Learning-Based Fuzzing of IoT Message Brokers

Bernhard K. Aichernig[1],
[1]*Institute of Software Technology*
Graz University of Technology
Graz, Austria
aichernig@ist.tugraz.at

Edi Muškardin[1,2]
[2]*TU-Graz SAL DES Lab*
Silicon Austria Labs
Graz, Austria
edi.muskardin@silicon-austria.com

Andrea Pferscher[1]
[1]*Institute of Software Technology*
Graz University of Technology
Graz, Austria
andrea.pferscher@ist.tugraz.at

***Abstract*—The number of devices in the Internet of Things (IoT) immensely grew in recent years. A frequent challenge in the assurance of the dependability of IoT systems is that components of the system appear as a black box. This paper presents a semi-automatic testing methodology for black-box systems that combines automata learning and fuzz testing. Our testing technique uses stateful fuzzing based on a model that is automatically inferred by automata learning. Applying this technique, we can simultaneously test multiple implementations for unexpected behavior and possible security vulnerabilities.**

We show the effectiveness of our learning-based fuzzing technique in a case study on the MQTT protocol. MQTT is a widely used publish/subscribe protocol in the IoT. Our case study reveals several inconsistencies between five different MQTT brokers. The found inconsistencies expose possible security vulnerabilities and violations of the MQTT specification.

***Keywords*-active automata learning; model inference; stateful fuzzing; conformance testing; MQTT; IoT**

I. Introduction

The Internet of Things (IoT) comprises tens of billions of devices and newly arising technologies, e.g. communication standards, will further push the growth of connected devices in the IoT [1], [2]. Since systems in the IoT are composed of multiple heterogeneous components, ensuring the dependability of these systems becomes a challenging task, especially if components appear as a black box.

Behavioral models create an understanding of complex interactions and can be used for structural testing, e.g. model-based testing of IoT protocols [3]. However, the manual creation of behavioral models could be a challenging task, especially in a black-box setting. The seminal work of Peled et al. [4] introduced automata learning as a promising tool to verify black-box systems. Using automata learning, they infer a behavioral model of the tested system, where the model is later used for model checking. Instead of verification, Tappler et al. [5] used automata learning for conformance testing. For this, they learned behavioral models of different implementations of the same system and, afterward, cross-checked the inferred models according to a conformance relation. The found inconsistencies between the models represented possible faults in the implementations.

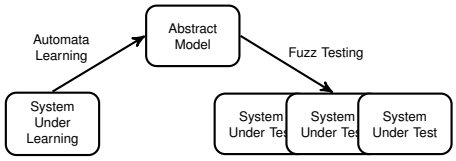

Figure 1. Fuzz-testing based on automata learning.

Depending on the abstraction level, the learned model might not cover behavior that leads to possible dependability issues. Fuzz testing is a testing technique that directly addresses the dependability of a system. The objective of fuzz testing is to force the system into an unexpected state by executing randomly generated input data. To observe unexpected behavior, the generated data can include malformed or invalid inputs. The combination of its ease of use and effectiveness made fuzzing a popular tool to test systems for faulty and unintended behavior. Furthermore, fuzz testing proved itself as a promising tool to reveal possible security issues [6]. The disadvantage of fuzzing is that the exposure to interesting behavior might require prior knowledge about the system. For example, authentication is required to explore other functionalities. Furthermore, it is difficult to define a stopping criterion for fuzz testing, especially when testing a black-box system.

In this paper, we present a technique that combines automata learning and fuzz testing. This synergy aims at in-depth security testing of black-box systems, where automata learning provides the required insight in a black-box system and fuzz testing reveals possible faulty behavior. Fig. 1 depicts the basic procedure of our testing technique. Our fuzzing target are different implementations of a system. First, we learn the behavioral model of one system. This system is denoted as the system under learning (SUL) and can either be randomly selected or represent a reference implementation. The systems that we fuzz are denoted as systems under test (SUTs). Note that the SUTs can also include the SUL. Using model-based testing, we test the SUTs based on the learned model. Since the model is more abstract than the SUTs, we need to concretize the generated test cases. The concretization is done by fuzzing techniques. After executing the concrete inputs, we check if the observed behavior from the SUT conforms to the model, i.e. our fuzzing technique is based on conformance testing.

We evaluate our learning-based fuzzing technique on the Message Queuing Telemetry Transport (MQTT) protocol [7]. MQTT is a widely used publish/subscribe protocol in the IoT. The MQTT protocol defines clients that connect to a broker. Connected clients can subscribe to topics and receive messages that are published to this topic or publish a message to a topic by themselves. The MQTT broker is a central server component that receives all requests from the clients and manages them. For example, a client sends a publish message on a specific topic, then the broker has to forward this message to all clients that are subscribed to this topic. The brokers' functionality is essential for the dependability of a network based on the MQTT protocol. Hence, we have chosen MQTT brokers as the first target for our new technique and fuzz different open-source implementations.

The contribution of this paper is threefold: Firstly, we present our learning-based fuzzing technique. Secondly, we evaluate our fuzzing technique in a case study on the MQTT protocol and report the found bugs in different MQTT brokers. Thirdly, the implementation of our learning-based fuzzing framework on MQTT brokers is available online[1].

The paper is structured as follows. In Sect. II, we introduce the used modeling formalism and the required techniques of automata learning and fuzz testing. We describe our learning-based fuzzing method in Sect. III and present our case study on MQTT brokers in Sect. IV . In Sect. V, we discuss related work. Finally, Sect. VI concludes the paper and gives an outlook on future work.

II. Preliminaries

A. Mealy Machines

Mealy machines are finite state machines that distinguish between input and output actions. Starting at an initial state, input actions can be performed, and every input triggers an output. Mealy machines are a popular modeling formalism for reactive systems and they are well supported by the automata learning libraries, e.g. LearnLib [8]. We formally define a Mealy machine as a 6-tuple $\langle Q, q_0, I, O, \delta, \lambda \rangle$ where Q is the finite set of states, $q_0 \in Q$ is the initial state, I is the finite set of input symbols, O is the finite set of output symbols, $\delta : Q \times I \to Q$ is the state-transition function, and $\lambda : Q \times I \to O$ is the output function. The Mealy machines used in this paper are *input-enabled* and *deterministic*, i.e. δ and λ are total functions. Input-enabledness establishes testability, since every input is defined for every state. Determinism ensures a unique successor state.

Let $s \in S^*$ be a sequence of input or output symbols, where either $S = I$ or $S = O$. We write ϵ for the empty sequence. The concatenation of two sequences $s, s' \in S^*$ is denoted by $s \cdot s'$. We also define a single element $e \in S$ as sequence, which enables $s \cdot e$. We extend the state-transition function δ and the output function λ for sequences. For this, we define the state-transition function $\delta^* : Q \times I^* \to Q$ for sequences which returns the state that is reached after performing a sequence of input symbols. Furthermore, let $\lambda^* : Q \times I^* \to O^*$ be an output function, that returns all observed output symbols after executing an input sequence.

Furthermore, two Mealy machines, $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ and $\mathcal{M}' = \langle Q', q_0', I, O, \delta', \lambda' \rangle$ are equivalent, if for all performed input sequences, the observed output sequences are equal, formally, $\forall s \in I^* : \lambda^*(q_0, s) = \lambda'^*(q_0', s)$. Hence, a counterexample to the equivalence between $\mathcal{M}$ and $\mathcal{M}'$ is an input sequence $s \in I^*$ where $\lambda^*(q_0, s) \neq \lambda'^*(q_0', s)$.

