LLM-Assisted Model-Based Fuzzing of Protocol Implementations

Changze Huang
hcz@stu.pku.edu.cn
Key Lab of HCST (PKU), MOE;
SCS, Peking University
Beijing, China

Di Wang
wangdi95@pku.edu.cn
Key Lab of HCST (PKU), MOE;
SCS, Peking University
Beijing, China

Zhi Quan Zhou
george.zhou@nio.com
NIO Inc.
Shanghai, China

Abstract

Testing network protocol implementations is critical for ensuring the reliability, security, and interoperability of distributed systems. Faults in protocol behavior can lead to vulnerabilities and system failures, especially in real-time and mission-critical applications. A common approach to protocol testing involves constructing Markovian models that capture the state transitions and expected behaviors of the protocol. However, building such models typically requires significant domain expertise and manual effort, making the process time-consuming and difficult to scale across diverse protocols and implementations.

We propose a novel method that leverages large language models (LLMs) to automatically generate sequences for testing network protocol implementations. Our approach begins by defining the full set of possible protocol states, from which the LLM selects a subset to model the target implementation. Using this state-based model, we prompt the LLM to generate code that produces sequences of states. This program serves as a protocol-specific sequences generator. The sequences generator then generates test inputs to call the protocol implementation under various conditions. We evaluated our approach on three widely used network protocol implementations and successfully identified 12 previously unknown vulnerabilities. We have reported them to the respective developers for confirmation. This demonstrates the practical effectiveness of our LLM-assisted fuzzing framework in uncovering real-world security issues.

Keywords

Protocol-Implementation Fuzzing, Model-Based Fuzzing, LLM-Assisted Testing

1 Introduction

Network-protocol implementations are widely deployed across systems from cloud services and web applications [11, 13] to embedded devices [15, 27] and industrial control systems [48]. Bugs in these implementations can persist for an extended period, compromising the security and stability of the systems. Studies show that even a single malformed input or unhandled edge case can disrupt services and cause catastrophic consequences [12, 28, 54]. Thus, identifying these bugs is an essential step when implementing protocols.

In this paper, we focus on *fuzzing* of protocol implementations. Studies show that fuzzing is an effective method for testing protocol implementations [21, 24, 25, 40]. In these methods, a fuzzer usually sends crafted protocol messages over a network interface or directly to a broker to trigger unexpected states, crashes, or abnormal responses. One major benefit of fuzzing is its capacity to explore a large state space without requiring a formal protocol specification.

Nevertheless, having domain knowledge about protocols can be helpful. *Model-based fuzzing* of protocol implementations [9, 10, 38, 39] is a method that uses knowledge about protocols. These methods use a predefined (usually coarse-grained) model of protocol behavior to generate *sequences* of messages, systematically exploring the state space of the protocol. One widely employed family of models is finite-state machines (FSMs) [20, 22, 42, 43]. These methods use an FSM to represent the states and transitions defined by a protocol, enabling the fuzzer to produce sequences of messages that reflect protocol-specific communication patterns. Following the predefined model, these methods improve input validity, enhance code coverage, and increase the chances of identifying bugs that depend on specific protocol states or message sequences.

However, the reliance on predefined models is a double-edged sword: the applicability and effectiveness of model-based fuzzing depend on the availability and quality of these models. Adapting model-based fuzzing to different protocols requires the manual construction of protocol models, and adjusting these models to optimize fuzzing performance would be an effort-consuming task.

In this paper, we propose ChatFuMe, a model-based protocol-implementation fuzzing method that leverages large language models (LLMs) [17] to construct and adjust protocol models *automatically*. LLMs have emerged as a novel tool for general fuzzing [52, 53], as well as protocol-implementation fuzzing [35, 37, 49]. These LLM-assisted fuzzing methods leverage the understanding and generation capabilities of LLMs to produce syntactically and semantically valid messages for different protocols. Unlike ChatFuMe, none of these are model-based: they rely on LLMs to directly generate a large number of sequences of protocol messages, resulting in long generation time and high token consumption.

Instead of direct generation of message sequences, ChatFuMe uses LLMs to automatically construct and adjust protocol models in the form of random sequence-generator *programs* that generate random message sequences. Our motivation is driven by the LLMs' strong capabilities in generating *programs*, inferring state transitions, and embedding domain knowledge from protocol documentation and usage patterns. Our design of ChatFuMe aims to strike a balance among the following desiderata:

- *Flexibility*. ChatFuMe handles implementations of different protocols with little adaptation effort.
- *Effectiveness*. ChatFuMe achieves comparable performance against model-based fuzzing with predefined models.
- *Cost-efficiency*. ChatFuMe consumes much fewer tokens than prior LLM-assisted protocol fuzzing methods.

ChatFuMe consists of two major components: (i) automatic model construction, and (ii) feedback-guided fuzzing loop. The model construction starts with identifying protocol states, using the protocol's documentation (possibly with some user prompts to

encode domain knowledge) as input. ChatFuMe then asks the LLM to summarize the protocol behavior, capturing high-level domain-specific patterns of the states and transitions. Next, ChatFuMe prompts the LLM to select key states and summarize the transition rules among them. These states and transitions form the basis of a coarse-grained protocol model. Rather than directly generating message sequences, ChatFuMe asks the LLM to construct a sequence generator, which is an executable program that samples state transitions and generates random message sequences.

After the model construction, ChatFuMe's fuzzing loop starts with pairing the random sequence generator with a user-provided payload generator that handles the low-level field formatting of messages. The user can once again use an LLM to program the payload generator in advance. Executing the random generator multiple times, ChatFuMe generates multiple message sequences and sends them to the protocol implementation being tested. ChatFuMe then collects the behavior of the protocol implementation, such as its responses, and prompts the LLM to evaluate these results and suggest adjustments to the protocol model. In particular, the LLM is supposed to provide feedback on whether to add new states, remove existing ones, or update transition probabilities. With the feedback, ChatFuMe's model-construction component adjusts the protocol model and the generator program. The fuzzing-loop component then begins another iteration using the adjusted model.

Our experiments demonstrate that ChatFuMe is reasonably flexible, effective, and cost-efficient. In testing three different real-world protocol implementations, our approach discovered 12 potential bugs, validating its fault detection capability. Compared to a prior model-based fuzzer, our method identified more new protocol states within the same time window, demonstrating its effectiveness in exploration. Additionally, compared to a prior LLM-based fuzzer, our technique achieves lower token consumption, highlighting its cost efficiency and scalability in practical fuzzing scenarios.

Contributions. The paper's contributions include the following:

- We propose ChatFuMe, an LLM-assisted model-based fuzzing method for protocol implementations. The key innovation is that it uses LLMs to construct and adjust a protocol-model program that generates message sequences.
- We implement ChatFuMe and conduct an experimental evaluation of it. Our experiments demonstrate the flexibility of ChatFuMe by applying it to three different protocols, its effectiveness by comparing it against a prior model-based method FUME [39], and its cost efficiency by comparing it against a prior LLM-assisted method ChatAFL [37].
- We apply ChatFuMe on three real-world protocol implementations (HMQ, PyModbus, and Moquette) and discover 12 potential bugs in these implementations.

