FUME: *Fuz*zing *Me*ssage Queuing Telemetry Transport Brokers

Bryan Pearson*, Yue Zhang†, Cliff Zou*, Xinwen Fu‡
*Dept. of Computer Science, University of Central Florida, USA; bpearson@knights.ucf.edu, czou@cs.ucf.edu
†Dept. of Computer Science, Jinan University; zyueinfosec@gmail.com
‡Dept. of Computer Science, University of Massachusetts Lowell, MA, USA; xinwen_fu@uml.edu

IEEE INFOCOM 2022 - IEEE Conference on Computer Communications | 978-1-6654-5822-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOM48880.2022.9796755

***Abstract*—Message Queuing Telemetry Transport (MQTT) is a popular communication protocol used to interconnect devices with considerable network restraints, such as those found in Internet of Things (IoT). MQTT directly impacts a large number of devices, but the software security of its server ("broker") implementations is not well studied. In this paper, we design, implement, and evaluate a novel fuzz testing model for MQTT. The fuzzer combines aspects of mutation guided fuzzing and generation guided fuzzing to rigorously exhaust the MQTT protocol and identify vulnerabilities in servers. We introduce Markov chains for mutation guided fuzzing and generation guided fuzzing that model the fuzzing engine according to a finite Bernoulli process. We implement "response feedback", a novel technique which monitors network and console activity to learn which inputs trigger new responses from the broker. In total, we found 7 major vulnerabilities across 9 different MQTT implementations, including 6 zero-day vulnerabilities and 2 CVEs. We show that when fuzzing these popular MQTT targets, our fuzzer compares favorably with other state-of-the-art fuzzing frameworks, such as BooFuzz and AFLNet.**

I. Introduction

Message Queuing Telemetry Transport (MQTT) [1] interconnects resource-constrained devices in applications such as smart home, smart city, and industrial Internet of Things (IoT). MQTT is used by hundreds of thousands of devices across the world [2], and it is estimated that 62% of all IoT solutions use MQTT [3]. It is often considered the "de-facto standard" for Internet of Things (IoT) communication due to its low overhead and immense popularity when compared to similar protocols such as CoAP and AMQP [4]. Many implementations of MQTT have been developed since its inception, including software libraries for clients and servers on a range of hardware, Operating Systems, and cloud platforms [5]. Brokers may serve thousands of unique clients at any given time.

MQTT security – in particular, the software security of broker/server implementations – has received little attention in the literature. Most works only focus on the lack of **network** security mechanisms in MQTT, such as authentication, access control, encryption, and integrity checking [5]–[8]. On the other hand, **software** vulnerabilities of brokers are not nearly as represented in the literature. To our best knowledge, we observed only a single example which performs a comprehensive assessment of MQTT software security from the perspective of brokers [9]. Based on this research gap, we believe there is an urgent need to investigate the software security of MQTT brokers.

One of the most prominent methods for software vulnerability discovery is **fuzz testing**, or simply fuzzing [10]. A fuzzing software ("fuzzer") will generate psuedo-random or invalid test cases which are then sent to the target application. The fuzzer then observes the application behavior. Popular fuzzing frameworks for network applications include BooFuzz [11], Spike [12], and AFLNet [13]. In the context to IoT security, IoTFuzzer is a blackbox fuzzing model that performs dynamic analysis of mobile apps to learn how to communicate with remote IoT devices [14]. The model can achieve protocol guided fuzzing without intimate knowledge of the protocol itself. However, IoTFuzzer only targets software vulnerabilities in network clients.

In this paper, we develop a novel fuzzing model for MQTT brokers, called **FUME**. This fuzzer implements mutation guided and generation guided fuzzing techniques according to Markov models, which describe the state of each fuzzing iteration independently from past iterations. We show that each Markov model can be described as a finite Bernoulli process, since each direct transition can be considered a Bernoulli trial with a probability of transitioning to the next state, independent from other state transitions. We also implement "response feedback," a technique where the fuzzer can listen to network activity and console output (i.e., stdout, stderr, or log files) from the broker. Inputs which trigger unique responses from the broker are saved and tested later on. FUME requires no source code and does not need to run on the same system as the target broker. In total, we discovered 7 major vulnerabilities across 9 different broker implementations, including 6 0-day vulnerabilities. Among these vulnerabilities are 2 CVEs in Mosquitto [15], a very popular MQTT platform developed by the Eclipse Foundation.

Our major contributions are summarized as follows:

- We discuss the principles of fuzz testing in terms of Markov modeling. Namely, we design 2 Markov chains and a Bernoulli process for modeling a mutation-and-generation guided fuzzer.
- We present FUME, a novel fuzzer that targets MQTT brokers. The fuzzer implements the aforementioned Markov models and leverages response feedback to dynamically select more intelligent inputs for mutation.
- We evaluate FUME against 9 different MQTT broker implementations. We discovered 7 major bugs, including 6 zero-day vulnerabilities and 2 CVEs. We show that our

fuzzer can detect these bugs favorably when compared to other state-of-the-art fuzzing frameworks.

Responsible disclosure: We have followed the responsible disclosure policy and reported all our findings to the entities of interest, who have patched their systems based on our report.

II. Background

In this section, we introduce the MQTT protocol and the principles of fuzz testing to the reader.

A. MQTT

Architecture. MQTT is a lightweight communication protocol that is published under the open OASIS standard ISO/IEC 20922. It was designed to meet the networking requirements of resource constrained devices, such as embedded systems and IoT devices. MQTT typically runs over TCP but is not required to. In MQTT, clients connect to a central broker and can either publish messages or subscribe to topics. When a client publishes a message, it specifies a topic filter, and the broker must forward these messages to any clients which have subscribed to the same topic filter. The broker facilitates all communication between clients, addresses session requirements, and authenticates clients. MQTT only supports password-based authentication, and other security requirements must be implemented by the application.

Control Packets. MQTT supports 15 different packet types called **control packets**. These include: CONNECT; CONNACK; PUBLISH; PUBACK; PUBREC; PUBREL; PUBCOMP; SUBSCRIBE; SUBACK; UNSUBSCRIBE; UNSUBACK; PINGREQ; PINGRESP; DISCONNECT; and AUTH. All MQTT packets contain the same structure, which is illustrated in Figure 1. Namely, each packet begins with a *fixed header*, which identifies the control packet type and specifies the length of the packet; a *variable header*, which lists some features of the packet; and the *payload*, which contains the payload of the message. Depending on the control packet type, the variable header and the payload may be optional or required, while the fixed header is always required. Version 5 of MQTT also supports a *properties sub-header*, containing a list of optional properties. The properties sub-header exists at the end of the variable header. The CONNECT packet may also specify a will topic and a will payload. This payload is published to all subscribers of the will topic if the client ever disconnects unexpectedly – e.g., the client did not send the DISCONNECT packet before closing the connection. In MQTT version 5, the will information includes a *will properties* field within the CONNECT payload. The name, identifier, and purpose of each control packet is shown in Table I.

