiveness. Across all tested programs, AFLNet equipped with seeds generated by ProtocolGuard consistently outperformed its baseline counterpart using random seeds, discovering on average 155.2% more assertion-triggered unique crashes. Notably, this performance gain was particularly significant for complex protocols such as DHCP and TLS, where ProtocolGuard achieved improvements ranging from 275% to 600%, demonstrating the clear advantage of semantically guided test case generation in exposing non-compliance bugs.

To understand the underlying reasons for this improvement, we further investigated why ProtocolGuard's test case generation approach is more effective than random seeds. We found that ProtocolGuard can generate high-quality, syntactically valid, and highly targeted initial seeds by leveraging protocol specifications and known violation patterns. These high-quality seeds guide the fuzzer to directly explore deep and potentially flawed logic paths. In contrast, a fuzzer starting with random seeds must rely on its inefficient mutation strategies to generate inputs that can trigger bugs, a task that is exceedingly difficult for complex protocols. In summary, the test case generation module of ProtocolGuard significantly enhances the effectiveness of fuzzing, improving its capability to validate potential non-compliance bugs in protocol implementations.

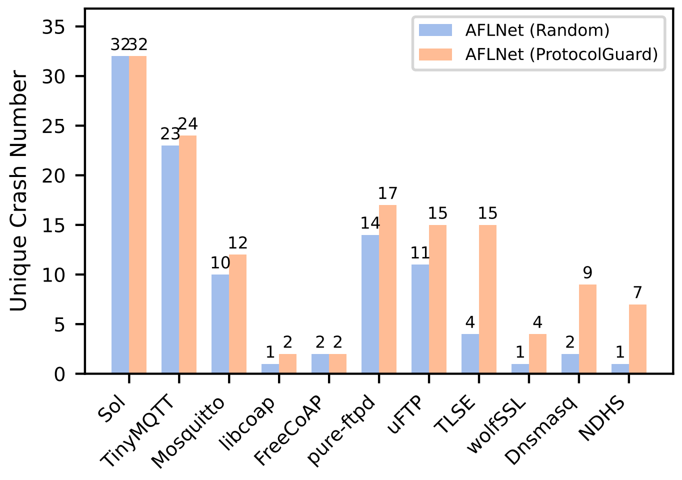


Fig. 2: Comparison of crash discovery results for different initial test case selections.

VI. Discussion and Limitations

ProtocolGuard has successfully identified many non-compliance bugs across various protocol implementations. However, it still has certain limitations. In this section, we discuss these limitations and potential solutions for future improvements.

Inaccuracy of LLM-guided Program Slicing. Due to the diversity of program design patterns and implementation styles in protocol software, our slicing approach faces several limitations that can affect its precision and completeness. Firstly, our slicing strategy relies on identifying handler-centric call paths. This design is less effective for protocol implementations that adopt decoupled, callback-based architectures, where critical logic such as connection cleanup or response transmission is implemented outside the direct call path of the message handler function. As a result, essential code may be excluded from the slice, leading to an incomplete context for LLM analysis and potential false negatives. Secondly, ProtocolGuard leverages SVF to resolve indirect function calls during slicing. However, SVF tends to be conservative, resolving each indirect call to a broad set of possible targets. This often introduces irrelevant functions into the slice process, unnecessarily enlarging the context passed to the LLM and potentially degrading analysis quality. Thirdly, the current implementation of ProtocolGuard is limited to C language-based protocol implementations. Extending support to the C++ language requires heavy engineering challenges, primarily due to the complexity introduced by object-oriented features, which complicate LLVM IR-based static analysis. In future work, we plan to enhance our framework with advanced analysis techniques capable of accurately resolving virtual function calls and recovering class hierarchies, thereby enabling robust support for C++-based protocol implementations.

Inaccuracy in Assertion Generation. ProtocolGuard employs LLM-generated assertion statements as bug oracles to validate potential non-compliance bugs through directed fuzzing. While this approach effectively uncovers subtle logic violations, it faces two primary limitations. Firstly, the accuracy of assertion generation relies heavily on sufficient contextual information and a precise understanding of the code. However, the current LLM agent (e.g., Cursor) operates solely on static code and lacks access to the program's dynamic execution state. Thus, it may misinterpret control flow or variable state, leading to the generation of invalid or ineffective assertions. In future work, we plan to incorporate dynamic execution information to help LLMs better understand program behavior and generate more precise assertions [35].