B. Conformance Testing

In conformance testing, we check if an implementation conforms to the specification, where the specification is described by a modeling formalism and the implementation is considered as black-box [9]. According to Tretmans [10] an implementation conforms to a specification if it implements the behavior defined in the specification. For this, Tretmans defines an implementation relation $\mathcal{I}$ **imp** $\mathcal{S}$ that is satisfied, if the implementation $\mathcal{I}$ conforms to the specification $\mathcal{S}$. To check if the conformance relation is satisfied, we use testing. For this, we define a test suite $T_\mathcal{S}$ that consists of test cases generated from the specification. The implementation relation is satisfied if the implementation successfully executes all tests of $T_\mathcal{S}$. A successful execution of a test t by an implementation $\mathcal{I}$ is denoted by $\mathcal{I}$ **passes** t, i.e. $\mathcal{I}$ behaves as expected by the test case, otherwise $\mathcal{I}$ **fails** t. Formally, we define conformance as follows.

$$\mathcal{I} \textbf{ imp } \mathcal{S} \iff \forall t \in T_\mathcal{S} : \mathcal{I} \textbf{ passes } t \tag{1}$$

For the creation of the test suite, a test-case generator is required. Ideally, a complete test suite [10] would be able to determine if an implementation conforms to a specification. However, in practice, this is usually infeasible. Therefore, different test-case generation techniques exist, e.g. W-Method [11], [12] or random walk, which are implemented in automata learning libraries like LearnLib [8].

C. Active Automata Learning

Automata learning creates a behavioral model of a black-box system based on a set of data. We call an automata-learning algorithm active if the required data to learn the behavioral model is actively queried by the learning algorithm. The seminal work of Angluin [13] proposes an active learning algorithm called L^*, which learns a deterministic finite automaton of a regular language. Today, many active learning algorithms are based on the basic concepts of L^*.

Angluin introduces the minimally adequate teacher (MAT) framework, which defines a *teacher* and a *learner*. The teacher answers questions asked by the learner about a black-box system, called SUL. We distinguish between two types of queries: *membership queries* and *equivalence queries*.

[1] https://github.com/DES-Lab/Learning-Based-Fuzzing

The learner asks membership queries to determine if a word is part of the regular language. The teacher answers this question either with *yes* if the word is in the language or *no*, if it is not. Equivalence queries are used to check if a proposed hypothesis model behaves equally to the SUL. In the case of a conforming hypothesis, the question is answered with *yes*. Otherwise, the teacher returns a counterexample showing non-conformance between the hypothesis and the SUL. Rivest and Schapire [14] proposed an improved L^*. They reduce the number of required membership queries by identifying the distinguishing suffix of a counterexample.

The L^* algorithm and its improvements have been adapted for other modeling formalism, like Mealy machines [15]–[17]. To learn a Mealy machine, these algorithms replace membership queries with *output queries*. The teacher returns instead of *yes* or *no*, the output sequence that was observed after executing the requested input sequence. The remaining concepts of the MAT framework stay unchanged.

Assuming an equivalence oracle that tells if a learned hypothesis behaves equally to the SUL is practically not applicable. In the literature, the equivalence oracle is replaced by conformance testing. In Sect. II-B, we explained that conformance testing is used to check if a formal model, i.e. the specification, conforms to a black-box implementation. In active automata learning, we observe an inverse situation, where we check if the hypothesis model conforms to the black-box SUL. Tappler [18] stresses that for an equivalence relation of an implementation $\mathcal{I}$ and for a hypothesis model $\mathcal{H}$ the relation $\mathcal{I}$ **imp** $\mathcal{H} \Leftrightarrow \mathcal{H}$ **imp** $\mathcal{I}$ holds. Therefore, we test during our learning procedure if $\mathcal{H}$ **imp** $\mathcal{I}$, where $\mathcal{H}$ is the hypothesis proposed by the learner and $\mathcal{I}$ is the SUL. We formally define the conformance relation for conformance testing in learning based on a finite test suite $T_\mathcal{S}$ as follows.

$$\mathcal{H} \textbf{ imp } \mathcal{I} \iff \forall t \in T_\mathcal{S} : \mathcal{H} \textbf{ passes } t \tag{2}$$

A failed test execution is a counterexample to the conformance between the learned hypothesis and SUL. This counterexample is then returned by the teacher. Otherwise, if the conformance test passes, the teacher returns *yes*, i.e. no counterexample can be found. The conforming hypothesis represents the final learned model.

The performance of active automata learning strongly depends on the alphabet-size of the SUL. For a Mealy machine, the alphabet is defined by the finite set of input and output actions. To overcome the problem of a large alphabet, Aarts et al. [19] proposed to use an abstracted input/output alphabet for learning. Hence, the learned model is an abstraction of the SUL. A mapper component is placed between the learning algorithm and the SUL. The mapper translates the abstract inputs from the learning algorithm to concrete inputs that can be executed on the SUL and the observed concrete outputs from the SUL to abstract outputs which are required by the learning algorithm.

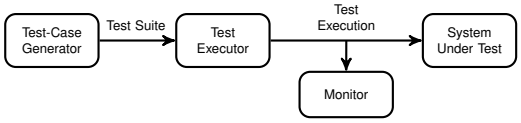

Figure 2. Components of a fuzzing framework.

D. Fuzz Testing

Fuzz testing (or fuzzing) is a testing technique that tests systems via a large randomly generated test suite. The goal of fuzzing is to detect bugs, especially security vulnerabilities, in the SUT through exhaustive testing with unexpected inputs. First fuzzing papers date back to 1990 where Miller et al. [20] fuzzed UNIX utilities. Due to the ease of use and effectiveness, fuzzing became a popular testing technique. The survey of Liang et al. [6] showed that fuzzing is a trending topic, with a high number of publications each year.

Since 1990 several different fuzzing techniques have emerged. In general, all the fuzzing methods build upon a similar architecture. Fig. 2 depicts the common components of a fuzzing framework which includes (1) the SUT, (2) a test-case generator, (3) a test executor, and (4) a monitor.

(1) The SUT represents the target device on which fuzzing is executed and monitored. Fuzzing distinguishes between three access levels of the SUT: white-box, gray-box, and black-box. In white-box fuzzing, the source code of the SUT is available and used for the test-case generation. This enables test-case generation based on code coverage or symbolic execution, e.g. used by SAGE [21]. In black-box fuzzing, we assume that no insight into the internal structure of the SUT, i.e. inputs can be executed and outputs observed. Gray-box fuzzing is situated between the previous levels and requires partial access to the SUT, e.g. to perform code instrumentation like american fuzzy lop (AFL) [22].

(2) The test-case generation in fuzzing can be categorized into two different strategies: *mutational fuzzing* and *generational fuzzing*. In mutational fuzzing, tests are generated via the mutation of existing data, e.g. through bit flips. The simplicity of this concept makes mutational fuzzing easily feasible, but with the drawback of possible low coverage of different functionalities of the SUT. Generational fuzzing possibly achieves better coverage, since test cases are generated according to a format specification requested by the SUT. Due to the required knowledge about the SUT, generational fuzzing requires more effort in the implementation and the applicability is limited. On top of these two test-case generation concepts, we find further techniques to improve the results of fuzzing [23], e.g. grammar-based fuzzing. In grammar-based fuzzing, test inputs are generated from an underlying formal grammar describing the input language.

(3) The tests are then executed on the SUT. For this, the test executor requires an interface to access the SUT.

(4) The monitor component observes the behavior of the SUT during the test execution and reports any unexpected behavior, e.g. a system crash or memory corruption.

III. METHOD

In the following, we present our stateful-fuzzing framework that is based on automata learning. First, we explain the composition of the components in our framework and discuss the interaction between them. Second, we describe the individual components and their specific adaptions required for learning and fuzzing the MQTT protocol.

