PSFuzzer: A Multi-Party Protocol Fuzzer for Publish/Subscribe Protocols

Xiangpu Song, Yingpei Zeng, Xiaofeng Liu, Xiaolei Liu, and Shanqing Guo

***Abstract*—Publish/subscribe protocols are widely adopted in IoT and distributed systems. Brokers mediate message delivery among multiple participants, acting as convergence points for complex multi-party dependencies and protocol states. As a result, flaws in broker implementations can compromise the security and stability of entire systems. Although fuzzing has proven effective for bug discovery, most existing fuzzers adopt a two-party model, which limits their ability to explore multi-party communication logic. Moreover, existing fuzzers typically focus on either memory corruption or logic errors without considering whether a broker implementation is specification-compliant.**

In this paper, we present PSFuzzer, a multi-party black-box fuzzer designed for pub/sub protocols. PSFuzzer employs a multi-party fuzzing framework with two concurrent senders to systematically explore multi-party interactions. We propose a unified intermediate representation for multi-party message transmission and introduce an LLM-driven dependency extraction approach to automatically extract multi-party dependency rules for various protocols. Guided by these rules, PSFuzzer coordinates concurrent message sending and combines random exploration with a message-priority scheduling strategy to improve fuzzing efficiency. To detect protocol non-compliance bugs, PSFuzzer leverages differential testing and an LLM-based analyzer to automatically validate and localize specification violations. We implemented PSFuzzer and evaluated it on 14 brokers across three categories of pub/sub protocols. PSFuzzer discovered 157 bugs, including 29 memory bugs and 128 non-compliance bugs, with 26 CVEs assigned. The comparison with state-of-the-art fuzzers indicates that PSFuzzer outperforms them in both code coverage and bug finding capabilities.

***Index Terms*—Multi-party Fuzzing, Protocol Fuzzing, Publish/Subscribe Protocols, Differential Testing.**

I. Introduction

The publish/subscribe (pub/sub) model is a fundamental messaging paradigm that enables scalable communication in distributed systems [11]. This model involves three key roles: publishers, subscribers, and brokers, thereby decoupling message production from message consumption. Due to its flexibility and scalability, the model is widely adopted in modern distributed systems and multi-party protocols like MQTT [6], which is one of the most popular protocols in the Internet of Things (IoT) ecosystem [1].

In pub/sub protocols, brokers are responsible for managing message distribution between publishers and subscribers, ensuring that messages are routed correctly according to topics and subscription rules. Publishers send messages with specific topics to the broker, while subscribers receive messages by registering their interests through the broker. As a result, dependencies arise not only between publisher-broker and subscriber-broker pairs, but also implicitly between publishers and subscribers through the broker's mediation. Given this central role, brokers become convergence points for complex communication dependencies and protocol states. Consequently, any flaw in a broker implementation may compromise the security and stability of the entire pub/sub system, posing serious risks to all connected participants.

Fuzzing is one of the most efficient methods for automatically discovering bugs and has achieved significant results [16], [61]. Recent studies have proposed various fuzzing approaches for pub/sub protocol brokers [17], [38], [43], [48], [51]. However, most existing approaches are designed around traditional two-party communication models. Such designs fail to capture the inherently multi-party interactions of pub/sub protocols [22], [43], limiting their ability to explore the complex multi-party logic within brokers. Additionally, existing fuzzers primarily focus on detecting memory corruption bugs, overlooking non-compliance bugs that can lead to unexpected behaviors and security vulnerabilities.

To address these challenges, we propose PSFuzzer, a novel multi-party fuzzing framework for brokers of pub/sub protocols, enabling systematic exploration of complex interaction logic and detection of both memory corruption and protocol non-compliance bugs. Specifically, we first introduce a unified intermediate representation (IR) to systematically represent multi-party interaction scenarios. Building on this representation, we design an LLM-driven dependency extraction approach that automatically extracts multi-party dependency rules for various protocols, guiding the coordinated message-sending behavior across concurrent senders. We then employ a message-priority scheduling strategy and combine it with dependency rules to guide the test case generation, enhancing the efficiency of bug discovery. In addition, we incorporate differential testing to detect protocol non-compliance bugs. To reduce the substantial manual effort typically required to validate such violations, we propose an LLM-based non-compliance analyzer that automatically confirms whether a detected behavior constitutes a specification violation and pinpoints the underlying rule being violated.

Xiangpu Song is currently a postdoctoral researcher with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong SAR, China. He was formerly with the School of Cyber Science and Technology, Shandong University, Qingdao 266237, China. E-mail: songxpu@gmail.com. Xiaofeng Liu and Shanqing Guo are with the School of Cyber Science and Technology, Shandong University, Qingdao, Shandong 266237, China. E-mail: 202590900104@sdu.edu.cn; guoshanqing@sdu.edu.cn. Xiaolei Liu is with the National Interdisciplinary Research Center of Engineering Physics, Mianyang, Sichuan 621000, China. E-mail: luxaole@gmail.com. Yingpei Zeng is with Hangzhou Dianzi University, Hangzhou, Zhejiang 310018, China. E-mail: yzeng@hdu.edu.cn.

Corresponding authors: Xiaolei Liu and Shanqing Guo.

We implemented PSFuzzer and evaluated it on multiple widely used protocols based on the pub/sub model, including MQTT [6], CoAP [4], and Stomp [14], across 14 different brokers. PSFuzzer discovered a total of 157 new bugs, including 29 memory bugs and 128 non-compliance bugs. We responsibly reported these bugs to the respective vendors, and at the time of writing, 139 bugs have been confirmed, with 97 bugs fixed and 26 CVEs assigned. The comparison with four state-of-the-art fuzzers demonstrates that PSFuzzer outperforms existing approaches in terms of code coverage and bug discovery. Furthermore, our evaluation of the LLM-based non-compliance analyzer indicates that it achieves an accuracy exceeding 90%.

Overall, we make the following contributions in this paper:

- We designed a novel multi-party black-box fuzzing framework for brokers of pub/sub protocols that considers dependencies across multiple parties.
- We proposed an LLM-driven dependency extraction approach to extract dependency rules for pub/sub protocols, enabling effective multi-party fuzzing.
- We proposed an LLM-based non-compliance bug analyzer to automatically validate and pinpoint the violations in these bugs, which achieves high accuracy.
- We implemented a prototype of PSFuzzer, and evaluated it on 14 various brokers. PSFuzzer revealed 157 new bugs and received 26 CVEs.

This paper extends our conference version, MBFuzzer, published at USENIX Security 2025 [43]. While MBFuzzer focuses on MQTT brokers and relies on manually identified multi-party dependency rules, this journal version generalizes the framework to support a broader range of pub/sub protocols. Specifically, we first define a unified IR to model multi-party communication behaviors in a protocol-irrelevant manner. Building on this IR, we develop an LLM-driven extraction approach that automatically derives dependency rules for different pub/sub protocols, reducing manual effort and enabling scalable adaptation. Consequently, we re-implemented the fuzzing framework as PSFuzzer. We further extended the evaluation beyond MQTT by adding experiments on two additional pub/sub protocols, CoAP and Stomp, covering eight more open-source brokers. These new experiments uncovered 36 previously unknown bugs, including 10 memory corruption bugs and 26 non-compliance bugs. We also revised the method for non-compliance bug counting to report bugs per affected broker implementation rather than the number of unique root causes, for clearer interpretation. In addition, we substantially revised and reorganized the paper for clarity and readability.

II. Background and Motivation

In this section, we first introduce the basics of the pub/sub protocol and use a real-world case to illustrate the motivation of our work and the limitations faced by existing research.

A. Publish/Subscribe Protocol Overview

Publish/subscribe protocols follow a broker-centric communication model in which publishers send messages to a broker and subscribers receive messages forwarded by the broker. A typical multi-party message transmission in pub/sub protocols proceeds in three stages. First, a subscriber establishes a connection with the broker and registers its interests by sending subscription requests. The broker then creates and maintains subscription-related state, such as topic filters and delivery options, for subsequent matching and forwarding. Second, a publisher establishes a connection with the broker and sends publish requests on specific topics, which the broker processes for matching and forwarding to subscribed clients. Third, upon receiving a publish request, the broker parses and validates the message, matches it against stored subscriptions, and forwards it to all eligible subscribers in accordance with the protocol's delivery rules.

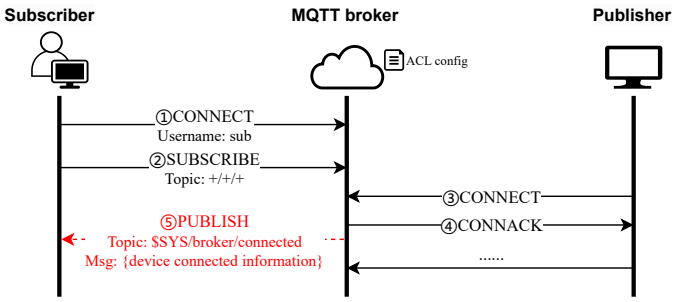

Fig. 1: Steps to trigger the access control bypass bug (CVE-2024-42655) in NanoMQ.

Such protocols are widely used in real-world systems that require scalable and many-to-many messaging, including IoT deployments and message-queue integration in distributed applications [1]. Prominent pub/sub protocols include MQTT [6] for IoT messaging, CoAP with the Observe extension [4], [9] for constrained devices, and Stomp [14] for message-oriented middleware interoperability.

B. Motivation

We use a real-world vulnerability in an MQTT broker as a case study to illustrate the motivation behind our approach and the limitations of existing work.

Real-world Example. The bug, assigned CVE-2024-42655, allows unauthorized access to sensitive system topics and leakage of sensitive information, such as the broker's version and details about connected devices, which can be highly valuable to hackers [15], [28]. Figure 1 illustrates how this bug can be triggered. Upon startup, the broker loads an access control configuration file that explicitly prohibits clients with the username `sub` from subscribing to topics that start with `$SYS`, which represents the system topic in MQTT [15]. A subscriber connects to the broker using the username `sub` (Step ①). Subsequently, the subscriber subscribes to the topic pattern `+/+/+` (Step ②), where the `+` symbol is a single-level wildcard that can represent any topic [8]. After that, a publisher connects to the broker and prepares to publish messages (Step ③). The broker responds to the publisher with a `CONNACK` message to acknowledge the connection (Step ④). Meanwhile, the broker constructs a `PUBLISH` message containing information about all currently connected devices to the topic `$SYS/broker/connected` and forwards this

message to the subscriber with the username `sub` (Step ⑤). However, the forwarding action violates the access control mechanism, as the user `sub` is explicitly prohibited from subscribing to system topics.

