MBFuzzer: A Multi-Party Protocol Fuzzer for MQTT Brokers

Xiangpu Song[1], *Jianliang Wu* [*2], *Yingpei Zeng* [*3], *Hao Pan*[1], *Chaoshun Zuo*[4], *Qingchuan Zhao*[5] *and Shanqing Guo* [*1,6]
[1]*Shandong University*
[2]*Simon Fraser University*
[3]*Hangzhou Dianzi University*
[4]*Ohio State University*
[5]*City University of Hong Kong*
[6]*Shandong Key Laboratory of Artificial Intelligence Security*

Abstract

MQTT is a multi-party communication protocol widely used in IoT environments, where MQTT brokers act as servers that connect with numerous devices. Consequently, any flaws in brokers will seriously impact all participants. Given the success of fuzzing techniques in finding bugs in programs, existing fuzzing works targeting MQTT brokers face the limitation of insufficient fuzzing input space because they all adopt a two-party fuzzing model. Accordingly, the code responsible for handling multi-party communication will not be examined. Moreover, existing fuzzers focus on either memory corruption bugs or logic errors without considering whether a broker implementation is specification-compliant.

In this paper, we design a black-box fuzzing approach, MBFuzzer, for brokers to address the above limitations. We first design a multi-party fuzzing framework containing two fuzzing input senders to facilitate the exploration of code space that handles multi-party communication. To improve fuzzing efficiency, we design a message priority scheduler, six dependency rules, and a dependency queue to guide test case generation and coordinate the message sending of the two senders, respectively. We leverage differential testing to identify non-compliance bugs and design an LLM-based non-compliance bug analysis method to automatically analyze the bug report and validate whether it is a non-compliance bug. We implemented a prototype MBFuzzer and evaluated it with six mainstream MQTT brokers. MBFuzzer successfully identified 73 bugs including 20 memory bugs and 53 non-compliance bugs with 11 CVEs assigned. The comparison with state-of-the-art fuzzers indicates that MBFuzzer outperforms them in both code coverage and bug finding capabilities.

1 Introduction

Message Queue Telemetry Transport (MQTT) is a popular Internet of Things (IoT) protocol that uses a publish/subscribe (pub/sub) message delivery model for multi-party communication [54, 60]. The communication of MQTT has three roles: publisher, subscriber, and broker, where the broker is the server-side protocol implementation for managing and routing messages. Brokers can simultaneously connect to numerous entities, such as clients, brokers, and commercial IoT platforms (e.g., AWS IoT Core [2]), enabling multi-party message transmission. With the flexible messaging model and reliability, brokers have been widely used in production environments [5, 26]. Therefore, any bug in brokers would pose a serious security risk to thousands of communication participants.

Notably, recent research has proposed various strategies to find bugs in brokers, including static analysis [33], formal verification [56, 59] and fuzzing [27, 48, 57, 60]. Among these strategies, fuzzing is one of the most efficient ways to discover bugs automatically and achieve great results [24, 66]. Despite the measurable success of fuzzing, existing fuzzers have limited ability to explore the logic of multi-party communication in brokers, thus limiting bug discovery.

In detail, the broker is responsible for routing and distributing all messages in MQTT. Publishers send messages with specific topics to the broker, and subscribers receive these messages by subscribing to the relevant topics through the broker. Consequently, there are message dependencies not only between the publisher and the broker or the subscriber and the broker but also between the publisher and the subscriber. However, existing fuzzers [29, 48, 50, 57] use a two-party fuzzing model (i.e., the fuzzer acts as a client, and the broker acts as the fuzz target) and only considers message dependencies in the two-party communication scenario (i.e., publisher-/subscriber-broker). These approaches limit the exploration of fuzzing input space for brokers, reducing the effectiveness of discovering bugs. Additionally, existing fuzzers either focus on memory corruption bugs [29, 48] or logic errors [59], without considering if an implementation is specification-compliant.

To address these challenges, we design MBFuzzer by proposing a novel black-box fuzzing approach for brokers to

*Corresponding authors.

discover both memory and non-compliance bugs. To explore the input space for multi-party communication, we design a multi-party fuzzing framework containing two message senders, playing the roles of subscriber and publisher. We first design six dependency rules and a dependency queue shared by senders, considering the message dependencies between publisher/subscriber and broker and the dependencies between the publisher and subscriber, to coordinate and facilitate the generation of fuzzing messages across different senders. Then, we use extended Petri net [41, 49] to model the communication behavior of these two parallel senders to clarify the process of collaboration. We design a message priority scheduler to improve the efficiency of exploring the input space and bug discovery. Moreover, we employ differential testing [44] to detect non-compliance bugs. To streamline the validation process for non-compliance bugs that would otherwise require substantial manual efforts, we propose a non-compliance bug analyzer based on Large Language Models (LLM). This tool automatically verifies whether a bug is non-compliant and pinpoints the specific violation behind this bug.

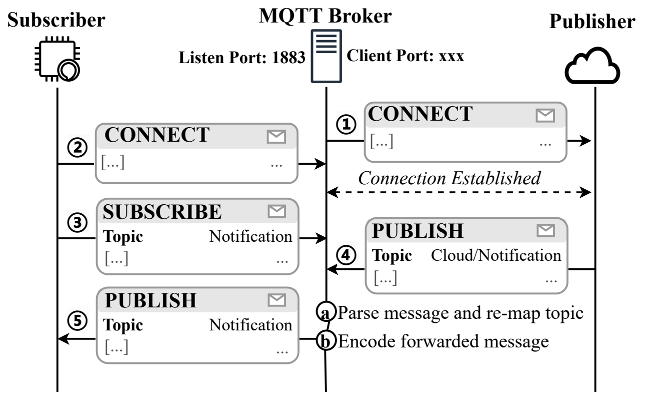
We implemented MBFuzzer and evaluated it on six open-source MQTT brokers, covering four different programming languages. MBFuzzer discovered a total of 73 new bugs, including 20 memory bugs and 53 non-compliance bugs. We responsibly reported these bugs to related vendors, and at the time of paper writing, 69 bugs are confirmed with 61 bugs fixed and 11 CVEs received. The comparison with three state-of-the-art fuzzers shows that MBFuzzer outperforms existing fuzzers in code coverage and bug discovery. Our evaluation of the LLM-based non-compliance bug analyzer indicates that it can achieve an accuracy of more than 90%.

Overall, we make the following contributions in this paper:

- We designed a novel multi-party black-box fuzzing framework for MQTT brokers that considers message dependencies across multiple parties.
- 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 MBFuzzer, and evaluated it on six popular brokers. MBFuzzer revealed 73 new bugs and received 11 CVEs.

2 Background and Motivation

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



Subscriber
MQTT Broker
Publisher
Listen Port: 1883
Client Port: xxx
① CONNECT [...] ...
② CONNECT [...] ...
Connection Established
③ SUBSCRIBE Topic Notification [...] ...
④ PUBLISH Topic Cloud/Notification [...] ...
⑤ PUBLISH Topic Notification [...] ...
ⓐ Parse message and re-map topic
ⓑ Encode forwarded message

Figure 1: Multi-party communication scenario of MQTT brokers.

2.1 MQTT Protocol

MQTT uses a pub/sub message delivery model [8], a multi-party communication paradigm widely used in distributed environments [54], consisting of four main components: publishers, subscribers, brokers, and topics. The client sending messages (publisher) and the client receiving messages (subscriber) do not need to establish a direct connection. The brokers handle all message routing and distribution based on topics. Notably, the brokers can not only connect with various clients but also enable message transmission through MQTT bridge [31] with other brokers and commercial IoT platforms (can be considered as a kind of broker). The MQTT bridge is a core feature and enables message interchange under various edge networks. In bridging scenarios, the broker that initiates the bridge request to another broker or IoT platform is called the *bridge broker* [31]. This functionality has proven useful in production environments with multiple levels of message aggregation [11], becoming a de facto standard [10].

Figure 1 shows a common multi-party communication scenario involving a client, a broker, and an IoT platform, where the client and platform are labeled *Subscriber* and *Publisher*. In this scenario, the broker must first initiate a connection request to the *Publisher* to establish a data transmission channel between them ①. Subsequently, the *Subscriber* within the local network connects to the broker ② and subscribes to the topic `Notification` ③. The *Publisher* then pushes a message with the topic `Cloud/Notification` to the broker ④. The broker processes the message by parsing and remapping the topic ⓐ, converting the topic from other platforms to the required local network topic `Notification`, which is a logic specific to paring messages from broker and IoT platform in the broker. Then, the broker encodes the message before forwarding it to the *Subscriber* ⓑ. At this point, the multi-party communication between the client, broker, and IoT platform is complete.

2.2 Motivation

We use a real-world bug to illustrate the motivation behind our approach and the limitations of existing work.