A. Learning-Based Fuzzing

In learning-based fuzzing, fuzzing is done based upon a behavioral model, which is created via automata learning. Combining learning with fuzzing, we can fuzz a black-box system and at the same time reason which part of the system leads to faulty behavior. Our technique is related to learning-based testing [5], but instead of testing only with expected inputs, we use fuzz testing to reveal faulty behavior.

Tappler et al. [5] presented learning-based testing as a promising technique to test black-box systems. They first learn the models of black-box systems with an automata learning algorithm. All of these black-box systems represent different implementations of the same system. To test the different SUTs, they compared their learned models via conformance testing. A test that shows non-conformance between the two models is considered as a possible witness for unintended behavior in one of the compared systems.

Our learning-based fuzzing approach differs from their learning-based testing in two aspects. Firstly, we only learn one model. Secondly, we check the conformance based on fuzz testing. Instead of learning the models of all SUTs, we only learn the model of one implementation. We denote this model as the model of the SUL. Learning only one model has several advantages. One benefit is that we do not have to repeat a possibly expensive automata learning algorithm for every SUT. Especially, the learning of systems with a large state space or a big input/output alphabet is costly in terms of the number of required queries. In addition, a time-costly interaction with the SUL can hamper the performance of the learning algorithm. Furthermore, this makes our approach also applicable if a model of the SUL already exists.

The learning-based fuzzing technique proposed by Tappler et al. [5] is limited by the comparison of abstracted models. However, our technique uses fuzz testing for conformance testing and, therefore, overcome the problems that possible faults are abstracted "away". We suspect that the SUT behaves faulty if we find non-conforming behavior between the model and SUT. To reveal faulty behavior, we randomly generate test cases that also contain input sequences with unexpected or invalid inputs. Such sequences should lead to differences between the defined behavior in the model and the observed behavior of the SUT.

Fig. 3 depicts our learning-based fuzzing framework, which consists of two major components: learning and fuzz-testing. We start with learning the behavioral model of the SUL. To make automata learning feasible for SULs with a large input/output alphabet, we use the abstraction technique proposed by Aarts et al. [19]. For this, we reduce the alphabet-size by defining more abstract input and output symbols. We introduce a mapper component that translates abstract inputs used by the learning algorithm to concrete inputs which can be executed on the SUL. In addition, the mapper translates the observed concrete outputs to abstract outputs, which are then processed by the learning algorithm.

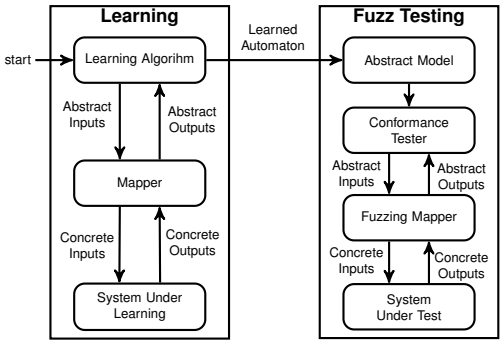

Figure 3. Learning-based fuzzing framework.

After the automata learning procedure, we continue with fuzzing the SUTs. Our targets for fuzz testing are SUTs that can be either different systems representing another implementation of the SUL, but also the SUL itself. The base for our fuzzing technique is an abstract model that is provided by the automata learning component. Since we learned the SUL based on the abstracted input/output alphabet, the model is on a more abstract level than the SUT. To test the conformance between our learned model and the SUT, we again have to adapt the level of abstraction. For this, we introduce another mapper component, which translates abstract inputs to concrete inputs and concrete outputs to abstract outputs. The main difference for this mapper is that the concrete inputs also contain randomly generated invalid or unexpected parts that should trigger unexpected behavior. These inputs are generated based on fuzz-testing techniques. Hence, we denote this mapper used in conformance testing as the *fuzzing mapper*.

A counterexample shows the non-conformance between the abstract model and the SUT. If we have found a counterexample in our fuzzing-based conformance testing, we analyze the reason for the different behavior between the model and the SUT. For this, we follow the procedure proposed by Tappler et al. [5]: Firstly, we execute the concrete input sequence of the found counterexample on the SUT and the SUL. Secondly, we analyze manually if the behavior of the SUT and SUL is correct. Another technique would be to analyze the learned model in advance. This could be done manually or by model-checking based on defined properties. Non-conforming behavior between the verified model and the SUT would then be a witness for a discrepancy between the SUT and the specification.

To fuzz the SUT, we generate a finite number of test cases

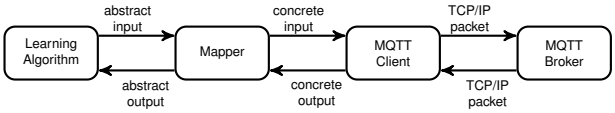

Figure 4. The interface proposed by Tappler et al. [5] to execute abstract queries from the learning algorithm on the MQTT broker.

and execute them on the SUT. Regarding the conformance relation defined in (1), we assume that the SUT conforms to the specification, which is our learned model of the SUL if all tests of the fuzzing test-suite pass.

B. Learning of MQTT

In this section, we describe our automata learning component and discuss the setup we used to learn the behavior of an MQTT broker. An MQTT broker can be described as a reactive system, where we observe for each performed input a corresponding output. Therefore, Mealy machines, as defined in Sect. II, are used as modeling formalism.

In automata learning, we distinguish between active and passive learning. In passive learning, we use a given set of data, e.g. log files. Therefore, the model is only as good as the given data set. In active learning, we actively query the SUL to gain knowledge about the systems' behavior. We use an active learning algorithm for our learning setup since an actively learned model is more suitable for subsequent fuzzing. Due to the actively querying, we gain more knowledge about the SUL and, therefore, we explore more unusual paths that can possibly not be seen in a passively generated data set. Our actively learned model describes behavior that might not be explored in the common usage of the SUL observable in the log files. Hence, the actively learned model contains possibly interesting paths for fuzzing.

One problem in learning the MQTT protocol is that the size of the input and output alphabet is extremely large and, therefore, automata learning does not scale. For this, we introduce a mapper component that abstracts and concretizes inputs and outputs. Fig. 4 depicts the interface that we used to enable the communication between the learning algorithm and the MQTT broker. The mapper takes the abstract input from the learning algorithm and concretizes the input by randomly choosing a corresponding concrete input. The MQTT client communicates with the MQTT broker. Therefore, the client processes the concrete input by generating TCP/IP packets that are sent to the broker. The broker responds with TCP/IP packets to the client. After receiving the response from the broker, the client extracts the concrete output and sends it to the mapper which abstracts the output for the learning algorithm.

A second problem in learning the MQTT broker is that we may observe non-deterministic behavior due to latency or timeouts. Also, other work [5], [24], [25] on automata learning of network protocols face the problem of observing non-deterministic behavior during learning. In our setup, it is sufficient to increase the timeout so that it accounts for possible higher latency between the client and the broker.

In contrast to other work [5], [26] on learning the MQTT protocol, we also consider invalid inputs that contain invalid characters according to the MQTT specification [7]. By using invalid inputs, our model also describes the error-handling behavior of the SUL, which we later require for our conformance test in our fuzzing component.

In active automata learning, we require an equivalence oracle that tells us if our intermediate learned model conforms to the behavior of the SUL. The existence of an equivalence oracle is not assumable in practice. Therefore, we replace the equivalence oracle with conformance testing. According to (2) we assume that a learned model $\mathcal{H}$ conforms to the SUL $\mathcal{I}$ if all tests of a finite test suite pass. Using Mealy machines as modeling formalism, we have sequences that consist of input and output actions. We assume that two Mealy machines show an equal behavior if all input sequences simulated on our hypothesis $\mathcal{H}$ produce the same output sequences as the execution on our SUL $\mathcal{I}$.

