Evaluating the Effectiveness of MQTT Fuzzing Tools Using Mutation Testing in IoT Systems

Darla Garcia Alves
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo
São Carlos, Brazil
darlagalves@usp.br

Bruno Ely Reis Garcia
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo
São Carlos, Brazil
bruno.erg@usp.br

Márcio Eduardo Delamaro
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo
São Carlos, Brazil
delamaro@icmc.usp.br

Simone do Rocio Senger de Souza
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo
São Carlos, Brazil
srocio@icmc.usp.br

Abstract

Internet of Things (IoT) systems are widely used in residential and industrial domains, with MQTT as a common communication protocol. However, testing MQTT-based systems remains challenging due to their event-driven behavior, preconfigured topics, asynchronous processing, and frequent silent failures. Although several MQTT fuzzing tools have been proposed, there is limited empirical evidence comparing their effectiveness in realistic IoT platforms using adequacy criteria beyond crash detection. This paper presents a controlled experiment evaluating MQTT fuzzing tools using Home Assistant, an open-source home automation platform deployed in a reproducible Docker-based environment. Effectiveness is assessed through mutation testing, with artificial faults injected into the source code and a stratified oracle detecting crashes, internal exceptions, incorrect entity states, and silent behavioral divergences. The evaluation uses mutation score as the main metric, complemented by an MQTT-adjusted mutation score, killed-mutant overlap, and post-hoc analysis of surviving mutants. The results provide empirical evidence on the ability of each fuzzer to expose application-level faults and highlight limitations related to reachability, configuration-dependent execution paths, and oracle observability in MQTT-based IoT systems.

Keywords

Fuzzing Testing; MQTT Protocol; IoT Systems; Mutation Testing; Home Assistant

1 Introduction

The Internet of Things (IoT) has become a major technological trend across domains such as transportation, healthcare, agriculture, environmental monitoring, home automation, and smart manufacturing [20], accompanied by a rapid increase in connected devices. According to Tanweer Alam [2], more than 75 billion IoT devices are expected to be connected globally by 2025. Broader projections, such as those reported by Cisco, indicate that this number may reach 500 billion by 2030 [13].

Smart homes are a major IoT application domain, integrating devices such as sensors, appliances, lighting, and heating systems through automated communication. In this context, lightweight protocols are essential for resource-constrained devices. Among these protocols, MQTT (Message Queuing Telemetry Transport) is described as the most widespread IoT protocol [14].

Despite their widespread adoption, MQTT-based applications remain susceptible to implementation errors, protocol violations, and anomalous message processing, such as accepting invalid packets or misinterpreting malformed payloads. Their consequences can range from explicit failures, such as service unavailability or crashes, to subtler effects, such as internal state corruption, behavioral inconsistencies, and silent failures.

These limitations motivate testing approaches that evaluate robustness to invalid, unexpected, or malformed inputs. Fuzzing testing is an automated technique that generates and injects diverse data into a target system to identify implementation defects, security vulnerabilities, and robustness issues. When applied to MQTT-based IoT systems, fuzzing explores the application's responses to malformed packets, corrupted payloads, and inconsistent message sequences, revealing protocol interactions and internal states that are difficult to exercise through traditional testing.

In this context, this paper evaluates the effectiveness of different fuzzing tools applied to the MQTT protocol for fault detection in IoT systems. Thus, we conduct a controlled experiment in a simulated environment based on the Home Assistant platform, using MQTT-based communication. A specialized test harness ensures generated inputs reach the system's internal message-processing logic by targeting valid MQTT topics and injecting corrupted messages, invalid packets, inconsistent JSON payloads, and unexpected message sequences. The analysis focuses on mutation score, MQTT-adjusted mutation score, killed-mutant overlap, and post-hoc classification of surviving mutants.

2 Related Work

MQTT has attracted increasing attention in fuzzing research because of its widespread adoption in IoT systems. Several tools have been proposed to generate or mutate MQTT messages to reveal faults in protocol implementations. FUME applies a hybrid strategy based on Markov chains and execution feedback [15]. At the same time, MQTTGRAM uses grammar-based generation to produce syntactically valid MQTT packets and reduce premature rejection by the target [4]. More recent approaches, such as MBFuzzer, explore

multi-participant MQTT interactions and message dependencies in broker-based communication scenarios [18].

Although these studies contribute to MQTT robustness evaluation, most focus on brokers such as Mosquitto, EMQX, or Moquette, assessing protocol compliance, routing, state-machine behavior, or memory safety. Less attention has been given to application-level MQTT processing in realistic IoT platforms, where payload interpretation, entity-state updates, and event-driven automations may introduce different classes of failures. This limitation is particularly relevant in systems such as Home Assistant, which usually process only messages published to topics previously associated with configured entities.

Traditional fuzzing also relies heavily on code coverage and crash-based oracles, which may miss semantic or behavioral faults. In event-driven IoT applications, failures may not cause process termination, but instead appear as incorrect entity states, missing updates, improperly triggered automations, internal exceptions, or silent failures. Therefore, evaluating fuzzers in this context requires oracles capable of observing more than explicit crashes.

Mutation testing has been used as a complementary criterion for evaluating automated testing techniques by injecting artificial faults, known as mutants, into the source code. In fuzzing research, mutation testing provides a controlled way to assess whether generated inputs not only reach program locations but also expose observable behavioral differences. Recent studies have explored this relationship in different domains, such as cyber-physical systems [10], Java programs [19], and fuzz-target design [8].

3 Background

3.1 MQTT and IoT Systems

IoT connects physical objects through communication networks, enabling automatic data collection, processing, exchange, and control of physical environments. In this sense, IoT represents a significant evolution in how computational systems interact with the physical world, integrating sensors, actuators, software, and communication networks into intelligent, responsive ecosystems [17].

To enable large-scale communication, the MQTT protocol has become a leading solution in IoT systems. Unlike the traditional client-server architecture, MQTT adopts the publish/subscribe model [11]. This asynchronous, decoupled model allows publishers and subscribers to communicate without direct knowledge of each other [9]. In this scenario, publishers correspond to devices, sensors, or applications responsible for generating data and publishing them to the network, whereas subscribers are devices, actuators, or applications that subscribe to receive data of specific interest [6].

MQTT uses a broker to manage connections and route publisher messages to the corresponding subscribers [1]. This routing is performed through topics (*topics*), which function as hierarchical categories, typically structured like directory paths, as in `home/livingroom/temperature`. Thus, a publisher sends its message to a specific topic, and the clients subscribed to that topic receive the corresponding updates [9].