Ineffectiveness of Fuzzing-based Dynamic Verification. The effectiveness of dynamic validation depends on the performance of fuzzing strategies. Although ProtocolGuard adopts fuzzers like AFLNet and SelectFuzz, these strategies often generate syntactically invalid inputs that are rejected during early message parsing. These inputs may fail to exercise the intended logic paths, ultimately preventing the triggering of assertion failures. To address this limitation, we plan to integrate protocol-aware mutation strategies or symbolic execution techniques to help fuzzers explore complex conditional paths that are otherwise difficult to reach.

Manual Effort. While ProtocolGuard is designed to operate with a high degree of automation, certain stages of its workflow still require manual intervention. Firstly, during assertion and test case generation, a single LLM session may not always complete the entire task in one pass. This can cause the process to terminate prematurely, requiring the user to manually prompt the agent to resume and complete the remaining steps. Secondly, the generated assertions can be mutually exclusive in practice. When one assertion is frequently triggered, it may dominate the fuzzing process and prevent the exploration of other assertion-instrumented paths. To ensure broader coverage, users must manually comment out the already-triggered assertions and restart fuzzing to enable discovery of the remaining ones. However, this limitation can be addressed by adopting a memory-based feedback, inspired by AFL++. Specifically, we can use a shared memory region to track the triggering status of all assertions. Once an assertion is triggered, its state is recorded in memory, and the assertion is automatically disabled in subsequent executions.

VII. Related Work

Static Analysis. Several static analysis techniques have been proposed to detect non-compliance bugs in protocol implementations. RIBDetector [32] extracts normative statements from RFCs as rules and uses heuristic patterns to determine whether the corresponding condition checking logic is present in the code. EBugDec [31] targets protocol bugs by analyzing inconsistencies introduced during the evolution of RFC documents. ParDiff [65] applies differential testing in static analysis to check for inconsistencies in network protocol parsers. PARCLEANSE [66] leverages LLMs to extract message formats from protocol specifications and uses them as bug oracles to validate parser correctness. However, existing work either relies on heuristic methods that suffer from low precision or is limited to specific functional modules within protocol implementations. Moreover, they require substantial manual effort to verify the numerous issues identified by static analysis.

Protocol Fuzzing. Fuzzing has been widely adopted for discovering bugs in protocol implementations. AFLNet [49] is the first gray-box protocol fuzzer that incorporates response codes as feedback. Recent work [47] has also explored incorporating LLMs into protocol fuzzing to improve test case generation. However, these approaches often rely on memory sanitizers to detect bugs, making them ineffective for identifying silent non-compliance bugs. Differential fuzzing is another prevalent strategy, which detects inconsistencies by cross-executing the same input across multiple implementations. This strategy has been applied to various protocols, including TCP [67], DNS [61], MQTT [54], and HTTP [48]. Despite its effectiveness, differential fuzzing has inherent limitations. If all implementations produce consistent outputs for a faulty behavior, the inconsistency may be undetectable. Additionally, not all inconsistencies imply actual bugs; thus, manual analysis is still required to determine root causes, imposing a heavy burden on developers. In contrast, ProtocolGuard unifies LLM reasoning capability, static analysis, and directed fuzzing to enhance detection accuracy for non-compliance bugs and minimize the need for manual validation.

Directed Fuzzing. Directed fuzzing aims to efficiently validate known or potential bugs by guiding input mutations toward specific program locations or paths. AFLGo [29] is the first directed fuzzer that uses distance metrics to guide fuzzing toward target code locations. Subsequent approaches, such as Beacon [37] and SelectFuzz [46], introduce path-pruning techniques to improve guidance precision by avoiding paths unrelated to the target. Dsfuzz [43] and SDFuzz [43] further enhance directed fuzzing by incorporating control and data dependencies. However, existing directed fuzzers are not designed for network protocols. To address this gap, ProtocolGuard integrates SelectFuzz into AFLNet, leveraging SelectFuzz's selected instrumentation strategy, which is readily compatible with AFLNet's protocol-aware execution model, to improve directed fuzzing performance in the protocol context.