The test suite for our conformance tests during learning is generated through random walks in the hypothesis $\mathcal{H}$. For this, we used the random-walk implementation provided in LearnLib [8]. In a random walk, we generate random inputs, which are later executed on the SUL $\mathcal{I}$. If the execution on $\mathcal{I}$ returns a different output than the simulation on the hypothesis, we found a counterexample to the conformance between $\mathcal{H}$ and $\mathcal{I}$. If such a counterexample exists the conformance relation is not satisfied and this counterexample is considered in the next round of our learning procedure. If we cannot find any counterexamples that show non-conformance between our hypothesis model and the SUL, our learning framework returns the finally learned hypothesis.

C. Fuzz Testing of MQTT

Our fuzzing technique is based on model-based testing, where the model describes how the SUT should behave. The model is previously created in our automata learning component and describes the behavior of the SUL. Therefore, we use the model as our specification $\mathcal{S}$ and the SUT is our currently tested implementation $\mathcal{I}$. To compare the behavior of the model and the SUT, we perform again conformance testing using our conformance relation defined in (1). Our goal is to find inputs that reveal non-conformance between the learned model and the SUT.

The test-case generation in our fuzz testing component is akin to the test-case generation for conformance testing we used during the active automata learning. To make fuzzing feasible, we limit the number of test cases to a finite number. The test cases are again generated by randomly walking through the model. This time, we created our own implementation for the random walk through the model, since we require an advanced error-handling for the communication to the MQTT broker. The additional error-handling is required

since the fuzzing test-cases could cause a denial of service of the SUT. In this random walk, we randomly select one input action from the alphabet and then execute the selected input on the SUT and on the model. Afterward, we check if the observed outputs from both systems are equal. If this is not the case, we have found a counterexample and return the input/output sequence that was generated so far. The number of inputs performed in one random walk is defined by a constant n_{step}. The number of performed random walks in the conformance check is defined by n_{test}.

The learned automaton is on a more abstract level than the SUT. Therefore, we require again a mapper component that translates the abstract sequences from the model to concrete sequences that can be executed on the SUT. Differently to our previously described mapper component, the mapper in our fuzzing component concretizes inputs using fuzz testing.

For the fuzz testing of the MQTT broker, we tested the topic filters and topic names used by the subscribe, unsubscribe, and publish commands of the protocol. In the MQTT protocol, a client can subscribe and unsubscribe to a topic filter, and publish messages to a topic name. The client receives the messages that are published to the topic for which the client is subscribed. The difference between the topic filter and the topic name is that the filter can contain wildcard characters, which enable the client to simultaneously subscribe or unsubscribe to several topics. Topic filters and names compromise levels, which are separated by the '`/`' character. We distinguish between the two wildcard characters '`+`' and '`#`'. The '`+`' character defines a one-level wildcard, which replaces the concrete name of one level, e.g. a subscription to `temp/+/` receives messages of the topics `temp/kitchen/` or `temp/office/`. The multi-level wildcard '`#`' must be the last character in a topic filter and defines a subscription two all subsequent levels, e.g. `temp/#` receive messages from `temp/gf/kitchen`. A message can only be published to a topic name. Therefore, it is not possible to publish a message to multiple topics using only a single publish command. We describe topic-filter structure by the following context-free grammar.

$$
\begin{aligned}
\langle \textit{TopicFilter} \rangle &::= \text{`\$'} \langle \textit{Body} \rangle \mid \langle \textit{Body} \rangle \\
\langle \textit{Body} \rangle &::= \text{`\#'} \mid \text{`+'} \langle \textit{EmptyLevel} \rangle \\
&\quad \mid \langle \textit{String} \rangle \langle \textit{EmptyLevel} \rangle \\
&\quad \mid \langle \textit{Level} \rangle \\
\langle \textit{EmptyLevel} \rangle &::= \langle \textit{Level} \rangle \mid \epsilon \\
\langle \textit{Level} \rangle &::= \text{`/'} \langle \textit{EmptyBody} \rangle \\
\langle \textit{EmptyBody} \rangle &::= \langle \textit{Body} \rangle \mid \epsilon \\
\langle \textit{String} \rangle &::= \text{`UTF-8 Characters'}
\end{aligned}
$$

The grammar for topic names is akin to the previously defined topic-filter grammar except that the wildcard characters are not included in the grammar. The topic filters and names must consist of UTF-8 characters. The MQTT specification [7] defines a subset of UTF-8 characters that must not be included, e.g. the `NULL` character (U+0000), or should not be included, e.g. control characters. Since we want to explore unexpected behavior during fuzz testing, we do not exclude these characters in our topic generation.

Our fuzzing mapper uses topic filters and topic names based on the defined grammar. The advantage of the grammar is that we can also use the grammar to modify existing topic names and filters. For example, we can modify a topic name to a corresponding topic filter with wildcards, enabling that a client that is subscribed to the topic filter also receives published messages for the topic name. Possible topic filters including wildcards for the topic name `/abc/def/gh` would be, e.g., `/abc/#` or `/abc/+/gh`.

The interface to our SUT during fuzzing is similar to the interface depicted in Fig. 4. However, the learning algorithm is replaced by our conformance-testing component that generates tests by randomly walking through the model. The randomly generated input sequences are then passed to the fuzzing mapper, which concretizes the inputs. For the concretization of subscribe and publish inputs, the mapper uses topic filters and names based on the defined grammar. A single MQTT client then passes the concrete inputs to the MQTT broker. We then check if the observations returned by the MQTT broker conform to the behavior defined by our model. If we found a non-conforming input/output sequence, we execute this sequence again on the SUL. If the SUL and the SUT show non-conforming behavior, we possibly found a bug in the SUT. In this case, we perform further manual analysis and check the cause for the non-conforming behavior. If we cannot find any counterexample during fuzz testing, we assume that the SUT conforms to the model of the SUL according to the executed tests.

IV. Case Study

In this section, we present technical details about the implementation of our learning-based fuzzing technique and report the found inconsistencies in the tested MQTT brokers. The implementation is available **online**[2] [27]. The evaluation of our proposed method is based on five different MQTT brokers that support the MQTT standard v5.0 [7].

A. Clients and Brokers

The MQTT protocol distinguishes between two different types of players: *broker* and *client*. The broker represents a server component that manages the interaction between multiple clients. Clients can connect to the broker and, afterward, publish messages or subscribe to a topic. The broker is required to handle the requests of the clients and forward the received messages. Therefore, the broker represents a key component in the MQTT protocol.

MQTT Client. We implemented our own MQTT client that supports the MQTT v5.0 specification. To enable fuzzing of the MQTT broker, our client does not perform any

[2]https://github.com/DES-Lab/Learning-Based-Fuzzing

packet validation or sanitation, like other available MQTT clients, e.g. the HiveMQ MQTT Client[3]. We expect that the dependability of an MQTT network should be independent of the used clients. Our client mimics a malicious component in an IoT system and allows us to pass malformed packets to the MQTT broker.