MQTT is particularly suitable for IoT systems because it addresses characteristic requirements of this domain, such as low resource consumption, lightweight communication, and operation in unstable network environments. Its minimum fixed header is only 2 bytes, reducing bandwidth consumption and requiring little processing, making it appropriate for constrained devices, such as battery-powered sensors. In addition, the protocol was designed to operate over unreliable or high-latency networks, a characteristic frequently found in IoT scenarios [3].

3.2 Home Assistant and MQTT Processing

Home Assistant (HA) is an open-source smart-home platform that integrates devices, sensors, actuators, and services to coordinate events and actions in an automated domestic environment. In this work, Home Assistant is adopted as the target system because it is a realistic, widely used IoT application in which MQTT message processing occurs at the application level rather than only at the broker level. This choice shifts the traditional focus of protocol testing, usually centered on the MQTT server, toward the client application's logic, an aspect that remains underexplored in systematic fuzzing evaluations for IoT [6].

The MQTT integration in Home Assistant follows the *publish/-subscribe* model described previously. In this model, HA acts as an MQTT client that connects to a broker and subscribes to topics previously associated with entities configured in the system. These entities represent devices or monitored states, such as temperature sensors, switches, lights, or actuators. When an IoT device publishes a message to a monitored topic, the broker forwards it to Home Assistant, which then extracts and interprets the payload, often in formats such as JSON or plain text. If the processing is successful, the internal state of the corresponding entity is updated in the platform's event bus.

State updates can trigger predefined rules and automations in response to observed conditions. For example, a change in the temperature sensor's value may trigger an automation rule that turns on a ventilation system or publishes a new MQTT command to another device [4]. Thus, processing a single MQTT message can trigger cascading effects across different components of the application, making the system's behavior dependent on the message's structure and the configuration context of entities and automations.

From a fuzz testing perspective, this characteristic poses a relevant challenge. Home Assistant does not generically process every MQTT message that passes through the broker. Instead, the application tends to react only to messages published to topics it has previously subscribed to and associated with configured entities. Fuzzer-generated messages sent to unrelated topics may therefore be ignored before reaching decoding, validation, state-update, or automation logic. This behavior may lead to a false perception of robustness, since malformed inputs may not effectively exercise the system's internal logic.

This restriction justifies the need for a specialized harness in the experiment. The harness serves as an adaptation layer between the fuzzing tools and Home Assistant, ensuring that the generated inputs are directed to topics that the application effectively monitors. Also, it enables prior configuration of the required entities, controls the topics used in the experiment, and correlates injected messages with the observed effects.

3.3 Fuzzing Testing

Fuzzing testing is a dynamic analysis technique used to identify failures and vulnerabilities in software by executing the system with a large number of inputs, including malformed, unexpected, or randomly mutated data. By exposing the SUT to input conditions that developers may not have anticipated, fuzzing can reveal crashes, exceptions, incorrect behaviors, and security vulnerabilities [12].

Unlike traditional testing strategies, which usually rely on predefined test cases and expected outputs, fuzzing follows a more exploratory approach. Its objective is to observe how the software behaves under unexpected inputs, particularly in execution paths that are rarely exercised by conventional tests. Compared with other vulnerability discovery techniques, such as manual code inspection or reverse engineering, fuzzing has the advantage of being highly automated and scalable, enabling extensive test campaigns with limited human intervention [7].

The term *fuzzer* refers to a program that generates, mutates, and submits test inputs to the SUT to evaluate its behavior.

Fuzzers differ in the amount of information they use and in how they generate inputs. Black-box fuzzers rely mainly on externally observable behavior, grey-box fuzzers use partial feedback such as logs, coverage, or runtime responses, and generation-based fuzzers rely on grammars, message templates, or protocol specifications. In protocol fuzzing, structured or syntactically valid inputs are often important because they are more likely to pass initial parsing stages and reach deeper logic [4, 7].

In the context of MQTT, the effectiveness of a fuzzer depends not only on its ability to generate malformed packets but also on its capacity to produce inputs that are actually processed by the target application. This requirement is particularly important in MQTT-based IoT systems, where messages are routed according to topics and where application-level components may ignore messages published to unknown or unsubscribed topics. Therefore, fuzzing MQTT applications requires considering both the protocol structure and the application-specific configuration that determines which messages reach the internal processing logic.

Recent research on MQTT fuzzing has proposed different approaches for testing brokers, protocol implementations, and IoT applications. Some tools focus primarily on the broker infrastructure, evaluating how MQTT servers handle malformed packets, invalid state transitions, or multi-participant interactions. For example, tools such as MBFuzzer target the behavior of MQTT brokers in complex communication scenarios, making them relevant for assessing protocol compliance and broker robustness [18]. However, this focus differs from the objective of the present work, which evaluates how an end-user IoT application reacts to corrupted or unexpected MQTT payloads at the application level.

4 Experimental Study

We compare MQTT fuzzing tools according to their ability to detect faults in IoT systems through the following research question:

Research question: How effective are the evaluated MQTT fuzzing tools at fault detection, as measured by mutation score?

To answer the research question, the experiment is guided by statistical hypothesis testing. The effectiveness of each fuzzing tool is measured using the MQTT-adjusted mutation score, and the hypotheses are defined as follows:

- **Null Hypothesis (H_0):** There is no statistically significant difference in the effectiveness of the evaluated MQTT fuzzing tools. For any pair of fuzzers under comparison, the probability of a mutant being killed by the first fuzzer and surviving the second is equal to the probability of it being killed by the second and surviving the first, i.e., $H_0 : p_{ij} = p_{ji}$.
- **Alternative Hypothesis (H_1):** There is a statistically significant difference in the effectiveness of the evaluated MQTT fuzzing tools. For a given pair of fuzzers, the probability of a mutant being killed by the first and surviving the second is not equal to the reverse scenario, i.e., $H_1 : p_{ij} \neq p_{ji}$.

Here, p_{ij} denotes the probability that a mutant is killed by $Fuzzer_i$ but survives $Fuzzer_j$. Rejecting H_0 indicates that the evaluated tools do not present equivalent effectiveness in detecting mutants under the experimental conditions defined in this study.

4.1 Experimental Design

The experiment was conducted as an *in silico* study in a controlled and reproducible environment. The target system was Home Assistant, executed in a Docker container and configured with an MQTT sensor entity subscribed to a valid MQTT topic. The experiment does not aim to evaluate the entire Home Assistant codebase or the MQTT broker implementation. Instead, it focuses on the subset of Home Assistant's application-level logic that is exercised after MQTT messages are delivered to configured entities.