VIII. Conclusion

In this paper, we proposed ProtocolGuard, a novel hybrid framework for detecting non-compliance bugs in protocol implementations. ProtocolGuard automatically extracts protocol rules from specifications and uses LLM-guided program slicing to generate rule-relevant code slices. It then analyzes semantic inconsistencies between rules and codes with the help of LLMs to identify potential non-compliance bugs. To validate these findings, ProtocolGuard generates assertion statements that convert the silent bugs into assertion failures and uses fuzzing to trigger them and generate concrete PoCs. In addition, ProtocolGuard incorporates a rule-specific test case generation approach to further enhance dynamic verification efficiency. We implemented a prototype of ProtocolGuard and evaluated it on 11 protocol implementations. The results demonstrate that ProtocolGuard effectively detected 158 non-compliance bugs with high precision, significantly reducing the manual effort required for bug validation.

Acknowledgments

We thank anonymous reviewers for their comprehensive feedback. This work was supported by Key R&D Program of Shandong Province, China(No. 2024CXGC010114), National Natural Science Foundation of China under Grant(No. 62372268), Shandong Provincial Natural Science Foundation, China (No. ZR2022LZH013, No. ZR2021LZH007). The work was also supported in part by the Zhejiang Provincial Natural Science Foundation of China (No. LY22F020022), in part by the National Natural Science Foundation of China (No. 61902098).

Ethics Considerations

We conducted this study by establishing ethical guidelines [62], and all findings were responsibly disclosed to the affected vendors. To prevent the discovered vulnerabilities from being exploited, all experiments were conducted in isolated local environments to ensure they did not affect any production systems.

References

[1] Aws: Conflicting mqtt client ids. https://docs.aws.amazon.com/iot-device-defender/latest/devguide/audit-chk-conflicting-client-ids.html, Accessed on 2025-7-19.

[2] Battle of the ai agents: Cursor vs. copilot. https://nearform.com/digital-community/battle-of-the-ai-agents/, Accessed on 2025-7-20.

[3] Clang 21.0.0git documentation: Introduction to the clang ast. https://clang.llvm.org/docs/IntroductionToTheClangAST.html, Accessed on 2025-7-14.

[4] Cursor docs: Agent tools. https://docs.cursor.com/en/agent/tools, Accessed on 2025-7-26.

[5] Cursor ide: Unleash lightning-fast automation. https://github.com/kleosr/cursorkleosr, Accessed on 2025-7-14.

[6] Cursor prompt engineering best practices. https://forum.cursor.com/t/cursor-prompt-engineering-best-practices/1592, Accessed on 2025-7-23.

[7] Cursor: The ai code editor. https://cursor.com/, Accessed on 2025-7-14.

[8] Cve-2022-46505 detail. https://nvd.nist.gov/vuln/detail/CVE-2022-46505, Accessed on 2025-7-20.

[9] Discussion abount deepseek r1' function calling. https://github.com/deepseek-ai/DeepSeek-R1/issues/9, Accessed on 2025-7-23.

[10] Eclipse mosquitto an open source mqtt broker. https://github.com/eclipse-mosquitto/mosquitto, Accessed on 2025-7-11.

[11] Github copilot · your ai pair programmer. https://github.com/features/copilot, Accessed on 2025-7-26.

[12] Github repository: A coap (rfc 7252) implementation in c. https://github.com/obgm/libcoap/blob/433f483f2c29b49a92e0a368d39beb6022eff88f/src/coap_net.c#L4383, Accessed on 2025-7-12.

[13] Github repository: Wireshak dissectors. https://github.com/wireshark/wireshark/tree/master/epan/dissectors, Accessed on 2025-7-10.

[14] Lightweight mqtt broker, written from scratch. io is handled by a super simple event loop based upon the most common io multiplexing implementations. https://github.com/codepr/sol, Accessed on 2025-7-19.

[15] Llvm programmer's manual. https://llvm.org/docs/ProgrammersManual.html#iterating-over-def-use-use-def-chains, Accessed on 2025-7-12.

[16] lxml - xml and html with python. https://lxml.de/, Accessed on 2025-7-26.

[17] Mqtt version 3.1.1. https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, Accessed on 2025-7-19.

[18] Nature: China's cheap, open ai model deepseek thrills scientists. https://www.nature.com/articles/d41586-025-00229-6, Accessed on 2025-7-14.

[19] Rfc 2228: Ftp security extensions. https://www.rfc-editor.org/rfc/rfc2228.txt, Accessed on 2025-7-22.

[20] Rfc 5246: The transport layer security (tls) protocol version 1.2. https://datatracker.ietf.org/doc/html/rfc5246, Accessed on 2025-7-12.