Real-world Example. The bug with a CVE number CVE-2024-42655 allows for 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 [21, 39]. Figure 2 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 `$SYS`, which represents the system topic in MQTT [21]. A subscriber connects to the broker using the username `sub` (Step ①). Subsequently, the subscriber subscribes to the topic pattern `+/+/+` (Step ②), where the `+` symbol represents a wildcard that can represent any topic [14]. After that, a publisher connects to the broker and prepares to publish messages (Step ③). The broker responds to the publisher with a `CONNACK` message to acknowledge the connection (Step ④). Meanwhile, the broker constructs a `PUBLISH` message containing information about all currently connected devices to the topic `$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 MQTT Fuzzers. The root cause of this bug is a failure to adhere to the protocol specification: *'The server must not match topic filters starting a wildcard character with topic names beginning with a $ character'*. Existing fuzzers [27, 29, 48, 50, 57, 60] 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, the unexpected behavior does not cause any crash in any participants, making it impossible for fuzzers relying on memory sanitizers to detect the bug [64,67]. 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.

3 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?** Not like existing fuzzers [29, 48] with only one sender, with two senders, after a test case is generated, it needs to be de-

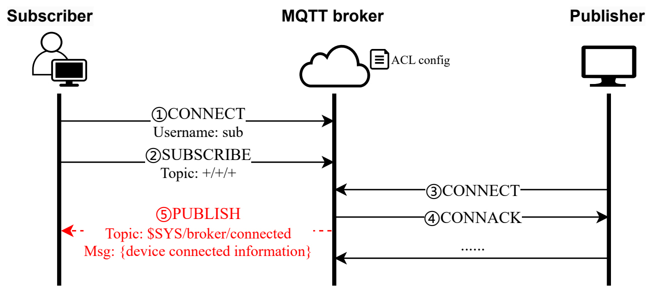


Figure 2: Steps to trigger the access control bypass bug (CVE-2024-42655) in NanoMQ.

termined which sender to 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 first extract six dependency rules from the messages involved in multi-party communication in the protocol specifications. Then, we design a dependency queue shared by two senders to coordinate their message sending. Lastly, we model the communication behavior of two senders in parallel using Petri net [49] to clarify the coordination process and demonstrate the effectiveness of the shared queue.

Black-box fuzzing does not require the instrumentation of source code and is well-suited for MQTT because most open-source brokers are implemented in different programming languages [59]. 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 the black-box fuzzer 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 [65]. Then, we use Q-learning [18] 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 [62]. 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 and effort, making it impractical for large-

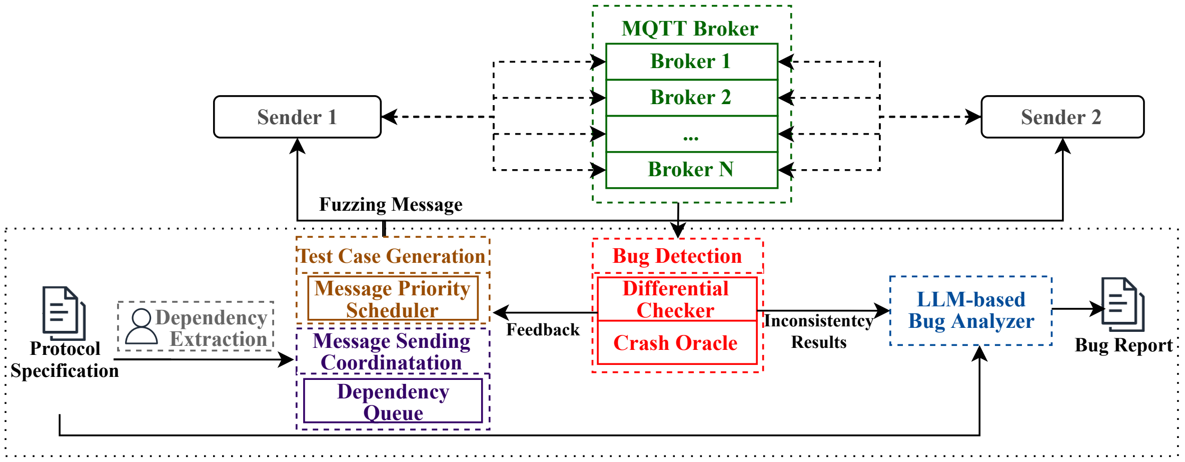


Figure 3: Fuzzing framework of MBFuzzer.

scale testing.

Solution: To address this challenge, we design a non-compliance bug analysis method based LLM, applying prompt chaining [17] and background-augmented prompting [55] strategy, to automatically validate and pinpoint the violation rules defined in specifications behind these bugs.

4 Design and Implementation

4.1 Overview

Figure 3 shows the overall framework of MBFuzzer, which compromises five components: message senders, test case generation, message sending coordination, bug detection, and LLM-based bug analyzer. 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 message they sent by the *Test Case Generation*. To test the relay code in brokers that communicate with other brokers, we set *Sender 2* as a bridged broker for receiving connection requests from brokers under test and sending messages once a connection is successfully established. Instead of using a real broker, we implement a Python script, which allows us to set a response cache so that it can return responses immediately after receiving the connection request to reduce processing delays and also enable us to send any format message.

The workflow of MBFuzzer could be divided into five steps: (1) We extract the dependency rules from MQTT protocol specifications. These rules 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.

4.2 Dependency Extraction

To efficiently coordinate message generation and sending among senders, we identify two types of dependencies influencing multi-party communication: message dependency and field dependency. *Message dependency* refers to pairs of message types that must be sent in a specific order to affect the communication process between senders and the broker. *Field dependency* refers to specific fields in dependent message pairs that the broker used for message matching and distribution. The values of these fields need to be matched across messages in multi-party communication. For example, in Figure 1, the `SUBSCRIBE` and `PUBLISH` messages 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 `SUBSCRIBE` and `PUBLISH` messages have a field dependency because it is used by the broker to match incoming messages with corresponding subscriptions.

We identify these dependencies and construct dependency rules following three steps: (1) We list all message types defined in the protocol specification and analyze their pairwise combinations to determine whether one message depends on another for multi-party communication. We name the first message in the pair *primary message*, which represents the initial message sent in the multi-party communication. The second is named *secondary message*, indicating it is sent subsequently. We consider a particular pair of messages to have a message dependency if their interaction facilitates the es-

No	Primary Message		Secondary Message		Affect Sender	Cache
	Type	Field	Type	Field		
1	CONNECT	will topic	SUBSCRIBE	topic	Another Sender	No
2	CONNECT[1]	client id	CONNECT	client id	Both Sender	Yes
3	SUBSCRIBE	topic	PUBLISH	topic	Both Sender	No
4	SUBSCRIBE	topic	UNSUBSCRIBE	topic	Same Sender	No
5	PUBLISH[2]	topic	SUBSCRIBE	topic	Both Sender	Yes
6	PUBLISH	topic alias	PUBLISH	topic alias	Same Sender	No

Table 1: Dependency rules for MQTT pub/sub message delivery model.

[1] The clean session flag should be 0.
[2] The retain flag should be 1.

tablishment or termination of multi-party communication. (2) We then analyze each field in the two messages to determine if the fields in the secondary message have a value dependency with the fields in the primary message. If there is a field in both messages used by the protocol for message matching or distribution, and their values must remain matched for message transmission, we consider it a field dependency for the current message pair. Additionally, we examine fields that trigger message caching in brokers, as the value of such fields can affect the order of multi-party communication. For instance, when a subscriber sends a subscription request after a message has already been published, the broker typically does not forward the published message to the subscriber because it has been discarded before the subscription request is processed. However, if a caching field is enabled, the broker will cache the published message and forward it to subsequent new subscribers. Therefore, if a caching field allows communication of a new combination of messages to affect multi-party communication, we consider it a field dependency as well. (3) Finally, we combine the identified message dependencies and field dependencies to construct dependency rules to coordinate senders for multi-party communication.

We manually analyzed the MQTT protocol specification and identified six dependency rules listed in Table 1. We use rule 1 as an example to explain the meaning of these rules and how they coordinate message sending across senders and affect multi-party message transmission. When a sender sends a `CONNECT` message containing a will message and then receives a `CONNACK` with a successful reason code, MBFuzzer copies the value of the `will topic` field to the `topic` field of the secondary message based on this rule and then saves the secondary message. Subsequently, MBFuzzer guides another sender to select the stored secondary `SUBSCRIBE` message according to the *Affect Sender* column, instantiates a subscription message, and sends it to the brokers. After that, if the sender that sent the primary `CONNECT` message accidentally disconnects, the broker will automatically send the will message to another sender according to the feature of the will message [12], completing the multi-party communication. Notably, if the sender that sent the primary message disconnects before another sender sends the `SUBSCRIBE`, the secondary message will expire and no longer be used to guide another sender to generate messages, as specified in the *Cache* column.