Specifically, the mutation campaign targets two modules associated with the MQTT data-processing flow: `components/mqtt/sensor.py` and `template.py`. The first module was selected because it is directly involved in processing MQTT payloads and updating the state of sensor entities. The second was selected because template processing is used by Home Assistant to interpret, transform, and derive values from incoming data before they affect the application state. Therefore, these modules represent two relevant points in the application-level processing pipeline triggered by MQTT inputs: payload-to-state conversion and template-based value interpretation.

This scope also separates the MQTT communication infrastructure from the application logic under evaluation. The MQTT broker is used only as a routing component responsible for delivering messages to subscribed clients. Faults are not injected into the broker or into the MQTT protocol implementation itself, but into Home Assistant modules that process the data after message delivery. Thus, the experiment evaluates whether fuzzing-generated MQTT inputs are capable of revealing faults in the application logic reached through valid MQTT topics and configured entities.

The fuzzing campaign was organized into two phases. In the first phase, each fuzzing tool was executed against the original, non-mutated version of Home Assistant in order to generate a tool-specific corpus of MQTT payloads. The collected corpora preserve the inputs produced by each tool and therefore represent the input-generation behavior of BooFuzz [16], FUME [15], and Scapy [5] under the same experimental configuration.

We used Scapy as a packet/payload generation baseline by implementing a custom MQTT-oriented generator on top of its packet-manipulation primitives. In this configuration, Scapy produced MQTT payload candidates from the same seed set and under the same campaign duration as the other tools. Therefore, the comparison treats Scapy not as an off-the-shelf MQTT fuzzer, but as a programmable generation baseline integrated into the same harness.

In the second phase, the collected corpora were replayed deterministically against each mutant generated by `mutmut` in the selected target modules. Instead of executing the fuzzers online during the mutation campaign, the experiment replayed the previously generated corpora under controlled conditions. This design reduces the influence of execution-time variability, network timing, and stochastic behavior when evaluating each mutant. As a result, each mutant is exercised with a fixed and reproducible input sequence associated with each fuzzer.

Because Home Assistant processes only MQTT messages published to topics associated with configured entities, the harness ensures that all replayed inputs are published to the monitored topic of the configured sensor entity. A semantic replay layer transforms each raw payload into a valid JSON message that the sensor configuration accepts. This transformation is deterministic and uniformly applied to all fuzzers, preserving differences among the generated corpora while ensuring the resulting messages reach Home Assistant's application-level processing logic.

Each campaign follows the same controlled workflow. First, the Home Assistant container is restarted to load the current mutant. Then, the harness verifies that the mutated source file is mounted inside the container and that the MQTT broker is reachable. Next, the semantic replay mechanism publishes the corpus associated with the evaluated fuzzer to the monitored MQTT topic. During and after execution, the harness collects evidence from the Home Assistant API, container status, and system logs. Finally, the observed behavior is compared against the reference behavior previously recorded for the original version of the system.

4.2 Mutation Dataset

Mutants were generated with mutmut [10], a source-level mutation testing tool suitable for Home Assistant's Python codebase.

The generated mutants represent small syntactic modifications intended to simulate realistic programming faults. In this study, *mutmut* was executed with its default mutation configuration, without explicitly enabling or disabling mutation types. Thus, the tool applied its standard transformations to the selected target files, including changes to conditional expressions, boolean and relational operators, constants, literals, strings, and statement-level constructs when applicable.

To illustrate the mutation process, Listings 1 and 2 present an example of a mutant generated by *mutmut* in `sensor.py`. In this case, the mutation changes a membership condition in a validation routine, replacing `in` with `notin`. This modification alters the logic used to validate MQTT sensor configuration parameters and may cause the system to accept invalid configurations or reject valid ones. Therefore, this mutant represents a realistic logical fault at the application level.

Listing 1: Original code in `sensor.py`.

```
def validate_sensor_state_class_config(config: ConfigType) -> ConfigType:
    """Validate the sensor state class config."""
    if (
        CONF_LAST_RESET_VALUE_TEMPLATE in config
        and (state_class := config.get(CONF_STATE_CLASS)) != SensorStateClass.
               TOTAL
    ):
        raise vol.Invalid(...)
```

Listing 2: Mutated version generated by *mutmut*.

```
def validate_sensor_state_class_config(config: ConfigType) -> ConfigType:
    """Validate the sensor state class config."""
    if (
        CONF_LAST_RESET_VALUE_TEMPLATE not in config
        and (state_class := config.get(CONF_STATE_CLASS)) != SensorStateClass.
               TOTAL
    ):
        raise vol.Invalid(...)
```

This mutant introduces a semantically relevant change in MQTT sensor validation; an observable divergence from the original behavior causes it to be classified as killed.

4.3 Experimental Environment

The experimental architecture consists of five main components. The first is the *Home Assistant* instance, which processes incoming MQTT messages, updates the state of configured entities, and executes associated automations. The second component is the MQTT broker, which acts as an intermediary in the communication between publishers and subscribers, forwarding messages to clients subscribed to the corresponding topics. The third component consists of fuzzing tools that generate invalid, unexpected, or mutated inputs. The fourth component is the harness, which integrates the fuzzers into the MQTT environment, ensuring that the generated inputs are published to topics that are effectively monitored by Home Assistant. Finally, the monitoring module is responsible for collecting execution evidence, including logs, exceptions, state changes, explicit failures, and possible incorrect behaviors.

In addition, the environment includes the *mutmut* mutation testing tool for generating faulty versions of the target modules.

The environment was implemented in Docker containers to ensure isolation among components, control dependencies, ease restarts, and improve reproducibility of the experimental campaigns. Each execution can be started from a known configuration, allowing the state of Home Assistant, the MQTT broker, and auxiliary services to be restored before evaluating each fuzzer or mutant.

The auxiliary scripts for the experiment, including routines for executing the fuzzers, controlling the campaigns, collecting logs, and processing results, were run in an isolated Python environment via a *virtual environment* (`venv`).

4.4 Fuzzing Campaign Design

The fuzzing campaign used different tools to generate MQTT-oriented payload corpora for application-level replay.

To ensure a fair comparison among the evaluated tools, we ran all campaigns under equivalent experimental conditions. Each tool used the same execution environment, Home Assistant configuration, MQTT broker, configured entities, initial set of *seeds*, execution interval, and computational infrastructure.

Each fuzzer generates a tool-specific corpus, which is later semantically normalized and replayed under deterministic conditions. From these messages, the fuzzers may generate invalid, unexpected, or mutated variations, including corrupted payloads, inconsistent JSON messages, values outside the expected domain, malformed packets, and unexpected message sequences.