[21] Rfc 8446: The transport layer security (tls) protocol version 1.3. https://datatracker.ietf.org/doc/html/rfc8446#page-39, Accessed on 2025-7-12.

[22] Scapy: the python-based interactive packet manipulation program & library. https://github.com/secdev/scapy, Accessed on 2025-7-13.

[23] Shodan: Matrixssl. https://www.shodan.io/search?query=MatrixSSL, Accessed on 2025-7-20.

[24] Tls 1.3 in practice:how tls 1.3 contributes to the internet. https://dl.acm.org/doi/fullHtml/10.1145/3442381.3450057, Accessed on 2025-7-22.

[25] Top 5 ai coding models of march 2025: A comparative review. https://kitemetric.com/blogs/top-5-ai-coding-models-of-march-2025-a-comparative-review, Accessed on 2025-7-22.

[26] tshark.dev capture lifecycle with tshark. https://tshark.dev/, Accessed on 2025-7-26.

[27] Whole program llvm: wllvm ported to go. https://github.com/SRI-CSL/gllvm, Accessed on 2025-7-14.

[28] Csfuzzer: A grey-box fuzzer for network protocol using context-aware state feedback. *Computers & Security*, 157:104581, 2025.

[29] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

[30] Larissa Braz, Enrico Fregnan, Gül Çalikli, and Alberto Bacchelli. Why don't developers detect improper input validation? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 499–511. IEEE, 2021.

[31] Jingting Chen, Feng Li, Qingfang Chen, Ping Li, Lili Xu, and Wei Huo. Ebugdec: Detecting inconsistency bugs caused by rfc evolution in protocol implementations. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 412–425, 2023.

[32] Jingting Chen, Feng Li, Mingjie Xu, Jianhua Zhou, and Wei Huo. Ribdetector: an rfc-guided inconsistency bug detecting approach for protocol implementations. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 641–651. IEEE, 2022.

[33] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023.

[34] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[35] Sijia Gu, Noor Nashid, and Ali Mesbah. Llm test generation via iterative hybrid program analysis. *arXiv preprint arXiv:2503.13580*, 2025.

[36] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[37] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed greybox fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE, 2022.

[38] Hailong Jiang, Jianfeng Zhu, Yao Wan, Bo Fang, Hongyu Zhang, Ruoming Jin, and Qiang Guan. Can large language models understand intermediate representations? *arXiv preprint arXiv:2502.06854*, 2025.

[39] Peiling Jiang, Fuling Sun, and Haijun Xia. Log-it: Supporting programming with interactive, contextual, structured, and visual logs. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–16, 2023.

[40] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, et al. When fuzzing meets llms: Challenges and opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 492–496, 2024.

[41] Mosh Levy, Alon Jacoby, and Yoav Goldberg. Same task, more tokens: the impact of input length on the reasoning performance of large language models. *arXiv preprint arXiv:2402.14848*, 2024.

[42] Hui Li, Zhen Dong, Siao Wang, Hui Zhang, Liwei Shen, Xin Peng, and Dongdong She. Extracting formal specifications from documents using llms for automated testing. *arXiv preprint arXiv:2504.01294*, 2025.

[43] Penghui Li, Wei Meng, and Chao Zhang. Sdfuzz: Target states driven directed fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2441–2457, 2024.

[44] Yang Li, Yingpei Zeng, Xiangpu Song, and Shanqing Guo. Improving seed quality with historical fuzzing results. *Information and Software Technology*, 179:107651, 2025.

[45] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[46] Changhua Luo, Wei Meng, and Penghui Li. Selectfuzz: Efficient directed fuzzing with selective path exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2693–2707. IEEE, 2023.

[47] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, volume 2024, 2024.

[48] Keran Mu, Jianjun Chen, Jianwei Zhuge, Qi Li, Haixin Duan, and Nick Feamster. The silent danger in http: Identifying http desync vulnerabilities with gray-box testing. 2025.

[49] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[50] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Transactions on Software Engineering and Methodology*, 32(6):1–26, 2023.

[51] Jannis Rautenstrauch and Ben Stock. Who's breaking the rules? studying conformance to the http specifications and its security impact. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 843–855, 2024.

[52] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pages 31210–31227. PMLR, 2023.

[53] Peyton Shields. *Hybrid testing: Combining static analysis and directed fuzzing*. PhD thesis, Massachusetts Institute of Technology, 2023.