2 Background: Model-Based Fuzzing of Protocol Implementations

In this section, we review FUME [39], a model-based fuzzing technique designed for Message Queuing Telemetry Transport (MQTT) implementations. We begin by reviewing the MQTT protocol, which we will use as a concrete protocol to demonstrate our method in Section 3. We then sketch FUME's predefined model that guides the fuzzing of MQTT implementations.

Table 1: A summary of MQTT control packets.

Name	Purpose	Components
CONNECT	Initiate connection to broker	FH, VH, payload
CONNACK	Acknowledge connection request	FH, VH
PINGREQ	Check if connection to broker is alive	FH
PINGRESP	Confirm connection is active	FH
DISCONNECT	Close network connection gracefully	FH
AUTH	Exchange authentication data	FH, VH
PUBLISH	Deliver message to subscribers	FH, VH, payload
PUBACK	Acknowledge QoS 1 PUBLISH	FH, VH
PUBREC	Acknowledge QoS 2 PUBLISH	FH, VH
PUBREL	Confirm receipt of PUBREC	FH, VH
PUBCOMP	Confirm receipt of PUBREL	FH, VH
SUBSCRIBE	Request subscription to topics	FH, VH, payload
SUBACK	Acknowledge SUBSCRIBE packet	FH, VH, payload
UNSUBSCRIBE	Request to cancel subscriptions	FH, VH, payload
UNSUBACK	Acknowledge UNSUBSCRIBE packet	FH, VH, payload

2.1 The MQTT Protocol

The Message Queuing Telemetry Transport (MQTT) protocol [4] has emerged as the de facto standard for messaging in the Internet of Things (IoT) and Industrial IoT (IIoT) domains [47]. Standardized by OASIS and ISO, MQTT is a lightweight, event-driven protocol designed for environments with limited bandwidth and high latency, making it ideal for devices such as sensors, embedded systems, and industrial PLCs. At its core, MQTT operates on a publish/subscribe model, which inherently decouples message *publishers* (i.e., senders) from *subscribers* (i.e., receivers). Communication is facilitated through *topics*, which serve as virtual channels for message exchange. The central component of this architecture is the MQTT *broker* (i.e., server), which manages connections, filters incoming messages from publishers based on their topics, and efficiently distributes them to all interested subscribers.

In MQTT, all communication between clients and brokers is facilitated through the exchange of *control packets*, which are the fundamental units of data transfer. These packets encapsulate various operational commands, enabling functions such as establishing connections, managing subscriptions, and publishing application messages. Each control packet adheres to a structured format consisting of up to three main components: a fixed header (FH), a variable header (VH), and the *payload*, which contains the actual data or message that may vary depending on the control packet type. MQTT supports 15 different packet types, including connection management, message publishing, and subscription management. Table 1 summarizes the 15 MQTT packet types.

The explicit connection management packets highlight MQTT's *stateful* nature. Such statefulness enables the broker to retain a client's subscriptions and buffer messages for delivery upon reconnection, which is crucial for maintaining persistent sessions. However, the statefulness results in a vast and complex input space for fuzzing. A fuzzing method needs domain knowledge about MQTT to generate valid message sequences (e.g., they should start

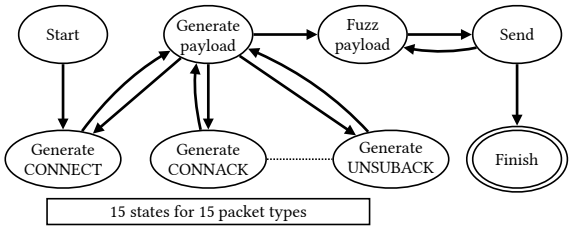

Figure 1: FUME's generation-guided fuzzing model.

with a CONNECT message) and explore deep states (e.g., some states can only be reached by subscribing to a specific topic).

2.2 FUME's Fuzzing Model for MQTT Brokers

FUME is a fuzzing method designed explicitly for testing MQTT implementations [39]. To this end, FUME is a model-based fuzzing method because it requires a predefined model of the MQTT protocol to guide the fuzzing process. FUME's model is coarse-grained: it only knows the 15 message types shown in Table 1, and every message sequence must start with CONNECT.

FUME then uses the MQTT model to construct two Markov models (i.e., FSMs with probabilistic transitions) to describe mutation-guided and generation-guided fuzzing, respectively. *Mutation*-guided fuzzing requires an input corpus of valid message sequences as test cases. On the other hand, *generation*-guided fuzzing requires domain knowledge of the protocol to generate valid message sequences. In this paper, we aim to develop a *flexible* fuzzing method applicable to different protocol implementations, where informal protocol descriptions are often more readily available than a corpus of test cases; thus, we focus on generation-guided fuzzing.

Figure 1 demonstates FUME's fuzzing model of one iteration of generation-guided fuzzing. We ignore the state-transition probabilities, so it appears just like an FSM. The model describes a random generation process: it starts with generating a CONNECT message and its payload, then stochastically generates some follow-up messages, applies fuzzing (e.g., insertion, deletion, and mutation) to the payloads multiple times, and finally sends the message sequences to the MQTT broker. FUME has another fuzzing model for its mutation-guided fuzzing process, and it alternates between the two fuzzing models to leverage the strengths of both models.

We take inspiration from FUME with a key observation: such fuzzing models can be easily expressed by *executable programs* that generate random message sequences. Moreover, payload generators are also executable programs. The observation motivates us to leverage the understanding and programming capabilities of LLMs to extract domain knowledge about protocols and generate executable programs that represent the fuzzing models.

3 Our Method

In this section, we describe the workflow and technical details of our ChatFuMe method. Section 3.1 presents an overview of ChatFuMe's workflow and its two major components: (i) automatic model construction, and (ii) feedback-guided fuzzing loop. Sections 3.2 and 3.3 use MQTT as a demonstration protocol to explain the two components, respectively.

3.1 Overview of the Workflow

Figure 2 presents an overview of ChatFuMe. Its key feature is combining model-based protocol-implementation fuzzing with LLMs: it uses LLMs to automate various steps in the workflow, including the construction and adjustment of the fuzzing model. In this way, ChatFuMe achieves flexibility to handle different protocols.

Automatic model construction. ChatFuMe begins with user-provided protocol documentation, which may include prompts to describe protocol states as a high-level summary of a protocol's specification. ChatFuMe uses an LLM to augment the set of states by extracting domain knowledge from the protocol documentation. With the augmented set of states, ChatFuMe again requests the LLM to perform state selection and summarize the state-transition rules, narrowing down to a small but essential subset of states as the starting point for fuzzing and capturing how the protocol allows moving between different states. These transitions, with the selected states, form a lightweight coarse-grained model of protocol behavior. This component also provides the functionality to adjust the model by prompting an LLM to modify the selected states based on feedback from the fuzzing loop, which we explain below.