The collected metrics include the number of killed and surviving mutants, the mutation score, the total campaign execution time, the size of the duplicated corpus, and the semantic trace outcomes produced during replay. Although the infrastructure records the oracle level responsible for killing a mutant, this version does not provide a precise time-to-failure measurement per mutant. Stability was assessed indirectly through campaign completion, absence of unexpected crashes in the control execution, and consistency of the semantic replay results. We did not use code coverage as a primary metric in the final campaigns reported in this version.

Table 1: Experimental configuration and replay parameters.

Parameter	Value
Target system	Home Assistant 2024.6.0
Target modules	`components/mqtt/sensor.py`; `helpers/template.py`
Mutation tool	Mutmut 2.5.1
Python version	Python 3.12.3
Execution environment	WSL Ubuntu; Docker containers
Home Assistant image	`ghcr.io/home-assistant/home-assistant:2024.6.0`
MQTT broker	Eclipse Mosquitto 2.1.2
Mosquitto image	`eclipse-mosquitto:latest`
Evaluated fuzzers	BooFuzz, FUME, Scapy
MQTT host/port	`127.0.0.1:1883`
MQTT topic	`homeassistant/sensor/temp`
Observed entity	`sensor.sensor_fuzzing`
Home Assistant API	`http://127.0.0.1:8123`
Corpus generation duration	60s for BooFuzz, FUME, and Scapy
Corpus deduplication	Duplicate payloads removed before replay
Replay limit	100 payloads per mutant execution
Replay delay	0.05s between payloads
Oracle	Semantic trace comparison of Home Assistant entity states
Baseline trace	Generated from the original, non-mutated Home Assistant version for each fuzzer corpus

4.5 The Role of the Harness and Points of Failure

The harness plays a central role in the experiment, serving as an adaptation layer between the fuzzing tools and the target system. Its function is not limited to forwarding inputs generated by the fuzzers; it also includes knowledge of the experimental environment's valid MQTT structure. This characteristic is essential in the context of Home Assistant, since the application tends to process only messages published to topics previously associated with configured entities.

Unlike a traditional network fuzzing scenario, in which the main objective is to send malformed packets directly to a network service, this experiment requires the generated inputs to effectively reach the application logic responsible for processing MQTT messages. Therefore, the harness must ensure that the data produced by the fuzzers are directed to topics monitored by Home Assistant, increasing the likelihood that the mutated payloads are interpreted by the internal routines responsible for decoding, validation, and state updates.

In this context, the harness is responsible for identifying the valid MQTT topics used by the configured entities, directing messages to effectively monitored topics, prioritizing mutations in MQTT payloads, and, when necessary, performing controlled mutations in topics. In addition, it can serve as an instrumentation mechanism for the environment, enabling the generated inputs to be correlated with observed effects in the system, such as state changes, internal exceptions, or anomalous behavior.

This instrumentation is particularly relevant because failures in Home Assistant do not always manifest as *crashes* or explicit service interruptions. The platform's MQTT integration includes internal exception-handling mechanisms that may discard invalid or unexpected inputs, record them only in logs, or cause subtle inconsistencies in the application's state. Thus, the evaluation of fuzzer effectiveness must consider not only abrupt failures but also less evident manifestations of incorrect behavior.

Observable failures in this experiment may include error messages in logs, internal exceptions handled by the system, inconsistent entity states, silent failures during processing, incorrectly triggered automations, and loss or corruption of internal state. These behaviors are especially relevant in IoT applications, where the absence of a *crash* does not necessarily imply that the system has processed the input correctly.

Among the main critical points in Home Assistant that are susceptible to failures are the *parsing* of malformed JSON payloads, invalid data type conversions, inadequate payload validation, incorrect handling of MQTT states, chained execution of automations, and the processing of Jinja2 *templates*[1]. These points encapsulate a significant portion of the application logic executed during MQTT message processing and therefore represent relevant targets for mutant-based evaluation. Given these characteristics, the experiment's monitoring module must go beyond simple detection of *crashes*. Evidence collection should include log analysis, exception tracing, verification of internal states, observation of automation triggering, and identification of silent failures. This approach enables a more comprehensive assessment of whether the inputs generated by the fuzzers can reveal behavioral divergences introduced by the mutants in the system's source code.

In the implemented experiment, the harness is responsible for more than invoking the fuzzing tools. It orchestrates the complete interaction between mutation testing, MQTT communication, Home Assistant, and the test oracle. For each mutant, the harness restarts the Home Assistant container, ensures that the mutated source code is executed inside the container, configures the fuzzer-specific execution context, triggers replay of the corresponding corpus, and records the resulting behavior of the target system.

A negative-control adapter, referred to as NOOP, was introduced to validate the harness. This adapter does not generate or replay fuzzing inputs. Therefore, any mutant killed during a NOOP campaign would indicate that the harness or the oracle is introducing false positives independently of the fuzzers. In the final configuration, the NOOP campaign killed zero mutants in both evaluated modules, providing evidence that detected divergences are associated with inputs derived from the fuzzing tools rather than with the harness or oracle alone.

[1]Jinja2 is a template engine developed for Python and widely used by Home Assistant for extracting, transforming, and formatting data from entities and MQTT messages. The engine allows the execution of expressions and data manipulation rules, making it a potential point of failure when subjected to unexpected or malformed inputs.

4.6 Test Oracle

The definition of a test oracle constitutes one of the main challenges of this experiment. In software testing, an oracle corresponds to the mechanism used to determine whether the observed behavior of the system under test is consistent with the expected behavior for a given input. In traditional fuzzing campaigns, this oracle is often implicit and relies on abrupt failures, such as *crashes*, execution aborts, or unhandled exceptions.

In the context of Home Assistant, this approach is insufficient. The platform's MQTT integration includes internal exception handling and fault-tolerance mechanisms, so invalid or unexpected inputs do not always interrupt the main process. In many cases, incorrect behaviors may manifest only as errors recorded in logs, inconsistent entity states, absence of state updates, incorrectly triggered automations, or silent failures during message processing.

For this reason, this experiment adopts a stratified test oracle, organized into different levels of severity and observability. The objective is to extend detection capabilities beyond explicit failures, enabling identification of subtle behavioral divergences introduced by the mutants. The considered levels are described as follows:

- **Level 1 — Process Termination**: corresponds to the occurrence of an unexpected termination of Home Assistant, an unplanned service restart, or application unavailability;
- **Level 2 — Internal Exception**: corresponds to the occurrence of exceptions, *stack traces*, or error messages recorded in the system logs, even when the main process remains running;
- **Level 3 — Incorrect Entity State**: corresponds to an incorrect update of the internal state of MQTT entities, including invalid, inconsistent, or domain-incompatible values;
- **Level 4 — Silent Failure**: corresponds to the absence of the expected behavior after receiving a structurally valid input, including the absence of state updates, failure to trigger automations, or behavioral inconsistencies not explicitly recorded as errors.