[54] Xiangpu Song, Jianliang Wu, Yingpei Zeng, Hao Pan, Chaoshun Zuo, Qingchuan Zhao, and Shanqing Guo. Mbfuzzer: A multi-party protocol fuzzer for mqtt brokers. In *Proceedings of the 34th USENIX Security*

Symposium, 2025.

[55] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[56] Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam, 1994.

[57] Jincheng Wang, Le Yu, and Xiapu Luo. Llmif: Augmented large language model for fuzzing iot devices. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 881–896. IEEE, 2024.

[58] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[59] Feifan Wu, Zhengxiong Luo, Yanyang Zhao, Qingpeng Du, Junze Yu, Ruikang Peng, Heyuan Shi, and Yu Jiang. Logos: Log guided fuzzing for protocol implementations. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1720–1732, 2024.

[60] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*, 2024.

[61] Qifan Zhang, Xuesong Bai, Xiang Li, Haixin Duan, Qi Li, and Zhou Li. Resolverfuzz: Automated discovery of dns resolver vulnerabilities with query-response fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4729–4746, 2024.

[62] Yiming Zhang, Mingxuan Liu, Mingming Zhang, Chaoyi Lu, and Haixin Duan. Ethics in security research: Visions, reality, and paths forward. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 538–545. IEEE, 2022.

[63] Ying-Zhou Zhang. Sympas: symbolic program slicing. *Journal of Computer Science and Technology*, 36(2):397–418, 2021.

[64] Zhen Zhao, Xiangpu Song, Qiuyu Zhong, Yingpei Zeng, Chengyu Hu, and Shanqing Guo. Tls-deepdiffer: message tuples-based deep differential fuzzing for tls protocol implementations. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 918–928. IEEE, 2024.

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

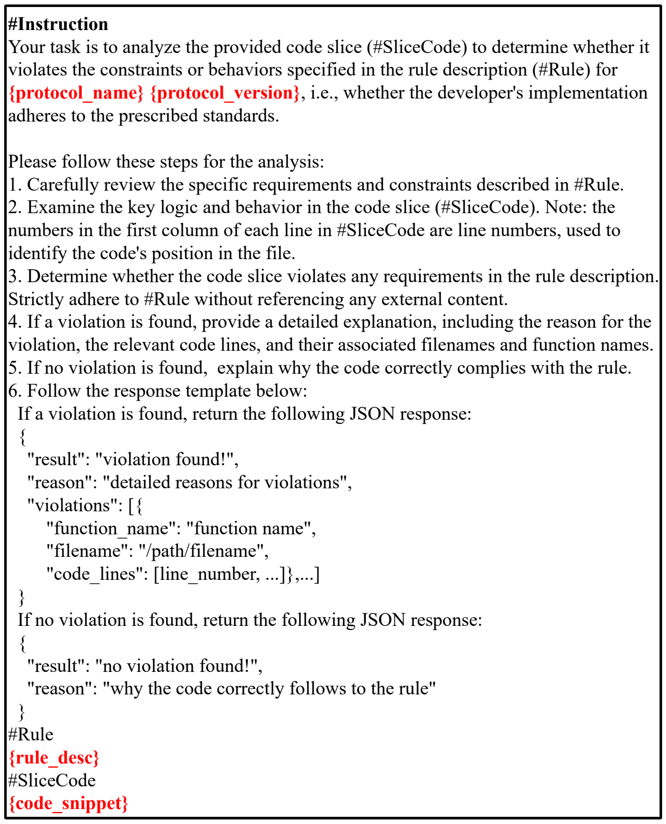
[66] Mingwei Zheng, Danning Xie, Qingkai Shi, Chengpeng Wang, and Xiangyu Zhang. Validating network protocol parsers with traceable rfc document interpretation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1772–1794, 2025.