Limitations in Existing Protocol Fuzzers. The root cause of this bug is a failure to adhere to the protocol specification: *'The server must not match topic filters starting with a wildcard character with topic names beginning with a $ character'*. Existing fuzzers [17], [18], [38], [39], [48], [51] are limited in detecting such multi-party non-compliance bugs because they are implemented on the two-party fuzzing model. This makes it impossible to trigger the broker to forward `PUBLISH` messages in step ⑤, as this behavior requires a certain message from a third participant. Moreover, existing fuzzers lack a semantic understanding of such multi-party dependencies. As a result, they can only blindly mutate the input messages, which makes it difficult for them to generate test cases that expose the message and field dependencies (i.e., those consisting of `SUBSCRIBE` and `PUBLISH` messages and their `topic` fields) required to reveal this bug. Lastly, the unexpected behavior does not cause any crash in any participant, making it impossible for fuzzers relying on memory sanitizers to detect the bug [56], [62]. All these limitations call for a fuzzer that supports multi-party communication and the detection of non-compliance bugs that do not result in crashes.

III. Challenges and Solutions

There are several challenges that need to be addressed in designing a fuzzer that supports multi-party communication and the detection of both memory and non-compliance bugs.

To support multi-party communication, an intuitive approach is to introduce another sender that can send fuzzing input to the broker. This leads to the first challenge **C1: how to coordinate message sending across different senders?** Unlike existing fuzzers [18], [38] with only one sender, with two senders, after a test case is generated, it needs to be determined which sender should send this test case. Allocating test cases randomly to a sender is not feasible since it ignores the dependencies between the messages sent by the two senders.

Solution: To address this challenge, we design a unified IR to represent multi-party interaction behaviors and employ an LLM-driven dependency extraction approach to extract multi-party dependency rules. We further introduce protocol mappers to bridge these rules with the fuzzing engine, guiding coordinated message-sending behavior across two senders during fuzzing.

Black-box fuzzing does not require the instrumentation of source code and is well-suited for protocol fuzzing because most open-source brokers are implemented in different programming languages [50]. The lack of feedback from the program leads to the second challenge **C2: how to set fuzzing feedback for improving the bug discovery efficiency?** We observe that protocol specifications describe the potential errors in processing each message at different lengths. More detailed descriptions indicate more complex parsing logic, which increases the likelihood of introducing errors in the implementation, including memory bugs or non-compliance bugs. Thus, ideal feedback would guide fuzzers to allocate more computational resources to send messages that are more prone to errors.

Solution: To overcome this challenge, we use the unique inconsistency found by differential testing as fuzzing feedback because it indicates that developers have different understandings of the same logic, which could be potentially buggy [60]. Then, we use Q-learning [12] to dynamically prioritize the sending of each message under different states based on the number of inconsistencies.

In contrast to memory bugs, non-compliance bugs do not exhibit any obvious error behavior and require subsequent manual analysis of the inconsistent results [54]. This leads to the third challenge **C3: how to efficiently and accurately confirm non-compliance bugs and pinpoint the violations?** Manual verification not only requires a deep understanding of the protocol from the analyst but also consumes a significant amount of time, making it impractical for large-scale testing.

Solution: To address this challenge, we design a non-compliance bug analysis method based on LLMs, applying prompt chaining [10] and background-augmented prompting [46] strategy, to automatically validate and pinpoint the violation rules defined in specifications behind these bugs.

IV. Design and Implementation

A. Overview

Figure 2 shows the overall framework of PSFuzzer. We set up two message senders to play the roles of publisher and subscriber to send fuzzing messages in parallel, and their roles are dynamically specified according to the messages they sent. The workflow of PSFuzzer can be divided into five steps: (1) The *Dependency Extraction* automatically extracts multi-party dependency rules using LLMs, which are provided for the *Message Sending Coordination*. (2) In the fuzzing loop, the *Test Case Generation* generates and mutates fuzzing messages for each sender. (3) The *Message Sending Coordination* coordinates message sending between senders by influencing message generation. (4) The *Bug Detection* discovers memory bugs and non-compliance bugs by monitoring socket status and differential testing. These newly discovered inconsistencies will be provided for the *Message Priority Scheduler* as feedback. (5) After fuzzing, the *LLM-based Bug Analyzer* replays all inconsistency test cases to validate and pinpoint the violations behind these non-compliance bugs.

B. Dependency Extraction

To coordinate message generation and sending among senders, we identify two types of dependencies influencing multi-party communication: message dependency and field dependency. The *message dependency* specifies the message types that must be sent in a specific order to affect the communication process between senders and the broker. The *field dependency* refers to specific fields in dependent message sequences that the broker uses for message matching and distribution. The values of these fields need to be matched across messages in multi-party communication. These dependencies

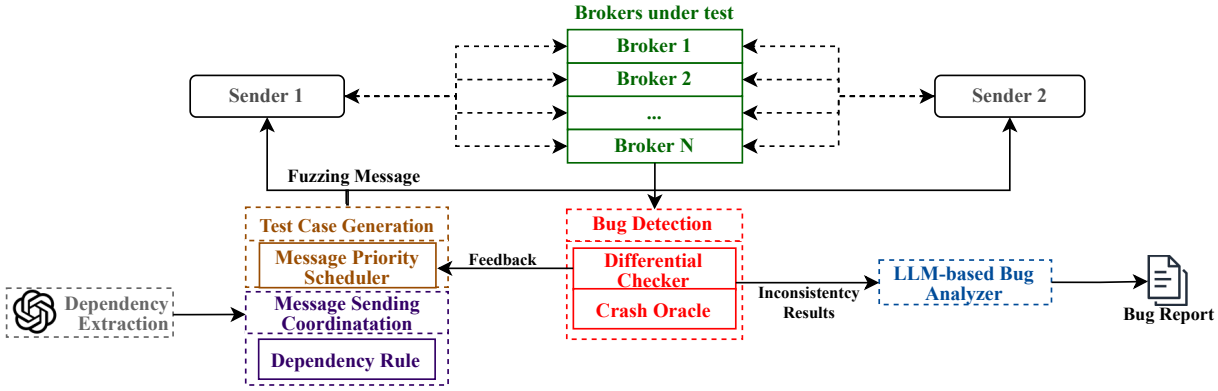

Fig. 2: Fuzzing framework of PSFuzzer.

are common in pub/sub protocols. For example, in Figure 1, the `SUBSCRIBE` and `PUBLISH` messages of MQTT have a message dependency as the two senders must establish a connection with the broker to register subscriptions and publish messages, respectively, which is a prerequisite for multi-party communication. The `topic` field in both messages has a field dependency because it is used by the broker to match incoming messages with corresponding subscriptions.

Building on these observations, we design an LLM-driven dependency extraction approach to automatically derive multi-party dependency rules. Instead of parsing protocol documents, we directly query LLMs, as they are trained on broad corpora that include many protocol specifications [36]. The extraction process proceeds in three steps. We first prompt the LLM to enumerate all message types defined by the target protocol and to return the field structure and a short semantic description for each message type. This step is used to build background knowledge that is later reused in subsequent prompts. Second, we prompt the LLM to summarize representative multi-party communication scenarios and express them as sequence diagrams in Mermaid format [5], since Mermaid provides a concise and text-based notation for specifying diagrams that LLMs can reliably generate [41]. In this step, we include the extracted knowledge (i.e., all message types and their field structures) in the prompt, enabling the LLM to systematically enumerate multi-party scenarios by reasoning over both message and field dependencies with fewer omissions. Notably, to enhance the completeness and stability of the LLM outputs, we employ a self-consistency [47] inspired approach in both steps: we generate multiple independent samples, cluster them according to protocol semantics, and instruct the LLM to perform iterative truthfulness assessment and refinement on the clustered results until all outputs are verified as consistent and faithful. Finally, we guide the LLM to abstract each scenario into a unified structured IR, from which dependency rules can be directly derived to guide multi-party protocol fuzzing. Due to space limitations, full prompts are available in the supplementary material.

To make the extracted rules executable in the fuzzing engine, we define a structured IR that transforms high-level multi-party scenarios into coordinated message-generation actions. Unlike conventional packet-format representations [52], which mainly describe the structure and constraints of individual messages, our IR is specifically tailored for multi-party interactions. In such scenarios, a single dependency rule typically involves multiple participants exchanging messages in a strict order, where the validity and effect of subsequent messages often depend on the protocol state or field values established by earlier messages from other parties. Representing these rules as independent messages fails to capture the critical temporal ordering and cross-message data-flow dependencies. To address this, we model each rule in the IR as an ordered interaction scenario. This abstraction explicitly defines the participating roles, the global message sequence among them, and the field-level constraints that bind or propagate values across messages. By focusing on these common structural elements rather than protocol-specific syntax, the IR provides a unified and protocol-agnostic way to represent multi-party dependencies. Consequently, fuzzers can instantiate multiple protocol endpoints in order, coordinate their communication sequence, and faithfully preserve inter-message dependencies, thereby enabling effective multi-party protocol fuzzing.

We define a unified structured IR consisting of three main components: *scenario metadata*, a list of *actors*, and an ordered list of interaction *steps*. The *scenario metadata* summarizes the scenario name, protocol version, and a short natural-language description, while *actors* declare the participants (e.g., clients and broker) and their roles involved in the scenario. Each step explicitly specifies the sender (`from`), receiver (`to`), message type (`type`), and a set of relevant field constraints (`fields`). The ordered `steps` encode constraints on message sending behavior across various participants, including which participant sends which message type and in what order (i.e., message dependencies), while `fields` specify field value constraints needed for multi-party transmission (i.e., field dependencies). To guide field instantiation, we introduce a set of annotations, written as brackets `[]` in the section `STEPS`. Specifically, `[RANDOM]` instructs the fuzzer to generate a value of the appropriate type for the field, `[BIND:X]` binds the instantiated value to a key X in a global value pool, and `[REF:X]` reuses the value previously bound to X when instantiating the current field, thereby enforcing field consistency across steps and participants. In addition to annotations, `fields` supports directly specifying literal values, which are utilized for instantiation during generation.