The identification of failures at Levels 3 and 4 requires an explicit definition of the expected system behavior. For this purpose, a reference trace, referred to as the *baseline*, is established for each fuzzer corpus by executing the original version of Home Assistant without injected mutations. During this reference campaign, valid and controlled MQTT messages were published to the topics monitored by the configured entities. The observed behavior, including entity states, automation execution, and log records, was stored for subsequent comparison.

In subsequent campaigns executed on mutated versions of the system, the monitoring module compared the observed behavior with that recorded in the *baseline*. Observable divergences between the original and mutated versions were interpreted as evidence of failure. Thus, in the context of *mutation testing*, a mutant was considered killed when the *fuzzer* produced an input that caused an observable divergence at least one of the established oracle levels. If no divergence is observed throughout the campaign, the mutant is classified as surviving.

We also analyzed invalid or non-executable mutants during the campaign validation process. These mutants correspond to source-code modifications that prevented Home Assistant from starting correctly, made the mutated module unavailable, or made the experimental campaign invalid regardless of the fuzzer input. Since such mutants do not allow a valid comparison among fuzzing tools, we excluded them from the mutation-score denominator.

We did not attempt to formally identify all equivalent mutants, since proving behavioral equivalence for all possible inputs is generally infeasible. Therefore, surviving mutants are not treated as equivalent by default. Instead, they are analyzed through the post-hoc classification presented in Section 5.5, which distinguishes cases related to low reachability, configuration paths not exercised, silent handling, oracle limitations, and generic framework logic.

The functional oracle is based on semantic traces. For each fuzzer, a baseline trace is first generated by replaying its corpus against the original version of Home Assistant, without mutations. This trace records the sequence of states observed in the configured MQTT sensor after each replayed input. During mutation testing, the same corpus is replayed against each mutant, producing an observed trace. A mutant is classified as killed when the observed trace diverges from the baseline trace for the same fuzzer. This comparison allows the oracle to detect behavioral differences that do not necessarily result in crashes or explicit exceptions.

The oracle remains stratified. Level 1 detects process-level failures, such as container termination or service unavailability. Level 2 detects internal exceptions and relevant MQTT-related errors in logs. Levels 3 and 4 are implemented by comparing semantic traces to capture incorrect entity states, missing updates, or silent behavioral divergences. This design is particularly important for Home Assistant, where invalid MQTT inputs may be handled internally and may not lead to abrupt failures.

4.7 Oracle Stability Check

Although the NOOP control verifies that the harness does not kill mutants when no input is produced, it does not exercise the asynchronous behavior involved in replaying MQTT messages, updating entities, processing logs, timers, and eventual consistency effects. Therefore, we performed an additional stability check for the semantic oracle.

For each valid fuzzer corpus, we replayed it five times against the original, unmodified Home Assistant version. Before each repetition, the Home Assistant container was restarted, and the resulting semantic trace was collected from the observed entity state sequence. We then compared the traces within each fuzzer corpus using a hash of the normalized trace.

Table 2 shows that all repeated baseline replays produced identical semantic traces. BooFuzz produced traces with 11 observations in all five runs, FUME produced traces with 205 observations in all five runs, and Scapy produced traces with 711 observations in all five runs. In all cases, only one unique trace hash was observed. This result suggests that the semantic oracle is stable under actual replay conditions, not only under the inactive NOOP control.

Table 2: Repeated baseline replay stability check.

Fuzzer	Repetitions	Trace length	Stable
BooFuzz	5	11	Yes
FUME	5	205	Yes
Scapy	5	711	Yes

4.8 Experiment Variables and Data Collection

The definition of the experimental variables is essential to ensure the organization of the study and the comparability among the evaluated tools. In this experiment, the variables were classified as independent, dependent, and controlled, as described below.

The **independent variables** are the factors manipulated or observed to analyze their effects on experimental results. In this study, the main factors considered are the fuzzing tool used and the set of mutants applied to the target system. The fuzzing tool is the primary factor for comparison, whereas the mutants define the artificial defects used to evaluate each tool's detection capability.

The **dependent variables** are the metrics used to evaluate the tools' effectiveness. The main metric is the mutation score, which measures the proportion of killed mutants among all valid mutants. In addition, the number of detected failures, the time required to detect each failure, and the stability of the executions were collected. These metrics allow the analysis not only of each fuzzer's ability to reveal defects, but also of its temporal efficiency and reliability during the experimental campaigns.

The **controlled variables** are the elements kept constant across executions to minimize external interference and ensure a fair comparison among the tools. These variables include the execution environment, the Home Assistant configuration, the MQTT broker used, the initial set of seeds, the campaign duration, the computational infrastructure, and the set of entities configured in the system. By keeping these factors constant, the study aims to ensure that the observed differences in the results are primarily attributable to the characteristics of the evaluated fuzzing tools.

During data collection, the monitoring module has recorded information on failures, internal exceptions, entity state changes, silent failures, detection times, and mutant execution results. These data were subsequently used to calculate the experimental metrics and support the comparative analysis among the tools.

In addition to the variables described above, the final campaign controls the replay configuration used to exercise each mutant. The replay limit, replay delay, MQTT topic, configured entity, Home Assistant configuration, Docker image, broker configuration, and semantic normalization rule are kept constant across all fuzzers. The only input that varies across tools is the corpus generated by each fuzzer.

The collected data include, for each fuzzer and seed, the number of generated corpus entries, the number of unique semantic states, the killed and surviving mutants, the mutation score, the total execution time, the exit status of the mutation campaign, and the hash of each mutant diff. These data are used to verify that all fuzzers were evaluated against the same mutant set and to support comparisons of mutation scores.

5 Results

This section describes the main results of the mutation-testing campaigns conducted on the Home Assistant IoT system.

5.1 Validation of the Experimental Harness

Before comparing the fuzzing tools, we executed a negative-control campaign using the NOOP adapter. The NOOP adapter does not generate or replay fuzzing inputs; therefore, any killed mutant in this campaign would indicate a false positive introduced by the harness, the replay infrastructure, or the test oracle.

The NOOP campaign was executed for both evaluated modules, sensor.py and template.py. In both cases, the NOOP adapter killed zero mutants. As shown in Table 3, the false-positive rate was 0.00% for both modules. This result provides evidence that the observed mutant kills in the fuzzing campaigns are caused by fuzzer-derived inputs rather than by deterministic setup actions, container restarts, baseline comparison artifacts, or the oracle itself.

Table 3: Negative-control results using the NOOP adapter.