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

Appendix

A. Prompt for LLM-based Inconsistency Detection

Figure 3 shows the prompt template used by ProtocolGuard for LLM-based inconsistency detection between protocol rules and code slices. The fields highlighted in red are automatically filled by ProtocolGuard: *protocol_name* and *protocol_version* specify the name and version of the protocol corresponding to the current rule under analysis, *rule_desc* contains the textual description of the protocol rule, and *code_snippet* contains the extracted code slice relevant to the rule.



```
#Instruction
Your task is to analyze the provided code slice (#SliceCode) to determine whether it
violates the constraints or behaviors specified in the rule description (#Rule) for
{protocol_name} {protocol_version}, i.e., whether the developer's implementation
adheres to the prescribed standards.

Please follow these steps for the analysis:
1. Carefully review the specific requirements and constraints described in #Rule.
2. Examine the key logic and behavior in the code slice (#SliceCode). Note: the
numbers in the first column of each line in #SliceCode are line numbers, used to
identify the code's position in the file.
3. Determine whether the code slice violates any requirements in the rule description.
Strictly adhere to #Rule without referencing any external content.
4. If a violation is found, provide a detailed explanation, including the reason for the
violation, the relevant code lines, and their associated filenames and function names.
5. If no violation is found,  explain why the code correctly complies with the rule.
6. Follow the response template below:
  If a violation is found, return the following JSON response:
  {
   "result": "violation found!",
   "reason": "detailed reasons for violations",
   "violations": [{
      "function_name": "function name",
      "filename": "/path/filename",
      "code_lines": [line_number, ...]},...]
  }
  If no violation is found, return the following JSON response:
  {
   "result": "no violation found!",
   "reason": "why the code correctly follows to the rule"
  }
#Rule
{rule_desc}
#SliceCode
{code_snippet}
```

Fig. 3: Prompt for LLM-based inconsistency detection in ProtocolGuard.

B. Details of Non-compliance Bugs Found by ProtocolGuard

Table V shows the detailed results of 158 non-compliance bugs discovered by ProtocolGuard, with 70 confirmed and 17 fixed. Most developers promise to fix the remaining unfixed bugs in the future. Due to space constraints, we omit some of the detailed descriptions of bugs. The complete table will be presented in our artifact.

C. Artifact Appendix

The ProtocolGuard artifact comprises three major components, each directly mirroring a core contribution of our paper: Protocol Rule Extraction, LLM-guided Program Slicing, and Fuzzing-based Dynamic Verification. The setup proceeds as follows.

1) Description & Requirements: In this section, we introduce how to obtain the artifact, including static analysis and fuzzers, along with the software and hardware requirements to run it.

a) How to access: We provide public access to our code and experiment setups through the following Zenodo link: ht tps://doi.org/10.5281/zenodo.17933922. You can also access it in Github: https://github.com/songxpu/ProtocolGuard. The artifact is licensed under the Apache License 2.0.

b) Hardware dependencies: ProtocolGuard can run on standard commercial servers or workstations without requiring specialized hardware. We recommend a minimum configuration of a 4-core CPU, 16 GB of RAM, and 128 GB of storage. It is important to note that resource-intensive tasks, such as those performed by code generation agents (e.g., Cursor) or large-scale fuzzing processes, can significantly increase system load.

c) Software dependencies: ProtocolGuard is installed and runs in a standalone Docker container based on Ubuntu 22.04, though it can also be deployed directly on a host operating system. To ensure proper operation, the environment must have LLVM 14, Python 3.10, and Go 1.18 installed.

d) Benchmarks: None.

2) Artifact Installation & Configuration: The ProtocolGuard artifact consists of four major components, each corresponding to a core contribution of the paper: Protocol Rule Extraction, LLM-guided Program Slicing, LLM-based Inconsistency Detection, and Fuzzing-based Dynamic Verification. The setup follows the following steps.

a) Step 1. Environment and Dependencies Setup: Install required system tools and third-party libraries on Ubuntu 22.04.3 LTS. Once installed, no additional system configuration is required.

b) Step 2. Static Analysis Initialization: This step involves preparing the target implementation for the subsequent LLM-guided slicing. We first compile the protocol's source code using LLVM to generate several intermediate artifacts, including the Control Flow Graph, SVF results, and AST metadata. Concurrently, the config/config.toml template must be properly filled in. The final output of this stage is the slicing-ready LLVM Intermediate Representation (IR) and an executable, which are used as the primary inputs for the next stage of LLM processing.

c) Step 3. LLM-based Inconsistency Detection: With the static analysis artifacts prepared, this step performs the core automated analysis. First, an LLM API key must be exported as an environment variable to authenticate requests. Then, the analysis is launched by executing a single script, which instructs the LLM to compare the extracted protocol rules against the program's behavior. Any violations found during this process are automatically parsed and recorded in an internal database. This produces the final output: a set of validated inconsistency results, which are then used as targets for dynamic verification.

d) Step 4. Fuzzing-based Dynamic Verification: This final step uses dynamic analysis to confirm the findings. For each violation detected by the LLM, the framework automatically generates corresponding test inputs and assertions. These are then fed into a directed protocol fuzzing campaign, which we conduct using AFLNet enhanced with our selective instrumentation.