Figure 3 presents an example rule for MQTTv3.1.1 rep-

```
# SCENARIO METADATA
name = "standard_qos0_pubsub"
protocol = "MQTTv3.1.1"
description = "Standard QoS 0 Publish/Subscribe: ClientA
subscribes to topic T with QoS 0, ClientB publishes a
message to T, and ClientA receives it."

# ACTORS
[[actors]]
id = "C1"; role = "client"
[[actors]]
id = "C2"; role = "client"
[[actors]]
id = "B"; role = "broker"

# STEPS
# --- Step 1: Client1 CONNECT ---
[[steps]]
from = "C1"; to = "B"; type = "CONNECT";
fields = { client_identifier = "[RANDOM][BIND:CLIENT_ID1]" }
# --- Step 2: Client1 SUBSCRIBE ---
[[steps]]
from = "C1"; to = "B"; type = "SUBSCRIBE";
fields = { packet_identifier = "[RANDOM][BIND:PID_SUB1]",
topic_filter = "[RANDOM][BIND:T]", requested_qos = 0 }
# --- Step 3: Client2 CONNECT ---
[[steps]]
from = "C2"; to = "B"; type = "CONNECT";
fields = { client_identifier = "[RANDOM][BIND:CLIENT_ID2]" }
# --- Step 4: Client2 PUBLISH ---
[[steps]]
from = "C2"; to = "B"; type = "PUBLISH";
fields = { topic_name = "[REF:T]", qos = 0, retain = 0,
payload = "[RANDOM][BIND:msg]" }
```

Fig. 3: An example rule of MQTTv3.1.1 represented in IR.

resented in IR, describing multi-party communication under Quality of Service (QoS) level zero. It defines three actors (i.e., clients `C1`, `C2` and a broker `B`) and their interactions described in `[[steps]]`. The step order specifies message dependencies: `C1` first sends `CONNECT` and `SUBSCRIBE` messages, and then `C2` sends `CONNECT` and `PUBLISH` messages, constituting a pub/sub transmission. The `fields` map in each step specifies how to instantiate field values. For instance, `topic_filter = [RANDOM][BIND:T]` in `SUBSCRIBE` instructs the fuzzer to randomly generate a valid topic filter and bind it to T in a global pool, while `topic_name = [REF:T]` in `PUBLISH` reuses the same value to satisfy the topic matching requirement for delivery. In step 4, the key-value pair `qos = 0` specifies a concrete value constraint, indicating that the QoS is instantiated with the fixed value zero rather than being randomly generated.

C. Test Case Generation

During fuzzing, PSFuzzer generates test cases for each sender based on either its current protocol state or the selected rule, following three steps: determining the message type, instantiating the message, and mutating it.

Determine Message Type. PSFuzzer determines the next message type via two complementary strategies. One is by selecting a message type from the selected rule (lines 1-6 of Algorithm 1), which we will explain in detail in Section IV-D. The other is that PSFuzzer randomly selects a message type with a higher priority based on the message priority scheduler

Algorithm 1 Workflow of Test Case Generation

Input: ProtocolState, Sender, GlobalValuePool, SelectedRule
Output: FuzzingMsg
// Step 1: Determine the message type
1: **if** (SelectedRule $\neq$ NONE **and** HASAVAILSTEP(SelectedRule, Sender)) **or** (RAND$(0,1) < 0.5$) **then**
// Rule-based generation
2: GenMode $\leftarrow$ RULEMODE
3: **if** SelectedRule == NONE **or not** HASAVAILSTEP(SelectedRule, Sender) **then**
4: SelectedRule $\leftarrow$ SELECTRANDOMRULE()
5: **end if**
6: MsgType $\leftarrow$ GETCURSTEPMSGTYPE(SelectedRule, Sender)
7: **else**
// Message priority-based generation
8: GenMode $\leftarrow$ PRIORITYMODE
9: MsgType $\leftarrow$ GETMSGBYPRIORITY(ProtocolState)
10: **end if**
// Step 2: Instantiate message content
11: **if** GenMode == RULEMODE **then**
12: MsgContent $\leftarrow$ CREATEFUZZMSG(MsgType, SelectedRule, Sender, GlobalValuePool)
13: **else**
14: MsgContent $\leftarrow$ CREATEFUZZMSG(MsgType)
15: **end if**
// Step 3: Mutate message content
16: FuzzingMsg $\leftarrow$ MUTATE(MsgContent, GlobalValuePool)

(lines 8-9 of Algorithm 1). We utilize a random number with a threshold of `0.5` to determine the two strategies, balancing collaboration with random exploration.

To enhance the effectiveness of random exploration, we observed that the protocol specification provides varying lengths of descriptions on parsing errors for each message type, suggesting that the likelihood of each message triggering a bug may differ accordingly. Next, we employ Q-learning [12], a model-free reinforcement learning method, to design a message priority scheduler that adjusts the strategy based on the number of bugs discovered during fuzzing. Q-learning allows the agent to dynamically learn a state-action policy, determining the optimal action based on the observed environment state. After each action, the agent receives a reward, updates its policy, and repeatedly learns how to maximize the accumulated reward. Q-learning is particularly suitable for scenarios with discrete states and limited action spaces [12], making it a natural fit for pub/sub protocols like MQTT. Compared to other popular methods, such as Markov chains [25], [38], it performs better in this application, as demonstrated in Section V-E.

With Q-learning, PSFuzzer dynamically learns a state-action policy that records the selection probability of each message type in different states. After sending the message, PSFuzzer receives a reward based on new inconsistency feedback and updates the policy. Following the policy update, PSFuzzer select the next message type based on the refined policy, continuously optimizing the test case generation to maximize the chances of triggering bugs. The detailed steps of the Q-learning method are shown as follows.

Capturing States. We represent the sender's state when communicating with the broker using a hash over key fields in the response messages, such as the message type and the status of handling the previous request. These fields contain the information of message type and handling state, typically

reflecting the internal state of the server handling the current request [39]. For example, in MQTT, we use the *fixed header* and *Reason Code*, while in CoAP, we rely on the message type and response code. Since PSFuzzer tests multiple brokers simultaneously and may receive different responses, we follow the principle of majority rules and use the most frequently occurring state values as the state to record.

Learning Policy. We use a two-dimensional Q-table to accumulate the experience learned by PSFuzzer, where each column corresponds to a specific type of message, and each row represents a distinct response state. Each cell stores the Q-value for the corresponding state-action pair, which is initially set to `0` and is iteratively updated based on feedback from the reward function. To facilitate bug discovery, we use new inconsistencies discovered as feedback and assign a constant reward value to the corresponding cell, as inconsistencies are usually considered potential bugs [62]. We do not consider memory bugs as feedback because the black-box fuzzing cannot infer the crash trigger stack of crashes without program instrumentation. The reward function is designed as follows.

$$R(s,a,s') = \begin{cases} 1, & \text{if } \textit{new inconsistency} \\ 0, & \text{if } \textit{no or duplicate inconsistency} \end{cases} \tag{1}$$

Once a reward is received, the scheduler uses the Q function $Q : S \times A \rightarrow R$ to update the Q-value for the specific state-action pair in the Q-table. After an action a is performed, a new state s' is generated from the current state s. We then update the Q-value using the following formula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R(s,a) + \gamma \max_{a'} Q\left(s', a'\right) - Q(s,a) \right] \tag{2}$$

where the $\alpha \in (0,1]$ is the learning rate and $\gamma \in (0,1]$ is the discount factor. We set a common value of 0.1 and 0.9 for them [19], which performed well in our experiments.

Getting Next Actions. Given a state, the scheduler first converts the Q-values corresponding to each message type into a probability distribution [45] using the softmax function. The probability $P(a_i|s)$ for each action is computed as follows:

$$P\left(a_i \mid s\right) = \frac{\exp\left(Q\left(s, a_i\right)/T\right)}{\sum_{j=1}^{j} \exp\left(Q\left(s, a_j\right)/T\right)} \tag{3}$$

where $Q(s, a_i)$ is the Q-value for action a_i in the state s, and T is the temperature parameter used to control the smoothness of the probability distribution. We set T to `0.5` to maintain a balance in the probability distribution.

Subsequently, the scheduler selects the message type through sampling from the probability distribution to find the first action with a cumulative probability $P(a_i|s)$ that exceeds the random number $p \in (0,1]$. This method ensures that messages with higher probabilities are more likely to be chosen, while messages with lower probabilities still have a chance of being selected [37].

Instantiate and Mutate Message. After determining the message type, PSFuzzer performs message instantiation based on the syntax-based generation strategy (line 11-15 of Algorithm 1). We first define the format template of each message based on the protocol specification, including the range of valid values for each field, to ensure the validity of the message format. PSFuzzer then generates random values based on the range for each field and assembles them in sequence to form a valid message. During message instantiation, if the current message type is determined from the selected rule, PSFuzzer uses the field values recorded in the *SelectedRule* and *GlobalValuePool* (explained later in Section IV-D) for instantiation instead of randomly generating to satisfy the field dependency (line 12 of Algorithm 1).

Lastly, PSFuzzer performs field-level mutation operations on messages generated by the syntax-based generation, as mutation is crucial for exploring diverse field values and detecting various bugs [24] (line 16 of Algorithm 1). The mutation process includes three types of operations: bit-flipping, arithmetic operations, and payload operations (e.g., deleting, injecting, and replacing a random segment of bytes). These mutations intentionally perturb initially valid field values and may generate boundary, extreme, or invalid values, thereby enabling PSFuzzer to exercise protocol implementations beyond normal-value ranges and explore corner cases in field processing. Notably, fields with dependencies stored in the *GlobalValuePool* are preserved without mutation to ensure that the field dependencies are not broken.

D. Message Sending Coordination

Multi-party protocol fuzzing requires not only generating individual messages, but also coordinating when and by whom they are sent so that both message and field dependencies across participants are satisfied. We address this challenge by introducing a coordination strategy that is based on rules.

To translate rules into executable actions, PSFuzzer implements protocol-specific mappers that link abstract message and field names used in rules to their corresponding message generation and field instantiation logic. For each protocol, we maintain a unified namespace of message and field identifiers to ensure consistent naming across rules and avoid ambiguity during execution. We then implement the mapper by manually aligning message and field identifiers with the fuzzer's instantiation logic via rule-aware hooks. These hooks intercept the default message and field generation logic and apply rule-defined constraints whenever applicable. With this mapping, a rule step can be directly instantiated as a concrete message and dispatched by the designated sender. For example, a rule step such as *'from = "C1", to = "B", type = "CONNECT"'* in Figure 3 can be directly translated into an executable action: PSFuzzer instantiates a `CONNECT` message and dispatches the message via sender `C1` to the broker `B`.