Module	Killed	Total	False-positive rate (%)
sensor.py	0	59	0.00
template.py	0	602	0.00

5.2 Corpus Generation and Campaign Validity

Before the mutation campaign, we executed each tool to generate a tool-specific MQTT payload corpus. BooFuzz, FUME, and Scapy produced non-empty deduplicated corpora and valid semantic baseline traces, and were therefore included in the mutation-score comparison. MQTTGRAM and the MitM/Polymorph adapter did not produce replayable payload corpora under the evaluated configuration and were excluded from the mutation campaign. Therefore, the reported mutation scores refer only to BooFuzz, FUME, and Scapy.

Table 4: Corpus generation results and campaign inclusion.

Tool	Duration	Unique payloads	Included	Reason
BooFuzz	60s	11	Yes	Replayable semantic corpus generated
FUME	60s	205	Yes	Replayable semantic corpus generated
Scapy	60s	711	Yes	Replayable semantic corpus generated
MQTTGRAM	120s	0	No	No payloads captured in the expected replay format
MitM/Polymorph	120s	0	No	No replayable artifacts generated under the configured harness

5.3 Mutation Testing Effectiveness

Table 5 presents the mutation scores for the valid campaigns. The MQTT replay covered 93 lines in `sensor.py` and 636 lines in `template.py`, resulting in 59 and 602 MQTT-reachable mutants, respectively.

Table 5: MQTT-adjusted mutation scores.

Module	**Tool**	**Killed**	**Observed**	MS_{MQTT}
`sensor.py`	BooFuzz	17	59	28.81%
`sensor.py`	FUME	19	59	32.20%
`sensor.py`	Scapy	17	59	28.81%
`template.py`	BooFuzz	127	602	21.10%
`template.py`	FUME	127	602	21.10%
`template.py`	Scapy	127	602	21.10%

Considering both modules, FUME killed 146 out of 661 observed mutants, corresponding to a combined MQTT-adjusted mutation score of 22.09%. BooFuzz and Scapy each killed 144 out of 661 observed mutants, corresponding to 21.79%. Therefore, the overall difference among the valid tools was small.

5.3.1 Sensor Module Results. The first mutation-testing campaign was conducted on the `components/mqtt/sensor.py` module. This module was selected because it is directly involved in MQTT payload processing and in updating the state of the configured sensor entity. In total, 126 mutants were generated for this module. After applying the MQTT-adjusted criterion, 59 mutants were included in the observed MQTT mutation space.

Before comparing the fuzzing tools, each campaign was validated according to whether the tool generated a non-empty input corpus and whether a corresponding semantic baseline trace could be produced. BooFuzz, FUME, and Scapy generated valid artifacts and were included in the mutation-score comparison.

In the observed MQTT mutation space, FUME achieved the highest score, killing 19 out of 59 mutants, corresponding to 32.20%. BooFuzz and Scapy killed 17 mutants each, corresponding to 28.81%. Thus, FUME exposed two additional behavioral divergences when compared with BooFuzz and Scapy.

These results indicate that, in the `sensor.py` campaign, FUME showed a slightly broader fault-detection capability. However, the difference among the tools remained small, suggesting a descriptive advantage for FUME rather than strong evidence of substantially superior effectiveness.

5.3.2 Template Module Results. The second mutation-testing campaign was conducted on the `template.py` module, which is relevant because template processing is part of the application-level logic that Home Assistant uses to interpret, transform, and derive values from incoming data. This module contains a substantially larger mutation space than `sensor.py`: in total, 1452 mutants were generated. After applying the MQTT-adjusted criterion, 602 mutants were included in the observed MQTT mutation space.

In this campaign, the valid fuzzing tools presented identical results. BooFuzz, FUME, and Scapy each killed 127 out of 602 observed mutants, corresponding to a mutation score of 21.10%. Unlike the `sensor.py` campaign, no tool killed additional mutants beyond those detected by the others.

This result indicates that, for the current template-processing scenario, the evaluated fuzzers exercised essentially the same observable behavior. The `template.py` campaign does not provide evidence of a relevant difference in effectiveness among BooFuzz, FUME, and Scapy. Instead, it highlights that the configured template scenario exposes only a subset of the module's mutation space.

Overall, the MQTT-adjusted mutation score shows that the evaluated fuzzers had similar effectiveness within the observed MQTT mutation space. Considering both modules, FUME killed 146 out of 661 observed mutants, corresponding to 22.09%, while BooFuzz and Scapy each killed 144 out of 661, corresponding to 21.79%. Therefore, the relative difference among the valid tools remained small. FUME showed a slight advantage in `sensor.py`, while all tools achieved identical results in `template.py`.

5.4 Fault Detection Overlap and Statistical Significance

5.4.1 Killed-Mutant Overlap Analysis. In addition to the MQTT-adjusted mutation score, we analyzed the overlap among the sets of mutants killed by each valid fuzzing campaign within the observed MQTT mutation space. This analysis is important because similar mutation scores may indicate either that the fuzzers detected the same faults or that they killed different subsets of mutants with comparable sizes.

For each pair of fuzzers, we computed the number of commonly and exclusively killed mutants and the Jaccard similarity coefficient, defined as $J(A, B) = |A \cap B|/|A \cup B|$.

Table 6: Paired comparison using McNemar's exact test.

Comparison	Both	Only A	Only B	Neither	p
sensor.py					
FUME–BooFuzz	17	2	0	40	0.500
FUME–Scapy	17	2	0	40	0.500
BooFuzz–Scapy	17	0	0	42	–
template.py					
FUME–BooFuzz	127	0	0	475	–
FUME–Scapy	127	0	0	475	–
BooFuzz–Scapy	127	0	0	475	–

The overlap analysis showed that the evaluated fuzzers killed highly similar sets of mutants. In the `sensor.py` campaign, BooFuzz and Scapy killed the same 17 mutants, while FUME killed these same mutants and additionally killed mutants 6 and 46. This resulted in a Jaccard similarity of 1.00 between BooFuzz and Scapy and 0.89 between FUME and each of the other two tools.

In the `template.py` campaign, the overlap was complete. BooFuzz, FUME, and Scapy killed exactly the same 127 mutants, resulting in a Jaccard similarity of 1.00 for all pairwise comparisons. Therefore, although the fuzzers generated different corpora, the observable behavior exposed by the current template-processing scenario was the same for all valid tools.

These results indicate that the evaluated fuzzers exercised a highly overlapping portion of the observed MQTT mutation space under the current experimental configuration. Thus, the main finding is not a clear ranking among tools, but rather that application-level MQTT fuzzing in Home Assistant is strongly constrained by the configured entity, the semantic replay mechanism, and the oracle's observability.