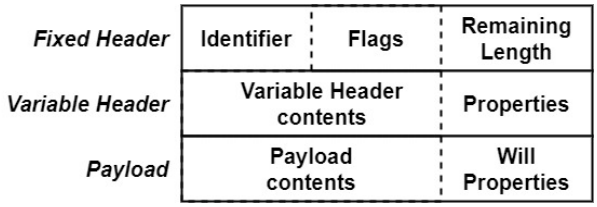

Fig. 1: MQTT packet structure. *Properties* only exists in MQTT version 5. *Will Properties* only exists in the CONNECT packet in version 5.

Name	ID	Purpose
CONNECT	0001	Request to connect to the broker
CONNACK	0010	Acknowledge the connect request
PUBLISH	0011	Send a message to subscribed clients
PUBACK	0100	Acknowledge the PUBLISH (QoS 1)
PUBREC	0101	Acknowledge the PUBLISH (QoS 2)
PUBREL	0110	Acknowledge the PUBREC (QoS 2)
PUBCOMP	0111	Acknowledge the PUBREL (QoS 2)
SUBSCRIBE	1000	Request to subscribe to a topic filter
SUBACK	1001	Acknowledge the SUBSCRIBE
UNSUBSCRIBE	1010	Stop listening to a topic filter
UNSUBACK	1011	Acknowledge the UNSUBSCRIBE packet
PINGREQ	1100	Ping the broker
PINGRESP	1101	Acknowledge the client's PINGREQ
DISCONNECT	1110	Send a request to disconnect
AUTH	1111	Exchange authentication data

TABLE I: A summary of MQTT control packets.

B. Fuzz Testing

To discover vulnerabilities in software, a fuzzer will generate psuedo-random or invalid test cases which are then sent to the target application; the fuzzer then observes the application behavior. If the application exhibits odd behavior, or crashes, then it is highly possible that a new vulnerability has been discovered; the researcher can then investigate this vulnerability more deeply. Fuzzers can be classified according to three factors: fuzzing method, knowledge of target, and vulnerability detection.

Fuzzing method: There are two primary fuzzing methods: generation guided and mutation guided. In *generation guided fuzzing*, data is generated randomly or from a user-defined model; for example, in *protocol guided fuzzing*, data is generated according to the protocol structure. This fuzzing method is appropriate when the user has a complete understanding of the syntax and semantics of the target protocol. In *mutation guided fuzzing*, payloads are sampled from a corpus of valid data inputs and fuzzed. This is appropriate when the target protocol is not well understood, or if the implementation differs from the specification. Another method, *genetic fuzzing*, may use either fuzzing method and apply genetic algorithms based on behavior exhibited from the target.

Knowledge of target: Depending on the knowledge of the target, a fuzzer might be classified as a blackbox fuzzer, a whitebox fuzzer, or a greybox fuzzer. A *blackbox fuzzer* has no knowledge of the target specification and can only see what is directly observable. A *whitebox fuzzer* is completely aware of the target's structure and may have access to its source code and specification. A *greybox fuzzer* has some knowledge of the specification and may use instrumentation or dynamic taint analysis to track the target's control flow.

Vulnerability Detection: To detect vulnerability in targets, a fuzzer may employ several techniques. For instance, the target may send an unexpected or malformed response, which can indicate a logic bug [14] [16]. The target may also hang,

i.e., the connection remains open but the target never sends a response [17]. Finally, the target may crash and close the connection; this behavior is almost universally observed by all fuzzers [11], [14], [16]–[19]. A program crash may indicate a severe vulnerability such as memory corruption.

III. Fuzz Testing Using Markov Modeling

In this section, we introduce the principles of mutation guided fuzzing and generation guided fuzzing in terms of two Markov models. We show that the models implement a finite Bernoulli process which describes the probabilistic behavior of input generation and payload fuzzing. We refer to the implementation of these models as the "mutation guided fuzzing engine" and "generation guided fuzzing engine".

A. Mutation Guided Fuzzing

The mutation guided fuzzing engine depends on the existence of an input corpus of semantically valid test cases. This engine can be broken down into two distinct phases: a *construction phase* and a *fuzzing phase*. In the *construction phase*, new packets are appended from the input corpus to the payload. In the *fuzzing phase*, the fuzzing engine can manipulate the payload using the byte-granular methods of injection, deletion, and mutation. The effects of these methods are as follows: *Injection* inserts new bytes into the payload; *Deletion* removes bytes from the payload; and *Mutation* changes the value of some bytes in the payload. Figure 3 illustrates the principle of each method using an MQTT SUBACK control packet with value `9003b80f07`.

The mutation guided fuzzing procedure can be modeled by a Markov chain, which is illustrated in Figure 2 (left). The model describes a single iteration of the fuzzing engine. The nodes represent states in the fuzzing engine, and the arcs represent probabilistic transitions; the transition probabilities are labeled next to their corresponding transitions. State S_0 represents the initial state of the fuzzing engine. State S_1 represents the *construction phase*. State S_2 represents the *fuzzing phase*. Finally, state S_f is the final state and concludes the current iteration of the fuzzing engine.

In the initial state S_0, the fuzzing engine may either transition to the construction phase, or it may select a payload from the response log. The response log is explained further in Section IV-B1; broadly speaking, it describes the set of test cases which have been added to the input corpus, i.e., those test cases which were not part of the original input corpus. The probability of selecting from the response log is b.

In the construction phase, the fuzzing engine randomly selects control packets from the input corpus. The probability of selecting CONNECT is c_1, CONNACK is c_2, etc. The sum of these probabilities is 1, i.e.,

$$\sum_{i=0}^{15} c_i = 1 \tag{1}$$

While the fuzzing engine is in state S_1, it has a X_1 probability of directly transitioning to state S_2, i.e., the fuzzing phase, and a $1 - X_1$ probability of selecting a new packet to append to the payload. In the model, appending a new packet is represented by the states `Add CONNECT`, `Add CONNACK`, and so forth. Based on the packet selection probabilities $c_i \mid i \in (1, 2, ..., 14, 15)$ and the probability of appending a new packet $1 - X_1$, the overall probability of adding a specific packet is $c_i - c_i X_1 \mid i \in (1, 2, ..., 14, 15)$.