PSFuzzer treats each rule as an execution plan that guides generation. At the beginning of each fuzzing campaign, PSFuzzer randomly selects a rule as the *SelectedRule*. The test case generator then switches to the RULEMODE mode and follows the rule step by step until no steps are available (lines 1-6 of Algorithm 1). At each step, PSFuzzer determines the next sender and message type specified by the rule and generates the corresponding message accordingly. In addition to message dependencies, PSFuzzer resolves field dependencies

by maintaining a global value pool, *GlobalValuePool*, which stores field values generated during rule execution as key-value pairs. To ensure field consistency across participants, PSFuzzer adopts a bind-and-reference strategy. When a field is annotated with `[BIND:ID]`, the generated value is recorded in the global pool under the identifier `ID`. Subsequently, when encountering a field annotated with `[REF:ID]`, PSFuzzer retrieves the previously bound value and assigns it to the current field, ensuring that dependent fields across messages and participants remain consistent.

E. Bug Detection

Differential Checker. To detect non-compliance bugs that do not typically cause program crashes, we design a differential checker based on differential testing [34]. This approach compares the same functionality across implementations, serving as mutual reference oracles, and has proven effective in detecting non-compliance bugs [56], [62].

The differential checker sequentially collects messages returned by each broker and analyzes them field by field to identify inconsistencies. Initially, the checker categorizes messages received by the senders: published messages are saved to the forwarding message queue, while all other messages are stored in the response queue. This is because a subscription request can trigger both a single response (e.g., `SUBACK` in MQTT) and multiple forwarded publish messages. Furthermore, according to protocol specifications [7], [8], brokers are allowed to send publish messages before the response. Such protocol features can cause false positives during differential testing. Next, the checker reorders and aligns all forwarding message queues using combinations of the topic and payload fields in published messages as digests. This alignment ensures the consistency of forwarded messages, as network latency and differences in broker processing logic may cause forwarded messages to arrive in varying orders for each sender. Finally, the checker analyzes all messages in the queues by comparing them field by field to identify inconsistencies. During this process, we classify fields into redundant and non-redundant categories [38] to avoid false positives. Redundant fields, such as the assigned client identifier in `CONNACK` of MQTT, typically have their values randomly generated by the broker. For these fields, only their presence is verified. Non-redundant fields, on the other hand, are checked for both presence and value consistency. The checker also filters duplicate inconsistencies to avoid impact on feedback and subsequent validation.

Crash Oracle. To detect bugs related to memory and non-memory crashes, we monitor the target brokers via socket connection state, such as connection refusal and timeouts. Notably, we enable ASAN [2] for open-source brokers implemented in the C/C++ programming language to improve the efficiency of memory bug detection as a complementary measure.

F. LLM-based Bug Analyzer

Inconsistency results do not always equate to non-compliance bugs and require manual verification to determine their validity and root cause [26], [56]. However, manual analysis is always daunting and labor-intensive. Inspired by recent advancements in LLMs for root cause analysis [40] and their deep understanding of protocols [36], we design a non-compliance bug analyzer based on LLMs to automatically verify and analyze the protocol violation behind inconsistencies.

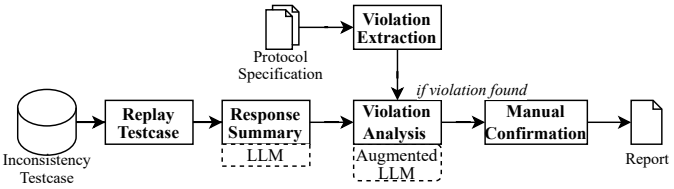

Fig. 4: Framework of LLM-based non-compliance bug analyzer.

We use the background-augmented prompting approach [46] to design prompts. Although the pre-training corpus of popular LLMs includes various protocol specifications [36], directly providing test case content to the LLM can lead to significant hallucination issues. Since protocol specifications often use a standardized format to indicate violations, such as keywords like *MUST* and *MUST NOT*, we incorporate these elements into the prompts provided to the LLM, making the prompts more accurate and generalized [46]. To overcome context limitations, we use regular expressions to automatically extract violation descriptions from specifications and store them in the format *'Protocol version - Message type'*. Only descriptions relevant to the protocol version and message type of the current test case are included in the prompt.

To ensure the stability of the LLM's analysis, we employ the prompt chaining strategy [10], dividing the analysis process into two parts, as shown in Figure 4. First, we replay the inconsistency test cases and convert the messages exchanged in the communication process into a readable text format. Next, we use the LLM to analyze and summarize the request and broker response information from the replay. The summary result is included in the prompts of violation analysis, with the augmented LLMs determining if the current test case exhibits any violations. If violations are found, we conduct further manual verification and report the findings to the developers. The full prompts are provided in the supplementary material.

V. Evaluation

We wrote about 6k lines of Python code to implement PSFuzzer. To evaluate the effectiveness, we conducted experiments on real-world pub/sub protocol brokers. Specifically, our evaluation aims to answer the following questions:

- **RQ1.** Can PSFuzzer find bugs in real-world pub/sub protocol brokers and what are their security impacts? (Section V-A)
- **RQ2.** How precise is the dependency rule extraction by PSFuzzer? (Section V-B)
- **RQ3.** Can PSFuzzer outperform other state-of-the-art protocol fuzzers? (Section V-C)
- **RQ4.** Can PSFuzzer outperform existing protocol compliance analysis tools in non-compliance bug detection? (Section V-D)
- **RQ5.** How does each component contribute to PSFuzzer? (Section V-E)

Benchmark. To evaluate the effectiveness and generality of PSFuzzer, we conduct experiments on three widely used protocols that adopt the pub/sub model: MQTT, CoAP (via the Observe extension [9]), and Stomp, which covers common IoT messaging and message-queue integration scenarios. Table I summarizes the benchmark subjects used in our evaluation. For MQTT, we select six mainstream, production-grade brokers. For CoAP and Stomp, we additionally include four implementations per protocol to broaden the diversity of codebases and language stacks.

Protocol	Subject	Language	LOC	Version
MQTT	EMQX	Erlang	–	5.6.0
	Mosquitto	C/C++	16.6K	2.0.18
	NanoMQ	C	104.6K	0.21.10
	VerneMQ	Erlang	-	1.13
	HiveMQ	Java	-	4.24.0
	FlashMQ	C/C++	24.6K	1.12.1
CoAP	ccoap	C	4.4K	77f55c4
	microcoap	C	0.5K	ef27289
	CoAPthon3	Python	10.7K	5b75e56
	freecoap	C	26.6K	3adc2ea
Stomp	stomper	C	0.6K	5e2741e
	coilmq	Python	1.4K	a2e580e
	Gozirra	Java	-	efb95dd
	activemq	Java	-	ef789aa

TABLE I: Benchmark subjects grouped by protocol. The *LOC* denotes the lines of code base in the corresponding language. The - indicates that the LOC is unavailable or not measured.

Setting. For the dependency extraction module, we employed the *Grok-4.1* model to enumerate protocol message types and fields, summarize representative multi-party scenarios, and derive dependency rules in the IR. For the LLM-based bug analyzer, we used the OpenAI *gpt-4o-2024-05-13* model with its default temperature setting to summarize replayed traces and reason about potential specification violations. Both models were accessed via their public APIs. In addition, we also set default parameters for the LLM, such as temperature.

A. Discovered Real-world Bugs

We evaluated the effectiveness of PSFuzzer in identifying bugs in brokers, which is the primary purpose of fuzzing [29].

Subject	Memory Bug			Non-Compliance Bug		
	Report	Confirmed	Fixed	Report	Confirmed	Fixed
EMQX	1	1	1	17	13	7
Mosquitto	0	0	0	11	9	3
NanoMQ	15	15	14	28	28	14
VerneMQ	0	0	0	16	15	15
HiveMQ	0	0	0	7	7	3
FlashMQ	3	3	3	23	23	22
ccoap	5	5	0	3	0	0
microcoap	0	0	0	2	0	0
CoAPthon3	0	0	0	1	0	0
freecoap	0	0	0	1	0	0
stomper	5	5	0	4	0	0
coilmq	0	0	0	6	0	0
Gozirra	0	0	0	6	6	6
activemq	0	0	0	3	3	3
Total	29	29	18	128	104	73

TABLE II: Detailed bug discovery results of PSFuzzer.

Table II presents the results of bug discovery across the 14 evaluated brokers. PSFuzzer discovered a total of 157 bugs, including 29 memory bugs (29 confirmed, with 18 fixed) and 128 non-compliance bugs (104 confirmed, and 73 were fixed). The remaining unconfirmed bugs are those for which the vendor had not yet responded as of the time of writing. Table III summarizes the detailed results of 29 memory bugs, most of which could lead to denial of service, memory leaks, or even remote code execution in brokers. Tables S.I, S.II, and S.III in the supplementary material present the detailed findings of 128 non-compliance bugs, including confirmation and fix statuses from each vendor. While most of these bugs do not directly compromise broker security, they can affect the functional availability of brokers or pose security risks to other communication participants.

```
1  int sub_ctx_handle(nano_work *w) {
2    nng_msg **retain = w->msg_ret;
3    while (tn) {
4      topic_str = tn->topic.body;
5      topic_exist = dbhash_check_topic(topic_str);
6      uint8_t rh = tn->retain_handling;
7      if (rh == 0 || (rh == 1 && !topic_exist))
8        retain = dbtree_find_retain(w->db_ret,
             topic_str);  // malloc retain
9       w->msg_ret = (w->msg_ret == NULL) ? retain :
            w->msg_ret;
10     for (i = 0; i < c_size(retain) && w->msg_ret
           != retain; i++)  // UAF here
11       cvector_push_back(w->msg_ret, retain[i]);
12     if (retain != w->msg_ret)
13       c_free(retain); // free retain
14     tn = tn->next;
15   }
16 }
```

Listing 1: Simplified code of case study 2.

To better understand the security impact of bugs found by PSFuzzer, we discuss three representative bugs as case studies, one of which has been introduced in Motivation (Section II-B). ***Case study 2: Use after free in NanoMQ.*** This vulnerability, labeled as bug M8 and assigned CVE-2024-42651 in Table III, occurs during the pub/sub message transmission process in NanoMQ. Listing 1 presents a simplified code snippet illustrating this issue. Specifically, the bug arises when NanoMQ processes subscribe to messages containing multiple subscription topics (lines 3-14). During this process, the broker extracts retained messages matching the subscription topics from the cache (line 8) and adds them to the message queue for subsequent delivery (line 11). However, since NanoMQ does not set the pointer `retain` to `NULL` after releasing retain messages (line 13), if a subsequent subscription topic matches the same retain message, and the retain handling option `rh` for this topic is set to 1, the broker would skip allocating the retain message (lines 7-8). Thus, a use-after-free bug occurs in line 10. The bug is hard to find with existing fuzzers because of the message and field dependency. PSFuzzer effectively utilizes dependency rules to guide the generation of retained publish messages and corresponding subscribe messages, enabling the successful detection of this issue.