4.3 Test Case Generation

In the fuzzing loop, MBFuzzer generates test cases for each sender, based on the protocol state they are under, through three steps: determining the message type, instantiating the message, and mutating it. The test case generation process is illustrated in Algorithm 1.

Algorithm 1 Workflow of Test Case Generation

```
Input: ProtocolState, Sender, SharedDepQueue
Output: FuzzingMsg
    // Step1: Determine message type
 1: if SharedDepQueue.HASAVAILABLEDEP(Sender) then
 2:     DepMsg ← SharedDepQueue.GETAVAILABLEDEPMSG()
 3:     MsgType ← DepMsg.GETMSGTYPE()
 4: else
 5:     MsgType ← GETMSGBYPRIORITY(ProtocolState)
 6: end if
    // Step2: Instantiate message content
 7: MsgContent ← CREATEFUZZMSG(MsgType, DepMsg)
    // Step3: Mutate message content
 8: FuzzingMsg ← MUTATE(MsgContent)
```

Determine Message Type. MBFuzzer determines the generated message type in two ways. One is by selecting a message type from the dependency queue shared by the two senders (lines 1-3 of Algorithm 1). The shared dependency queue is responsible for coordinating the message sending between senders and records secondary messages defined by the dependency rules, which we will explain in detail in Section 4.4. The other is that when the dependency queue does not have a secondary message available for the sender, MBFuzzer randomly selects a message type with a higher priority based on the message priority scheduler (line 5 of Algorithm 1). To achieve this, we first 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 [18], a model-free reinforcement learning method, to design a message priority scheduler 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 [18], making it a natural fit for MQTT. Compared to other popular methods, such as Markov chains [35, 48], it performs better in this application, as demonstrated in Section 5.3.

With Q-learning, MBFuzzer dynamically learns a state-action policy that records the selection probability of each message type in different states. After sending the message, MBFuzzer receives a reward based on new inconsistency feedback and updates the policy. Following the policy update, MBFuzzer would 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 the hash value of the combination of the *fixed header* and *reason code* fields from response messages received from the broker. These fields contain the information of message type and handling state, typically reflecting the internal state of the server handling the current request [50]. Since MBFuzzer 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 in Q-learning.

Learning Policy. We use a two-dimensional Q-table to cumulate the experience learned by MBFuzzer, 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 the feedback and assign a constant reward value to the corresponding cell, as inconsistencies are usually considered potential bugs [67]. 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 [30], respectively, 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 [52] 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 $P(a_i|s)$ 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 [46].

Instantiate and Mutate Message. After determining the message type, MBFuzzer performs message instantiation based on the syntax-based generation strategy (line 7 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. MBFuzzer then generates random values based on the range for each field and assembles them in sequence to form a valid MQTT message. During message instantiation, if the current message type is determined from the dependency queue, MBFuzzer uses the field values recorded in the secondary message for instantiation instead of randomly generating them to satisfy the field dependency (line 7 of Algorithm 1). Lastly, MBFuzzer performs field-level mutation operations on messages generated by the syntax-based generation, as the mutation is crucial for detecting various bugs [34] (line 8 of Algorithm 1). The mutation process includes three types of mutation operations, such as bit-flipping, arithmetic operation, and payload operation (e.g., deleting, injecting, and replacing a random segment of bytes). Notably, fields with dependencies in messages are preserved without mutation to ensure that the field dependencies are not broken.

4.4 Message Sending Coordination

To collaborate message sending between two senders and facilitate multi-party communication, we design a shared dependency queue for them and apply the constructed dependency rules.

The dependency queue is responsible for managing all secondary messages defined by the dependency rules and coordinating message sending between senders by influencing the test case generation process. First, after each sender in

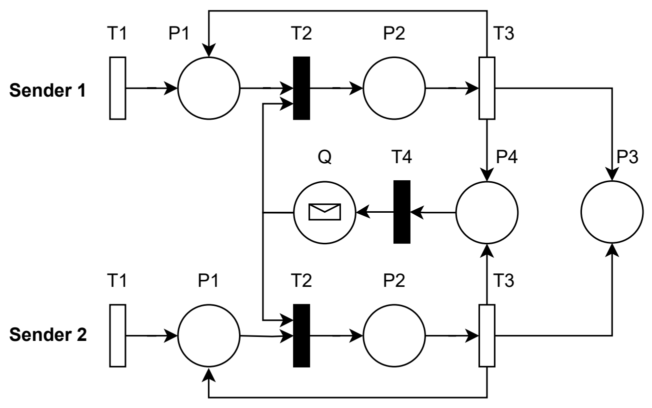


Figure 4: A Petri net model of the communication behavior of two parallel senders based on the shared dependency queue.

Figure 3 sends a request and receives a response, MBFuzzer examines the response code to determine whether the broker has successfully parsed the request. Next, based on the dependency rules, MBFuzzer checks whether the request has a dependent message, i.e., the secondary message. If a message dependency is found, MBFuzzer selects the corresponding secondary message, copies the dependent field values from the request into the secondary message, and stores it in the dependency queue. Subsequently, during message generation for each sender, MBFuzzer inspects the dependency queue for available secondary messages and sequentially selects a message from the queue. Using the selected message type and the associated field values recorded in it, MBFuzzer generates the next fuzzing message for the sender. Since certain secondary messages are only valid for multi-party communication when the sender of the primary message is connected, MBFuzzer prioritizes these messages in the queue to prevent them from expiring before being sent. Through the shared dependency queue, MBFuzzer can synchronize the message generation and sending of two concurrently running senders, facilitating multi-party communication.

To further clarify the process of collaborative multi-party message sending, we employ an extended Petri net [41, 49] to formally model the behavior of parallel senders based on the shared dependency queue. Petri net is an intuitive model widely used to describe and analyze the dynamic behavior of events in distributed systems [41]. Figure 4 illustrates the overall communication process, including two message senders, their runtime state represented by the circle, and the associated events represented by the box. During fuzzing, sender 1 and sender 2 start the process by establishing a connection via event *T1* and entering the waiting state for message generation and sending (*P1*), respectively. Subsequently, MBFuzzer generates a message for sender 1 in event *T2* to send, either by priority selection or by determining the message type from the shared queue *Q*. Sender 1 then transitions to state *P2*, awaiting and processing response messages in event *T3*. If any error occurs, sender 1 transitions to state *P3*, terminating the session. Otherwise, sender 1 returns to the waiting state for message generation and sending (*P1*), repeating the process. Meanwhile, MBFuzzer checks whether the message sent by the current sender has any dependency based on the rules and stores the corresponding dependent message into the shared queue *Q* during event *T4*. Sender 2 follows a similar process to sender 1, beginning with establishing a connection and cycling through states (*P1*, *P2*, *P3*) based on the events (*T1*, *T2*, *T3*). With the model, it is possible to visually demonstrate that the shared queue *Q* effectively coordinates message generation and sending across senders, enabling them to follow dependency rules and generate messages sequentially.

4.5 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 [44]. This approach compares the same functionality across implementations, serving as mutual reference oracles, and has proven effective in detecting non-compliance bugs [64, 67].

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`) and multiple forwarded publish messages. Furthermore, according to protocol specifications [13, 14], 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 [48] to avoid false positives. Redundant fields, such as the assigned client identifier in `CONNACK`, 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 during fuzzing 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 like connection refusal and timeouts. Notably, we enable ASAN [1] for open-source brokers implemented in

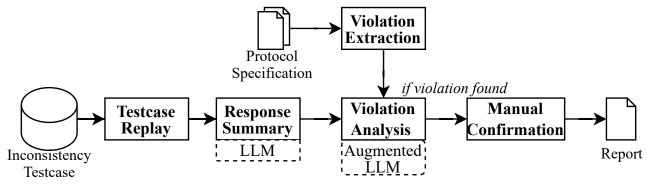


Figure 5: Framework of LLM-based non-compliance bug analyzer.

Prompt A

# Instruction
(1) The following is a sequence of MQTT requests sent by the sender.
{req_id} {message_name}: {message_description}
(2)The following is the sequence of responses from several brokers.
{broker_name}: {response_message_description}
(3) Now you are an expert in the MQTT protocol, please describe what the differences between these brokers in responding to handle such request. Please provide a detail description of the analysis.

Prompt B

# Instruction
(1) The following is protocol violations related to several MQTT messages. Message **{message_name}** has following relevant violation definitions: **{protocol violation details}.**
(2) The following is a sequence of MQTT requests sent by the sender.
{req_id} {message_name}: {message_description}
(3) The processing of the above requests by each broker has the following differences:
{response differences summarized by LLM}
(4) Now that you are an MQTT protocol expert, determine whether the client request violates the violations listed in (1) based on the response differences. Answer in such format: first give the analysis process, starting with '# Analysis Process.'. Then give the final conclusion, if there is an explicit violation, response with 'Protocol Violation Found!' or 'No violation!'