In the fuzzing phase, the fuzzing engine can either transition to the `Inject` state, `Delete` state, or `Mutate` state, or it can transition to a `Send` state, which sends the fuzzed payload to the broker. The `Inject` state can transition to a `BOF` state or a `Non-BOF` state. In the former state, a large number of bytes are injected into the payload in an attempt to trigger a buffer overflow attack. In the latter state, the fuzzing engine only injects a small number of bytes – in the implementation, the number of injected bytes can never exceed the length of the original payload. The fuzzing states `Inject`, `Delete`, and `Mutate` have probabilities d_1, d_2, and d_3, respectively, such that $d_1 + d_2 + d_3 = 1$. The state `BOF` has probability d_4.

The probability of directly transitioning to the `Send` state is X_2. Based on the fuzzing state probabilities and the probability of transitioning to the `Send` state, the overall probability of choosing a specific fuzzing state is $d_i - d_i X_2 \mid i \in (1, 2, 3)$.

Finally, in the `Send` state, the fuzzing engine has a X_3 probability of transitioning to S_f and ending the current fuzzing iteration. Otherwise, there is a $1 - X_3$ probability to return to S_2 and restore the payload obtained from the construction phase.

B. Generation Guided Fuzzing

Generation guided fuzzing depends on deep knowledge of the protocol to generate semantically valid test cases. Figure 2 (right) illustrates the Markov model for generation guided fuzzing. The fuzzer generates a CONNECT packet first before generating other packets at random. Steps S_0 and S_1 comprise the *payload generator* component. Step S_2 performs the actual fuzzing operation. For simplicity, we have condensed the `Inject`, `Delete`, and `Mutate` states into a single `I/D/M` state. The probabilities for state transitions $S_2 \rightarrow Send$, $S_2 \rightarrow$ `I/D/M`, $Send \rightarrow S_2$, and $Send \rightarrow S_f$ are consistent between both models. In fact, both models are identical once state S_2 is reached, because the actual fuzzing of the payload is independent from how to obtain that payload.

C. Markov Modeling as a Bernoulli Process

Since each state transition depends solely on its transition probabilities, and each probability is assumed to be random, then we may also demonstrate each Markov model as a finite Bernoulli process. Namely, we can describe each Markov chain as a sequence $\bigcup_{i,j}^{S}(X_{s_i \rightarrow s_j}) \mid s_i, s_j \in S$, where S is the set of states in the Markov model and $s_i \rightarrow s_j$ describes a direct transition from state s_i to state s_j. Each state transition $s_i \rightarrow s_j$ is a Bernoulli trial with Bernoulli variable $X_{ij} = X_{s_i \rightarrow s_j}$. The probability of the fuzzing engine transitioning from state s_i to state s_j is $p_{X_{ij}}$. This value is simply the probability value given for that corresponding transition in Figure 2.

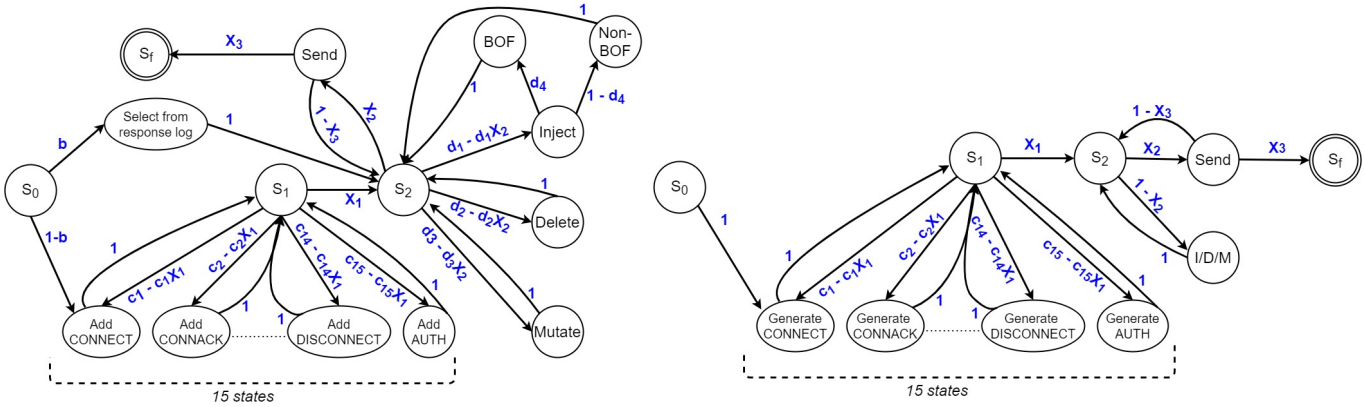

Fig. 2: Markov chains for mutation guided fuzzing and generation guided fuzzing.

Original	9003b80f07
Injection	904503b8f60f07a2
Deletion	9003|07
Mutation	9022b80f07

Fig. 3: Different payload manipulation methods.

IV. FUME: A Fuzzer for MQTT Brokers

In this section, we present FUME, a generation-and-mutation guided fuzzer for MQTT brokers. We first introduce the architecture of our fuzzing model. We then discuss each component of the architecture.

A. Overview: Architecture

First we introduce a high-level overview of FUME, which can be seen in Figure 4. There are five major components to the modeled architecture: the central component (simply called "FUME"), the user-defined parameters, the payload generator, the user filesystem, and the broker. The role of each component can be briefly summarized as follows:

- **Central component ("FUME")**: This contains the two fuzzing engines. It also handles communication and response monitoring from the target broker.
- **User-defined parameters**: Allows the user to configure aspects of the fuzzer, such as the probabilistic values X_1, X_2, and X_3.
- **Payload generator**: Generates a sequence of structurally valid control packets from scratch.
- **Filesystem**: Stores the input corpus and logs more test cases when new responses are observed from the broker.
- **Broker**: The target broker. May be local or remote.