Case study 3: Non-compliance Bug#N41 in NanoMQ. This non-compliance bug, labeled as bug N41 in Table S.I occurs when NanoMQ processes a publish message containing invalid property values. According to the protocol specification [8], *'A Control Packet which contains an Identifier which is not*

ID	Project	Version	Bug Description	Potential Security Issue	0 Day	Status
M1	EMQX	5.6.0	Segmentation fault in the function dump_module_literals	Denial of Service	×	Confirmed & Fixed
M2	FlashMQ	1.12.1	Crash in handling will `PUBLISH` message	Denial of Service	✓	Confirmed & Fixed
M3	FlashMQ	1.14.0	Assertion in saving retain `PUBLISH` message to the database	Denial of Service	✓	CVE-2024-42645
M4	FlashMQ	1.14.0	Assertion in forwarding will `PUBLISH` message	Denial of Service	✓	CVE-2024-42644
M5	NanoMQ	0.17.5	Segmentation fault in handling `PUBLISH` message	Denial of Service	✓	CVE-2024-42650
M6	NanoMQ	0.17.5	Segmentation fault in handling `CONNECT` message	Denial of Service	✓	CVE-2023-34488
M7	NanoMQ	0.17.5	Heap buffer overflow in handling `SUBSCRIBE` message	Remote code execution	✓	CVE-2024-54832
M8	NanoMQ	0.17.9	Heap use after free in handling `SUBSCRIBE` message	Remote code execution	✓	CVE-2024-42651
M9	NanoMQ	0.17.9	Segmentation fault in handling `PUBLISH` message	Denial of Service	✓	CVE-2024-54836
M10	NanoMQ	0.21.10	Socket file description exhaustion caused by session keeping feature	Denial of Service	✓	Confirmed
M11	NanoMQ	0.21.10	Segmentation fault in handling client requests	Denial of Service	✓	CVE-2024-42646
M12	NanoMQ	0.21.10	Stack buffer overflow in handling `PUBLISH` message	Remote code execution	✓	CVE-2024-54834
M13	NanoMQ	0.22.1	Stack buffer overflow in handling `PUBLISH` message	Remote code execution	✓	CVE-2024-42647
M14	NanoMQ	0.22.1	Memory Leak after receiving message	Memory leakage	✓	CVE-2024-42649
M15	NanoMQ	0.22.1	Heap buffer overflow in handling `CONNECT` message	Remote code execution	✓	CVE-2024-42648
M16	NanoMQ	0.22.4	Heap use after free in handling `PUBLISH` message	Remote code execution	✓	CVE-2024-54831
M17	NanoMQ	0.22.4	Memory leak in receiving message	Memory leakage	✓	CVE-2024-54830
M18	NanoMQ	0.22.4	Segmentation fault in message	Denial of Service	✓	CVE-2024-54833
M19	NanoMQ	0.22.4	Memcpy-param-overlap in handling `PUBLISH` message	Memory Corruption	✓	CVE-2024-54830
M20	stomper	5e2741e	Segmentation fault in handling `SEND` message	Denial of Service	✓	CVE-2026-26445
M21	stomper	5e2741e	Unhandled `SIGPIPE` in handling `CONNECT` message	Denial of Service	✓	CVE-2026-26446
M22	stomper	5e2741e	Use-after-free in handling duplicate `SUBSCRIBE` message	Denial of Service	✓	CVE-2026-26447
M23	stomper	5e2741e	Use-after-free in handling duplicate `CONNECT` message	Denial of Service	✓	CVE-2026-26448
M24	stomper	5e2741e	Denial of Service caused by partial `SEND` message with EPOLLET	Denial of Service	✓	CVE-2026-26449
M25	ccoap	77f55c4	Segmentation fault in handling malformed CoAP options	Denial of Service	✓	CVE-2026-26459
M26	ccoap	77f55c4	Segmentation fault in handling zero-length CoAP options	Denial of Service	✓	CVE-2026-26457
M27	ccoap	77f55c4	Segmentation fault in handling malformed `URI_PATH` options	Denial of Service	✓	CVE-2026-26453
M28	ccoap	77f55c4	Segmentation fault caused by invalid CoAP option number handling	Denial of Service	✓	CVE-2026-26452
M29	ccoap	77f55c4	Use-after-free due to race condition in session handling	Denial of Service	✓	CVE-2026-26456

TABLE III: Memory bugs discovered in brokers by PSFuzzer.

valid for its packet type, or contains a value not of the specified data type, is a Malformed Packet. If received, use a CONNACK or DISCONNECT packet with Reason Code 0x81 (Malformed Packet)'. However, NanoMQ skips illegal property values during parsing and subsequently forwards the malformed packet to eligible subscribers without following the specification to disconnect. The bug is hard for existing fuzzers to detect because it does not have obvious erroneous behaviors. PSFuzzer successfully discovered and identified the violation of this bug through differential testing and the LLM-based bug analyzer. Although the bug does not pose a direct security risk to the broker itself, we found it can lead to unexpected behavior in other participants connected to the broker. For instance, we observed that `mosquitto_sub` is affected by this bug and it triggers an assertion error and crashes when receiving such messages. The `mosquitto_sub` is a C library provided by Mosquitto, widely integrated into various devices as clients. This case study proves the importance of adhering to protocol specifications in multi-party IoT environments. Non-compliance bugs in implementations can also lead to significant issues, highlighting the need for rigorous fuzzing.

B. Dependency Extraction Effectiveness

To evaluate the effectiveness of our LLM-driven dependency extraction, we compare the extracted rules against manually constructed ground truth. Specifically, for each evaluated protocol, two researchers independently constructed the set of unique multi-party dependency rules as ground truth by following the manual analysis steps described in [43], with disagreements resolved through discussion. We then assess whether PSFuzzer can accurately and completely identify multi-party interaction rules from protocol specifications.

Table IV summarizes the results of LLM-driven dependency extraction. PSFuzzer initially extracts 60 raw rules, which are consolidated into 48 semantically unique rules after deduplication. We consider two extracted rules equivalent when they describe the same underlying multi-party interaction and are established by the same cross-party dependency, even if they additionally contain different auxiliary field conditions. For example, multiple MQTT rules may describe message delivery established by the same topic-matching dependency, while some additionally specify properties such as the `Payload Format Indicator` or `Reason String`. Since these auxiliary properties do not establish a distinct cross-party dependency, we treat such rules as the same underlying dependency rather than separate multi-party interaction rules. We therefore evaluate precision and recall based on these semantically unique rules. Among the 48 unique rules, 44 match the ground truth (TP), with 4 false positives (FP), yielding an overall precision of 91.7%. Against 49 ground-truth rules, PSFuzzer misses 5 rules (FN), achieving an overall recall of 89.8%.

Protocol	Rule	GT	TP	FP	FN	Precision	Recall
CoAP	6(4)	4	4	0	0	100%	100%
MQTTv3.1.1	10(9)	10	9	0	1	100%	90.0%
MQTTv5.0	15(12)	14	12	0	2	100%	85.7%
STOMPv1.0	9(6)	5	5	1	0	83.3%	100%
STOMPv1.1	10(8)	8	7	1	1	87.5%	87.5%
STOMPv1.2	10(9)	8	7	2	1	77.8%	87.5%
Overall						**91.7%**	**89.8%**

TABLE IV: Effectiveness of LLM-driven dependency extraction. *Rule* reports raw extracted rules, with deduplicated semantically unique rules shown in parentheses. *GT* denotes manually identified ground-truth rules. *TP*, *FP*, and *FN* are computed by matching unique extracted rules against the ground truth, and the overall precision and recall are computed over all protocols.

We further analyze how the FP and FN rules affect the

Subject	AFLNet		SGFuzz		Fume		MPFuzz		PSFuzzer	
	Coverage	Bug	Coverage	Bug	Coverage	Bug	Coverage	Bug	Coverage	Bug
EMQX	-	-	-	-	-	**1**/0	-	0/0	-	**1/17**
Mosquitto	2296	0/0	2046	0/0	3306	0/0	2592	0/0	**3684**	**0/11**
NanoMQ	9747	1/0	-	-	11436	**3**/0	9965	1/0	**11621**	**3/28**
VerneMQ	-	-	-	-	-	0/0	-	0/0	-	0/**16**
HiveMQ	-	-	-	-	-	0/0	-	0/0	-	0/**7**
FlashMQ	4130	1/0	4581	2/0	4263	2/0	4272	**3**/0	**4653**	**3/23**
		2/0		2/0		6/0	-	4/0		**7/102**

TABLE V: Detailed fuzzing results on MQTT brokers over 24 hours of PSFuzzer and SOTA fuzzers. The *Bug* is presented as the format *"Memory bugs/Non-compliance bugs"*. The symbol - means that the fuzzer or metric cannot work in that subject.

subsequent fuzzing process. The few FP rules may cause PSFuzzer to select rules that do not actually require multi-party coordination or that duplicate existing rules, thereby reducing the efficiency of multi-party exploration. The FN rules, in contrast, may leave some multi-party dependencies unexplored because the corresponding coordinated interactions are not explicitly generated. Nevertheless, these cases are limited in number and thus have only a modest impact on the overall fuzzing process. Moreover, all generated messages are still subjected to fuzzing mutations and executed by the target implementation, so these cases do not prevent PSFuzzer from exploring message parsing and processing logic.

We also analyzed the extracted rules in details. For MQTT, the rules cover message transmission under different QoS levels, retained message delivery, will message delivery, request/response patterns, persistent-session delivery, message expiry, and subscription management (`SUBSCRIBE` and `UNSUBSCRIBE`). Additionally, MQTTv5.0-specific rules capture advanced features including wildcard subscriptions, shared subscriptions, subscription identifiers, topic aliases, the no-local flag, and retained-message handling options. For CoAP, the rules align with its four basic message types (`CON`, `NON`, `ACK`, and `RST`), modeling confirmable transfers (`CON` requiring `ACK`), non-confirmable transfers (`NON` without acknowledgment), and subscription lifecycle management via the Observe extension (Observe=0 to subscribe; Observe=1 or RST to unsubscribe). For Stomp, the rules capture fundamental pub/sub delivery (`SEND` followed by `MESSAGE`), queue/topic delivery semantics, transactional messaging for both publishers and subscribers (`BEGIN`, `SEND`, `ACK`/`NACK`, `COMMIT`, and `ABORT`), request/response patterns via temporary reply queues, and acknowledgment semantics across modes (`auto`, `client`, and `client-individual`). They also cover subscription termination and non-durable delivery behavior after disconnection. Overall, these rules provide executable multi-party interaction ways that guide coordinated message generation, allowing PSFuzzer to explore various multi-party logic that is difficult to exercise under a two-party fuzzing model.