Figure 6: Prompt templates used in bug analyzer.

the C/C++ programming language to improve the efficiency of memory bug detection as a complementary measure.

4.6 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 [36, 64]. However, manual analysis is always daunting and labor-intensive. Inspired by recent advancements in LLM for root cause analysis [51] and their deep understanding of protocols [45], we design a non-compliance bug analyzer based on LLM to automatically verify and analyze the protocol violation behind inconsistencies.

We use the background-augmented prompting approach [55] to design prompts. Although the pre-training corpus of popular LLMs includes various protocol specifications [45], directly providing test case content to the LLM can lead to significant hallucination issues. Since MQTT specifications often use a standardized format to indicate violations, such as phrases like *'It is a Protocol Error'* and keywords like *MUST* and *MUST NOT*, we incorporate these elements into the prompts provided to the LLM, making the prompts more accurate and generalized [55]. To overcome context limitations, we use regular expressions to automatically extract violation descriptions from MQTT specifications, and store them in the format *'Protocol version | Message type'*. Only violations 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 [17], dividing the analysis process into two parts, as shown in Figure 5. 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 *Prompt A* in Figure 6 shows the prompt template for this step. The summary result is included in the prompts of violation analysis, with the augmented LLM determining if the current test case exhibits any violations as shown in *Prompt B*. If violations are found, we conduct further manual verification and report the findings to the developers. We used the OpenAI *gpt-4o-2024-05-13* model with its default temperature parameter for our analysis.

5 Evaluation

We wrote about 6k lines of Python code to implement MBFuzzer. To evaluate the effectiveness, we conducted experiments on real-world and production-level MQTT brokers. Specifically, our evaluation aims to answer the following questions:

- **RQ1.** Can MBFuzzer find bugs in real-world MQTT brokers and what about the security influence of these bugs? (Section 5.1)
- **RQ2.** Can MBFuzzer outperform other state-of-the-art MQTT fuzzers? (Section 5.2)
- **RQ3.** How does each component contribute to MBFuzzer? (Section 5.3)

Benchmark. We selected six mainstream MQTT brokers from the open-source community for production environments as the benchmark: EMQX [9], Mosquitto [4], NanoMQ [22], VerneMQ [23], HiveMQ [7] and FlashMQ [6] as shown in Table 2. These brokers were chosen based on four criteria: (1) High community impact [3] (as indicated by the *Stars* in Table 2), where the vendors of EMQX, NanoMQ, and HiveMQ are the members of OASIS MQTT Technical Committee [15]; (2) Active project maintenance, as indicated by the *Recent Commits* column, which shows the most recent year of developer commits as of August 2024); (3) Comprehensive support for all MQTT version features; (4) Support for MQTT bridge.
Environment. We ran all experiments through Docker images on a local machine with one Intel(R) Xeon(R) Gold 6226R CPU and 256 GB RAM, and a Ubuntu 20.04 LTS system.

Subject	Language	Stars	Recent Commits	Version
EMQX	Erlang	13.7k	4022	5.6.0
Mosquitto	C/C++	8.8k	39	2.0.18
NanoMQ	C	1.5k	806	0.21.10
VerneMQ	Erlang	3.2k	98	1.13
HiveMQ	Java	1.1k	151	4.24.0
FlashMQ	C/C++	0.18k	305	1.12.1

Table 2: MQTT brokers used for comparison. The *Stars* denotes the number of stars the GitHub repository acquires. The *Recent Commits* denotes the number of commits by developers in the past year.

5.1 Discovered Real-world Bugs

We evaluated the effectiveness of MBFuzzer in identifying bugs in MQTT brokers, as the primary purpose of fuzzing is to discover such issues [40].

Table 3 presents the comprehensive results of the bug discovery process across the six evaluated brokers. MBFuzzer discovered a total of 73 bugs, including 20 memory bugs (all confirmed, with 19 fixed) and 53 protocol non-compliance bugs (49 confirmed, and 42 were fixed). Notably, some non-compliance bugs are shared across different brokers. As a result, the total number of non-compliance bugs reported in the last row of Table 3 is not a simple sum of the bugs found in each individual broker. Table 4 summarizes the detailed results of 20 memory bugs, most of which could lead to denial of service, memory leaks, or even remote code execution in brokers. Table 10 presents the detailed findings of 53 protocol 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. Due to space constraints, the full details of Table 10 are provided in Appendix A.1.

Subject	Memory Bug			Non-Compliance Bug		
	Report	Confirmed	Fixed	Report	Confirmed	Fixed
EMQX	1	1	1	17	13	7
Mosquitto	1	1	1	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
Sum	20	20	19	53	49	42

Table 3: Detail bug discovery results of MBFuzzer.

We further analyzed these two categories of bugs and observed that they are predominantly associated with a few MQTT message types. For instance, 10 memory bugs were related to `PUBLISH` messages, while `SUBSCRIBE` and `CONNECT` each accounted for two. Similarly, 98% of the non-compliance bugs were concentrated in `CONNECT`, `SUBSCRIBE`, `UNSUBSCRIBE`, and `PUBLISH` message types. This concentration of bugs is unsurprising, as these four message types are central to pub/sub message delivery and involve more extensive code and parsing logic. For instance, the MQTT specification provides significantly more detail for `CONNECT`, `PUBLISH`, and `SUBSCRIBE` than for the other 13 message types. Correspondingly, their handler functions in Mosquitto also have the most extensive codebases, increasing the likelihood of bugs [53].

To better understand the security impact of bugs found by MBFuzzer, we discuss four representative confirmed bugs as case studies, one of which has been introduced in Motivation (Section 2.2).

Case study 2: Double free in Mosquitto. This vulnerability is classified as medium severity, is marked as M2 in Table 4, and is assigned CVE-2024-3935. Listing 1 in Appendix A.2 presents the simplified code where the vulnerability occurs. It occurs when Mosquitto parses published messages received from other brokers. Unlike parsing published messages from clients, the broker would perform topic remapping, converting the topic in the message to a new topic that conforms to the current network communication format (line 3). When the broker receives a publish message with an invalid format topic, the topic remapping function returns `INVALID_VALUE` and subsequently frees the topic (lines 12-13). However, the broker would free the message that contains the topic, leading to a double-free occurrence (lines 4-5). MBFuzzer discovered this bug using a novel multi-party fuzzing framework that enables the exploration of inter-broker interactions to find bugs. The vulnerability impacts all Mosquitto v2.x releases from 2020 to the present while existing MQTT fuzzers proposed during this period could not detect it. Bug M10 in Table 4 is similar to this vulnerability, which occurs in the code that parses published messages from brokers.

Case study 3: Use after free in NanoMQ. This vulnerability, labeled as bug M9 and assigned CVE-2024-42651 in Table 4, occurs during the pub/sub message transmission process in NanoMQ. Listing 2 in Appendix A.2 presents a simplified code snippet illustrating this issue. Specifically, the bug arises when NanoMQ processes subscribe messages containing multiple subscription topics (lines 3-16). 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 14), 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 approaches because of the message and field dependency. MBFuzzer effectively utilizes dependency rules to guide the generation of retained publish messages and corresponding subscribe messages, enabling the successful detection of this issue. Bug M5, M14, and M20 listed in Table 4 and CVE-2024-42655 in Table 10 are the same category of bugs that are triggered during the process of multi-party pub/sub message transmission in brokers.

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	Mosquitto	2.0.18	Double free in handling `PUBLISH` message from brokers	Denial of Service	✓	CVE-2024-3935
M3	FlashMQ	1.12.1	Crash in handling will `PUBLISH` message	Denial of Service	✓	Confirmed & Fixed
M4	FlashMQ	1.14.0	Assertion in saving retain `PUBLISH` message to the database	Denial of Service	✓	CVE-2024-42645
M5	FlashMQ	1.14.0	Assertion in forwarding will `PUBLISH` message	Denial of Service	✓	CVE-2024-42644
M6	NanoMQ	0.17.5	Segmentation Fault in handling `PUBLISH` message	Denial of Service	✓	CVE-2024-42650
M7	NanoMQ	0.17.5	Segmentation Fault in handling `CONNECT` message	Denial of Service	✓	CVE-2023-34488
M8	NanoMQ	0.17.5	Heap buffer overflow in handling `SUBSCRIBE` message	Remote code execution	✓	CVE-requested
M9	NanoMQ	0.17.9	Heap use after free in handling `SUBSCRIBE` message	Remote code execution	✓	CVE-2024-42651
M10	NanoMQ	0.17.9	Segmentation Fault in handling `PUBLISH` message	Denial of Service	✓	CVE-requested
M11	NanoMQ	0.21.10	Socket file description exhaustion caused by session keeping feature	Denial of Service	✓	Confirmed
M12	NanoMQ	0.21.10	Segmentation Fault in handling client requests	Denial of Service	✓	CVE-2024-42646
M13	NanoMQ	0.21.10	Stack buffer overflow in handling `PUBLISH` message	Remote code execution	✓	CVE-requested
M14	NanoMQ	0.22.1	Stack buffer overflow in handling `PUBLISH` message	Remote code execution	✓	CVE-2024-42647
M15	NanoMQ	0.22.1	Memory Leak after receiving message	Memory leakage	✓	CVE-2024-42649
M16	NanoMQ	0.22.1	Heap buffer overflow in handling `CONNECT` message	Remote code execution	✓	CVE-2024-42648
M17	NanoMQ	0.22.4	Heap use after free in handling `PUBLISH` message	Remote code execution	✓	CVE-requested
M18	NanoMQ	0.22.4	Memory leak in receiving message	Memory leakage	✓	CVE-requested
M19	NanoMQ	0.22.4	Segmentation fault in message	Denial of Service	✓	CVE-requested
M20	NanoMQ	0.22.4	Memcpy-param-overlap in handling `PUBLISH` message	Memory Corruption	✓	CVE-requested

Table 4: Memory bugs discovered in MQTT brokers by MBFuzzer.