Feedback-guided fuzzing loop. Based on the constructed protocol model, ChatFuMe uses an LLM to generate a sequence-generator program, which produces randomized state sequences that conform to transition rules. The sequence generator is paired with a user-provided payload generator—which an LLM could generate in advance—to generate complete message sequences. ChatFuMe then sends these sequences to the protocol implementation and monitors its behavior, e.g., whether it crashes or reports abnormal responses. After collecting the testing results, ChatFuMe decides whether to adjust the protocol model. We employ two mechanisms for this decision: (i) with a predefined probability, ChatFuMe resets the protocol model to the initial one constructed at the beginning of the workflow, i.e., incorporates *random restart*; or (ii) ChatFuMe prompts the LLM to analyze the testing results and decide if the model needs refinement, and if so, further prompts it to identify states that contribute to effective testing outcomes. Typically, the LLM would add a few new states or remove existing ones from the protocol model. If the model gets adjusted, ChatFuMe loops back to the phase of generating a sequence-generator program via an LLM. Otherwise, it keeps the sequence-generator program unchanged and starts the next loop iteration.

Incorporation of LLMs. We use LLMs because they provide capabilities for understanding documentation (for model construction), generating code (for program generation), and analyzing testing results (for model adjustment). Furthermore, for widely used protocols, even if the user does not possess domain knowledge, LLMs' familiarity with these protocols enables ChatFuMe to perform fuzzing effectively with minimal guidance. Instead of asking the LLM to generate individual test cases directly, we prompt it to generate an executable program as a random sequence generator. This generator-based design significantly reduces token consumption

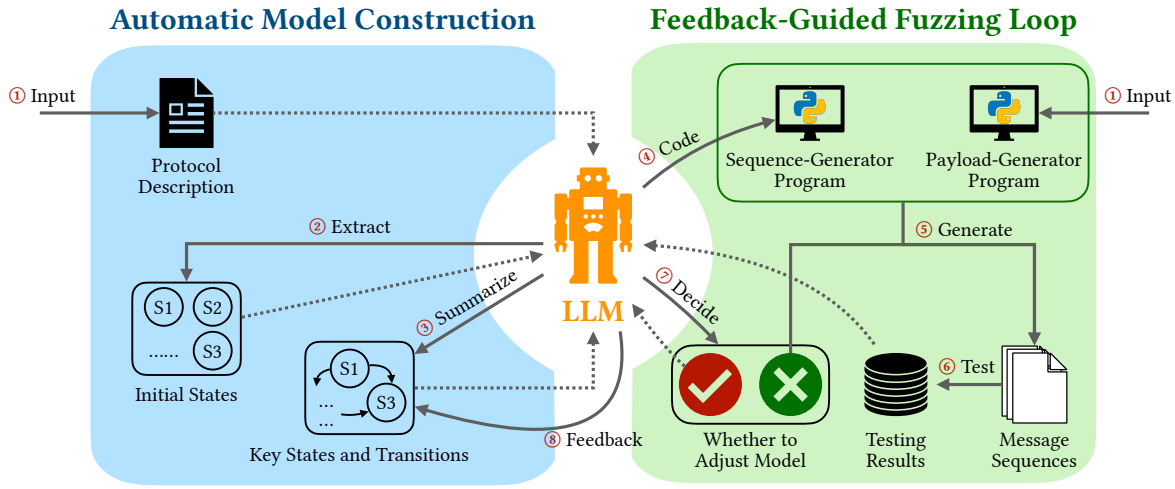

Figure 2: An overview of ChatFuMe's workflow. Dotted arrows indicate prompting the LLM for specific tasks.

by generating numerous message sequences from a single generator, making ChatFuMe cost-efficient for LLM-assisted fuzzing of real-world protocol implementations.

3.2 Automatic Model Construction

The construction of our fuzzing model begins with user-provided identification of protocol states, which can be derived from existing work, specifications, documentation, or manual analysis. Listing 1 provides an example of the initial states of MQTT. These initial states can be extracted from the MQTT specification [4]. These states provide a coarse-grained foundation for understanding the protocol's high-level behavior. To enrich this initial model, we prompt the LLM to incorporate domain knowledge, such as typical client-server interactions and expected message sequences. This step enhances the completeness and semantic accuracy of state definitions, enabling better downstream reasoning during fuzzing. By combining user input with LLM-driven knowledge expansion, we strike a balance between manual guidance and automated insight.

Listing 1: An example of initial states. The list above represents the primary MQTT control packet types. Each state corresponds to a specific control packet used in the communication between MQTT clients and brokers.

```
states = [ "CONNECT", "CONNACK", "PUBLISH",
           "PUBACK", "PUBREC", "PUBREL",
           "PUBCOMP", "SUBSCRIBE", "SUBACK",
           "UNSUBSCRIBE", "UNSUBACK", "PINGREQ",
           "PINGRESP", "DISCONNECT", "AUTH" ]
```

Once domain knowledge is incorporated, we leverage the LLM to refine the state space by selecting a concise set of essential protocol states for testing. This step abstracts the protocol into a manageable number of representative states that retain sufficient semantic coverage while reducing unnecessary complexity. As illustrated in Listings 2 and 3, we construct a two-part prompt to guide the LLM in identifying essential protocol states for testing. The Listings 2 provides contextual information about the protocol and the goal of reducing the input space by selecting a small number of high-impact states. The Listings 3 specifies a structured output format, requiring the LLM to return a JSON array of state names and concise justifications for their importance. This automated abstraction enables ChatFuMe to focus on high-impact areas of the protocol, balancing coverage and efficiency in the input space exploration.

Listing 2: The prompt for state selection.

```
Prompt_for_States_Selection = f'''I am designing a model-
    based testing framework for the {protocol}. To
    reduce the search space, I want to abstract the
    protocol into a small number of essential states
    that capture the most important aspects of its
    behavior for testing purposes.
Please help me identify {number} essential states above
    that cover the core functionality of {protocol}
    while preserving enough semantics to be useful for
    testing client and broker implementations.
Output Format: {example}'''
```

Listing 3: The Example for state selection.

```
Example_for_States_Selection = f'''
Return a JSON array of {number} objects.
Each object should have ONLY the following fields:
`"select"`: A short name for the state
(e.g., `"CONNECT"`, `"PUBLISH"`)
`"reason"`: A concise explanation of why this state is
essential for {protocol} testing
'''
```

3.3 Feedback-Guided Fuzzing Loop

In the feedback-guided fuzzing loop, we leverage the LLM to analyze execution results and guide the adjustment of the protocol model. The loop begins with the LLM-generated sequence generator, which produces message sequences following the current state transition model. These sequences serve as high-level plans for how a client might interact with a protocol implementation.

To guide the LLM in producing a realistic sequence generator, we apply an *autoprompting* strategy [17, 50, 52] to create a high-quality prompt that encapsulates the protocol specification. Given a set of selected states, we ask the LLM to construct a prompt that will later be used to generate Python code implementing a random state sequence generator. This prompt includes constraints to preserve realistic protocol behavior, such as capturing state transition probabilities, enforcing randomness, and ensuring variable-length sequences. In this way, we distill the protocol specification into a reusable instruction, forming a bridge between abstract state modeling and a concrete generation of the generator.

After generating a prompt tailored for protocol behavior, we use the LLM to produce a Python program as the sequence generator, which we write into a module and dynamically import for execution. This integration step serves both as a code validity check and as the mechanism to link LLM output with our fuzzing pipeline. If the generated code passes import and runtime validation, we invoke it repeatedly in the loop. Listing 4 presents an example of the code generated by the LLM, which incorporates both the protocol specification and random control structures.