MQTT Brokers. Hence, the broker's packet processing and sanitation must not rely on well-performing clients. For example, the MQTT standard [7] defines which topics must not or should not be accepted. Our case study aims to test if the brokers conform to the topic specification. We performed fuzz testing on the following MQTT brokers which implement v5.0 of the MQTT protocol:

- Eclipse Mosquitto 1.6.8 [4]
- HiveMq 2020.2 [5]
- EMQ X v4.0.0 [6]
- VerneMQ 1.11.0 [7]
- ejabberd 20.7.0 [8]

To efficiently apply learning and subsequent fuzzing, we simultaneously ran all brokers and assigned different ports to each broker. Furthermore, the network communication to the MQTT broker depends on time. To deal with this timed behavior, we chose a timeout that defines the maximum amount of time we expected to receive a response from the broker. We observed that some clients responded faster, while others required more time. We set the timeout of each broker to 200 milliseconds, which was the required response time by the slowest broker.

B. Learning Setup

Our learning-based fuzzing framework requires a behavioral model of the SUL, which is compared to the SUTs. To create this model, we used active automata learning. We based our learning setup on the Java library LearnLib 8. LearnLib provides a convenient interface to many different learning algorithms and equivalence-testing techniques. We used Rivest and Schapire's improved L^* algorithm [14] and applied a conformance-testing technique based on random walks provided by LearnLib. Following setup was used for the random walk: the probability of resetting the SUL was set to 0.09, the maximum number of steps was $3\,000$ and the step counter was reset after every found counterexample.

For our case study on the MQTT protocol, we learned the model of the Eclipse Mosquitto broker. We have chosen the Eclipse Mosquitto broker as SUL, since the case studies of Tappler et al. [5] and Mladenov [28] showed that the broker implementation of Eclipse Mosquitto has fewer discrepancies to the MQTT specification than other investigated broker implementations. Note that we could have chosen any other broker as SUL, since we later perform a conformance check that compares each broker with the model.

[3]https://github.com/hivemq/hivemq-mqtt-client
[4]https://mosquitto.org/
[5]https://github.com/hivemq/hivemq-community-edition
[6]https://github.com/emqx/emqx
[7]https://github.com/vernemq/vernemq
[8]https://www.ejabberd.im/

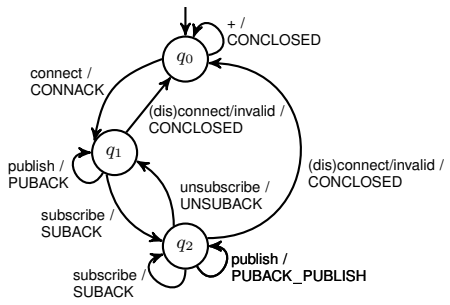

Figure 5. The learned model of the Eclipse Mosquitto broker interacting with one client. The "+"-symbol unifies the remaining input transitions.

Due to the large input and output alphabet of the MQTT protocol, we used the abstraction mechanism proposed by Aarts et al. [19] to reduce the input/output alphabet. For this, we implemented a static mapper component similar to one proposed by Aichernig et al. [26]. We used the following abstracted input alphabet $I^{\mathcal{A}} = \{\mathsf{connect}, \mathsf{disconnect}, \mathsf{subscribe}, \mathsf{unsubscribe}, \mathsf{publish}, \mathsf{invalid}\}$ and the concrete input alphabet $I = \{\mathsf{connect}, \mathsf{disconnect}, \mathsf{subscribe}(\mathit{topicFilter}), \mathsf{publish}(\mathit{topicName}, \mathit{message}), \mathsf{unsubscribe}(\mathit{topicFilter})\}$, where $\mathit{topicFilter} \in \{\mathtt{NULL}, \mathrm{S}\}$, $\mathit{topicName} \in \{\mathtt{NULL}, \mathrm{S}\}$, and $\mathit{message} \in \mathrm{S}$. Let S be the set of valid UTF-8 characters that are accepted by the protocol and `NULL` be the unicode character U+0000.

For the concretization of the abstract input alphabet, we used a setup where one client interacted with the MQTT broker. Topic filters/names and messages of publish, subscribe and unsubscribe inputs are translated to accepted characters. The mapper translates the invalid input to publish, subscribe and unsubscribe commands that contain the Unicode Character `NULL` (U+0000), which is prohibited by the MQTT specification. To keep the learning of the MQTT broker feasible, we always selected matching topic filters and names during learning, e.g. `test/test/test` and `test/test/+`. Otherwise, we would have to refine the abstracted alphabet since the used level of abstraction would have lead to non-determinism. The concrete output alphabet $O = \{\mathsf{CONNACK}, \mathsf{CONCLOSED}, \mathsf{PUBACK}, \mathsf{SUBACK}, \mathsf{UNSUBACK}, \mathsf{PUBACK\_PUBLISH}(\mathit{topicName}, \mathit{message})\}$ is abstracted to $O^{\mathcal{A}} = \{\mathsf{CONNACK}, \mathsf{CONCLOSED}, \mathsf{PUBACK}, \mathsf{SUBACK}, \mathsf{UNSUBACK}, \mathsf{PUBACK\_PUBLISH}\}$, i.e. we only removed the parameters of the publish message.

Fig. 5 depicts the learned model of the Eclipse Mosquitto broker. The model is relatively simple due to the consideration of only one client. It can be seen that the broker correctly disconnects the client (CONCLOSED) when it provides an invalid input. Learning of this model took approximately 260 seconds. The learning procedure required

a single equivalence query and 114 output queries that account for the majority of the runtime. We claim that the behavior described by the model conforms to the Eclipse Mosquitto broker for the used learning alphabet and performed conformance tests.

C. Fuzzing Setup

For fuzzing, we implemented a conformance tester based on model-based testing. Our fuzzing implementation requires a model and an interface to the SUTs. As previously explained, we created the model via automata learning. However, any behavioral model, e.g. an existing or a manually created one, can be used for our fuzzing technique. For this, we implemented a parser for `dot`-files of Mealy machines that conform to the syntax used by LearnLib.

The implementation of the interface to the MQTT brokers is similar to the interface required for learning. Hence, we require again an MQTT client that enables the interaction with the broker and a mapper component for the conformance test between the SUT and the model. The previously used MQTT client implementation remained unchanged. However, we adapted the mapper implementation for fuzz testing.

The *fuzzing mapper* abstracts and concretizes inputs and outputs, like our static mapper in learning. Since the abstract model is given, the abstract input and output alphabet is equal to the one used for our learning algorithm. To fuzz the MQTT protocol, the generation of concrete inputs changed for publish and subscribe messages. We generated topics based on the grammar for topic filters defined in Sect. III-C. To generate invalid topics, we used characters that are defined by the MQTT specification as prohibited or inadvisable. As mentioned previously, one character that must not be included is the `NULL` character (U+0000). Furthermore, the MQTT specification [7] defines, e.g., that the Unicode characters from U+0001 to U+001F should not be included. For fuzzing, we concretized the input $invalid$ to actions (subscribe, unsubscribe, publish) that are invalid. Invalid actions are not only topics that contain characters that are prohibited by the MQTT specification. They also encompass actions such as publishing to the system-level topics or publishing to topic names containing a wildcard. The implementation of our fuzzing mapper ensured again that the topic filter and names match.

We developed the conformance test between the model and the SUT for our fuzz-testing technique. For this, we did not reuse the equivalence-checking algorithm provided by LearnLib. Our fuzzing technique requires advanced error handling since the execution of malformed input can lead to a crash of the SUT. The conformance testing is again based on random walks through the model. We assume that random walks are well-suited for fuzzing since possibly unexpected input sequences are generated by traversing the model. The generated sequences are then executed on the SUT through the fuzzing mapper. The aim of the conformance testing is to find an input sequence that generates different output sequences on the model and on the SUT.

In our case study, we fuzzed all five MQTT brokers that are listed in Sect. IV-A, including the Eclipse Mosquitto broker which served as the SUL. We used an equal parameter setup for all five brokers. For the random walk, we set the number of inputs performed in one random walk n_{step} to 50 and the number of performed random walks n_{test} to $1\,000$. If we found a counterexample via fuzzing that shows that the model and the SUT behave differently, we manually investigate the causes for the found inconsistency.