Case study 4: Non-compliance Bug#N41 in NaonMQ. This non-compliance bug, labeled as bug N41 in Table 10 occurs when NanoMQ processes a publish message containing invalid property values. According to the protocol specification [14], *'A Control Packet which contains an Identifier which is not 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. MBFuzzer 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. Bug N19, N20, and N50 listed in Table 10 are similar non-compliance bugs that pose a security risk to other communication participants. 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.

5.2 Comparison to Existing MQTT Fuzzers

We compared MBFuzzer with three state-of-the-art protocol fuzzers (AFLNet [50], SGFuzz [29], and Fume [48]) using six benchmark brokers in Table 2. AFLNet and SGFuzz are grey-box state-guided protocol fuzzers, while Fume is a black-box MQTT fuzzer. The comparison focused on three key metrics: the number of messages sent, code coverage, and bug discovery effectiveness. We used the *gcov* tool to measure the branch coverage, which is applicable specifically to programs written in C/C++ program language. To mitigate randomness in fuzzing results, we repeated each fuzzing campaign five times, with each run lasting 24 hours.

Table 5 presents detailed fuzzing results of MBFuzzer and other SOTA fuzzers over 24 hours in terms of the number of messages sent, code coverage, and bug discovery. Table 6 summarizes the comparative results of memory bug discovery among these fuzzers, where the column *Bug ID* corresponds to the column *ID* in Table 4. Notably, the comparison was conducted using the broker versions listed in Table 2. As a result, the bug discovery results only cover specific versions of brokers. Additionally, Bug M4 and M5 affect the program versions listed in Table 2, and therefore they are included in the results.

Overall, MBFuzzer sent on average 36% more messages than AFLNet, but 3% and 92.3% fewer messages than SGFuzz and Fume, respectively. However, MBFuzzer outperformed the others in both bug discovery and code coverage. MBFuzzer achieved higher code coverage than AFLNet, SGFuzz, and Fume on the same benchmarks, with increases of 24%, 26%, and 5%, respectively. We employed the Mann-Whitney U-test [38] to calculate the p-value for code coverage comparisons between MBFuzzer and the other fuzzers, all of which were below 0.05, indicating statistical significance. In terms of bug discovery, MBFuzzer found 8 memory bugs and 53 non-compliance bugs across all categories. In contrast, AFLNet, SGFuzz, and Fume could only find 2, 2, and 6 memory bugs respectively, and none were able to detect non-compliance

Subject	AFLNet			SGFuzz			Fume			MBFuzzer		
	Message	Coverage	Bug	Message	Coverage	Bug	Message	Coverage	Bug	Message	Coverage	Bug
EMQX	-	-	-	-	-	-	32886k	-	1/0	1060k	-	**1/17**
Mosquitto	503k	2296	0/0	461k	2046	0/0	20465k	3306	0/0	1060k	**3724**	**1/11**
NanoMQ	609k	9747	1/0	-	-	-	6091k	11436	3/0	1060k	**11704**	**3/28**
VerneMQ	-	-	-	-	-	-	10213k	-	0/0	1060k	-	0/**16**
HiveMQ	-	-	-	-	-	-	2148k	-	0/0	1060k	-	0/**7**
FlashMQ	1220k	4130	1/0	1733k	4581	2/0	10849k	4263	2/0	1060k	**4648**	**3/23**
			2/0			2/0			6/0			**8/53**

Table 5: Detailed fuzzing results over 24 hours of MBFuzzer 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.

Bug ID	AFLNet	SGFuzz	Fume	MBFuzzer
M1	×	×	✓	✓
M2	×	×	×	✓
M3	✓	✓	✓	✓
M4	×	✓	✓	✓
M5	×	×	×	✓
M11	×	×	✓	✓
M12	×	×	✓	✓
M13	✓	×	✓	✓

Table 6: Comparison of memory bug discovery results by MBFuzzer and SOTA fuzzers after 24 hours.

bugs.

We further investigated why MBFuzzer performs better than other fuzzers. First, MBFuzzer discovered numerous non-compliance bugs through differential testing, whereas existing fuzzers only support the detection of memory bugs. Moreover, most of the inconsistency results in the experiments are concentrated in four message types, `CONNECT`, `SUBSCRIBE`, `UNSUBSCRIBE`, and `PUBLISH`. MBFuzzer 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. Since these messages typically involve the most parsing logic in brokers and specifications, the scheduler also helps achieve higher code coverage. Second, using the proposed new multi-party fuzzing framework and dependency rules, MBFuzzer successfully detects vulnerabilities in the inter-broker interaction code (Bug M2 in Table 6) and pub/sub message transmission (Bug M5 in Table 6), while other fuzzers fail to detect. It also demonstrates the effectiveness of our approach in guiding the exploration of multi-party communication input space. Lastly, MBFuzzer incurs some performance overhead because it must collect and analyze each response and forwarded message from six brokers simultaneously, leading to fewer fuzzing messages being sent. The performance overhead is justifiable because differential testing allows MBFuzzer to uncover more non-compliance bugs and dynamically optimize the message generation strategy, effectively offsetting this performance trade-off.

Fume sets an even message generation probability for each message type manually and cannot dynamically adjust the generation based on fuzzing runtime information. Additionally, Fume uses response contents and approximate log contents as the fuzzing feedback, which may lack a correlation with bugs. Therefore, it reduces the performance of Fume for exploring brokers and detecting bugs in them. SGFuzz and AFLNet lack awareness of the MQTT message format, making it easy to break message formats and dependencies during mutations, which results in many messages being dropped by the broker. In addition, they perform poorly in Mosquitto and NanoMQ due to slower speeds. Lastly, SGFuzz and AFLNet do not support fuzzing brokers implemented in Erlang and Java, such as EMQX, VerneMQ, and HiveMQ. Furthermore, the *netdriver* used by SGFuzz does not support the fork system calls [29] as well and therefore does not support NanoMQ.

5.3 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. Specifically, we designed three variants: MBFuzzer$_{!d}$, MBFuzzer$_{!r}$, and MBFuzzer$_{!m}$. In MBFuzzer$_{!d}$, the capability of dependency rules is disabled to evaluate their effectiveness in facilitating the exploration of the input space for multi-party communication. In MBFuzzer$_{!r}$, 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 MBFuzzer$_{!m}$, which replaces Q-learning with a rule-based approach. This approach uses the widely used Markov chain [35, 48] to model message generation, assigning selection probabilities based on the percentage of unique inconsistent responses each message triggers [63]. The ablation study was conducted on the same three metrics using the same experimental configuration and program version detailed in Section 5.2.

Table 7 presents the results of the ablation study on MBFuzzer, focusing on the number of messages sent, code coverage, and bug discovery. Table 8 details the findings of bugs in the ablation study, containing all memory bugs and non-compliance bugs that are assigned CVE numbers.

The results demonstrate the observable impact of each

Subject	MBFuzzer$_{!d}$			MBFuzzer$_{!r}$			MBFuzzer$_{!m}$			MBFuzzer		
	Message	Coverage	Bug	Message	Coverage	Bug	Message	Coverage	Bug	Message	Coverage	Bug
EMQX	1023k	-	**1**/**17**	1035k	-	0/**17**	991K	-	**1**/16	1060k	-	**1**/**17**
Mosquitto	1023k	3327	**1**/10	1035k	3209	**1**/10	991K	3482	**1**/10	1060k	**3724**	**1**/**11**
NanoMQ	1023k	11695	**3**/26	1035k	11414	2/25	991K	11550	**3**/24	1060k	**11704**	**3**/**28**
VerneMQ	1023k	-	0/**16**	1035k	-	0/14	991K	-	0/15	1060k	-	0/**16**
HiveMQ	1023k	-	0/6	1035k	-	0/6	991K	-	0/**7**	1060k	-	0/**7**
FlashMQ	1023k	4342	2/20	1035k	4582	**3**/20	991K	4382	2/19	1060k	**4648**	**3**/**23**
			7/51			6/48			7/49			**8**/**53**

Table 7: Results of ablation study over 24 hours of MBFuzzer. 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.