Listing 4: The Python code of an LLM-generated random sequence generator for the MQTT protocol.

```python
import random

def MQTT_state_generator():
    states = ['CONNECT', 'CONNACK', 'PUBLISH', 'SUBSCRIBE', 'DISCONNECT', 'PINGREQ', 'PUBACK']
    state_sequence = []
    current_state = 'CONNECT'
    state_sequence.append(current_state)

    while True:
        if current_state == 'CONNECT':
            next_state = random.choices(['CONNACK', 'SUBSCRIBE'], weights=[70, 30])[0]
        elif current_state == 'CONNACK':
            next_state = random.choices(['PUBLISH', 'SUBSCRIBE', 'DISCONNECT'], weights=[50, 30, 20])[0]
        elif current_state == 'PUBLISH':
            next_state = random.choices(['PUBACK', 'SUBSCRIBE', 'DISCONNECT'], weights=[60, 30, 10])[0]
        elif current_state == 'SUBSCRIBE':
            next_state = random.choices(['PUBLISH', 'DISCONNECT', 'PINGREQ'], weights=[40, 30, 30])[0]
        elif current_state == 'DISCONNECT':
            if random.random() < 0.5:
                break
            else:
                next_state = 'CONNECT'
        elif current_state == 'PINGREQ':
            next_state = random.choices(['DISCONNECT', 'PUBACK'], weights=[70, 30])[0]
        elif current_state == 'PUBACK':
            next_state = random.choices(['PUBLISH', 'DISCONNECT'], weights=[60, 40])[0]

        state_sequence.append(next_state)
        current_state = next_state

        if random.random() < 0.5:
            break

    return state_sequence
```

The generated sequences are then passed to a user-provided payload generator, which translates each sequence into concrete protocol messages. Note that the payload generator could also be generated in advance by an LLM. The resulting payloads are sent to the target protocol implementation, and ChatFuMe records the feedback from each run to inform further model adjustment.

After sending a sufficient number of fuzzing sequences (ranging from 20,000 to 50,000 in our experiments), ChatFuMe analyzes the feedback from the target protocol implementation and evaluates the fuzzing effectiveness, guiding model adjustment. The analysis focuses on identifying failure patterns by computing statistics such as the number of failures per protocol function, total request distribution, and failure rates. Specifically, we categorize responses like timeouts or connection resets as failures and calculate their frequency relative to the total number of generated sequences. This analysis provides insight into which parts of the protocol are more error-prone, helping to inform the LLM-driven model adjustment in subsequent fuzzing iterations.

The analysis above is then formatted and sent as part of a prompt to the LLM, asking it to interpret the results in the context of the target protocol and suggest whether any states should be added or removed from the model to improve test effectiveness, as shown in Listing 5. Notably, we only ask the LLM to interpret the statistical results and enhance test effectiveness. We do not explicitly ask it to analyze or reason about correlations between different types of failures. To ensure reliability, we validate any LLM-suggested states to avoid hallucinated or irrelevant protocol behavior. This LLM-driven evaluation allows our fuzzing process to adapt intelligently over time, refining the model based on concrete feedback from protocol implementations.

Listing 5: The prompt for model adjustment.

```
Prompt_for_Decision = f'''{reuslt_summary}
Above is the summary of fuzzing results of a {protocol} implementation using these states:
{states}
Do you think I should add or remove more states in the search space?
Give your result in JSON format.
It should ONLY have two fields:
`"decision"`: ADD or DELETE answer for should I add more states in search space or delete one state in the search space?
`"reason"`: A concise explanation of why I should add more states.'''
```

To enhance exploration and prevent convergence to a local optimum, we introduce controlled randomness into the loop by occasionally re-initializing the model from the beginning, allowing the system to escape stagnant state configurations. (Note that this mechanism is not illustrated in Figure 2.) The fuzzing loop continues until a crash or critical fault is observed, ensuring prolonged exploration when necessary. When analyzing results, if adjustment is needed, we prompt the LLM to suggest a new state to add (from previously unselected candidates) or recommend removing underperforming states. These decisions are based on summarized feedback from past executions, and each recommendation includes a justification to preserve model clarity. By integrating randomness and iterative adjustment, the loop strikes a balance between exploiting known effective sequences and exploring new protocol behaviors.

Table 2: A summary of protocol implementations used in our evaluation. "#Stars" indicates the number of GitHub stars (as of July 2025). "Used in" denotes which research question(s) (RQ1–RQ3) each implementation contributed to.

Name	Protocol	Language	#Stars	Used in
HMQ	MQTT	Go	1359	RQ1
Moquette	MQTT	Java	2382	RQ1
Mosquitto	MQTT	C	9928	RQ1 & RQ2
Aedes	MQTT	JavaScript	1873	RQ2
Pymodbus	Modbus	Python	2494	RQ1
OwnTone	DAAP	C	2287	RQ1 & RQ3

4 Experimental Design

In this section, we describe our experimental design to evaluate our ChatFuMe method. We propose the following three research questions, concerning flexibility, effectiveness, and cost efficiency:

- **RQ1**: How flexible and effective is ChatFuMe in discovering faults across different protocols and implementations?
- **RQ2**: How does ChatFuMe compare to a prior model-based fuzzing method in protocol testing effectiveness?
- **RQ3**: What are the characteristics of ChatFuMe in terms of token usage compared with an existing LLM-based fuzzer?

4.1 Systems Under Test and Baselines

We selected three network protocols and six real-world software implementations in total, with varying levels of maturity and popularity, as reflected by their GitHub star counts. Table 2 presents the statistics, including the protocol and programming language they are based on, their popularity (GitHub stars), and the specific RQs they were used to evaluate. This setup helps ensure that ChatFuMe is not tied to any single protocol or implementation style, reinforcing its flexibility and practical applicability.

RQ1. To demonstrate the flexibility and effectiveness of our approach, we conduct experiments across three different protocol implementations: MQTT, Modbus, and Digital Audio Access Protocol (DAAP). These protocols span different formats, transport layers, and usage domains. The goal of this part of the experiment is to demonstrate that ChatFuMe is not limited to any specific protocol or domain.

MQTT is a lightweight publish-subscribe messaging protocol commonly used in IoT systems. We evaluated our approach on three different MQTT broker implementations: HMQ [31], Moquette [3], and Mosquitto [5]. This demonstrates ChatFuMe's flexibility across languages and ecosystems. HMQ is a high-performance MQTT broker written in Go, designed for scalability and compatibility with MQTT 3.1.1 and standard clients. Moquette is a lightweight, embeddable Java broker that supports MQTT versions 3 and 5, featuring session expiration and topic aliasing. Finally, Mosquitto is a widely used C implementation that offers a compact MQTT broker and client suite. Testing across these diverse implementations helps establish that ChatFuMe is protocol-agnostic and language-agnostic. It is capable of handling varying runtime environments and codebases.

Modbus is an industrial control protocol based on function codes. We selected Pymodbus [7] as the target implementation for Modbus in our evaluation. Pymodbus is a full-featured, open-source Modbus protocol stack written in Python, supporting both synchronous and asynchronous APIs. It provides built-in client and server simulators, payload builder/decoder functions, and supports both standard and extended Modbus function codes with minimal external dependencies.