D. Bug Hunt

In the following, we describe the found bugs and cases of non-conformance. The experiments were conducted with a MacBook Pro 2018 with an Intel Quad-Core i5 operating at 2.3 GHz and with 16 GB RAM. We separate our findings into three categories: (1) topic character acceptance, (2) topics beginning with `$`, and (3) number of received publications.

(1) *Topic character acceptance.* As explained earlier, our fuzzing mapper generates topic filters and names containing characters that must not be allowed or should not be allowed by the broker. We were able to reveal inconsistencies between the brokers even by adding single invalid characters. The most severe bug was found in the VerneMQ broker. It seems that this broker does not perform any string validation or sanitization. The simple input trace $\mathsf{connect} \cdot \mathsf{publish}(invalid)$ shows non-conforming behavior. Our client can subscribe and publish to topics containing prohibited characters, e.g. `/te\u0000st`, as well as to topics containing inadvisable characters, e.g. `/te\u000Ast`. Such behavior shows clearly a violation of the MQTT specification. This bug could also lead to a possible security vulnerability in the MQTT network since a topic containing the `NULL` character could be forwarded to a subscribed client. Such a message could cause possible memory leakages in an MQTT client written in C.

We can exploit VerneMQ vulnerability by the following setup. For this, we require our malicious MQTT client, an additional client written in C, e.g. wolfMQTT[9], and the VerneMQ broker. Following steps are performed.

1) wolfMQTT & malicious client: connect to VerneMQ
2) wolfMQTT: subscribe to `test/#`
3) malicious client: publish to `test/te\u0000st`
4) wolfMQTT: process truncated topic name `test/te`

The brokers EMQ X and ejabberd also showed questionable behavior. We could not reveal any violations against the MQTT specification, but these brokers accept characters that should – according to the specification – not be included. Allowing Unicode control characters, e.g. U+0001…U+001F, also enables possible attacks. For example, consider the setup of the previously mentioned exploit. We change the

[9]https://github.com/wolfSSL/wolfMQTT

publish topic and, instead, publish a message to the topic name `test/\u001b[0;31mred`. Printing this topic name on the client side changes the font color of the terminal to red. Using Unicode control characters, we can change the font color of messages to an arbitrary value, e.g. to hide messages. Note that this vulnerability is independent of the implementation language of the MQTT client.

Eclipse Mosquitto and HiveMQ showed for all topic characters a behavior conforming to the specification. One difference was that HiveMQ sends additional disconnect packages, which is valid but not required behavior.

(2) *Topics beginning with `$`.* The MQTT specification [7] defines that brokers should prevent clients from using topics beginning with the `$` symbol. Additionally, the specification states in non-normative comments that applications cannot use topics with a leading `$` and that topics beginning with `$SYS/` are widely used for the broker communication. `$SYS/` topics are special meta-topics that the broker can use to publish internal information and its client sessions. Hence, `$SYS/` topics should be read-only for MQTT clients.

In our case study, we investigated the behavior on topic filters and names beginning with `$`. We found that the handling of these topics is inconsistent between the tested brokers. The only broker that behaves conforming to our learned model is HiveMQ. The broker allows subscriptions to such topics but does not allow any publish messages. It even disconnects the client, when it sends a publish message to such a topic. VerneMQ shows an akin behavior, but instead of disconnecting the client it simply does not send a response to the invalid publish message and the client stays connected. The broker ejabberd responds with an acknowledge package but does not forward the publish message to the client. Surprisingly, the Mosquitto broker, which served as our SUL, showed unexpected behavior. With Mosquitto it was possible to exchange messages via topics beginning with `$`, i.e. it is possible to subscribe and publish to such topics. Fig. 6 depicts the behavior of the Eclipse Mosquitto broker, where the red transitions show the behavior on publish messages on topics with a leading `$`. We observe that there is no difference in the behavior compared to a valid publish message. However, Mosquitto prohibits publications to `$SYS/` topics. EMQ X showed the most laissez-fair behavior since it was also possible to use `$SYS/` topics for client-to-client communication.

(3) *Number of received publications.* MQTT clients can simultaneously subscribe to multiple topics . Due to possible wildcards in topic filters, subscriptions can also overlap, e.g. a client subscribes to `test/#` and `test/test`. In the case of an overlapping subscription, all tested brokers except HiveMQ forwarded for each subscription a separate publish message. HiveMQ, instead, sent a single publish message. According to the specification this is a valid behavior. The specification only defines that this single message has to contain all subscription identifiers, if the client had provided such identifiers. We also checked this requirement manually and all brokers conform to this. The example of overlapping subscriptions shows us, that non-conforming traces not always reveal inconsistencies to the specification and, therefore, manual analysis is still required.

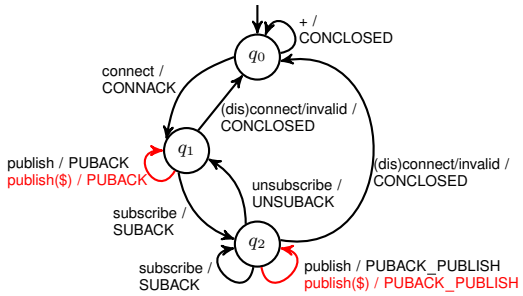

Figure 6. The Eclipse Mosquitto model learned with a separate publish(`$`) input. The red transitions indicate the non-conforming behavior.

V. Related Work

Fuzz testing the MQTT protocol became a prominent topic in the literature due to the rapid increase of devices in the IoT and the correlating popularity of MQTT. The existing fuzzing methods mainly follow black-box techniques [29]–[34], but there also exist gray-box fuzzing methods [35]. Rodriguez and Batista [36] analyze different tools and techniques that apply black-box fuzzing to MQTT brokers and clients. Ramos et al. [29] perform denial-of-service attacks on one MQTT client and broker by using mutation-based fuzzing based on templates for MQTT packages. Casteur et al. [30] refine the fuzzing test suite by assigning scores to test cases that lead to a crash or abnormal behavior. Since the black-box fuzzing techniques of Ramos et al. and Casteur et al. only value the crashes or errors of SUT, they do not recognize any possible unintended state transitions like our stateful fuzzing method. Sochor et al. [32] propose a security-testing framework for MQTT brokers that generates test cases from attack patterns. Similar to our technique, they execute the tests on the Eclipse Mosquitto broker and then check if other brokers behave differently. However, they do not cross-check the Eclipse Mosquitto broker itself. Palmieri et al. [31] present an MQTT security-analysis tool that performs different attacks, including fuzz testing, on the tested broker and then automatically summarizes the findings in a report. Some of their performed checks are similar to our technique, e.g. they test if `$SYS/-` topics are accessible. However, they do not check unexpected state transitions. Gray-box fuzzing is used by Zeng et al. [35]. For this, they integrate in their framework the AFL [22] tool.

We already introduced the work of Tappler et al. [5] which uses learning-based testing to test different MQTT brokers. Using this testing technique, they could reveal several inconsistencies between the broker implementations. Similarly, several works in the literature use active automata learning to find security vulnerabilities in protocols like TLS

[24], DTLS [25], or the 802.11 4-Way Handshake [37]. They refer to this technique as protocol-state fuzzing, since they fuzz the system with various input sequences to learn the SUT. Furthermore, this technique was also applied to test the security of other stateful systems like bank cards [38], biometric passports [39], or hand-held smartcard readers [40]. All of the mentioned security-testing methods learn the systems and perform subsequent manual analysis of the learned models. In contrast to our proposed method, they do not apply additional fuzzing techniques.