5.4.2 Paired Statistical Comparison. Since all valid fuzzing tools were evaluated against the same observed MQTT mutation space, we performed a paired comparison of killed and non-killed mutants. For each pair of fuzzers, each mutant was classified into one of four categories: killed by both fuzzers, killed only by the first fuzzer, killed only by the second fuzzer, or killed by neither fuzzer.

We used McNemar's exact test to compare the discordant pairs, that is, mutants killed by only one of the two fuzzers. This test is suitable for paired binary outcomes and avoids relying only on aggregate mutation scores.

Table 6 presents the paired comparison results within the observed MQTT mutation space. In the `sensor.py` campaign, FUME killed all mutants detected by BooFuzz and Scapy and additionally killed two mutants. However, the number of discordant pairs was very small, and McNemar's exact test did not indicate a statistically significant difference between FUME and the other tools. Therefore, the advantage observed for FUME in this module should be interpreted as a descriptive difference rather than as strong evidence of superior effectiveness.

In the `template.py` campaign, there were no discordant pairs among the valid fuzzers. BooFuzz, FUME, and Scapy killed exactly the same set of 127 mutants. Consequently, McNemar's test was not informative for this module, and the result indicates complete agreement among the tools for the killed-mutant set.

Overall, the paired analysis reinforces that the evaluated fuzzers exercised a highly overlapping portion of the observed MQTT mutation space under the current experimental configuration. The results do not provide statistical evidence of a substantial difference in effectiveness among the valid tools.

5.5 Post-hoc Mutant Classification

To better interpret the low mutation scores, we performed a post-hoc classification focused only on mutants related to the MQTT scenario exercised in the experiment. This analysis considers the MQTT-observed mutation space, which contains 59 mutants in `sensor.py` and 602 mutants in `template.py`. Mutants outside this space were excluded from this classification.

In `sensor.py`, mutants were grouped according to the MQTT behavior they affect: availability, payload or template handling, state updates, topic/subscription handling, and MQTT schema/configuration. In `template.py`, the relation with MQTT is indirect; therefore, mutants were grouped into value-template rendering/interpreting and template infrastructure potentially used by MQTT.

The classification shows that the low mutation scores are partly explained by MQTT-related behaviors that were not sufficiently exercised or observed by the current scenario. In `sensor.py`, most surviving MQTT-related mutants were associated with availability and expiration behavior. This indicates that the current harness mainly exercised immediate payload-to-state updates, but did not sufficiently stress time-dependent behavior such as expiration timers and availability transitions. In `template.py`, all MQTT-related surviving mutants were classified as value-template interpreting mutants. This suggests that the evaluated scenario exercised only a limited portion of the template behavior used to transform MQTT payloads into sensor states. These results reinforce the need for application-aware harnesses, more diverse MQTT configurations, time-aware checks, and stronger behavioral oracles.

Table 7: Post-hoc classification of MQTT-related mutants.

Class	sensor.py	template.py
K1: killed by semantic trace	19	127
S1: MQTT availability/expiration	34	0
S2: MQTT payload/template handling	5	0
S3: MQTT state update	1	0
T1: value-template rendering/interpreting	0	475
Total MQTT-related mutants	59	602

5.6 Discussion

The results suggest that the main bottleneck is not only input generation but also application-level reachability and observability. Although the fuzzers generated different corpora, the semantic replay mechanism and the single MQTT sensor configuration constrained the behaviors exercised in Home Assistant, contributing to the high overlap among killed mutants, especially in `template.py`.

The NOOP results strengthen the interpretation of the mutation campaigns by showing that the oracle and harness did not introduce observable false positives in the absence of fuzzer-derived inputs.

The MQTT-adjusted mutation score provides additional evidence for this interpretation. By focusing on the observed MQTT mutation space, the score excludes mutants that were neither covered during MQTT replay nor killed by any MQTT fuzzing campaign. In `sensor.py`, 59 mutants were included in this observed space. FUME killed 19 of them, resulting in a score of 32.20%, while BooFuzz and Scapy each killed 17, resulting in 28.81%. In `template.py`, 602 mutants were included in the observed space, and all valid fuzzers killed the same 127 mutants, resulting in an identical score of 21.10%.

These results show that the evaluated fuzzers were effective only over a limited portion of the MQTT-observed mutation space. However, the relative difference among tools remained small. FUME achieved the highest score in `sensor.py`, but the advantage was limited to two additional mutants. In `template.py`, the tools were indistinguishable, killing exactly the same set of mutants. Therefore, the results are insufficient to support a strong ranking among fuzzers. Instead, the main finding is that the current application-level MQTT scenario led the evaluated tools to exercise largely overlapping behaviors.

This result indicates that applying MQTT fuzzers directly to an IoT platform may be insufficient when the application processes only preconfigured topics and entity-specific payloads. Effective fuzzing in this context requires an application-aware harness that can map generated inputs to meaningful application states.

For researchers, the results highlight a methodological challenge in evaluating MQTT fuzzers beyond broker-level testing. Mutation testing can reveal not only differences among fuzzers, but also limitations of the fuzzing setup itself. The MQTT-adjusted score mitigates part of this issue by focusing the analysis on mutants observed in the evaluated MQTT scenario. Nevertheless, the results show that reachability alone is not enough: stronger behavioral oracles, more diverse entity configurations, and broader MQTT scenarios are still necessary to expose additional faults.

5.7 Threats to Validity

This section discusses the main threats identified and the mitigation strategies adopted.

Conclusion validity. A threat to conclusion validity is that the study uses one corpus generation campaign per fuzzer. Since fuzzers may exhibit stochastic behavior, different seeds or execution windows could produce different corpora and potentially different killed-mutant sets. To reduce this threat during mutant evaluation, the generated corpora were replayed deterministically against the same mutant set under the same environment. Thus, the results should be interpreted as a comparison of the evaluated fuzzer-derived corpora, rather than as a complete characterization of each fuzzer across multiple independent executions.

Internal validity. Internal validity refers to the possibility that factors other than the fuzzing tools themselves may influence the observed results. In this experiment, such factors include variations in computational resource consumption, interference among environmental components, broker instability, differences in setup across executions, and failures introduced by the harness. To reduce these risks, the experimental environment was executed in Docker containers, enabling component isolation, dependency control, and restoration of the system to a known state before each campaign. In addition, to mitigate the risk of false positives caused by non-deterministic behavior in the semantic oracle, we complemented the NOOP control with a repeated baseline replay check. The same corpora were replayed five times against the original Home Assistant version, and the resulting semantic traces were identical for all valid fuzzers. This reduces the likelihood that killed mutants were caused by unstable replay behavior rather than by behavioral differences introduced by mutation.