B. The Central Component

The central component "FUME" handles three tasks, each of which is handled by a sub-component. The *first task* is to alternate (perhaps randomly) between running the mutation

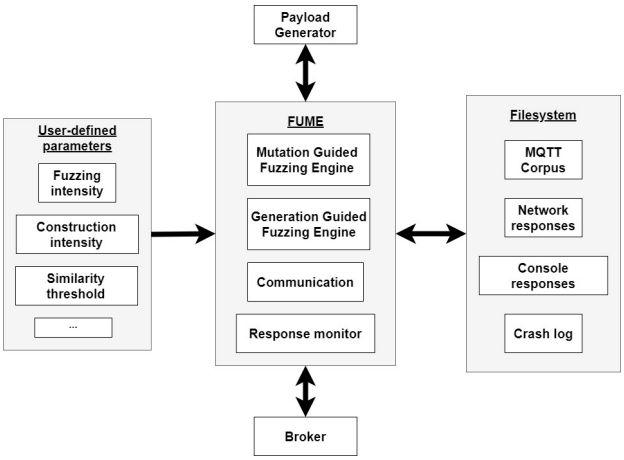

Fig. 4: FUME fuzzer architecture.

guided fuzzing engine and the generation guided fuzzing engine. These engines implement the Markov models described in Section III. During mutation guided fuzzing, the engine will access the filesystem component for appending new control packets to the payload or selecting inputs from the response log. During generation guided fuzzing, the engine will access the payload generator component to perform the actual generation of control packets. The *second task* is to establish a connection with the target broker and send fuzzed inputs over to the target. The `Send` state in the model defers responsibility to this sub-component to handle the connection requirements. The *third task* is to listen for network responses and console responses, and log them to the filesystem if necessary. The logging operation accesses the filesystem component.

1) Response Feedback: FUME monitors two major types of activity from the target broker: network response and console response. **Network responses** comprise the MQTT packets sent from the broker to the client. **Console responses** refer to the standard out and standard error file descriptors of the broker. When a unique response is observed by the fuzzer, the payload which triggered this response is logged to the

filesystem. These payloads can be fuzzed by the mutation guided fuzzing engine later on. This behavior is modeled by the mutation guided Markov chain, in the transition from initial state S_0 to the state `Select from response log`. Note that if the broker runs remotely, then console output will not be accessible on the local filesystem. However, some cloud platforms record console activity from running software – e.g., AWS CloudWatch [20] – which can be leveraged in this case. The fuzzer can always observe network responses.

A pitfall to response feedback is that responses may be redundant. For example, a CONNACK packet from the broker can contain an assigned client identifier, which may be randomly generated; then each CONNACK packet contains no real new information despite technically being unique. To address this drawback, we implemented a *packet parser* that can accurately derive each field in the payload from the broker. FUME monitors the fields that contain only a concise number of possible values (we call these fields "interesting"), and it ignores those redundant fields described above. When the broker sends a response that contains a new set of interesting fields, the response is considered unique, and it is logged to the filesystem.

However, the packet parser can only derive fields from network responses, but not from console responses. To limit redundancies in console response, we implemented a *similarity threshold* that ignores responses which are too similar to past responses. The threshold value is a percentage value defined by the user. For instance, a threshold of 75% means that console responses will not be logged if they are at least 75% similar to any previously logged response. We evaluate the impact of the similarity threshold on our fuzzer in Section V-D.

C. User-defined Parameters

The fuzzer accepts several values from the *user-defined-parameters* component which will influence its behavior. *Fuzzing intensity*, denoted as `fi`, is a percentage value that indicates what percentage of bytes should be fuzzed in a packet. For instance, a fuzzing intensity of 50% means that up to 50% of a payload should be fuzzed. Fuzzing intensity also determines how frequently a payload should be fuzzed in one iteration of the fuzzing engine, i.e., how long the model should remain in state S_2. *Construction intensity*, denoted as `ci`, is a non-negative integer indicating the desired number of packets in a payload sequence. For example, an intensity of 7 means that, on average, the payload shall be constructed of 7 distinct MQTT packets (it is only an average due to the stochastic property of the model). This sequence always begins with a CONNECT packet.

1) Mapping User Parameters to Markov Variables: For a user who wants to employ FUME without worrying about the details of the Markov model, the concepts of fuzzing intensity and construction intensity parameters may be more intuitive, while the Markov model may not be. To solve this issue, we offer a simple method to map `fi` and `ci` to X_1, X_2, and X_3. We assume the states of selecting/generating packets have discrete uniform distribution, i.e., $b = \frac{1}{2}$ and $c_i = \frac{1}{15} \mid i \in (1, 2, ..., 14, 15)$. We assume the same for the fuzzing states, i.e., $d_i = \frac{1}{3} \mid i \in (1, 2, 3)$, and $d_4 = \frac{1}{2}$. Note that in our implementation of FUME, all parameters and variables can be configured by the user, including X_1, X_2, and X_3. Our mapping is defined as follows:

- Let $X_1 = \frac{1}{ci}$
- Let $X_2 = 1 - fi$
- Let $X_3 = 1 - 2 * log(1 + fi)$

D. Payload Generator

To meet the requirements of generation guided fuzzing, the *payload generator* component can generate valid payloads for each of the 15 MQTT control packets. Algorithm 1 demonstrates how the payload generator will construct a CONNECT packet, i.e., state `Generate CONNECT` in the Markov model. The algorithm constructs the fixed header, variable header, and payload as shown in Figure 1. The variable header contains a *flags* field that indicates the presence of a username/password pair, protocol name and version, will information, and session information. Depending on the protocol version, the variable header may contain a *properties* field. The payload contains a client identifier and possibly a username, a password, and will properties.

Algorithm 1: Payload generator pseudocode for a CONNECT packet.

```
input: protocol_version
fixed_header.ID = 0x10;
variable_header.name = "MQTT";
variable_header.version = protocol_version;
variable_header.flags.username = random(0..1);
variable_header.flags.password = random(0..1);
variable_header.flags.will_retain = random(0..1);
variable_header.flags.will_qos = random(0..2);
variable_header.flags.will_flag = random(0..1);
variable_header.flags.clean_start = random(0..1);
variable_header.keepalive = random(0, 0xffff);
if protocol_version == 5 then
    variable_header.properties = random_properties();
payload.clientid = random_string();
if variable_header.flags.will_flag == 1 then
    if protocol_version == 5 then
        payload.will_properties = random_properties();
    payload.will_topic = random_string();
    payload.will_payload = random_string();
if variable_header.flags.username == 1 then
    payload.username = random_string();
if variable_header.flags.password == 1 then
    payload.password = random_string();
packet_length = len(variable_header) + len(payload);
fixed_header.remaining_length = packet_length;
packet = concat(fixed_header, variable_header,
 payload);
return packet;
```

E. The Filesystem

The user's filesystem stores the input corpus and the crash log. Payloads which trigger unique network or console responses from the broker – according to the similarity threshold – are also logged into the filesystem. In future iterations, these inputs are accessed by the mutation guided fuzzing engine so that they may be fuzzed.