C. Comparison to Existing Protocol Fuzzers

To compare PSFuzzer with existing fuzzers, we use protocol-specific baseline fuzzers whenever available. For MQTT, we select three state-of-the-art fuzzers: AFLNet [39], SGFuzz [18], and Fume [38]. We configure AFLNet to skip the deterministic stage for improved performance [23]. For SGFuzz, we employ *NetDriver* to enable protocol fuzzing as recommended. For CoAP, we adopt FuzzCoAP [35], a CoAP-specific black-box fuzzer, as the baseline. We also include MPFuzz [32] as an MQTT baseline because it is the closest existing work that supports collaborative protocol fuzzing for IoT messaging protocols. Although MPFuzz also targets CoAP, its public repository does not provide the corresponding CoAP configuration files. Therefore, we include MPFuzz in the MQTT comparison and exclude it from the CoAP comparison. For Stomp, to the best of our knowledge, there is no publicly available fuzzer that focuses on the Stomp protocol. Therefore, we exclude Stomp from the comparison. The comparison focused on two key metrics: code coverage, and bug discovery. We used the *gcov* tool to measure the branch coverage, which is applicable specifically to programs written in the C/C++ programming language. To mitigate randomness in fuzzing results, we repeated each fuzzing campaign five times, with each run lasting 24 hours.

Subject	FuzzCoAP		PSFuzzer	
	Coverage	Bug	Coverage	Bug
ccoap	648	4/0	**803**	**5/3**
microcoap	275	0/0	**281**	0/**2**
CoAPthon3	-	0/0	-	0/**1**
freecoap	862	0/0	**875**	0/**1**
		4/0		**5/7**

TABLE VI: Detailed fuzzing results on CoAP brokers over 24 hours of PSFuzzer and SOTA fuzzers (CoAP). The *Bug* is presented as the format *"Memory bugs/Non-compliance bugs"*. The symbol - means that the fuzzer or metric cannot work in that subject.

Table V and Table VI present detailed fuzzing results on MQTT and CoAP brokers for PSFuzzer and other SOTA fuzzers over 24 hours in terms of code coverage and bug discovery. The comparison was conducted using the broker versions listed in Table I. Overall, PSFuzzer outperformed existing state-of-the-art fuzzers in terms of code coverage and bug discovery on both MQTT and CoAP brokers. For MQTT brokers, PSFuzzer achieved higher code coverage than AFLNet, SGFuzz, Fume, and MPFuzz on the same comparable benchmarks, with average increases of 24%, 26%, 5%, and 19%, respectively. We employed the Mann-Whitney U-test [27] to calculate the p-value for code coverage comparisons between PSFuzzer and the other fuzzers, all of which were below 0.05, indicating statistical significance. In terms of bug discovery, PSFuzzer found 7 memory bugs and 102 non-compliance bugs across all MQTT brokers. In contrast, AFLNet, SGFuzz, Fume, and MPFuzz found 2, 2, 6, and 4 memory bugs, respectively, and none of them detected non-compliance bugs. It also demonstrates the effectiveness of

PSFuzzer in guiding the exploration of the multi-party communication input space compared to MPFuzz. For CoAP brokers, PSFuzzer also achieved the best overall results. Compared with FuzzCoAP, PSFuzzer achieved higher code coverage with an average improvement of 10%. Moreover, PSFuzzer found 5 memory bugs and 7 non-compliance bugs, whereas FuzzCoAP could only find 4 memory bugs and no non-compliance bugs.

We further investigated why PSFuzzer performs better than other fuzzers. Firstly, PSFuzzer discovered numerous non-compliance bugs through differential testing, whereas existing fuzzers only support the detection of memory bugs. Secondly, PSFuzzer dynamically adjusts the priority of message types through the message priority scheduler based on inconsistency feedback, thus assigning more computational resources to send these messages, enhancing bug discovery efficiency. For example, for MQTT brokers, most of the inconsistency results in the experiments are concentrated in four message types, `CONNECT`, `SUBSCRIBE`, `UNSUBSCRIBE`, and `PUBLISH`. Since these messages typically involve the most parsing logic in brokers and specifications, the scheduler also helps achieve higher code coverage. Compared with MPFuzz, PSFuzzer differs in coordination granularity and feedback signals. MPFuzz improves field-level consistency by synchronizing critical fields across fuzzing instances, but complex pub/sub behaviors often require higher-level message dependencies, such as subscription before publication. This means that MPFuzz can only generate test cases that satisfy this dependency through random mutations. PSFuzzer explicitly encodes such dependencies as rules to coordinate participant roles, message order, and cross-message field relations. Moreover, using the proposed new multi-party fuzzing framework and dependency rules, PSFuzzer successfully detects vulnerabilities in the pub/sub message transmission (Bug M4 and M23 in Table III), while other fuzzers fail to detect. It also demonstrates the effectiveness of our approach in guiding the exploration of the multi-party communication input space.

D. Comparison to Existing Protocol Compliance Analysis Tools

To evaluate the effectiveness of PSFuzzer in detecting non-compliance bugs through differential testing, we compare it with two state-of-the-art static protocol compliance analysis tools, ParDiff [57] and RFCAudit [21]. We conduct the comparison on three widely used MQTT implementations: Mosquitto, NanoMQ, and FlashMQ. These subjects are implemented in C/C++ and have been widely adopted in real-world MQTT deployments. For ParDiff, since it is designed for static differential analysis between two C-language codebases and mainly focuses on message parsers, we apply it to Mosquitto and NanoMQ, which are both implemented in C. Following its intended usage guideline, we manually identify the message parsing entry functions in the two implementations and provide them as analysis targets. For RFCAudit, we replace its default LLM with DeepSeek V4 Flash, which provides stronger code understanding capability at lower cost, following the model comparison used in the original RFCAudit evaluation [3]. Since different tools may discover overlapping or distinct bugs, we first cluster all reported results and count the number of unique bugs for each subject in the evaluation.

Protocol	Total Bugs	ParDiff	RFCAUDIT	PSFuzzer
Mosquitto	17	-	9	11
NanoMQ	36	-	10	28
FlashMQ	30	-	10	23
Total	**83**	-	**29**	**62**

TABLE VII: Comparison results with existing protocol compliance analysis tools in detecting non-compliance bugs. The symbol - indicates that the tool failed to produce valid analysis results for the target subject.

Table VII summarizes the comparison results with existing RFC compliance detection tools. Overall, PSFuzzer detects 62 out of 83 unique non-compliance bugs, significantly outperforming RFCAudit (29 bugs) and ParDiff (which fails to produce any valid analysis results in these subjects). These results demonstrate the superior effectiveness of PSFuzzer in exposing non-compliance bugs across real-world MQTT implementations.

We further analyze the performance gap. ParDiff primarily identifies parser-level code differences between C-based implementations. However, in Mosquitto and NanoMQ, the message parsing logic is distributed across multiple functions and involves indirect calls, making it difficult for ParDiff to establish accurate parser-level correspondences. Although RFCAudit performs better than ParDiff, its effectiveness remains limited. RFCAudit relies on code summarization and LLM-based retrieval to locate specification-relevant code fragments. These summaries may omit critical implementation details necessary for accurate compliance checking [44]. Moreover, since RFCAudit organizes analysis according to specification sections, the retrieved code context is often large and noisy, which degrades the accuracy of subsequent LLM-based reasoning [31]. In contrast, PSFuzzer adopts dynamic differential testing instead of purely static analysis. By directly validating behavioral inconsistencies during execution, it eliminates the need for additional dynamic confirmation. Furthermore, its LLM-based bug analyzer effectively summarizes the root causes of observed inconsistencies, substantially reducing manual effort in post-analysis.

E. Ablation Study

Effectiveness of Dependency Rules and Message Priority Scheduler. To assess the impact of dependency rules and the message priority scheduler based on inconsistency feedback, we conducted an ablation study by disabling these methods. We designed three variants: PSFuzzer$_{!d}$, PSFuzzer$_{!r}$, and PSFuzzer$_{!m}$. In PSFuzzer$_{!d}$, the capability of dependency rules is disabled to evaluate their effectiveness in facilitating the exploration of the input space for multi-party communication. In PSFuzzer$_{!r}$, the message priority scheduler is disabled and replaced with a random message selection strategy to understand whether they can improve bug discovery performance. To evaluate the effectiveness of Q-learning within the message priority scheduler, we designed PSFuzzer$_{!m}$, which replaces Q-learning with a rule-based approach. This approach uses the widely used Markov chain [25], [38] to model

Subject	PSFuzzer$_{!d}$		PSFuzzer$_{!r}$		PSFuzzer$_{!m}$		PSFuzzer	
	Coverage	Bug	Coverage	Bug	Coverage	Bug	Coverage	Bug
EMQX	-	**1**/**17**	-	0/**17**	-	**1**/16	-	**1/17**
Mosquitto	3327	0/10	3209	0/10	3482	0/10	**3684**	0/**11**
NanoMQ	**11695**	**3**/26	11414	2/25	11550	**3**/24	11621	**3/28**
VerneMQ	-	0/**16**	-	0/14	-	0/15	-	0/**16**
HiveMQ	-	0/6	-	0/6	-	0/**7**	-	0/**7**
FlashMQ	4342	2/20	4582	**3**/20	4382	2/19	**4653**	**3/23**
ccoap	787	4/**4**	801	4/**4**	**803**	**5/4**	**803**	**5/4**
microcoap	213	0/1	**282**	0/1	272	0/1	281	0/**2**
CoAPthon3	-	0/**1**	-	0/**1**	-	0/**1**	-	0/**1**
freecoap	764	0/**1**	869	0/**1**	871	0/**1**	**875**	0/**1**
		10/102		9/99		11/98		**12/110**

TABLE VIII: Results of ablation study over 24 hours of PSFuzzer. The *Bug* is presented as the format *"Memory bugs/Non-compliance bugs"* . The symbol - means that the fuzzer or metric cannot work in that subject.

message generation, assigning selection probabilities based on the percentage of unique inconsistent responses each message triggers [55]. The ablation study was conducted on the same benchmarks and metrics detailed in Section V-C.