ID	Project	Category	Bug Description	New	Status
1	Dnsmasq	Parsing	Missing Multicast Destination Check in dhcp6_packet Allows Unicast Message Processing	Yes	Confirmed
2	Dnsmasq	State	Missing Rapid Commit Check in dhcp6_no_relay Allows Unauthorized Rebind Bindings	Yes	Confirmed
3	Dnsmasq	State	Unconditional Lease Creation in dhcp6_no_relay for Rebind Messages	Yes	Confirmed
4	Dnsmasq	Parsing	Missing Zero Link-Address Check in dhcp6_maybe_relay	Yes	Confirmed
5	Dnsmasq	Parsing	Incorrect Hop-Count Check in relay_upstream6 Allows HOP_COUNT_LIMIT Messages	Yes	Confirmed
6	Dnsmasq	State	Missing Interface-Id Option in relay_upstream6 for Unusable Link-Address	Yes	Confirmed
7	Dnsmasq	Parsing	Improper Interface-Id Option Inclusion in dhcp6_no_relay Non-Relay Messages	Yes	Confirmed
8	Dnsmasq	Error	Incorrect NoAddrsAvail Status Placement in dhcp6_no_relay Reply Messages	Yes	Confirmed
9	Dnsmasq	State	Missing Requested Options in dhcp6_no_relay Advertise Messages	Yes	Confirmed
10	Dnsmasq	State	Missing IA_PD Handling in check_ia and dhcp6_no_relay for Solicit Messages	Yes	Confirmed
11	Dnsmasq	Parsing	Missing Required Options in dhcp6_no_relay for Solicit Message ORO	Yes	Confirmed
12	NDHS	Parsing	Missing Unicast Destination Check in process_receive Allows Invalid Message Processing	Yes	Reported
13	NDHS	Error	Improper Reply Sending in process_receive for Invalid CONFIRM Conditions	Yes	Confirmed
14	NDHS	Parsing	Missing ORO Content Validation in process_receive for Required DHCPv6 Options	Yes	Confirmed
15	NDHS	State	Missing Information Refresh Time Option in process_receive for Information-request Replies	Yes	Confirmed
16	NDHS	State	Unrequested Rapid Commit Option in process_receive Responses	Yes	Reported
17	NDHS	Parsing	Missing IA Content Validation in process_receive for Rebind Messages	Yes	Fixed
			...		
25	libcoap	Parsing	Missing ETag Validation in handle_request and coap_send_* for Multicast GET Requests	Yes	Confirmed
26	libcoap	State	Improper Option Retention in coap_add_data_large_response_lkd for Error Responses	Yes	Confirmed
27	freecoap	Parsing	Missing Context-Based Option Validation in coap_msg_check_critical_ops	Yes	Reported
28	freecoap	Error	Missing 4.05 Response in coap_server_exchange for Unrecognized Method Codes	Yes	Reported
29	PureFTPD	State	Missing EPSV ALL Check for PASV in parser Allows Unauthorized Connection Setup	Yes	Reported
			...		
39	PureFTPD	State	Missing PBSZ Check Before PROT in parser	Yes	Reported
40	PureFTPD	State	Missing REST Command Sequence Validation in parser	Yes	Reported
41	PureFTPD	State	Missing RNFR-RNTO Sequence Enforcement in parser and dornto	Yes	Reported
42	uFTP	Security	Missing Security Exchange Check for CCC in parseCommandCcc	Yes	Fixed
			...		
49	uFTP	Security	Missing Security Exchange and Argument Validation for PBSZ in parseCommandPbsz	Yes	Fixed
50	uFTP	Security	Missing PBSZ State Tracking for PROT in parseCommandPbsz	Yes	Fixed
51	uFTP	Security	Missing PBSZ Negotiation Check for PROT in parseCommandProt	Yes	Fixed
52	uFTP	State	Missing REST Sequence Enforcement in parseCommandRest and processCommand	Yes	Fixed
53	uFTP	Error	Missing Partial Transfer Handling in parseCommandRest and parseCommandRetr	Yes	Fixed
54	uFTP	State	Missing RNFR-RNTO Sequence Enforcement in parseCommandRnfr	Yes	Fixed
55	uFTP	Security	Missing Reauthorization Enforcement in parseCommandAuth After AUTH Command	Yes	Fixed