DAAP is a binary protocol layered over HTTP used for media sharing. We chose OwnTone [36] as our DAAP server for evaluation. OwnTone is an open-source media server written in C, designed to serve audio content over the DAAP. It supports sharing and streaming music via DAAP, making it a versatile and realistic target for fuzz testing. By incorporating OwnTone as our test subject, we demonstrate that our framework can handle binary protocol implementations layered over HTTP.

RQ2. To evaluate the testing effectiveness of our approach, we compare it against FUME, a model-based fuzzing technique explicitly designed for MQTT. FUME combines mutation-based and generation-based fuzzing strategies and introduces Markov chains to guide both payload mutation and generation. It models the fuzzing process as a finite Bernoulli process to explore MQTT protocol behaviors and uncover vulnerabilities thoroughly.

This research question is evaluated in two parts. First, we measure and compare the number of test cases generated by each method within a fixed time window, assessing the throughput and exploration capability of the fuzzers. Second, we analyze the crash discovery speed, which is the rate at which each tool triggers a fault or crash in the target protocol implementation. For the throughput comparison, we evaluate on Mosquitto. For the crash speed comparison, we use Aedes [6], a popular MQTT broker implemented in JavaScript.

RQ3. For token usage analysis, we compare our approach with ChatAFL [37], a recent LLM-guided protocol fuzzer. ChatAFL leverages large language models trained on human-readable protocol specifications to extract protocol message grammars and predict stateful interactions. It uses LLMs to generate message sequences and detect states in protocol implementations, combining grammar construction with mutation and sequence prediction. We conduct a comparison focused on: the total number of tokens consumed and the number of LLM API calls required during the fuzzing process.

4.2 Our Implementation

ChatFuMe is primarily implemented in Python, with an emphasis on cost-efficiency and broad applicability. To maximize flexibility, all experiments in RQ1 and RQ2 were conducted using GPT-4o-mini, a lightweight language model that is readily interchangeable with many popular alternatives on the market. This is made possible by our design choice to decompose the overall fuzzing workflow into smaller, modular tasks. This eliminates the need for large token windows or high-capacity models. For RQ3, we additionally evaluated ChatFuMe using GPT-3.5 Turbo, which successfully handled all required subtasks, further demonstrating the adaptability of our approach to different LLM configurations. In all experiments,

the temperature parameter was set to 0.5 to introduce controlled randomness and encourage diverse outputs from the LLM.

We obtain the initial states and payload generators for MQTT, Modbus, and Digital Audio Access Protocol (DAAP) using the following approach:

- For MQTT, we extracted protocol states directly from the official specification[4] and adopted the existing payload generator from FUME.
- For Modbus, we similarly derived states from the official specification [2]. To generate payloads, we provided ChatGPT with examples from the specification. ChatGPT generates a protocol-aware payload generator totaling 501 lines of code.
- For DAAP, which is layered over HTTP and features loosely structured binary payloads, we used ChatGPT to create a lightweight generator based on the specification[1] and integrated simple mutation strategies. The resulting generator is compact—just over 100 lines of code—yet effective.

4.3 Experimental Setup

We designed fuzzing campaigns tailored to each research question while balancing resource constraints and consistency.

RQ1. We ran each fuzzing campaign for five hours to allow sufficient exploration while limiting the cost of LLM API calls due to hardware and budget constraints.

RQ2. We conducted a more controlled comparison with FUME. To mitigate randomness, we ran both ChatFuMe and FUME for one hour, repeated across three independent trials, and measured the total number of unique test cases, the total number of test cases generated, the average test case length, and the new response found. To evaluate the crash discovery speed, we ran both tools three times and recorded the time it took for each to trigger the first crash.

RQ3. Since ChatAFL does not store all generated test cases, we cannot directly compare the number of test cases over a fixed time. Therefore, we focus on comparing token usage and the number of LLM calls. To ensure a fair comparison, we use GPT-3.5 Turbo instead of GPT-4o-mini in our evaluation.

Environment. All experiments were conducted on a virtual machine running Ubuntu 20.04. The VM was allocated 11.4 GB of memory, 4 CPU cores, and a 50 GB SCSI hard disk. The host machine is equipped with a 13th Gen Intel(R) Core(TM) i9-13900H @ 2.60 GHz and 32 GB of RAM. All LLM calls in our experiments were made via the OpenAI API.

Metrics. Because ChatFuMe does not rely on instrumentation, we do not report traditional code coverage metrics [16, 51]. For the comparison in RQ2 regarding test case generation, we measure the total number of test cases generated, the number of unique test cases, the average test case length, and the number of distinct responses discovered. For RQ3, we focus on efficiency metrics by comparing the total number of LLM tokens used and the number of LLM API calls made.

5 Evaluation and Discussion

By evaluating across general-purpose protocols and comparing with both traditional and LLM-based fuzzers, we demonstrate that ChatFuMe is broadly applicable, capable of uncovering real-world bugs, and cost-efficient in terms of token usage.

5.1 RQ1: Flexibility and Effectiveness Across Protocol Implementations

ChatFuMe successfully discovered multiple bugs across diverse protocol implementations, demonstrating both effectiveness and flexibility. Table 3 summarizes the 12 potential bugs discovered by ChatFuMe across multiple protocol implementations.

For the MQTT protocol, we evaluated three broker implementations: HMQ, Moquette, and Mosquitto. On HMQ, we identified a critical bug that causes the system to crash. On Moquette, ChatFuMe uncovered three distinct issues that trigger exceptions, though the broker remains operational. No bugs were discovered in Mosquitto, likely because we reused the payload generator from FUME, which has already been used extensively to test Mosquitto, and most known issues have been patched.

Listing 6 shows the buggy code we discovered in the HMQ MQTT broker. The issue lies in the conditional statement `conn != nil && conn.RemoteAddr() != nil`. While it appears safe, calling `conn.RemoteAddr()` when `conn` is `nil` will still cause a runtime panic in Go, as method calls on a `nil` interface result in dereferencing a `nil` pointer. This leads to a crash if the code path is ever executed with a `nil` conn, which our generated input successfully triggered. This example illustrates how ChatFuMe can uncover subtle but critical edge-case bugs by exploring under-tested execution paths.

Listing 6: Buggy code in HMQ where line 2 contains a faulty conditional check: calling `conn.RemoteAddr()` without ensuring `conn` is non-`nil` leads to a runtime panic.

```
1 // add remote connection address
2 if !wsEnabled && conn != nil && conn.RemoteAddr() != nil
3 {
4     result = append(result, zap.Stringer("addr", conn.
      RemoteAddr()))
5 }
6 else if wsEnabled && wsConn != nil && wsConn.Request() !=
       nil
7 {
8     result = append(result, zap.String("addr", wsConn.
      Request().RemoteAddr))
9 }
```

For Modbus, we tested the Pymodbus implementation. Our approach identified eight unique bugs, each causing exceptions without crashing the system. They are all related to incorrect buffer lengths during binary unpacking. These exceptions point to improper handling of specific malformed or edge-case inputs and reflect the method's ability to exercise error-handling paths even in well-established protocol libraries.