Table VIII presents the results of the ablation study on PSFuzzer, focusing on code coverage and bug discovery. The results demonstrate the observable impact of each component on code coverage and bug discovery performance. Specifically, disabling dependency rules (PSFuzzer$_{!d}$) reduces the fuzzer's ability to explore the specific execution paths involving multi-party communication, leading to an average decrease of 16.5% in code coverage. Moreover, without dependency guidance, PSFuzzer$_{!d}$ struggles to detect bugs such as M4 and N29, which are related to the pub/sub message transmission. Similarly, replacing the message priority scheduler with a random message selection strategy (PSFuzzer$_{!r}$) treats all message types equally during test case generation. This approach diminishes the fuzzer's efficiency in both discovering bugs and exploring parsing logic in brokers. Most bugs and parsing logic are concentrated in a small subset of message contexts. Thus, PSFuzzer$_{!r}$ fails to detect bugs like M1 and M10, while code coverage declines by an average of 16.4%. For instance, detecting bug M10 requires sending a large number of `CONNECT` messages of MQTT with varying client identifiers for persistent sessions within a short time, which is challenging to achieve using a random strategy. Lastly, while PSFuzzer$_{!m}$ dynamically calculates selection probabilities for each message based on historical data, it overlooks the impact of protocol state changes on message selection. This limitation causes it to mistakenly generalize message priority in some states to priority in all states. For example, 93% of the MQTT messages sent by PSFuzzer$_{!m}$ were `CONNECT` and `PUBLISH`, limiting its exploration of other message types, such as `SUBSCRIBE`. This imbalance results in an average 15.2% reduction in code coverage and prevents the detection of bugs associated with other message types. Notably, for CoAP implementations, the coverage differences among PSFuzzer, PSFuzzer$_{!r}$, and PSFuzzer$_{!m}$ are marginal. This is expected because CoAP has a much simpler state machine and substantially fewer core message contexts, leaving limited room for state-aware prioritization to outperform random or Markov-chain-based selection.

Accuracy of LLM-based Bug Analyzer. To evaluate the accuracy of the LLM-based bug analyzer in verifying and analyzing inconsistency test cases generated by the differential checker, we construct the evaluation dataset from MQTT experiments. We chose MQTT because it yields the largest number and the richest diversity of inconsistency outcomes in our evaluation, providing sufficient statistical power and a representative mix of message types and error-handling behaviors for assessing violation identification. Specifically, we randomly selected 500 inconsistency test cases that lead to inconsistent behaviors of different broker implementations as the evaluation dataset. This dataset also allows us to quantify the bug detection rate of inconsistencies identified by differential testing correspond to real bugs, since all selected cases were first produced by the differential checker and then manually validated against the protocol specifications. We then invited two researchers to manually analyze the results generated by the module, referencing the protocol specifications. After completing the analysis, we compared the results from the two researchers. Any discrepancies were further given to a third analyzer for secondary analysis to eliminate potential biases in the manual evaluation. We also recorded the number of test cases linked to each non-compliance bug during manual validation, considering that one test case can trigger multiple non-compliance bugs or multiple test cases can lead to the same bug [30]. This resulted in duplicate results in the LLM-based bug analyzer.

LLM (GPT 4o)		Truth	
		Violation (Unique/Total)	Non-Violation (Unique/Total)
Prediction	Violation	53/340	10/24
	Non-Violation	4/11	125[1]

TABLE IX: Results of validation of the accuracy of LLM-based bug analyzer. The row *Prediction* represents the classification of results by the LLM, while the column *Truth* indicates the manually verified outcomes.

[1] This data does not include duplicate violations, as neither the LLM predictions nor the facts contain any violation.

Table IX shows the results of manually confirming the accuracy of the LLM-based bug analyzer. The LLM identified 72.8% (364/500) of the test cases as containing violations, while the remaining 27.2% (136/500) were determined to have no violations. Our manual analysis confirmed that 340 of the 364 test cases flagged by the LLM indeed contained actual violations, while 125 of the 136 test cases contained non-violations. Overall, the LLM-based bug analyzer attains

a precision of 93.4% and a recall of 96.9% on the dataset, highlighting its effectiveness in accurately detecting and identifying protocol violations. From the perspective of differential testing, 351 out of the 500 sampled inconsistency cases correspond to real protocol violations, while the remaining 149 cases are implementation differences that do not constitute non-compliance bugs. This corresponds to a precision of 70.2% and a FP rate of 29.8% for raw differential inconsistency reporting. Among the 340 results correctly identified by the LLM as containing violations, we found 53 unique violations (i.e., non-compliance bugs in Table S.I). Additionally, in the 11 cases where the LLM incorrectly identified no violations, 4 unique violations were discovered. Finally, in the 24 cases where the LLM incorrectly identified violations, it mistakenly flagged 10 unique violations. Similarly, the column *Count* in Table S.I records the number of effective inconsistency test cases in the dataset associated with each non-compliance bug. On average, each non-compliance bug is triggered by seven test cases, with at least one test case triggering every bug.

The LLM-based bug analyzer demonstrated an accuracy of over 90% in detecting both protocol violations and non-violations, showcasing the effectiveness of the designed prompts and workflow. This accuracy significantly enhances the efficiency of manually analyzing inconsistencies generated by differential testing, as it reduces the time and effort required for human intervention. For instance, the LLM takes approximately 30 seconds to perform an analysis of a test case. Researchers in our study then spend an average of just one minute per case to verify the LLM's results, making this approach substantially faster than conducting a full manual analysis for each test case. Additionally, the LLM-generated reports offer valuable insights and serve as validation support, further streamlining the confirmation process and improving overall analysis efficiency.

VI. Discussion and Limitations

PSFuzzer has successfully identified numerous bugs, yet it still has certain limitations. In this section, we discuss these limitations and solutions for future improvements.

Multi-party Fuzzing Framework. PSFuzzer employs two dynamically assigned senders to act as the primary participant roles in pub/sub protocols, namely, a publisher and a subscriber, while treating the broker as the target under test. To coordinate these senders' message types, field values, and sending order, PSFuzzer introduces a set of dependency rules, which together cover most role-based multi-party interaction scenarios. However, this rule-guided approach imposes a predefined sending order, which inherently limits the framework's ability to trigger bugs related to concurrent multi-party message processing—currently, such exploration relies on random scheduling. To address this, we plan to introduce concurrency-aware scheduling in future work, enabling systematic exploration of these behaviors.

LLM-based Dependency Extraction. The LLM significantly reduces the manual effort needed to construct multi-party dependency rules for pub/sub protocols. In our experience, manually deriving such rules requires an expert to analyze specifications and infer dependencies, typically taking over one hour for MQTT [43]. In contrast, the LLM can generate high-quality candidate rules in approximately five minutes, greatly facilitating the extension of PSFuzzer to new protocols. LLMs may still introduce hallucinations or unstable outputs, such as redundant rules or a small number of non-multi-party interaction rules. To mitigate this issue, we employ multiple independent samplings and auditing steps to improve the stability of the extracted results. Nevertheless, since such rules account for only a small portion, they have limited impact on the overall fuzzing effectiveness.

Manual Effort. While PSFuzzer automatically extracts multi-party dependency rules for new protocols, integrating a new protocol still requires minor manual effort. Specifically, we manually implement protocol mappers that link message and field identifiers to the corresponding message generators and field instantiation functions in the fuzzer. However, this effort is lightweight and one-time per protocol, and we expect it can be further reduced by leveraging LLM agents in future work.

Bug Analyzer. The LLM-based bug analyzer focuses on violations that are explicitly described in the protocol specification and thus may miss those that are not related to the specification. Besides, the bug analyzer automatically filters out test cases that LLM determines to be non-violating without further manual analysis, which may result in a small number of test cases that violate the specification being discarded incorrectly. Nonetheless, the results of LLM analysis can provide useful insights into manual validation and significantly improve the efficiency of manual efforts. It is worth noting that the current evaluation is limited to MQTT, as it produced the largest and most diverse set of inconsistency cases in our experiments. Consequently, its effectiveness on other protocols remains to be validated, and we leave such validation for future work.

Differential Checker. Due to the inherent limitations of differential testing [56], PSFuzzer may fail to detect certain types of non-compliance bugs. Specifically, when all brokers exhibit the same responses to a given request, differential testing cannot identify inconsistencies, potentially overlooking existing bugs. In future work, we will consider integrating the LLMs into the fuzzing process as a complement to the differential checker to enhance violation detection.

Generalizability of PSFuzzer. PSFuzzer focuses on various multi-party protocols based on the pub/sub model. Nevertheless, the dependency extraction workflow and IR are not tied to a specific protocol or specification, as they model dependencies through roles, message sequences, and field-level constraints. Therefore, they can potentially be applied to other protocols with similar cross-participant and cross-message dependency patterns. However, extending PSFuzzer to protocols with fundamentally different architectures or more complex coordination semantics, such as Raft [13], may require adapting the coordination framework and dependency representation, which we leave as future work.

VII. Related Work

Protocol Fuzzing. There have been several fuzzing techniques proposed and applied to fuzz pub/sub protocol implementations. MultiFuzz [51] is a grey-box multi-party fuzzer for MQTT brokers, but it simply adds client socket connections. Fume [38] uses Markov chains to model the process of message generation for fuzzing MQTT brokers. SHADOWFUZZER [49] proposes a novel fuzzing approach to fuzz MQTT clients via brokers. SGANFuzz [48] uses generative adversarial networks to guide test cases for brokers. FuzzCoAP [35] is a black-box fuzzer for CoAP implementations that generates test cases based on the protocol specification. In addition, there are generic protocol fuzzers that can also be used to fuzz brokers. AFLNet [39] is the first grey-box protocol fuzzer that uses the response code as the state feedback. SGFuzz [18] recognizes the enum-type variables in code as state variables automatically to construct the state feedback. ChatAFL [36] introduces LLM to process Request for Comments (RFCs) and generate fuzzing message sequences. mGPTFuzz [33] leverages LLM to automatically convert human-readable content to machine-readable information to construct finite-state machines for guiding black-box fuzzing for IoT devices. Although many fuzzers could be used for fuzzing brokers, they cannot fully explore the input space of multi-party communications, limiting bug discovery capabilities. PSFuzzer addressed these challenges by proposing a new multi-party fuzzing framework and a series of strategies.