Bug ID	MBFuzzer$_{!d}$	MBFuzzer$_{!r}$	MBFuzzer$_{!m}$	MBFuzzer
M1	✓	×	✓	✓
M2	✓	✓	✓	✓
M3	✓	✓	✓	✓
M4	✓	✓	✓	✓
M5	×	✓	×	✓
M11	✓	×	✓	✓
M12	✓	✓	✓	✓
M13	✓	✓	✓	✓
N29	×	✓	×	✓

Table 8: Comparison of bug discovery results for MBFuzzer in ablation study after 24 hours.

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

Table 9: 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.

component on code coverage and bug discovery performance. Specifically, disabling dependency rules (MBFuzzer$_{!d}$) reduces the fuzzer's ability to explore the specific execution paths involving multi-party communication, leading to an average decrease of 3.6% in code coverage. Moreover, without dependency guidance, MBFuzzer$_{!d}$ struggles to detect bugs such as M5 and N29, which are related to the pub/sub message transmission, as shown in Table 8. Similarly, replacing the message priority scheduler with a random message selection strategy (MBFuzzer$_{!r}$) 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 codes are concentrated in a small subset of message contexts. Thus, MBFuzzer$_{!r}$ fails to detect bugs like M1 and M11, while code coverage declines by an average of 4.5%. For instance, detecting bug M11 requires sending a large number of `CONNECT` messages with varying client identifiers for persistent sessions within a short time, which is challenging to achieve using a random strategy. Lastly, while MBFuzzer$_{!m}$ 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 messages sent by MBFuzzer$_{!m}$ were `CONNECT` and `PUBLISH`, limiting its exploration of the other types, such as `SUBSCRIBE`. This imbalance results in an average 3.4% reduction in code coverage and prevents the detection of bugs associated with other message types.

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 randomly selected 500 inconsistency test cases that lead to inconsistent behaviors of different broker implementations as the evaluation dataset. We then invited two researchers to manually analyze the results generated by the module, referencing the protocol specifications. After completing the analysis process, 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 [42]. This resulted in duplicate results in the LLM-based bug analyzer.

Table 9 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. Among the 340 results correctly identified by the LLM as containing violations, we found 53 unique violations (i.e., non-compliance bugs in Table 10). 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 10 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.

Overall, 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.

6 Discussion and Limitations

MBFuzzer has successfully identified many bugs on six different open-source brokers. However, it still has certain limitations. In this section, we discuss these limitations and solutions for future improvements.

Manual Effort. In our prototype, MBFuzzer requires manual effort to extract the dependency rules for multi-party communication from the protocol specification. As experts familiar with MQTT, we spent approximately one hour following the extraction method to build rules. Non-experts might need more time to understand the specification and derive rules. To further reduce required manual efforts, we believe that LLMs, with their advanced ability to deeply understand protocols [55], could accelerate and even automate this process. Additionally, the dependency rules we identified may only apply to the MQTT protocol. Nevertheless, we believe that our dependency extraction approach can be applied to other multi-party communication protocols, such as DDS [47], OPC UA [16], and STOMP [20], as well as message middleware systems like RabbitMQ [19].

Efficiency. The sender 2 in the fuzzing framework that acts as a broker starts sending fuzzing messages only after the connection from the brokers under test is established. If the connection is accidentally disconnected, it usually takes one to two seconds for the broker to reconnect automatically. During this period, the sender remains idle, which may slightly degrade the fuzzing performance. However, we can use the approach of SnapFuzz [28], which implements adaptive timeouts in the network software by rewriting the code, which can be used to eliminate delays during reconnection of bridge interrupts and solve this problem.

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.

Differential Checker. Due to the inherent limitations of differential testing [64], MBFuzzer 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.

7 Related Work

Protocol Fuzzing. There have been several fuzzing techniques have been proposed and applied for fuzzing MQTT protocol implementations. MultiFuzz [60] is a grey-box multi-party fuzzer for brokers, but it simply adds client socket connections. Fume [48] uses Markov chains to model the process of message generation for fuzzing MQTT brokers. SHADOWFUZZER [58] proposes a novel fuzzing approach to fuzz MQTT clients via brokers. SGANFuzz [57] uses generative adversarial networks to guide test cases for brokers. In addition, there are generic protocol fuzzers that can also be used to fuzz brokers. AFLNet [50] is the first grey-box protocol fuzzer that uses the response code as the state feedback. SGFuzz [29] recognizes the enum-type variables in codes as state variables automatically to construct the state feedback. ChatAFL [45] introduces LLM to process Request for Comments (RFCs) and generate fuzzing message sequences. mGPTFuzz [43] leverage 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 MQTT brokers, they cannot fully explore the input space of multi-party communications, limiting bug discovery capabilities. MBFuzzer 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 [67] applies differential testing to fuzz the TCP stack to find semantic bugs. ResolFuzz [32] and ResolverFuzz [61] use differential testing to test DNS resolvers to find non-crash vulnerabilities in them. Moreover, TLS-DeepDiffer [64] detects logic flaws throughout TLS interactions based on message tuple and differential testing. However, existing methods are not directly applicable to MQTT. 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. MBFuzzer designs the first differential checker for MQTT and introduces LLM to bug analysis to automatically pinpoint the violations behind inconsistency results.

Formal Verification. Formal verification is the use of formal mathematical methods to verify the correctness of a software implementation and is currently also being used in the detection of logical flaws in MQTT brokers. MPInspector [56] uses active learning to automatically infer the state model of brokers and then uses formal verification to identify violations related to the secrecy and authentication properties. MQTTactic [59] identifies flaws in authorization-related properties based on static analysis and model check. These methods typically focus on logic bugs in specific security properties. MBFuzzer applies differential testing to detect non-compliance bugs in all messages and fields during the parsing of messages by brokers.

8 Conclusion

In this paper, we proposed MBFuzzer, a novel multi-party black-box fuzzer for brokers to detect both memory and non-compliance bugs. MBFuzzer models the communication behavior of multiple senders using extended Petri net and designs six dependency rules to coordinate the message sending between different senders. MBFuzzer then uses a message priority schedule approach based on inconsistency feedback to adjust the priority of messages in different states dynamically to improve the efficiency of bug discovery. Subsequently, MBFuzzer 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 MBFuzzer is effective in finding bugs in MQTT brokers, with 73 bugs discovered, of which 11 CVEs have been assigned. It could also achieve higher code coverage and vulnerability discovery than state-of-the-art fuzzers.

Acknowledgments

We thank anonymous reviewers for their comprehensive feedback. This work is supported by the National Natural Science Foundation of China (No. 62372268), the Major Scientific and Technological Innovation Projects of Shandong Province, China (No. 2024CXGC010114), and the Shandong Provincial Natural Science Foundation, China (No. ZR2022LZH013, No. ZR2021LZH007). It is also supported in part by the Zhejiang Provincial Natural Science Foundation of China (No. LY22F020022) and the National Natural Science Foundation of China (No. 61902098).

Ethics Considerations

Adhering to ethical guidelines [37], we carefully design our measurements and report our findings to relevant vendors. The primary ethical concern of this study is avoiding the vulnerabilities discovered from being exploited in the real world. Therefore, we deployed brokers and conducted experiments in local environments to avoid impacting brokers running in production environments. We also reported all of our findings to the affected vendors, most of which were confirmed and fixed, and the few remaining unfixed bugs that do not directly pose a security risk to the brokers are planned to be solved by the developers in the future.

Open Science

This study follows the principles of open science. To facilitate further research, we have made MBFuzzer publicly available on [25].

References

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

[2] Aws iot core. https://aws.amazon.com/iot-core/. Accessed on 2024-8-8.

[3] Comparison of open source mqtt brokers 2024. https://www.emqx.com/en/blog/a-comprehensive-comparison-of-open-source-mqtt-brokers-in-2023. Accessed on 2024-8-8.

[4] Eclipse mosquitto - an open source mqtt broker. https://github.com/eclipse/mosquitto. Accessed on 2024-8-8.

[5] Emqx: The #1 mqtt platform for iot, iiot and connected cars. https://www.emqx.com/. Accessed on 2024-8-8.

[6] Flashmq – high speed mqtt. https://www.flashmq.org/. Accessed on 2024-8-8.

[7] Hivemq- the most trusted mqtt platform to transform your business. https://www.hivemq.com/. Accessed on 2024-8-8.