V. Evaluation

In this section, we present our vulnerability findings and discuss the details of each vulnerability. We compare the vulnerability discovery speed of our fuzzer to three other popular fuzzing frameworks. We also discuss how mutation guided fuzzing compares to generation guided fuzzing. Finally, we explore the efficiency of response feedback across 3 different brokers for different similarity threshold values.

A. Experimental Setup

Software/Hardware: Our fuzzer is written in Python 3. All experiments are conducted in a Kali Linux 2021.1 virtual machine, which was allocated with 8 GB of RAM and 4 processor cores. The host machine is a Dell XPS 15 9570 laptop with Intel Core i7-8750H CPU. We have provided public access to our code at https://github.com/PBearson/FUME-Fuzzing-MQTT-Brokers.

User-defined Parameters: For all experiments, the fuzzing intensity and construction intensity was fixed at 0.1 and 3, respectively. This means X_1 was set to 0.33, X_2 was set to 0.9, and X_3 was set to 0.917.

Input Corpus: The predefined input corpus was collected systematically by connecting a client to the Mosquitto, HiveMQ, and VerneMQ brokers and collecting MQTT traffic using Wireshark. The inputs are stored in the filesystem. In total, we collected 50 distinct MQTT packets to seed the initial corpus.

Targets: In total, we tested 9 different MQTT broker implementations, including Mosquitto [15], HiveMQ [21], VerneMQ [22], aedes [23], EMQX [24], KMQTT [25], mqttools [26], hrotti [27], and moquette [28]. We fuzzed each broker for approximately 12 hours using a combination of the mutation guided and generation guided techniques. Table II shows the version number of each broker as well as the programming language that the broker was developed in. Note that hrotti does not have an official version number, so we just report the commit ID from Github.

Broker	Version	Language
Mosquitto	2.0.7	C
HiveMQ	2021.1	Java
VerneMQ	1.11.8	Erlang
aedes	0.45	JavaScipt
EMQX	4.3.3	Erlang
KMQTT	0.2.5	Kotlin
mqttools	0.47.0	Python
hrotti	Commit 087b33bb	Go
moquette	0.16	Java

TABLE II: Brokers tested

B. Vulnerability Findings

In total, our fuzzer discovered 7 vulnerabilities, including 6 zero-day vulnerabilities and 1 n-day vulnerability. All vulnerabilities result in immediate termination of the broker, causing denial-of-service. Aside from hrotti, which has abandoned development since 2017, all vulnerabilities have been reported to the developers and patched due to our responsible disclosure. Table III lists the complete set of crashes and a brief error summary. More details follow.

Index	Broker	Zero-day	Error Summary
0	Mosquitto	Yes	Malformed CONNACK in MQTT v5
1	Mosquitto	Yes	PUBLISH topic length = 0
2	KMQTT	Yes	Broken pipe error
3	aedes	Yes	Malformed DISCONNECT
4	hrotti	Yes	Malformed PUBLISH
5	hrotti	Yes	UNSUBSCRIBE topic length = 0
6	hrotti	No	Malformed CONNECT

TABLE III: A summary of crashes found.

Mosquitto: We discovered two vulnerabilities in Mosquitto version 2.0.7. The first vulnerability occurs in MQTT v5 when an authenticated client sends a malformed CONNACK control packet, causing a null pointer dereference and crashing the server. This vulnerability was reported to Eclipse and assigned to CVE-2021-28166 [29]. It has been patched in version 2.0.10. The second vulnerability occurs when a client sends a PUBLISH control packet with a topic length set to 0, causing the server to crash. This bug had previously been patched in version 2.0.8 at the time of our discovery; however, the patch was intended to address a bug in the Mosquitto *client* library, and the bug in the server was not originally recognized as a vulnerability. It has been assigned to CVE-2021-34432 [30].

KMQTT: We discovered a vulnerability in version 0.2.5. On some Linux systems, the server would throw a SIGPIPE signal if the broker tried to send a payload to a closed TCP connection – for instance, if it tried to respond to a SUBSCRIBE request with a SUBACK response. This bug was reported to the project's maintainer and patched in version 0.2.6.

aedes: In version 0.45.0, a vulnerability occurred if the client closed a connection with a malformed DISCONNECT packet. The bug only occurred if the following sequence of packets were sent: [CONNECT, PUBREL, DISCONNECT]. The sequence causes a buffer overflow to occur and crashes the server. We learned the vulnerability was due to a bug in *mqtt-packet* version 6.9.0, which is a Node.js package that aedes depends on. The bug in mqtt-packet was patched in version 6.9.1, and aedes version 0.45.1 now points to the patched package version.

hrotti: We discovered three vulnerabilities in hrotti. It should be noted that the project has apparently halted development since 2017. Therefore, at the time of writing this, all bugs are still present in the code. The first vulnerability is a parsing error that occurs when the client sends a valid CONNECT packet followed by a malformed PUBLISH packet. The second vulnerability occurs when the client sends an UNSUBSCRIBE packet with a topic length of 0. The final vulnerability occurs

when the client sends a malformed CONNECT packet. This vulnerability was first reported by Github user Alexander Sieg in an issue in September 2017 [31].

C. Vulnerability Discovery Speed

We now evaluate the discovery speed of the seven discovered vulnerabilities. We compare FUME against three fuzzing engines: BooFuzz [11], mqtt_fuzz [32], and AFLNet [13]. **BooFuzz** is a fuzzing framework written in Python that generates a fixed number of test cases according to a given input corpus. **mqtt_fuzz** is a mutation-based fuzzer for MQTT. **AFLNet** is an extension of AFL with added support for network applications. It should be noted that only FUME implements a generation guided fuzzing approach while the other fuzzers use a mutation guided approach. For the sake of fairness, we only utilize the mutation guided fuzzing engine of FUME in this evaluation. We also share the same original input corpus among all four fuzzers.

The final results of our evaluation can be seen in Table IV. In general, FUME discovered all vulnerabilities faster than any other broker. The only exception is that BooFuzz discovered the first Mosquitto vulnerability in 23 seconds, while FUME took 2 minutes and 29 seconds to discover the same vulnerability. Some brokers could not find a vulnerability after 12 hours of fuzzing (the cells labelled "N/A" in the table). In particular, mqtt_fuzz could only find the aedes vulnerability.