Differential Testing. Differential testing has been widely used in testing network protocol implementations to identify semantic bugs. TCPFuzz [62] applies differential testing to fuzz the TCP stack to find semantic bugs. ResolFuzz [20] and ResolverFuzz [53] use differential testing to test DNS resolvers to find non-crash vulnerabilities in them. Moreover, TLS-DeepDiffer [56] detects logic flaws throughout TLS interactions based on message tuples and differential testing. However, existing fuzzers are not directly applicable to pub/sub protocols. In addition to differences in protocol semantics, multi-party communication can also cause response messages to be disordered. Moreover, most existing methods also rely on manual analysis to analyze the root causes of non-compliance bugs. PSFuzzer designs the first differential checker for pub/sub protocols and introduces LLM to bug analysis to automatically pinpoint the violations behind inconsistency results.

Protocol Compliance Analysis. Several specialized techniques have been proposed for detecting non-compliance bugs in protocol implementations. RIBDetector [21] extracts normative statements from RFCs as rules and uses heuristic patterns to identify corresponding condition-checking logic in the code. ParDiff [57] applies differential testing in static analysis to detect inconsistencies in network protocol parsers. PARCLEANSE [59] leverages LLMs to extract message formats from protocol specifications and uses them as oracles to validate parser correctness. RFCAudit [58] employs LLMs to audit protocol implementation compliance against specifications. However, most existing approaches rely primarily on static analysis. While effective at locating potential issues, they still require substantial manual effort for dynamic verification and confirmation of the reported problems [42]. In contrast, PSFuzzer combines differential testing with LLM-based bug analysis. This approach not only achieves higher accuracy in identifying protocol compliance violations, but also effectively detects other logical implementation issues that are unrelated to the specifications.

VIII. Conclusion

In this paper, we proposed PSFuzzer, a novel multi-party black-box fuzzer for pub/sub protocols to detect both memory and non-compliance bugs. PSFuzzer introduces a unified intermediate representation to model multi-party interactions and an LLM-driven dependency extraction approach to derive rules across various protocols. PSFuzzer then uses a message-priority scheduling approach based on inconsistency feedback to adjust the priority of messages in different states dynamically to improve the efficiency of bug discovery. Subsequently, PSFuzzer introduces differential testing and LLM-based bug analysis methods to detect and automatically verify non-compliance bugs. We implemented a prototype, and experiments show that PSFuzzer is effective in finding bugs in brokers, with 157 bugs discovered, of which 26 CVEs have been assigned. It could also achieve higher code coverage and vulnerability discovery than state-of-the-art fuzzers.

References

[1] 2022 survey shows mqtt adoption is high for industry. https://www.hivemq.com/blog/2022-survey-shows-mqtt-adoption-is-high-for-industry/. Accessed on 2025-12-22.

[2] Addresssanitizer wiki. https://github.com/google/sanitizers/wiki/AddressSanitizer. Accessed on 2024-8-9.

[3] Claude 3.5 sonnet vs deepseek v4 flash. https://benchlm.ai/compare/claude-3-5-sonnet-vs-deepseek-v4-flash. Accessed on 2026-6-19.

[4] The constrained application protocol (coap). https://datatracker.ietf.org/doc/html/rfc7252. Accessed on 2025-12-23.

[5] Mermaid diagramming and charting tool. https://mermaid.js.org/. Accessed on 2025-12-24.

[6] Mqtt: The standard for iot messaging. https://mqtt.org/. Accessed on 2025-12-22.

[7] Mqtt version 3.1.1 specification. https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. Accessed on 2024-8-9.

[8] Mqtt version 5.0 specification. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. Accessed on 2024-8-9.

[9] Observing resources in the constrained application protocol (coap). https://datatracker.ietf.org/doc/html/rfc7641. Accessed on 2025-12-25.

[10] Prompt chaining. https://www.promptingguide.ai/techniques/prompt_chainin. Accessed on 2024-8-13.

[11] Publish-subscribe_pattern. https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern. Accessed on 2025-12-22.

[12] Q-learning wikipedia. https://en.wikipedia.org/wiki/Q-learning. Accessed on 2024-8-21.

[13] The raft consensus algorithm raft. https://raft.github.io/. Accessed on 2026-6-19.

[14] Stomp: The simple text oriented messaging protocol. https://stomp.github.io/. Accessed on 2025-12-23.

[15] Sys-topics: Why you shouldn't use sys-topics for monitoring. https://www.hivemq.com/blog/why-you-shouldnt-use-sys-topics-for-monitoring/. Accessed on 2024-8-9.

[16] Why fuzzing over formal verification. https://blog.trailofbits.com/2024/03/22/why-fuzzing-over-formal-verification/. Accessed on 2024-8-8.

[17] Bernhard K Aichernig, Edi Muškardin, and Andrea Pferscher. Learning-based fuzzing of iot message brokers. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 47–58. IEEE, 2021.

[18] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, 2022.

[19] Anders Blomqvist and Christian Andersson. Exploring the parameter space of q-learning for faster convergence using snake, 2022.

[20] Jonas Bushart and Christian Rossow. Resolfuzz: Differential fuzzing of dns resolvers. In *European Symposium on Research in Computer Security*, pages 62–80. Springer, 2023.

[21] 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.

[22] Samin Y Chowdhury, Ruimin Sun, and Brandon Dudley. A survey for mqtt fuzzing. In *Proceedings of the 2025 Workshop on Re-design Industrial Control Systems with Security*, pages 16–25, 2025.

[23] 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.

[24] Philipp Görz, Björn Mathis, Keno Hassler, Emre Güler, Thorsten Holz, Andreas Zeller, and Rahul Gopinath. Systematic assessment of fuzzers using mutation analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4535–4552, 2023.

[25] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle. A tool for model-checking markov chains. *International Journal on Software Tools for Technology Transfer*, 4:153–172, 2003.

[26] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[28] Wenhao Li, Selvakumar Manickam, Priyadarsi Nanda, Ayman Khallel Al-Ani, Shankar Karuppayah, et al. Securing mqtt ecosystem: Exploring vulnerabilities, mitigations, and future trajectories. *IEEE Access*, 2024.

[29] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. Unifuzz: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2777–2794, 2021.

[30] Jiawei Liu, Yuheng Huang, Zhijie Wang, Lei Ma, Chunrong Fang, Mingzheng Gu, Xufan Zhang, and Zhenyu Chen. Generation-based differential fuzzing for deep learning libraries. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–28, 2023.

[31] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

[32] Zhengxiong Luo, Junze Yu, Qingpeng Du, Yanyang Zhao, Feifan Wu, Heyuan Shi, Wanli Chang, and Yu Jiang. Parallel fuzzing of iot messaging protocols through collaborative packet generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3431–3442, 2024.

[33] Xiaoyue Ma, Lannan Luo, and Qiang Zeng. From one thousand pages of specification to unveiling hidden bugs: Large language model assisted fuzzing of matter {IoT} devices. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4783–4800, 2024.

[34] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[35] Bruno da S Melo and Paulo Lício de Geus. Robustness testing of coap server-side implementations through black-box fuzzing techniques. In *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*, pages 533–540. SBC, 2017.

[36] 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)*, 2024.

[37] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox fuzzing of distributed systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1615–1629, 2023.

[38] Bryan Pearson, Yue Zhang, Cliff Zou, and Xinwen Fu. Fume: Fuzzing message queuing telemetry transport brokers. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 1699–1708. IEEE, 2022.

[39] 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.

[40] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. Exploring llm-based agents for root cause analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 208–219, 2024.

[41] Basel Shbita, Farhan Ahmed, and Chad DeLuca. Mermaidseqbench: An evaluation benchmark for llm-to-mermaid sequence diagram generation. *arXiv preprint arXiv:2511.14967*, 2025.

[42] Xiangpu Song, Longjia Pei, Jianliang Wu, Yingpei Zeng, Gaoshuo He, Chaoshun Zuo, Xiaofeng Liu, Qingchuan Zhao, and Shanqing Guo. Protocolguard: Detecting protocol non-compliance bugs via llm-guided static analysis and dynamic verification. In *NDSS*, 2026.

[43] 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 *34th USENIX Security Symposium (USENIX Security 25)*, pages 6179–6197, 2025.

[44] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source code summarization in the era of large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1882–1894. IEEE, 2025.

[45] S Syafiie, F Tadeo, and E Martinez. Softmax and ε-greedy policies applied to process control. *IFAC Proceedings Volumes*, 37(12):729–734, 2004.

[46] 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 196–196. IEEE Computer Society, 2024.

[47] 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.

[48] Zhiqiang Wei, Xijia Wei, Xinghua Zhao, Zongtang Hu, and Chu Xu. Sganfuzz: A deep learning-based mqtt fuzzing method using generative adversarial networks. *IEEE Access*, 2024.

[49] Huikai Xu, Miao Yu, Yanhao Wang, Yue Liu, Qinsheng Hou, Zhenbang Ma, Haixin Duan, Jianwei Zhuge, and Baojun Liu. Trampoline over the air: Breaking in iot devices through mqtt brokers. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 171–187. IEEE, 2022.

[50] Bin Yuan, Zhanxiang Song, Yan Jia, Zhenyu Lu, Deqing Zou, Hai Jin, and Luyi Xing. Mqttactic: Security analysis and verification for logic flaws in mqtt implementations. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2385–2403. IEEE, 2024.

[51] Yingpei Zeng, Mingmin Lin, Shanqing Guo, Yanzhao Shen, Tingting Cui, Ting Wu, Qiuhua Zheng, and Qiuhua Wang. Multifuzz: A coverage-based multiparty-protocol fuzzer for iot publish/subscribe protocols. *Sensors*, 20(18):5194, 2020.

[52] 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.

[53] 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.

[54] Xiaohan Zhang, Cen Zhang, Xinghua Li, Zhengjie Du, Bing Mao, Yuekang Li, Yaowen Zheng, Yeting Li, Li Pan, Yang Liu, and Robert Deng. A survey of protocol fuzzing. *ACM Comput. Surv.*, 57(2), 2024.

[55] Lei Zhao, Yue Duan, and Jifeng XUAN. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. Network and Distributed System Security Symposium (NDSS), 2019.

[56] 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.

[57] 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.

[58] Mingwei Zheng, Chengpeng Wang, Xuwei Liu, Jinyao Guo, Shiwei Feng, and Xiangyu Zhang. Rfcaudit: Ai agent for auditing protocol

implementations against rfc specifications. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1221–1233. IEEE, 2025.

[59] 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.

[60] Qingyang Zhou, Qiushi Wu, Dinghao Liu, Shouling Ji, and Kangjie Lu. Non-distinguishable inconsistencies as a deterministic oracle for detecting security bugs. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3253–3267, 2022.

[61] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

[62] 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.