Listing 7: Buggy code in Pymodbus where line 1 attempts to unpack 5 bytes from a buffer without validating its length, leading to a `struct.error` when the data is too short.

```
1 self.address, count, _byte_count = struct.unpack(">HHB",
      data[0:5])
```

The bug occurs in the Modbus implementation when parsing a request frame with an unexpected or malformed length. Specifically, the code in Listing 7 attempts to unpack 5 bytes from the incoming `data` buffer using `struct.unpack(">HHB", data[0:5])`.

Table 3: A summary of identified potential bugs by protocol, software, error type, and description.

#	Protocol	Subject	Error Type	Description
1	Modbus	PyModbus	`struct.error`	Buffer too small for `>HH` in `register_message.py:180`.
2	Modbus	PyModbus	`struct.error`	Buffer too short for `>HHB` in `bit_message.py:134`.
3	Modbus	PyModbus	`struct.error`	Buffer too short for `>H` in `file_message.py:238`.
4	Modbus	PyModbus	`struct.error`	Buffer too short for `>BBB` in `mei_message.py:52`.
5	Modbus	PyModbus	`struct.error`	Buffer too short for `>HH` in `bit_message.py:30`.
6	Modbus	PyModbus	`struct.error`	Buffer too short `>HH` in `register_message.py:26`.
7	Modbus	PyModbus	`struct.error`	Buffer too short for `>BHHH` in `file_message.py:61`.
8	Modbus	PyModbus	`struct.error`	Buffer too short for `>HHB` in `register_message.py:225`.
9	MQTT	HMP	`nil pointer dereference`	Crash if `conn.RemoteAddr()` is nil
10	MQTT	Moquette	`IOException`	Invalid MQTT message caused channel closure.
11	MQTT	Moquette	`NullPointerException`	Null access during `PUBLISH` message handling.
12	MQTT	Moquette	`StacklessClosedChannelException`	Connection closed before sending `CONNACK`.

Table 4: The comparison of test-case generation between ChatFuMe and FUME over three runs (on Mosquitto).

Run	ChatFuMe			FUME		
	Total Cases	Unique Cases	Avg. Length	Total Cases	Unique Cases	Avg. Length
1	**3,333,073**	**1,796,307**	127.06	1,875,234	1,138,751	**139.26**
2	**2,125,908**	**1,179,660**	**137.91**	1,874,732	1,137,339	135.46
3	**1,610,051**	899,816	**144.34**	1,519,627	**925,739**	129.8

However, if the `data` is shorter than 5 bytes, it results in a `struct.error` with the message `unpack` requires a buffer of 5 bytes. This indicates a missing length check before unpacking, which can cause the program to crash or throw an exception at runtime.

In the case of DAAP, we tested the OwnTone media server. Our testing did not uncover new bugs. One challenge here is that DAAP is layered over HTTP, and our system currently models only the DAAP-specific parts, without generating complete HTTP requests. Furthermore, OwnTone has already been tested by ChatAFL [37], which may have addressed some common issues. Nevertheless, ChatFuMe was still able to process and explore DAAP's binary structure with minimal manual adjustment, underlining its general applicability.

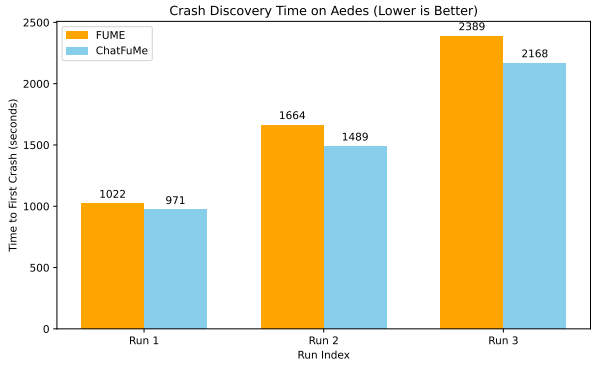

Figure 3: The comparison of crash discovery time between ChatFuMe and FUME over three runs (on Aedes).

5.2 RQ2: Effectiveness vs. Model-Based Fuzzers

To evaluate how our LLM-assisted fuzzing approach compares to existing model-based fuzzers, we conducted a head-to-head comparison with FUME, a domain-specific MQTT fuzzer. Our evaluation consists of two parts: the first measures the ability of each tool to generate diverse and voluminous test cases within a fixed time budget; the second assesses the efficiency of each method in triggering faults by comparing the time taken to induce a crash.

Table 4 presents the test case generation results of our approach and FUME across three independent one-hour fuzzing runs on the Mosquitto MQTT broker. ChatFuMe consistently generates a higher number of total and unique test cases compared to FUME. For example, in the first run, ChatFuMe produced over 3.3 million total cases and nearly 1.8 million unique ones, whereas FUME generated only 1.8 million total cases and 1.1 million unique cases. This trend holds across all three runs, demonstrating the higher test case diversity and generation throughput of our LLM-assisted approach.

While the average payload length varies slightly, both tools produce messages of comparable size, indicating similar levels of complexity in the generated data. This suggests that ChatFuMe is not simply generating larger or noisier inputs to inflate test coverage, but is instead producing diverse, well-formed messages that are competitive in structure and semantics. Moreover, the consistently higher number of unique cases indicates broader exploration of the input space, which can lead to uncovering more edge cases and rare protocol behaviors.

To assess crash discovery efficiency, we compared our LLM-assisted fuzzer against FUME on the Aedes MQTT broker under

Table 5: The comparison of LLM token and API call usage between ChatFuMe and ChatAFL over a one-hour run.

Method	Tokens Used	LLM Calls
ChatFuMe	5,980	16
ChatAFL	216,596	160

identical conditions over three trials. ChatFuMe located the first crash in an average of 1543.0 seconds, compared to 1691.7 seconds for FUME, which is an improvement of roughly 9%. In every individual run, our approach detected faults faster (16m11s vs. 17m02s, 24m50s vs. 27m44s, and 36m08s vs. 39m49s), demonstrating that our LLM-guided sequence generation can accelerate the identification of critical vulnerabilities.

5.3 RQ3: Cost Efficiency vs. LLM-Based Fuzzers

To assess the efficiency of ChatFuMe compared to existing LLM-based fuzzing approaches, we conducted a one-hour experiment using both ChatFuMe and ChatAFL. We measured the total number of tokens consumed and the number of LLM calls made during the fuzzing process. As shown in Table 5, our approach consumed only 5,980 tokens and made 16 LLM calls, while ChatAFL consumed 216,596 tokens and issued 160 LLM calls in the same period. This demonstrates that ChatFuMe is more token-efficient—using roughly 36 times fewer tokens and 10 times fewer LLM calls—making it more practical for long-running or cost-sensitive fuzzing campaigns.

While the token and call statistics provide a clear comparison of efficiency, it is important to note that a direct comparison of the generated test cases between ChatFuMe and ChatAFL is limited due to differences in how the two tools store their outputs. Specifically, ChatAFL only retains test cases that trigger new execution paths, whereas ChatFuMe stores all generated cases for analysis and replay. Upon inspection of the test cases retained by ChatAFL, we found that they primarily consisted of HTTP-like requests such as `GET` and `POST`, which aligns with the nature of DAAP as an HTTP-based protocol. In contrast, the test cases produced by ChatFuMe are raw hexadecimal payloads that conform to the DAAP message format.