[8] Introduction to mqtt publish-subscribe pattern. https://www.emqx.com/en/blog/mqtt-5-introduction-to-publish-subscribe-model. Accessed on 2024-8-9.

[9] The most scalable open-source mqtt broker for iot, iiot, and connected vehicles. https://github.com/emqx/emqx. Accessed on 2024-8-8.

[10] Mqtt bridge. https://docs.vernemq.com/configuring-vernemq/bridge. Accessed on 2024-8-8.

[11] Mqtt integrations. configuring mosquitto bridge. https://cedalo.com/blog/mosquitto-bridge-configuration/l. Accessed on 2024-8-9.

[12] Mqtt specification 5.0. will message. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html#_Toc479576982. Accessed on 2024-8-25.

[13] 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.

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

[15] Oasis message queuing telemetry transport (mqtt) tc. https://groups.oasis-open.org/communities/tc-community-home2?CommunityKey=99c86e3a-593c-4448-b7c5-018dc7d3f2f6. Accessed on 2024-8-14.

[16] Opc unified architecture (ua). https://opcfoundation.org/about/opc-technologies/opc-ua/. Accessed on 2024-12-20.

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

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

[19] Rabbitmq one broker to queue them all. https://www.rabbitmq.com/. Accessed on 2024-12-20.

[20] Stomp protocol specification, version 1.2. https://stomp.github.io/stomp-specification-1.2.html. Accessed on 2024-12-20.

[21] 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.

[22] An ultra-lightweight and blazing-fast messaging broker/bus for iot edge & sdv. https://github.com/nanomq/nanomq. Accessed on 2024-8-8.

[23] Vernemq: A distributed mqtt message broker based on erlang/otp. https://github.com/vernemq/vernemq. Accessed on 2024-8-8.

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

[25] Zenodo MBFuzzer. https://doi.org/10.5281/zenodo.14710570. 2025.

[26] Zoomey: "moqsuitto". https://www.zoomeye.org/searchResult?q=app%3A%22Mosquitto%22. Accessed on 2024-8-8.

[27] 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.

[28] Anastasios Andronidis and Cristian Cadar. Snapfuzz: high-throughput fuzzing of network applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 340–351, New York, NY, USA, 2022. Association for Computing Machinery.

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

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

[31] Francesco Buccafurri, Vincenzo De Angelis, and Sara Lazzaro. Mqtt-a: A broker-bridging p2p architecture to achieve anonymity in mqtt. *IEEE Internet of Things Journal*, 10(17):15443–15463, 2023.

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

[33] Pietro Ferrara, Amit Kr Mandal, Agostino Cortesi, and Fausto Spoto. Static analysis for discovering iot vulnerabilities. *International Journal on Software Tools for Technology Transfer*, 23:71–88, 2021.

[34] 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.

[35] 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.

[36] 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.

[37] Erin Kenneally and David Dittrich. The menlo report: Ethical principles guiding information and communication technology research. *Available at SSRN 2445102*, 2012.

[38] 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.

[39] 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.

[40] 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.

[41] Guanjun Liu, Mengchu Zhou, and Changjun Jiang. Petri net models and collaborativeness for parallel processes with resource sharing and message passing. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(4):1–20, 2017.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] Gerardo Pardo-Castellote. Omg data-distribution service: Architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 200–206. IEEE, 2003.

[48] 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.

[49] James L Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977.

[50] 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.

[51] 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.

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

[53] A.S. Tanenbaum. *Computer Networks*. Computer Networks. Prentice Hall PTR, 2003.

[54] Chenxi Wang, Antonio Carzaniga, David Evans, and Alexander L Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *Proceedings of the 35th annual hawaii international conference on system sciences*, pages 3940–3947. IEEE, 2002.

[55] 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.

[56] Qinying Wang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Yuhong Kan, Zhaowei Lin, Changting Lin, Shuiguang Deng, Alex X Liu, et al. Mpinspector: A systematic and automatic approach for evaluating the security of {IoT} messaging protocols. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4205–4222, 2021.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

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

[62] 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.

[63] 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.

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

[65] 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.

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

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

```
int handle__publish(...) {
  ...
  rc = bridge__remap_topic_in(context, &msg->topic);
  if(rc){
    db__msg_store_free(msg);  // double free
    return rc;
  }
}
int bridge__remap_topic_in(..., char **topic) {
  rc = mosquitto_topic_matches_sub(..., *topic);
  if(rc){
    mosquitto__free(*topic);  // first free
    return rc;  // rc == INVALID_VALUE 3
  }
}
```

Listing 1: Simplified code of case study 2.

```
int sub_ctx_handle(nano_work *w) {
  nng_msg **retain = w->msg_ret;
  while (tn) {
    topic_str = tn->topic.body;
    topic_exist = dbhash_check_topic(w->pid.id, topic_str
        );
    uint8_t rh = tn->retain_handling;
    if (rh == 0 || (rh == 1 && !topic_exist))
      retain = dbtree_find_retain(w->db_ret, topic_str);
          // malloc retain
     w->msg_ret = (w->msg_ret == NULL) ? retain : w->
        msg_ret;
    for (i = 0; i < c_size(retain) && w->msg_ret !=
        retain; i++) {  // UAF here
      cvector_push_back(w->msg_ret, retain[i]);
    }
    if (retain != w->msg_ret)
      c_free(retain); // free retain
    tn = tn->next;
  }
}
```

Listing 2: Simplified code of case study 3.

A Appendix

A.1 Details of Non-compliance Bugs Found by MBFuzzer

Table 10 shows the detailed results of 53 non-compliance bugs discovered by MBFuzzer, where 49 of them have been confirmed and 42 have been fixed. Most developers promise to fix the remaining unfixed bugs in the future. Due to space constraints, we use aliases in *Bug Description*, most of which are formatted by *"protocol version"-"mandatory or customized rule in specifications"*. The full description corresponding to each bug can be found in Table 11.