Construct validity. Construct validity concerns the adequacy of the metrics, oracles, and experimental mechanisms used to measure the effectiveness of the fuzzing tools. Although the mutation score is a well-established metric, its use in MQTT-based IoT systems may

not capture all relevant failures, especially behavioral inconsistencies or silent failures. To mitigate this threat, the study employs a stratified test oracle that can observe different levels of failure. The use of semantic replay introduces an additional construct-validity threat. Since raw payloads generated by the fuzzers are transformed into valid JSON messages before being submitted to the Home Assistant sensor, the experiment does not evaluate the raw MQTT traffic produced by each tool exactly as produced. Instead, it evaluates each fuzzer's ability to generate input corpora that, after a deterministic, uniform semantic mapping, exercise Home Assistant's application-level MQTT payload-processing logic. To mitigate this threat, the same normalization rule is applied to all fuzzers, and the raw corpora are preserved as experimental artifacts for inspection and reproducibility.

External validity. External validity concerns the generalization of the results to other systems.Since this experiment uses Home Assistant as the target system, the results may reflect specific features of its architecture. Moreover, the experiment focuses on the MQTT sensor module and on a single configured sensor entity. Therefore, the findings characterize the effectiveness of the evaluated fuzzers in exercising application-level MQTT payload processing in this specific context. They should not be directly generalized to other Home Assistant integrations. To mitigate this limitation, the study explicitly defines its experimental scope, documents the adopted configuration, and provides the artifacts necessary to support replication and extension of the experiment.

6 Conclusion

This paper evaluated MQTT fuzzing tools for application-level fault detection in Home Assistant using mutation testing. The experiment used a controlled Docker-based environment, an application-aware harness, deterministic corpus replay, and a stratified oracle to detect crashes, exceptions, incorrect entity states, and silent behavioral divergences.

The results showed that the evaluated fuzzers achieved limited effectiveness within the observed MQTT mutation space. FUME showed a small advantage in `sensor.py`, killing 19 out of 59 observed mutants, while BooFuzz and Scapy each killed 17. In `template.py`, all valid fuzzers killed the same 127 out of 602 observed mutants. Thus, under the current configuration, the evaluated fuzzers exhibited similar effectiveness and largely overlapping behavior.

The post hoc analysis indicated that many surviving or non-observed mutants were associated with configuration-dependent paths, generic template processing, and silent handling. These findings suggest that evaluating MQTT fuzzers in realistic IoT applications requires not only input generation, but also application-aware harnesses, diverse entity configurations, coverage-aware analysis, and stronger behavioral oracles. Future work includes expanding the experiment to additional Home Assistant integrations, improving oracle observability, and evaluating raw MQTT traffic alongside semantically normalized replay.

Artifact Availability

The experimental artifacts are publicly available at:

`https://github.com/darlagalves/SAST2026-FuzzingMQTT`

Acknowledgements

The authors thank CNPq for its support under grants 306719/2025-8 and 406941/2025-4. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

References

[1] Bernhard K. Aichernig, Edi Muškardin, and Andrea Pferscher. 2021. Learning-Based Fuzzing of IoT Message Brokers. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 47–58.

[2] Tanweer Alam. 2018. A Reliable Communication Framework and Its Use in Internet of Things (IoT). *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 3, 5 (2018).

[3] Luis Gustavo Araujo Rodriguez and Daniel Macêdo Batista. 2020. Program-Aware Fuzzing for MQTT Applications. In *29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*. ACM, 582–586.

[4] Luis Gustavo Araujo Rodriguez and Daniel Macêdo Batista. 2021. Towards Improving Fuzzer Efficiency for the MQTT Protocol. In *2021 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 1–7.

[5] Philippe Biondi and The Scapy Community. 2023. Scapy: A powerful interactive packet manipulation program. `https://scapy.net/`.

[6] Samin Y. Chowdhury, Ruimin Sun, and Brandon Dudley. 2025. A Survey for MQTT Fuzzing. In *Proceedings of the 2025 Workshop on Re-design Industrial Control Systems with Security (RICSS '25)*. ACM.

[7] Maialen Eceiza, Jose Luis Flores, and Mikel Iturbe. 2021. Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems. *IEEE Internet of Things Journal* 8, 13 (2021), 10390–10411.

[8] Bruno Garcia and Simone Souza. 2025. An empirical evaluation of fuzz targets using mutation testing. In *Anais do X Simpósio Brasileiro de Testes de Software Sistemático e Automatizado* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 75–83.

[9] Santiago Hernández Ramos, M. Teresa Villalba, and Raquel Lacuesta. 2018. MQTT Security: A Novel Fuzzing Approach. *Wireless Communications and Mobile Computing* 2018 (2018).

[10] Jaekwon Lee, Enrico Viganò, Fabrizio Pastore, and Lionel Briand. 2024. MOTIF: A tool for Mutation Testing with Fuzzing. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 451–453.

[11] Xinpeng Liu, Qinying Wang, Peiyu Liu, Wenhai Wang, and Shouling Ji. 2025. MQueez: Specification-Driven Fuzzing for MQTT Broker (Registered Report). In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis Companion (ISSTA Companion '25)*. ACM.

[12] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. doi:10.1109/TSE.2019.2946563

[13] Jean Baptiste Minani, Fatima Sabir, Naouel Moha, and Yann-Gaël Gueéneuc. 2024. A Systematic Review of IoT Systems Testing: Objectives, Approaches, Tools, and Challenges. *IEEE Transactions on Software Engineering* (2024).

[14] Biswajeeban Mishra and Attila Kertész. 2020. The Use of MQTT in M2M and IoT Systems: A Survey. *IEEE Access* (2020). doi:10.1109/ACCESS.2020.3035849

[15] 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.

[16] Jesse Pereyda. 2024. BooFuzz: Network Protocol Fuzzing for Humans. `https://github.com/jtpereyda/boofuzz`.

[17] Pallavi Sethi and Smruti Sarangi. 2017. Internet of Things: Architectures, Protocols, and Applications. *J. of Electrical and Computer Engineering* (2017), 1–25.

[18] Xiangpu Song, Jianliang Wu, Yingpei Zeng, Hao Pan, Chaoshun Zuo, Qingchuan Zhao, and Shanqing Guo. 2025. MBFuzzer: A Multi-Party Protocol Fuzzer for MQTT Brokers. In *34th USENIX Security Symposium*. USENIX Association.

[19] Vasudev Vikram, Isabella Laybourn, Ao Li, Nicole Nair, Kelton OBrien, Rafaello Sanna, and Rohan Padhye. 2023. Guiding Greybox Fuzzing with Mutation Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 929–941. doi:10.1145/3597926.3598107