5.4 Discussion

Limitations. We do not include an ablation study in this work because our goal is to minimize manual effort and demonstrate how LLMs can be leveraged to model protocol structures with minimal human input automatically. Rather than manually deconstructing and varying components, our focus is on showcasing the feasibility and effectiveness of using LLMs in a streamlined and integrated way. The strength of our approach lies in its simplicity and automation. We focus our evaluation on comparing the complete system against established baselines, including traditional fuzzers and existing LLM-based methods.

Our findings suggest that LLMs can be used not only to automate the creation of fuzzers but also to generate malicious tools for exploiting vulnerabilities. This highlights a broader concern about the dual-use nature of large language models, suggesting that future work should consider safeguards and responsible deployment practices.

Theat to construct validity. One limitation of our method, common to many fuzzing techniques, is that it primarily exposes surface-level bugs: crashes or exceptions caused by malformed inputs. While our method generates syntactically diverse and realistic messages, it does not capture deeper semantic behaviors that may be necessary to uncover subtle logic bugs. This may limit the types of vulnerabilities our tool can detect, potentially underestimating deeper security issues present in the target systems.

Threat to internal validity. Malformed inputs may not directly cause some bugs discovered during fuzzing, but rather be caused by unrelated factors such as system configuration or dependency issues. We mitigate this by confirming that crashes are triggered deterministically with repeated inputs.

Threat to external validity. Our evaluation focuses on a select set of protocols (MQTT, Modbus, DAAP) and open-source implementations. While they cover multiple transport layers and application domains, generalizing our results to all protocol-based software may be limited. Proprietary systems, real-time protocols, or those with more complex state machines may exhibit different behavior.

In some cases, the initial protocol specifications used by our payload generator were derived from LLM output (e.g., ChatGPT). While this allows automation, it also introduces potential inaccuracies or omissions compared to official standards. The effectiveness of our fuzzing may partially rely on the correctness of LLM-generated specifications, which may not generalize well to protocols with complex or poorly documented semantics.

6 Related Work

6.1 Protocol Implementation Fuzzing

Protocol implementation fuzzing is a testing technique that systematically sends malformed and unexpected inputs to network protocol implementations to uncover bugs, vulnerabilities, or unexpected behavior. Fuzzing techniques in terms of input generation are commonly categorized into generation-based and mutation-based approaches. Generation-based fuzzing [14, 23, 41, 46] constructs inputs from predefined specifications, ensuring syntactic correctness and compliance with the protocol. Mutation-based fuzzing [10, 29, 38, 40] modifies existing valid inputs to create test cases, relying on randomness or heuristics to explore unexpected behaviors. A common approach for fuzzers to improve performance on semantic constraints is to build a protocol communication model. The model enables fuzzers to generate structured and context-sensitive message sequences. FUME [39] manually constructs a communication model for MQTT and integrates generation-based and mutation-based fuzzing in this model. There are some works that use automated methods to build communication models [26, 34, 55]. For example, Pulsar [26] automatically builds a communication model by analyzing traffic loads.

There are also several LLM-based fuzzers designed for testing protocol implementations. These approaches typically provide the

LLM with protocol inputs or documentation and prompt it to generate test cases in the form of protocol payloads. ChatAFL [37] is a general fuzzing framework that directly uses LLM to extract information and generate initial inputs. In this framework, LLM plays an important role in initializing the seed of fuzzing and provides guidance for mutation based on coverage. However, ChatAFL is primarily designed for string-based protocol implementations, leveraging the strengths of LLMs in understanding and generating structured text data. mGPTFuzz [35] is an LLM-based fuzzing framework for Matter IoT Devices [8]. In mGPTFuzz's fuzzing loop, LLM is first asked to extract information from Matter's specification. Users then prompt the LLM to build finite state machines (FSMs) based on the extracted information. Finally, it generates inputs based on FSMs and a user-defined policy. LLMIF [49] is an LLM-based fuzzing framework for Zigbee IoT devices [27], utilizing LLM in the process of protocol information extraction and response reasoning.

Our work differs from existing approaches in several ways. Compared to manually crafted model-based fuzzers and protocol-specific LLM-based fuzzes like mGPTFuzz, our method uses LLMs to automatically build and adapt different protocol models, making it more general and less dependent on expert input. In contrast to LLM-based fuzzers that directly generate individual test cases, our method uses the LLM to program a protocol-aware sequence generator, providing better control over input structure and reducing token consumption.

6.2 LLM in Testing

Large language models (LLMs) have shown strong performance across multiple tasks in software engineering [32, 33, 45]. By leveraging their ability to understand and generate code, LLMs can help identify edge cases, create meaningful test inputs, and detect potential vulnerabilities. Fuzz4all[52] is a universal fuzzing framework for compiler testing. It outperforms different baseline tools in 6 different programming languages. Whitefox [53] is a white-box fuzzer for testing logic bugs. SymPrompt[44] presents a prompting strategy for test generation. By implementing a multi-stage workflow, SymPrompt reaches higher code coverage in several open-source Python projects.

Our work differs from existing LLM-based fuzzers in other domains, such as compiler testing or code-based test generation, by focusing on general protocol implementation fuzzing. Unlike those approaches that typically generate code or API calls as test cases, our method finally generates protocol message payloads that must conform to specific communication sequences. This requires handling stateful interactions and semantic constraints unique to network protocols, which we address by using LLMs to model protocol behavior and guide input generation, rather than producing test cases directly.

6.3 Feedback-Guided Testing

Feedback-guided testing [18] is a software cybernetics [19] approach to software testing where test strategies evolve dynamically based on real-time feedback from previous test executions. Unlike traditional static testing methods, this approach treats the software under test as a controlled object and the testing process as a feedback control loop. The central idea is to collect the outcome data from the executed test cases. For instance, in the Controlled Markov Chain model [30], the testing process is governed by an estimated state, and optimal actions are selected to meet reliability goals with minimal resource consumption.

Our method incorporates the idea of feedback-guided testing by using LLMs to iteratively adjust the input generator based on feedback from previous test executions. We adapt the generator by modifying protocol states or transitions, allowing the fuzzer to explore diverse and previously untested behaviors. This approach brings the principles of ART into the LLM era, enabling automated, feedback-driven refinement of test strategies in a structured and scalable way.

7 Conclusion

In this work, we propose a novel LLM-assisted fuzzing method ChatFuMe that automates protocol modeling and test case generation with minimal manual effort. ChatFuMe demonstrates strong flexibility across multiple protocols and is effective in identifying real-world bugs in diverse software systems. Through extensive evaluation, we show that our approach generates more diverse and higher-volume test cases than traditional fuzzers, while being significantly more efficient in discovering crashes. Moreover, our method achieves these results using far fewer LLM tokens and calls compared to other LLM-based fuzzers, highlighting its efficiency and practicality. This study illustrates the potential of large language models to streamline and enhance fuzz testing, opening new directions for intelligent and automated software testing.

References

[1] [n. d.]. GitHub - bjoernricks/daap-protocol: Digital Audio Access Protocol (DAAP) documentation — github.com. https://github.com/bjoernricks/daap-protocol. [Accessed 17-07-2025].

[2] [n. d.]. Modbus Specifications and Implementation Guides — modbus.org. https://www.modbus.org/specs.php. [Accessed 17-07-2025].