A.2 Source Code of Case Study

ID	Bug Description	EMQX	Mosquitto	NanoMQ	VerneMQ	HiveMQ	FlashMQ	Count
N1	MQTTv3.1.1-[MQTT-2.3.1-1]	✓	×	✓†‡	✓†‡	✓†‡	×	2
N2	MQTTv3.1.1-[MQTT-3.1.2-15]	×	×	×	×	×	✓†‡	8
N3	MQTTv3.1.1-[MQTT-3.1.2-19]	✓†‡	✓†	×	✓†‡	✓†	×	8
N4	MQTTv3.1.1-[MQTT-3.1.2-22]	✓†‡	×	✓†	×	×	×	18
N5	MQTTv3.1.1-[MQTT-3.1.2-9]	×	×	×	×	✓†‡	×	2
N6	MQTTv3.1.1-[MQTT-3.1.3-11]	×	×	×	✓†‡	✓†‡	×	11
N7	MQTTv3.1.1-[MQTT-3.1.3-4]	×	×	×	✓†‡	×	×	1
N8	MQTTv3.1.1-[MQTT-3.1.3-8]	×	×	✓†	×	×	×	9
N9	MQTTv5.0-[MQTT-3.1.0-1]	×	×	×	×	×	✓†‡	1
N10	MQTTv5.0-[MQTT-3.1.2-11]	✓†‡	×	✓†‡	×	×	✓†‡	6
N11	MQTTv5.0-[MQTT-3.1.2-13]	✓†‡	×	✓†‡	×	×	✓†‡	8
N12	MQTTv5.0-[MQTT-3.1.2-3]	×	×	✓†‡	×	×	×	2
N13	MQTTv5.0-[MQTT-3.1.2-9]	×	×	×	×	×	✓†‡	2
N14	MQTTv5.0-[MQTT-3.1.3-12]	×	×	×	✓†‡	✓	×†	1
N15	MQTTv5.0-[MQTT-3.10.3-1]	✓†	×	×	✓†‡	×	×	5
N16	MQTTv5.0-[MQTT-3.10.3-2]	×	✓	×	×	×	×	1
N17	MQTTv5.0-[MQTT-3.2.0-2]	×	×	✓†‡	×	×	×	1
N18	MQTTv5.0-[MQTT-3.3.1-2]	×	×	✓†‡	×	×	×	30
N19	MQTTv5.0-[MQTT-3.3.2-1]	×	×	×	✓†‡	×	×	47
N20	MQTTv5.0-[MQTT-3.3.2-14]	×	✓†	✓†	✓†‡	×	×	4
N21	MQTTv5.0-[MQTT-3.3.2-19]	×	×	✓†‡	×	×	×	3
N22	MQTTv5.0-[MQTT-3.3.2-9]	×	×	✓†‡	×	×	×	3
N23	MQTTv5.0-[MQTT-3.3.4-6]	✓†	✓†	✓†	✓†‡	×	✓†‡	82
N24	MQTTv5.0-[MQTT-3.4.2-1]	✓†	×	✓†	×	×	✓†	5
N25	MQTTv5.0-[MQTT-3.6.1-1]	×	×	✓†‡	×	×	✓†‡	7
N26	MQTTv5.0-[MQTT-3.8.3-1]	✓†	×	×	✓†‡	×	×	2
N27	MQTTv5.0-[MQTT-3.8.3-2]	×	✓	×	×	×	×	6
N28	MQTTv5.0-[MQTT-3.8.3-4]	×	✓†‡	×	✓†‡	×	✓†‡	1
N29	MQTTv5.0-[MQTT-4.7.2-1]	×	×	✓†[1]	×	×	✓†‡	1
N30	MQTTv5.0-[MQTT-4.8.2-2]	×	✓†‡	✓†‡	✓†‡	×	✓†‡	1
N31	MQTTv5.0-[Authentication Data rule]	✓†	✓†	✓†	✓†‡	×	✓†‡	2
N32	MQTTv5.0-[Authentication Method rule]	×	×	✓†	×	×	×	1
N33	MQTTv5.0-[Content Type rule]	×	×	×	×	×	✓†‡	2
N34	MQTTv5.0-[Correlation Data rule]	✓	×	×	×	×	×	2
N35	MQTTv5.0-[Maximum QoS rule]	✓†	×	×	×	×	×	1
N36	MQTTv5.0-[Message Expiry Interval rule]	×	×	✓†	×	×	✓†‡	5
N37	MQTTv5.0-[Payload Format Indicator rule1]	×	×	✓†‡	×	×	✓†‡	6
N38	MQTTv5.0-[Payload Format Indicator rule2]	✓	✓†	×	✓†‡	✓†	✓†‡	1
N39	MQTTv5.0-[Payload Format Indicator rule3]	×	✓†	✓†‡	✓†‡	✓†	✓†‡	1
N40	MQTTv5.0-[Payload Format Indicator rule4]	✓†‡	×	×	×	×	×	3
N41	MQTTv5.0-[Property rule]	×	×	✓†	×	×	×	2
N42	MQTTv5.0-[Receive Maximum rule]	×	×	×	×	×	✓†‡	24
N43	MQTTv5.0-[Request Problem Information rule]	×	×	✓†	×	×	✓†‡	3
N44	MQTTv5.0-[Request Response Information rule1]	×	×	✓†	×	×	×	5
N45	MQTTv5.0-[Request Response Information rule2]	✓†‡	×	×	×	×	×	2
N46	MQTTv5.0-[Response Topic rule1]	×	×	×	×	×	✓†‡	4
N47	MQTTv5.0-[Response Topic rule2]	✓	✓†‡	✓†	×	×	✓†‡	1
N48	MQTTv5.0-[Subscription Identifier rule]	×	×	✓†‡	×	×	×	3
N49	MQTTv5.0-[Topic Alias rule]	✓†‡	×	×	×	×	✓†‡	9
N50	MQTTv5.0-[User Property rule1]	×	×	✓†	×	×	×	15
N51	MQTTv5.0-[Will QoS rule]	×	×	×	✓	×	×	1
N52	MQTTv5.0-[User Property rule2]	×	×	×	×	×	✓†‡	1
N53	MQTTv5.0-[Will Topic rule]	×	×	✓†‡	×	×	×	2

Table 10: Non-compliance bugs in MQTT brokers discovered by MBFuzzer. The *Bug Description* indicates the description of the bug in the protocol specifications or the bug itself. The ✓ indicates that the current broker is affected by this bug, and the × indicates that it is not. The † indicates that the developer has confirmed the bug, and ‡ indicates that the bug has been fixed. The *Count* column indicates the number of test cases that triggered that bug in the ablation study.

[1] CVE-2024-42655.

ID	Detailed Bug Description
N1	SUBSCRIBE, UNSUBSCRIBE, and PUBLISH (in cases where QoS >0) Control Packets MUST contain a non-zero 16-bit Packet Identifier.
N2	If the Will Flag is set to 0, then the Will Retain Flag MUST be set to 0.
N3	If the User Name Flag is set to 1, a user name MUST be present in the payload.
N4	If the User Name Flag is set to 0, the Password Flag MUST be set to 0.
N5	If the Will Flag is set to 1, the Will QoS and Will Retain fields in the Connect Flags will be used by the Server, and the Will Topic and Will Message fields MUST be present in the payload.
N6	The User Name MUST be a UTF-8 encoded string
N7	The ClientId MUST be a UTF-8 encoded string
N8	If the Client supplies a zero-byte ClientId with CleanSession set to 0, the Server MUST respond to the CONNECT Packet with a CONNACK return code 0x02 (Identifier rejected) and then close the Network Connection.
N9	After a Network Connection is established by a Client to a Server, the first packet sent from the Client to the Server MUST be a CONNECT packet.
N10	If the Will Flag is set to 0, then the Will QoS MUST be set to 0 (0x00).
N11	If the Will Flag is set to 0, then Will Retain MUST be set to 0.
N12	The Server MUST validate that the reserved flag in the CONNECT packet is set to 0.
N13	If the Will Flag is set to 1, the Will QoS and Will Retain fields in the Connect Flags will be used by the Server, and the Will Properties, Will Topic, and Will Message fields MUST be present in the Payload.
N14	If the User Name Flag is set to 1, the User Name is the next field in the Payload. The User Name MUST be a UTF-8 Encoded String.
N15	The Topic Filters in an UNSUBSCRIBE packet MUST be UTF-8 Encoded Strings.
N16	The Payload of an UNSUBSCRIBE packet MUST contain at least one Topic Filter.
N17	The Server MUST NOT send more than one CONNACK in a Network Connection.
N18	The DUP flag MUST be set to 0 for all QoS 0 messages.
N19	The Topic Name MUST be present as the first field in the PUBLISH packet Variable Header. It MUST be a UTF-8 Encoded String.
N20	The Response Topic MUST NOT contain wildcard characters.
N21	The Content Type MUST be a UTF-8 Encoded String.
N22	A Client MUST NOT send a PUBLISH packet with a Topic Alias greater than the Topic Alias Maximum value returned by the Server in the CONNACK packet.
N23	A PUBLISH packet sent from a Client to a Server MUST NOT contain a Subscription Identifier.
N24	The Client or Server sending the PUBACK packet MUST use one of the PUBACK Reason Codes.
N25	Bits 3,2,1 and 0 of the Fixed Header in the PUBREL packet are reserved and MUST be set to 0,0,1 and 0 respectively. The Server MUST treat any other value as malformed and close the Network Connection.
N26	The Topic Filters MUST be a UTF-8 Encoded String.
N27	The Payload MUST contain at least one Topic Filter and Subscription Options pair.
N28	It is a Protocol Error to set the No Local bit to 1 on a Shared Subscription.
N29	The Server MUST NOT match Topic Filters starting with a wildcard character (# or +) with Topic Names beginning with a $ character.
N30	The ShareName MUST NOT contain the characters "/", "+" or "#", but MUST be followed by a "/" character. This "/" character MUST be followed by a Topic Filter.
N31	It is a Protocol Error to include Authentication Data if there is no Authentication Method.
N32	The authentication method's name is a UTF-8 encoded string.
N33	It is a Protocol Error to include the Content Type more than once.
N34	It is a Protocol Error to include Correlation Data more than once.
N35	It is a Protocol Error if the Maximum QoS field has the value 3.
N36	It is a Protocol Error to include the Message Expiry Interval more than once.
N37	The Payload Format Indicator value is either 0 or 1.
N38	The Payload Format Indicator value 1 (0x01) means the Will Message is UTF-8 encoded.
N39	The Payload Format Indicator value 1 (0x01) means the Payload is UTF-8 encoded.
N40	It is a Protocol Error to include the Payload Format Indicator more than once.
N41	A Control Packet which contains an Identifier that is not valid for its packet type, or contains a value not of the specified data type, is a Malformed Packet.
N42	It is a Protocol Error to include the Receive Maximum value more than once or for it to have the value 0
N43	It is a Protocol Error to include Request Problem Information more than once or to have a value other than 0 or 1.
N44	It is a Protocol Error that Request Response Information has a value other than 0 or 1.
N45	It is a Protocol Error to include the Request Response Information more than once.
N46	It is a Protocol Error to include the Response Topic more than once.
N47	The Response Topic should not be NULL.
N48	The Subscription Identifier can have a value of 1 to 268,435,455.
N49	It is a Protocol Error to include the Topic Alias value more than once.
N50	The User Property is a UTF-8 String Pair.
N51	The value 3(0x03) of Will QoS is a Malformed Packet.
N52	The length of Properties in the UNSUBSCRIBE packet Variable Header encoded as a Variable Byte Integer.
N53	The Will Topic can not be empty.

Table 11: Detail bug description of non-compliance bugs.