While testing AFLNet against some of the target brokers, We discovered that AFLNet will fail to identify and report the bugs in KMQTT, aedes, and hrotti. However, we confirmed that AFLNet eventually generates the payloads necessary to crash those brokers, and the lack of bug reporting may be the result of a bug in AFLNet. To address this, we started each server in a separate window before running AFLNet in non-instrumentation mode; this allowed us to visually observe the state of each server during the fuzzing process, but it removes AFLNet's code coverage features. After doing this, we were able to detect that all three brokers eventually crash as expected. In the case of Mosquitto, instrumenting it with *afl-gcc* provides the expected code coverage functionality supported by AFLNet, and it can detect the bugs.

Mosquitto Findings: The first two rows in Table IV correspond to the Mosquitto vulnerabilities (vulnerability index 0 and 1). For the first vulnerability, it can be seen that FUME triggered the crash in approximately 2 and a half minutes, while BooFuzz found the bug in 23 seconds and AFLNet took 45 minutes. For the second vulnerability, FUME found the crash in more than 4 minutes, and BooFuzz found it in 12 and a half minutes. AFLNet did not find the second vulnerability. mqtt_fuzz did not find either vulnerability. We also observed that AFLNet fuzzed targets much more slowly compared to the other fuzzers, sending on average between 1 and 2 payloads per second to the target, while the other fuzzers can send between 10 and 100 requests per second on average.

KMQTT Findings: From Table IV, it can be seen that Mosquitto discovered the KMQTT vulnerability in about 0.2 seconds, BooFuzz discovers the vulnerability in 50 seconds, and AFLNet discovers it in almost 3 minutes. Triggering this vulnerability requires the client to send a valid PUBLISH, SUBSCRIBE or UNSUBSCRIBE packet followed by immediately closing the connection. FUME is more likely to send valid control packets due to our fine-grained fuzzing strategy. In addition, the required valid packets are already present in the input corpus, which explains why our fuzzer detects the vulnerability so quickly. After 12 hours, mqtt_fuzz could not find the vulnerability, similar to before.

aedes Findings: Vulnerability index 3 indicates the results for the aedes vulnerability. FUME and mqtt_fuzz found the bug immediately, at 1.423 seconds and 1.773 seconds respectively. Meanwhile, BooFuzz found the bug in 16 seconds, while AFLNet found the bug in 19 secondg. During these experiments, we discovered that our input corpus actually contains a valid [CONNECT, PUBREL, DISCONNECT] sequence that triggers the crash without any fuzzing needed; this is why the bug is discovered so quickly by all four fuzzers. In the case of AFLNet, the crash occurs during the "dry run" phase in the beginning of the run, during which AFLNet will send each payload verbatim to the broker.

hrotti Findings: The last 3 rows in Table IV showcase the experimental results for hrotti. FUME could find all vulnerabilities in less than one second. BooFuzz discovered the first vulnerability in 7.8 seconds and the third vulnerability in 0.655 seconds. Since the third vulnerability depends on sending a malformed CONNECT packet, we adjusted our Python script to only send valid CONNECT packets in order to avoid triggering the third vulnerability multiple times. However, we could not trigger the second vulnerability despite multiple attempts; this is due to how *hrotti* only supports 65535 concurrent sessions, leading BooFuzz to quickly exhaust all of them and causing *hrotti* to hang. mqtt_fuzz could not find any of the vulnerabilities. AFLNet triggered the second vulnerability in 2 minutes and 21 seconds. We removed the input case that triggered this crash in hopes of triggering the other crashes; however, AFLNet failed to detect those crashes.

Vuln. Index	FUME	BooFuzz	mqtt_fuzz	AFLNet
0	149	23	N/A	2700
1	255	748	N/A	N/A
2	0.196	50	N/A	177
3	1.423	16	1.773	19
4	0.170	7.8	N/A	N/A
5	0.677	N/A	N/A	141
6	0.192	0.655	N/A	N/A

TABLE IV: Time to discover each vulnerability. Times are reported in seconds.

We now empirically compare the mutation guided fuzzing engine to the generation guided fuzzing engine. Table V compares the vulnerability discovery speeds between both approaches. In the case of Mosquitto, the mutation guided approach detects the first vulnerability in 149 seconds, while the generation guided approach takes over 38 minutes to find the same vulnerability. This can be attributed to the nature of the vulnerability, which requires a specially-crafted CONNACK packet that triggers a null pointer dereference in

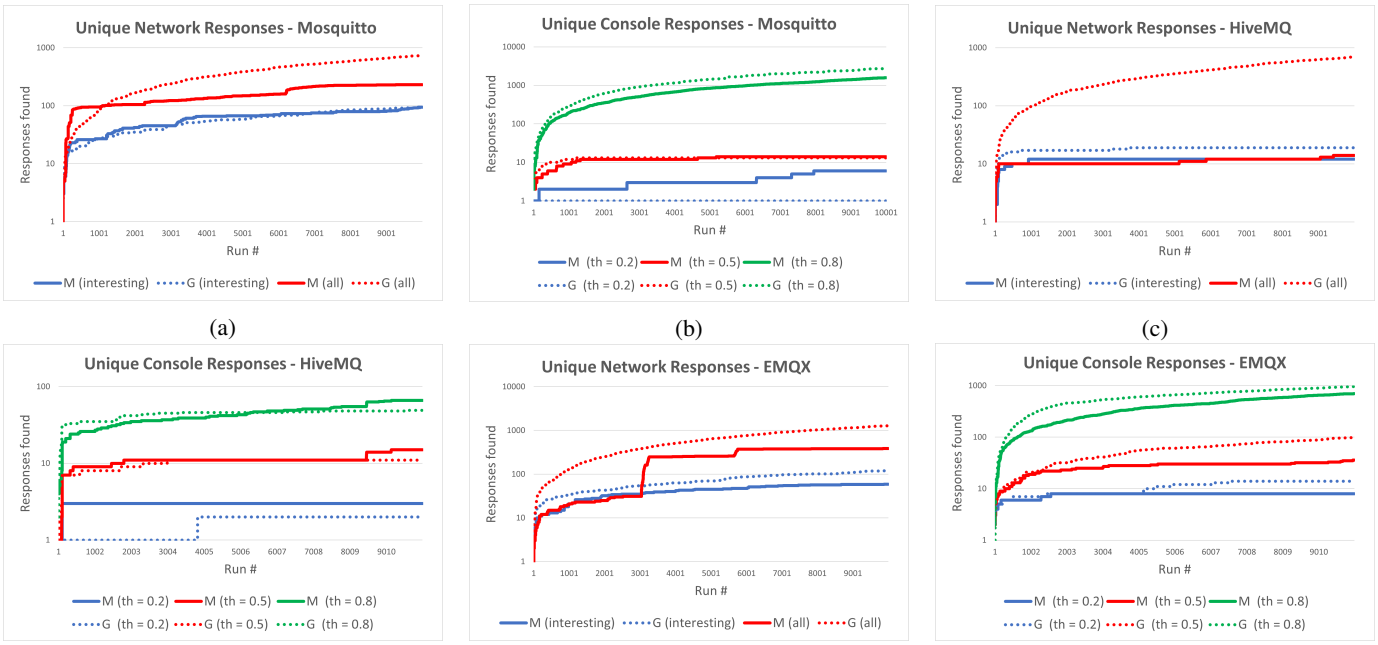

Fig. 5: State feedback evaluation – number of unique responses detected in mutation (M) versus generation (G) guided fuzzing. Tested for similarity thresholds (`th`) = 0.2, 0.5, and 0.8. `Interesting` means that only the interesting MQTT fields were monitored, while `all` means that we observed all MQTT fields.