[3] [n. d.]. Moquette Broker — moquette-io.github.io. https://moquette-io.github.io/moquette/. [Accessed 19-07-2025].

[4] [n. d.]. MQTT Specification — mqtt.org. https://mqtt.org/mqtt-specification/. [Accessed 14-07-2025].

[5] 2025. Eclipse Mosquitto — mosquitto.org. https://mosquitto.org/. [Accessed 19-07-2025].

[6] 2025. GitHub - moscajs/aedes: Barebone MQTT broker that can run on any stream server, the node way — github.com. https://github.com/moscajs/aedes. [Accessed 19-07-2025].

[7] 2025. GitHub - pymodbus-dev/pymodbus: A full modbus protocol written in python — github.com. https://github.com/pymodbus-dev/pymodbus. [Accessed 19-07-2025].

[8] Connectivity Standards Alliance. 2023. Matter Specification Version 1.2. https://csa-iot.org/wp-content/uploads/2023/10/Matter-1.2-Core-Specification.pdf. [Accessed 27-05-2025].

[9] Max Ammann, Lucca Hirschi, and Steve Kremer. 2024. DY fuzzing: formal Dolev-Yao models meet cryptographic protocol fuzz testing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1481–1499.

[10] Paschal C Amusuo, Ricardo Andrés Calvo Méndez, Zhongwei Xu, Aravind Machiry, and James C Davis. 2023. Systematically detecting packet validation vulnerabilities in embedded network stacks. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 926–938.

[11] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H Katz. 1995. Improving TCP/IP performance over wireless networks. In *Proceedings of the 1st annual international conference on Mobile computing and networking*. 2–11.

[12] Christian Berger, Philipp Eichhammer, Hans P Reiser, Jörg Domaschka, Franz J Hauck, and Gerhard Habiger. 2021. A survey on resilience in the iot: Taxonomy, classification, and discussion of resilience mechanisms. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–39.

[13] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. 1996. RFC1945: Hypertext Transfer Protocol–HTTP/1.0.

[14] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2017. A messy state of the union: Taming the composite state machines of TLS. *Commun. ACM* 60, 2 (2017), 99–107.

[15] SIG Bluetooth. 2010. Bluetooth Specification Version 2.0. *http://www. bluetooth. com/* (2010).

[16] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*. 1621–1633.

[17] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[18] Kai-Yuan Cai. 2002. Optimal software testing and adaptive software testing in the context of software cybernetics. *Information and Software Technology* 44, 14 (2002), 841–855.

[19] Kai-Yuan Cai, João W Cangussu, Raymond A DeCarlo, and Aditya P Mathur. 2003. An overview of software cybernetics. In *Eleventh Annual International Workshop on Software Technology and Engineering Practice*. IEEE, 77–86.

[20] Joeri De Ruiter and Erik Poll. 2015. Protocol state fuzzing of {TLS} implementations. In *24th USENIX Security Symposium (USENIX Security 15)*. 193–206.

[21] Dongliang Fang, Zhanwei Song, Le Guan, Puzhuo Liu, Anni Peng, Kai Cheng, Yaowen Zheng, Peng Liu, Hongsong Zhu, and Limin Sun. 2021. Ics3fuzzer: A framework for discovering protocol implementation bugs in ics supervisory software by fuzzing. In *Proceedings of the 37th Annual Computer Security Applications Conference*. 849–860.

[22] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri De Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of {DTLS} implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2523–2540.

[23] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2023. Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations.. In *NDSS*.

[24] Matheus E Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. 2022. {BrakTooth}: Causing havoc on bluetooth link manager via directed fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 1025–1042.

[25] Matheus E Garbelini, Chundong Wang, and Sudipta Chattopadhyay. 2020. Greyhound: Directed greybox wi-fi fuzzing. *IEEE Transactions on Dependable and Secure Computing* 19, 2 (2020), 817–834.

[26] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 330–347.

[27] Drew Gislason. 2008. *Zigbee wireless networking*. Newnes.

[28] Lav Gupta, Raj Jain, and Gabor Vaszkun. 2015. Survey of important issues in UAV communication networks. *IEEE communications surveys & tutorials* 18, 2 (2015), 1123–1152.

[29] Fengjiao He, Wenchuan Yang, Baojiang Cui, and Jia Cui. 2022. Intelligent fuzzing algorithm for 5g nas protocol based on predefined rules. In *2022 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 1–7.

[30] Hai Hu, Chang-Hai Jiang, and Kai-Yuan Cai. 2008. Adaptive software testing in the context of an improved controlled Markov chain model. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*. IEEE, 853–858.

[31] joy.zhou, chowyu, dependabot[bot], Lucas Vieira, spit4520, gerdstolpmann, muXxer, Marc Magnin, Rajiv Shah, Thomas, TrickTt, Luca Moser, Ron Evans, Yog, chujiangke, foosinn, Jason, YangYuDong, winglq, Michael Stapelberg, Marc Magnin, Lijin, Husy, Jayden, Giovanni Rosa, Gary Barnett, and Aleksey Myasnikov. 2025. fhmq/hmq. https://github.com/fhmq/hmq. https://github.com/fhmq/hmq

[32] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of LLM-based explainable fault localization. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1424–1446.

[33] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.

[34] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. 2019. Polar: Function code aware fuzz testing of ics protocol. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–22.

[35] Xiaoyue Ma, Lannan Luo, and Qiang Zeng. 2024. 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)*. 4783–4800.

[36] OwnTone maintainers. 2025. OwnTone — owntone.github.io. https://owntone.github.io/owntone-server/. [Accessed 19-07-2025].

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

[38] Roberto Natella. 2022. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering* 27, 7 (2022), 191.

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

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

[41] Gaganjeet Singh Reen and Christian Rossow. 2020. DPIFuzz: a differential fuzzing framework to detect DPI elusion strategies for QUIC. In *Proceedings of the 36th Annual Computer Security Applications Conference*. 332–344.

[42] Mengfei Ren, Xiaolei Ren, Huadong Feng, Jiang Ming, and Yu Lei. 2021. Z-fuzzer: Device-agnostic fuzzing of zigbee protocol implementation. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 347–358.

[43] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. 2020. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *29th USENIX Security Symposium (USENIX Security 20)*. 19–36.

[44] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 951–971.

[45] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.

[46] Juraj Somorovsky. 2016. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 1492–1504.

[47] Dipa Soni and Ashwin Makwana. 2017. A survey on mqtt: a protocol of internet of things (iot). In *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*, Vol. 20.

[48] George Thomas. 2008. Introduction to the modbus protocol. *The Extension* 9, 4 (2008), 1–4.

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

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

[51] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*. 995–1007.

[52] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[53] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 709–735.

[54] Xiaohan Zhang, Cen Zhang, Xinghua Li, Zhengjie Du, Bing Mao, Yuekang Li, Yaowen Zheng, Yeting Li, Li Pan, Yang Liu, et al. 2024. A survey of protocol fuzzing. *Comput. Surveys* 57, 2 (2024), 1–36.

[55] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. 2019. SeqFuzzer: An industrial protocol fuzzing framework from a deep learning perspective. In *2019 12th IEEE Conference on software testing, validation and verification (ICST)*. IEEE, 59–67.