There already exists fuzzing techniques [41], [42] that previously infer a model of the SUT and then use the model to fuzz the SUT. In contrast to our active learning approach, these techniques are based on passive learning algorithms, more specifically on state merging. Doupé et al. [41] crawled web applications and used the obtained traces to infer a model of the web application. Afterward, they use the traces of the crawler again, but manipulate the HTTP request with a fuzzing component. Comparetti et al. [42] proposed a framework that follows a similar technique. Their tool infers models of communication protocols. The model is then used as input for a stateful fuzzer that tests the protocols based on a provided grammar for the messages. The disadvantage of passive techniques is that the quality of the inferred model heavily depends on the provided traces.

A fuzzing framework that tests a system based on a behavioral model of the SUT is SNOOZE [43]. Beside the model, this stateful fuzzer also requires user-defined fuzzing scenarios. Based on the model and the scenarios, the tool generates malicious inputs with provided fuzzing techniques. In contrast to our method, they assume a manually crafted model that is derived from the specification.

Smeters et al. [44] combine active automata learning and fuzzing. In contrast to our technique, they use fuzzing as an equivalence oracle during learning. Using this method they can learn a model that better describes the error-handling of the SUL. However, they dismiss the black-box assumption and use the gray-box fuzzer AFL [22].

VI. Conclusion

Summary. We propose a new fuzz-testing technique for stateful black-box systems. We used active automata learning to infer a behavioral model of a software system. The inferred model served as a base for subsequent fuzz testing on different implementations of the learned system. We combined model-based testing and fuzzing techniques to find inconsistencies between the model and the tested systems. The effectiveness of our method has been shown in a case study on a widely used IoT protocol. Our technique found several inconsistencies in five MQTT brokers and a subsequent manual investigation revealed possible vulnerabilities. We managed to forward malicious packets, which could be harmful to components in an MQTT network. We could also mimic internal server communication by a simple user application.

Discussion. Our new approach overcomes the limitations of existing techniques. Promising fuzzing techniques, e.g. [22], may not be applicable, since they require a white-box or gray-box setting. Standard black-box fuzzing, however, hardly achieves in-depth testing. With active automata learning, we can improve this situation by gaining insight into the behavior of the SUT. By actively querying the system, we expect to explore more interesting behavior. We noted that the mapper creation required manual effort. This effort, though, is only needed once and can be compared to other fuzzing techniques that require, e.g., a manual attack derivation [32]. Additionally, automatic vulnerability analysis could support our manual technique.

Future Work. Currently, we are investigating further case studies. A natural progression of this work is to test further functionalities of the MQTT protocol. Existing work on testing MQTT mainly focuses on the older standard [45]. The newer standard v5.0, used in this paper, introduces several new features that considerably increase the power of MQTT, e.g. message expiry or topic aliases. Furthermore, Tappler et al. [5] proposed to also test the MQTT clients and we have seen in our case study that there exist client implementations that are vulnerable to malicious inputs.

Non-deterministic behavior was a challenge that occurred during learning and fuzzing. First, we noticed that the behavior of an MQTT broker depends on time. Ignoring the timed behavior introduces non-determinism. Second, the usage of a too abstract alphabet could lead to non-deterministic observations. In previous work [46], we proposed a learning algorithm for abstracted non-deterministic systems. Using this learning technique can overcome our problems dealing with non-determinism since this algorithm addresses both types of non-determinism. Furthermore, the proposed abstraction mechanism could decrease the size of the learned model, which would be beneficial when considering a multi-client setup.

Acknowledgment

This work was supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments", and the "University SAL Labs" initiative of Silicon Austria Labs (SAL) and its Austrian partner universities for applied fundamental research for electronic based systems. We thank Martin Tappler for the useful hints on MQTT, Jorrit Stramer for implementing the MQTT Client, and the developers of LearnLib.

References

[1] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the internet of things: A survey," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 720–734, 2016. [Online]. Available: https://doi.org/10.1109/JIOT.2015.2505901

[2] T. Frühwirth, L. Krammer, and W. Kastner, "Dependability demands and state of the art in the internet of things," in *ETFA 2015, Luxembourg, September 8-11, 2015*. IEEE, 2015, pp. 1–4. [Online]. Available: https://doi.org/10.1109/ETFA.2015.7301592

[3] B. K. Aichernig and R. Schumi, "How fast is MQTT? - Statistical model checking and testing of IoT protocols," in *QEST 2018, Beijing, China, September 4-7, 2018*, ser. LNCS, A. McIver and A. Horváth, Eds., vol. 11024. Springer, 2018, pp. 36–52. [Online]. Available: https://doi.org/10.1007/978-3-319-99154-2_3

[4] D. A. Peled, M. Y. Vardi, and M. Yannakakis, "Black box checking," *J. Autom. Lang. Comb.*, vol. 7, no. 2, pp. 225–246, 2002. [Online]. Available: https://doi.org/10.25596/jalc-2002-225

[5] M. Tappler, B. K. Aichernig, and R. Bloem, "Model-based testing IoT communication via active automata learning," in *ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE, 2017, pp. 276–287. [Online]. Available: https://doi.org/10.1109/ICST.2017.32

[6] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Trans. Reliab.*, vol. 67, no. 3, pp. 1199–1218, 2018. [Online]. Available: https://doi.org/10.1109/TR.2018.2834476

[7] "OASIS message queuing telemetry transport (MQTT) TC," Organization for the Advancement of Structured Information Standards, Burlington, MA, USA, Standard, 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[8] M. Isberner, F. Howar, and B. Steffen, "The open-source LearnLib - A framework for active automata learning," in *CAV 2015, San Francisco, CA, USA, July 18-24, 2015*, ser. LNCS, D. Kroening and C. S. Pasareanu, Eds., vol. 9206. Springer, 2015, pp. 487–495. [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4_32

[9] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[10] J. Tretmans, "Model based testing with labelled transition systems," in *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, ser. LNCS, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., vol. 4949. Springer, 2008, pp. 1–38. [Online]. Available: https://doi.org/10.1007/978-3-540-78917-8_1

[11] M. P. Vasilevskii, "Failure diagnosis of automata," *Cybernetics*, vol. 9, no. 4, pp. 653–665, Jul 1973. [Online]. Available: https://doi.org/10.1007/BF01068590

[12] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. Software Eng.*, vol. 4, no. 3, pp. 178–187, 1978. [Online]. Available: https://doi.org/10.1109/TSE.1978.231496

[13] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987. [Online]. Available: https://doi.org/10.1016/0890-5401(87)90052-6

[14] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," *Inf. Comput.*, vol. 103, no. 2, pp. 299–347, 1993. [Online]. Available: https://doi.org/10.1006/inco.1993.1021

[15] T. Margaria, O. Niese, H. Raffelt, and B. Steffen, "Efficient test-based model generation for legacy reactive systems," in *Ninth IEEE International High-Level Design Validation and Test Workshop 2004, Sonoma Valley, CA, USA, November 10-12, 2004*. IEEE, 2004, pp. 95–100. [Online]. Available: https://doi.org/10.1109/HLDVT.2004.1431246

[16] O. Niese, "An integrated approach to testing complex systems," Ph.D. dissertation, TU Dortmund, Germany, 2003. [Online]. Available: http://eldorado.uni-dortmund.de:8080/0x81d98002_0x0007b62b

[17] M. Shahbaz and R. Groz, "Inferring Mealy machines," in *FM 2009, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, ser. LNCS, A. Cavalcanti and D. Dams, Eds., vol. 5850. Springer, 2009, pp. 207–222. [Online]. Available: https://doi.org/10.1007/978-3-642-05089-3_14

[18] M. Tappler, "Learning-based testing in networked environments in the presence of timed and stochastic behaviour," Ph.D. dissertation, TU Graz, 2019. [Online]. Available: https://mtappler.files.wordpress.com/2019/12/thesis.pdf

[19] F. Aarts, B. Jonsson, J. Uijen, and F. W. Vaandrager, "Generating models of infinite-state communication protocols using regular inference with abstraction," *Formal Methods Syst. Des.*, vol. 46, no. 1, pp. 1–41, 2015. [Online]. Available: https://doi.org/10.1007/s10703-014-0216-x

[20] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *ACM*, vol. 33, no. 12, pp. 32–44, 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[21] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS 2008, San Diego, CA, USA, February 10-13, 2008*. The Internet Society, 2008. [Online]. Available: https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[22] M. Zalewski, "American fuzzy lop'," https://lcamtuf.coredump.cx/afl/, 2013, Accessed: 2020-10-18.