Vuln. Index	Mutation guided	Generation guided
0	149	2286
1	255	83
2	0.196	0.806
3	1.423	1.561
4	0.170	0.131
5	0.677	0.069
6	0.192	0.143

TABLE V: Time to discover each vulnerability. Mutation guided approach versus generation guided approach. Times are reported in seconds.

the program. This occurs much faster in the mutation-based approach, because the input corpus contains a CONNACK packet that closely matches the contents of the malformed packet; however, the generation guided approach can generate hundreds of valid CONNACK packets, and most of them will be too dissimilar such that the fuzzing step cannot generate the malicious packet. On the other hand, the second vulnerability is found more quickly by the generation guided fuzzer – that is, only 83 seconds compared to the 255 seconds in the case of mutation guided fuzzing. This result is expected. This particular Mosquitto vulnerability occurs when a valid PUBLISH packet contains a topic length of 0. The generation guided approach can generate this packet reasonably quickly, while the mutation guided approach must successfully mutate the "topic length" field in a PUBLISH packet without corrupting the rest of the packet. All other vulnerabilities were found in less than 2 seconds using both fuzzing approaches.

D. Response Feedback Benchmarks

Using response feedback, FUME can monitor unique responses observed in a broker. The uniqueness of a network response depends on the field values of the control packet, while the uniqueness of console response is dictated according to a similarity threshold. In both cases, we attempt to minimize the number of redundant cases in the input queue. We have collected unique responses among 3 brokers: Mosquitto, HiveMQ, and EMQX. In the case of Mosquitto, we used the patched version so that we did not risk triggering a crash during our experiments. By default, HiveMQ and EMQX write log contents to an output file on disk, but we changed this by adjusting their configuration files so that log contents are printed to stdout. For each broker, we measured the number of unique responses across 10 thousand runs using generation guided and mutation guided fuzzing. Figure 5 shows the full results. Note that the vertical axes are logarithmic for the purpose of readability. For console responses, we measured the number of responses for similarity thresholds (i.e., `th`) 0.2, 0.5, and 0.8. For network responses, we measured how many unique responses were captured when we only monitored interesting MQTT fields (as described in Section IV-B1), and when we monitored all fields. We denote the *interesting fields monitor mode* as `G-I` and `M-I` for generation-guided and mutation-guided fuzzing, respectively, and the *all fields monitor mode* as `G-A` and `M-A`.

Mosquitto: From Figure 5a, we can see that `G-I` and `M-I`, i.e., only interesting response fields, perform very similarly. This is not too surprising, since many of the packets from our

input corpus were collected while running Mosquitto, allowing the mutation fuzzer to trigger many "hits" in the network responses early on. For G−A and M−A, i.e., all response fields, the number of unique responses is much higher. We attribute the rise in unique responses to the client ID field in the CONNACK response packet, which contains a random byte string generated by Mosquitto. FUME ignores this value when it only monitors interesting fields. Figure 5b plots the number of unique console responses observed in Mosquitto. For th = 0.5, the findings for mutation guided fuzzing and generation guided fuzzing are nearly identical. Again, we attribute this to the input corpus that we generated from Mosquitto. In the case of th = 0.8, generation guided fuzzing actually detected more unique responses; however, most of these were redundant cases since Mosquitto prints the client ID to the console. In the case of th = 0.2, generation guided fuzzing only logged a single response.

HiveMQ: Figures 5c and 5d plot the number of unique responses in HiveMQ. For network responses, we found that mutation-guided fuzzing receives relatively few responses "hits" regardless of which fuzzing field values are monitored. G−I performed slightly better (19 responses), and G−A discovered almost 700 responses. For console responses, the number of observations in generation guided fuzzing and mutation guided fuzzing are similar, especially when th = 0.5.

EMQX: Finally, Figures 5e and 5f plot the number of unique responses in EMQX. For network responses, we observed similar behavior to Figure 5c, with the exception of M−A; between runs 3041 and 3257, the number of hits increases rapidly. For console responses, we also observe similar behavior to Figure 5d. For th = 0.2 and th = 0.5, generation-guided fuzzing narrowly outperforms mutation-guided fuzzing. As usual, when th = 0.8, we find the majority of logged responses are redundancies of previous responses.

VI. Related Work

Fuzzing is a widely popular approach for finding software bugs. AFL-type fuzzers are arguably the most popular class of fuzzing frameworks [13], [18], [33]–[35]. AFL is a coverage-based greybox fuzzer (CGF); it instruments the target by injecting instructions into the assembly code at compile-time, and during fuzzing, the instrumented target informs AFL whenever it reaches a new path in the code. **AFLFast** [33] improves on AFL by tweaking the frequency at which a selected seed is fuzzed (its *energy*); AFLFast gives higher energy to seeds which execute low-frequency paths, thereby increasing the odds of finding a new path. **AFLGo** [34] enables *directed fuzzing* toward a target code location; the instrumented binary reports back to the fuzzer both the code coverage and the seed distance, i.e., the covered distance of a seed input from the target code location. Then AFLGo selects seeds which are more likely to minimize this seed distance. Other fuzzing frameworks might combine the CGF approach popularized by AFL with symbolic/concolic analysis [36]–[38], dynamic taint analysis [14], [19], [39], and grammar construction [40].