56	uFTP	State	Missing USERNAME-PASSWORD Sequence Enforcement in parseCommandPass	Yes	Fixed
57	wolfSSL	Parsing	Missing OID Value Validation in SendTls13Certificate for Client Certificates	Yes	Confirmed
58	wolfSSL	Parsing	Missing Extension Correspondence Check in SendTls13Certificate and ProcessPeerCerts	Yes	Confirmed
59	wolfSSL	Parsing	Missing Signature Algorithm Validation in DoTls13CertificateVerify	Yes	Confirmed
60	wolfSSL	Security	Incorrect Version Downgrade in DoTls13ClientHello Without supported_versions	Yes	Confirmed
61	wolfSSL	Security	Improper Use of legacy_version in DoTls13ClientHello	Yes	Fixed
62	wolfSSL	Security	Missing Startup Time Check for 0-RTT in DoClientTicketCheck	Yes	Confirmed
63	wolfSSL	Parsing	Missing Duplicate Extension Detection in TLSX_Push and TLSX_Parse	Yes	Confirmed
64	TLSE	Parsing	Missing OID Value Validation in tls_parse_certificate for Client Certificates	Yes	Confirmed
65	TLSE	Parsing	Missing Extension Correspondence Check in tls_parse_certificate and tls_certificate_request	Yes	Confirmed
66	TLSE	Parsing	Incorrect Signature Parameter Validation in tls_parse_* Functions	Yes	Confirmed
67	TLSE	Parsing	Missing Signature Algorithm Validation in tls_parse_verify_tls13 and tls_parse_payload	Yes	Confirmed
68	TLSE	Security	Missing psk_key_exchange_modes Validation in tls_parse_hello	Yes	Confirmed
69	TLSE	Error	Missing Handshake Abort for Group Mismatch in tls_parse_hello	Yes	Confirmed
70	TLSE	Security	Incorrect Version Negotiation in tls_parse_hello Without supported_versions	Yes	Confirmed
71	TLSE	Security	Missing KeyShareEntry Validation in tls_parse_hello and _private_tls_parse_key_share	Yes	Confirmed
72	TLSE	Security	Missing PSK Key Exchange Mode Validation in tls_parse_hello	Yes	Confirmed
			...		
89	Sol	Session	Missing Will Message Removal in read_callback After Publication	Yes	Reported
90	Sol	Parsing	Missing Will QoS Validation in connect_handler and unpack_mqtt_connect	Yes	Reported
91	Sol	Parsing	Missing Validation for Will Fields in unpack_mqtt_connect When Will Flag is 0	Yes	Reported
92	Sol	Parsing	Missing Prohibition of Will QoS 3 in connect_handler	Yes	Reported
93	Sol	State	Missing CONNECT Packet Enforcement in process_message and read_callback	Yes	Reported
94	Sol	Session	Missing Retain Flag in Will Message PUBLISH in connect_handler	Yes	Reported
95	Sol	Parsing	Incorrect Client ID Length Restriction in connect_handler	Yes	Reported
			...		
128	TinyMQTT	Parsing	Missing Will Retain Flag Validation in parse_connect_packet	Yes	Reported
129	TinyMQTT	State	Missing CONNACK Timeout Enforcement in parse_connect_packet and mqtt_connect	Yes	Reported
130	TinyMQTT	Error	Missing Rejection of Subsequent CONNECT Packets in decode_tcp_message_	Yes	Reported
			...		
155	Mosquitto	Parsing	Missing ShareName Character Validation in sub__topic_tokenise and sub__add	No	Confirmed
156	Mosquitto	Parsing	Missing Subscription Identifier Validation in mosquitto_property_check_command	No	Confirmed
157	Mosquitto	State	Incorrect QoS for Overlapping Subscriptions in subs__send	Yes	Confirmed
158	Mosquitto	State	Incorrect Topic Alias Handling in send__real_publish	Yes	Confirmed

TABLE V: Detail description of non-compliance bugs discovered by ProtocolGuard. The *Category* indicates the functional module where the bug occurred, *New* indicates whether the bug is a new finding, and *Status* represents the current status of the bug.