[23] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: The state of the art," Defence Science and Technology Organisation, Edinburgh, Australia, Tech. Rep., 2012.

[24] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[25] P. Fiterău-Broştean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS implementations using protocol state fuzzing," in *USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 2523–2540. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

[26] B. K. Aichernig, R. Bloem, M. Ebrahimi, M. Tappler, and J. Winter, "Automata learning for symbolic execution," in *FMCAD 2018, Austin, TX, USA, Oct 30 - Nov 2, 2018*, N. Bjørner and A. Gurfinkel, Eds. IEEE, 2018, pp. 1–9. [Online]. Available: https://doi.org/10.23919/FMCAD.2018.8602991

[27] E. Muškardin and A. Pferscher, "Supplemental materials for "Learning-based fuzzing of IoT message brokers"," Jan 2021. [Online]. Available: https://github.com/DES-Lab/Learning-Based-Fuzzing

[28] K. Mladenov, "Formal verification of the implementation of the MQTT protocol in IoT devices," University of Amsterdam, Netherlands, Tech. Rep., 2017.

[29] S. H. Ramos, M. T. Villalba, and R. Lacuesta, "MQTT security: A novel fuzzing approach," *Wirel. Commun. Mob. Comput.*, vol. 2018, 2018. [Online]. Available: https://doi.org/10.1155/2018/8261746

[30] G. Casteur, A. Aubaret, B. Blondeau, V. Clouet, A. Quemat, V. Pical, and R. Zitouni, "Fuzzing attacks for vulnerability discovery within MQTT protocol," in *IWCMC 2020, Limassol, Cyprus, June 15-19, 2020*. IEEE, 2020, pp. 420–425. [Online]. Available: https://doi.org/10.1109/IWCMC48107.2020.9148320

[31] A. Palmieri, P. Prem, S. Ranise, U. Morelli, and T. Ahmad, "MQTTSA: A tool for automatically assisting the secure deployments of MQTT brokers," in *SERVICES 2019, Milan, Italy, July 8-13, 2019*, C. K. Chang, P. Chen, M. Goul, K. Oyama, S. Reiff-Marganiec, Y. Sun, S. Wang, and Z. Wang, Eds. IEEE, 2019, pp. 47–53. [Online]. Available: https://doi.org/10.1109/SERVICES.2019.00023

[32] H. Sochor, F. Ferrarotti, and R. Ramler, "Automated security test generation for MQTT using attack patterns," in *ARES 2020, Ireland, August 25-28, 2020*, M. Volkamer and C. Wressnegger, Eds. ACM, 2020, pp. 97:1–97:9. [Online]. Available: https://doi.org/10.1145/3407023.3407078

[33] F-Secure Corporation, "mqtt_fuzz," https://github.com/F-Secure/mqtt_fuzz, Accessed: 2020-10-18.

[34] Eclipse Foundation, "Fuzzing Proxy," https://iottestware.readthedocs.io/en/development/smart_fuzzer.html, Accessed: 2020-10-18.

[35] Y. Zeng, M. Lin, S. Guo, Y. Shen, T. Cui, T. Wu, Q. Zheng, and Q. Wang, "Multifuzz: A coverage-based multiparty-protocol fuzzer for iot publish/subscribe protocols," *Sensors*, vol. 20, no. 18, p. 5194, 2020. [Online]. Available: https://doi.org/10.3390/s20185194

[36] L. G. A. Rodriguez and D. M. Batista, "Program-aware fuzzing for MQTT applications," in *ISSTA '20, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds. ACM, 2020, pp. 582–586. [Online]. Available: https://doi.org/10.1145/3395363.3402645

[37] C. M. Stone, T. Chothia, and J. de Ruiter, "Extending automated protocol state learning for the 802.11 4-way handshake," in *ESORICS 2018, Barcelona, Spain, September 3-7, 2018*, ser. LNCS, J. López, J. Zhou, and M. Soriano, Eds., vol. 11098. Springer, 2018, pp. 325–345. [Online]. Available: https://doi.org/10.1007/978-3-319-99073-6_16

[38] F. Aarts, J. de Ruiter, and E. Poll, "Formal models of bank cards for free," in *ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE, 2013, pp. 461–468. [Online]. Available: https://doi.org/10.1109/ICSTW.2013.60

[39] F. Aarts, J. Schmaltz, and F. W. Vaandrager, "Inference and abstraction of the biometric passport," in *ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010*, ser. LNCS, T. Margaria and B. Steffen, Eds., vol. 6415. Springer, 2010, pp. 673–686. [Online]. Available: https://doi.org/10.1007/978-3-642-16558-0_54

[40] G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter, "Automated reverse engineering using Lego®," in *WOOT '14, San Diego, CA, USA, August 19, 2014*, S. Bratus and F. F. Lindner, Eds. USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/chalupar

[41] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *21th USENIX Security, Bellevue, WA, USA, August 8-10, 2012*, T. Kohno, Ed. USENIX Association, 2012, pp. 523–538. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe

[42] P. M. Comparetti, G. Wondracek, C. Krügel, and E. Kirda, "Prospex: Protocol specification extraction," in *S&P 2009, 17-20 May 2009, Oakland, CA, USA*. IEEE, 2009, pp. 110–125. [Online]. Available: https://doi.org/10.1109/SP.2009.14

[43] G. Banks, M. Cova, V. Felmetsger, K. C. Almeroth, R. A. Kemmerer, and G. Vigna, "SNOOZE: toward a stateful network protocol fuzzer," in *ISC 2006, Samos Island, Greece, Aug 30 - Sep 2, 2006*, ser. LNCS, S. K. Katsikas, J. López, M. Backes, S. Gritzalis, and B. Preneel, Eds., vol. 4176. Springer, 2006, pp. 343–358. [Online]. Available: https://doi.org/10.1007/11836810_25

[44] R. Smetsers, J. Moerman, M. Janssen, and S. Verwer, "Complementing model learning with mutation-based fuzzing," *CoRR*, vol. abs/1611.02429, 2016. [Online]. Available: http://arxiv.org/abs/1611.02429

[45] "Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1, ISO/IEC 20922:2016," International Organization for Standardization, Geneva, CH, Standard, Jun. 2016.

[46] A. Pferscher and B. K. Aichernig, "Learning abstracted non-deterministic finite state machines," in *ICTSS 2020, Naples, Italy, December 9-11, 2020*, ser. LNCS, V. Casola, A. D. Benedictis, and M. Rak, Eds., vol. 12543. Springer, 2020, pp. 52–69. [Online]. Available: https://doi.org/10.1007/978-3-030-64881-7_4