The major advantages of these fuzzing frameworks over FUME is 1) they are agnostic to the target software or protocol, and 2) they can monitor code coverage directly using either instrumentation (compiler-level or binary-level), or dynamic taint analysis, while FUME can only estimate coverage. On the other hand, all of these approaches require a great deal of "setup" on the part of the user. For example, most of these frameworks rely strictly on mutation-guided fuzzing since they have no knowledge of the target. Thus, their efficiency depends entirely on the seed corpus supplied by the user, which may be incomplete. Skyfire [40] constructs a probabilistic context-sensitive grammar (PCSG) to generate syntactically- and semantically-valid input seeds. However, Skyfire requires the user to supply an input corpus and context-free grammar. VUzzer [19] requires the user to perform static analysis on the target by constructing a control flow graph (CFG) and running analysis scripts. AFLNet [13] runs a persistent target program and requires a "cleanup script" to discard any changes to the program's state over a single run. In contrast to these approaches, FUME requires almost no setup from the user, since it also supports generation-guided fuzzing; moreover, FUME does not need a cleanup script for the persistent target program since it monitors the network and console channels for response feedback, which only capture the most important state changes. Finally, as opposed to other fuzzing frameworks, FUME requires no instrumentation or dynamic taint analysis, which are not always available for every application.

VII. Conclusion

In this paper, we designed a fuzzer based on Markov modeling for servers in MQTT-connected systems. MQTT affects hundreds of thousands of devices, particularly resource-constrained devices such as those found in IoT. Our fuzzer combines the techniques of mutation guided fuzzing and generation guided fuzzing to rigorously stress test the MQTT protocol. Response feedback from the target broker is monitored for tracking unique activity, which provides new test cases for the input corpus. We discussed three fuzzing methods that emphasize fine-grained manipulation of the payload. We have shown that state-of-the-art MQTT implementations such as Mosquitto contain serious vulnerabilities that can lead directly to denial-of-service attacks and threaten the reliability of the entire network. In total, we discovered 7 vulnerabilities, including 6 zero-day vulnerabilities. Finally, we compared our fuzzer against three popular fuzzing frameworks and demonstrated that our model can find MQTT vulnerabilities more effectively and rapidly in nearly all cases.

Acknowledgments

This research was supported in part by US National Science Foundation (NSF) Awards 2042996, 1643835, 1931871, and 1915780, US Department of Energy (DOE) Award DE-EE0009152. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] O. Open, "Mqtt version 5.0." https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html#_Toc3901006, March 2019.

[2] Shodan, "Shodan report for mqtt." https://www.shodan.io/search/report?query=mqtt, July 2021.

[3] B. Cabé, "Key trends from the iot developer survey 2018." https://blog.benjamin-cabe.com/2018/04/17/key-trends-iot-developer-survey-2018, April 2018.

[4] I. Skerrett, "Why mqtt has become the de-facto iot standard." https://dzone.com/articles/why-mqtt-has-become-the-de-facto-iot-standard, October 2019.

[5] Y. Jia, L. Xing, Y. Mao, D. Zhao, X. Wang, S. Zhao, and Y. Zhang, "Burglars' iot paradise: Understanding and mitigating security risks of general messaging protocols on iot clouds," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pp. 465–481, IEEE, 2020.

[6] G. Perrone, M. Vecchio, R. Pecori, and R. Giaffreda, "The day after mirai: A survey on MQTT security solutions after the largest cyber-attack carried out through an army of iot devices," in *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security, IoTBDS 2017, Porto, Portugal, April 24-26, 2017* (M. Ramachandran, V. M. Muñoz, V. Kantere, G. B. Wills, R. J. Walters, and V. Chang, eds.), pp. 246–253, SciTePress, 2017.

[7] S. Andy, B. Rahardjo, and B. Hanindhito, "Attack scenarios and security analysis of mqtt communication protocol in iot system," *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pp. 1–6, 2017.

[8] M. S. Harsha, B. M. Bhavani, and K. R. Kundhavai, "Analysis of vulnerabilities in MQTT security using shodan API and implementation of its countermeasures via authentication and acls," in *2018 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2018, Bangalore, India, September 19-22, 2018*, pp. 2244–2250, IEEE, 2018.

[9] S. H. Ramos, M. T. Villalba, and R. Lacuesta, "MQTT security: A novel fuzzing approach," *Wirel. Commun. Mob. Comput.*, vol. 2018, 2018.

[10] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecur.*, vol. 1, no. 1, p. 6, 2018.

[11] J. Pereyda, "boofuzz: Network protocol fuzzing for humans." https://github.com/jtpereyda/boofuzz, 2021.

[12] D. Aitel, "An introduction to spike, the fuzzer creation kit." https://www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt, 2002.

[13] V. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, pp. 460–465, IEEE, 2020.

[14] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, The Internet Society, 2018.

[15] T. E. Foundation, "Eclipse mosquitto." https://mosquitto.org/, 2021.

[16] T. Petsios, A. Tang, S. J. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pp. 615–632, IEEE Computer Society, 2017.

[17] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), pp. 2155–2168, ACM, 2017.

[18] Google, "American fuzzy lop." https://github.com/google/AFL, 2020.

[19] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing.," in *NDSS*, vol. 17, pp. 1–14, 2017.

[20] I. Amazon Web Services, "Amazon cloudwatch." https://aws.amazon.com/cloudwatch/, 2021.

[21] H. GmbH, "Hivemq." https://www.hivemq.com/, 2021.

[22] O. L. AG, "Vernemq." https://vernemq.com/, 2021.

[23] MoscaJS, "aedes." https://github.com/moscajs/aedes, 2021.

[24] L. EMQ Technologies Co., "Emq." https://www.emqx.io/, 2021.

[25] D. Pianca, "Kmqtt." https://github.com/davidepianca98/KMQTT, 2021.

[26] E. Moqvist, "mqttools." https://github.com/eerimoq/mqttools, 2020.

[27] A. S-M, "hrotti." https://github.com/alsm/hrotti, 2017.

[28] moquette io, "Moquette." https://github.com/moquette-io/moquette, 2021.

[29] T. M. Corporation, "Cve-2021-28166." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28166, March 2021.

[30] T. M. Corporation, "Cve-2021-28166." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-28166, July 2021.

[31] A. Sieg, "Server crash after client send zero-byte username but no zero byte password." https://github.com/alsm/hrotti/issues/15, September 2017.

[32] F.-S. Corporation, "mqtt-fuzz." https://github.com/F-Secure/mqtt_fuzz, 2021.

[33] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2019.

[34] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344, 2017.

[35] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 679–696, IEEE, 2018.

[36] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution.," in *NDSS*, vol. 16, pp. 1–16, 2016.

[37] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 745–761, 2018.

[38] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 697–710, IEEE, 2018.

[39] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 711–725, IEEE, 2018.

[40] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 579–594, IEEE, 2017.