Fingerprinting Bluetooth Low Energy Devices via Active Automata Learning

Andrea Pferscher and Bernhard K. Aichernig[0000−0002−3484−5584]

Institute of Software Technology, Graz University of Technology, Graz, Austria
{apfersch,aichernig}@ist.tugraz.at

Abstract. Active automata learning is a technique to automatically infer behavioral models of black-box systems. Today's learning algorithms enable the deduction of models that describe complex system properties, e.g., timed or stochastic behavior. Despite recent improvements in the scalability of learning algorithms, their practical applicability is still an open issue. Little work exists that actually learns models of physical black-box systems. To fill this gap in the literature, we present a case study on applying automata learning on the Bluetooth Low Energy (BLE) protocol. It shows that not the size of the system limits the applicability of automata learning. Instead, the interaction with the system under learning, is a major bottleneck that is rarely discussed. In this paper, we propose a general automata learning architecture for learning a behavioral model of the BLE protocol implemented by a physical device. With this framework, we can successfully learn the behavior of five investigated BLE devices. The learned models reveal several behavioral differences. This shows that automata learning can be used for fingerprinting black-box devices, i.e., identifying systems via their specific learned models. Based on the fingerprint, an attacker may exploit vulnerabilities specific to a device.

Keywords: Active automata learning · Model inference · Learning-based testing · Fingerprinting · Bluetooth Low Energy · IoT

1 Introduction

Bluetooth is a key communication technology in many different fields. Currently, it is assumed that 4.5 billion Bluetooth devices are shipped annually and that the number will grow to 6.4 billion by 2025 [9]. This growth mainly refers to the increase of peripheral devices that support Bluetooth Low Energy (BLE). With BLE, Bluetooth became also accessible for low-energy devices. Hence, BLE is a vital technology in the Internet of Things (IoT).

The amount of heterogeneous devices in the IoT makes the assurance of dependability a challenging task. Additionally, the insight into IoT components is frequently limited. Therefore, the system under test must be considered as a black-box. Enabling in-depth testing of black-box systems is difficult, but can be achieved with model-based testing techniques. Garbelini et al. [17] successfully used a generic model of the BLE protocol to detect security vulnerabilities

of BLE devices via model-based fuzzing. However, their work states that the creation of such a comprehensive model was challenging since the BLE protocol has high degrees of freedom. In practice, the creation of such a model is an error-prone process and is usually not feasible.

To overcome the problem of model availability, learning-based testing techniques have been proposed [4]. In learning-based testing, we use automata learning algorithms to automatically infer a behavioral model of a black-box system. The learned model could then be used for further verification. Motivated by promising results of learning-based testing, various automata learning algorithms have been proposed to extend learning for more complex system properties like timed [32,6] or stochastic behavior [30]. However, few of these algorithms have been evaluated on systems in practice.

In this paper, we present a case study that applies active automata learning on real physical devices. Our objective is to learn the behavioral model of the BLE protocol implementation. For this, we propose a general automata-learning framework that automatically infers the behavioral model of BLE devices. Our presented framework uses state-of-the-art automata learning techniques. We adapt these algorithms considering practical challenges that occur in learning real network components.

In our case study, we present our results on learning five different BLE devices. Based on these results, we stress two different findings. First, we observe that the implementations of the BLE stacks differ from device to device. Using this observation, we show that active automata learning can be used to identify black-box systems. That is, our proposed framework generates a fingerprint of a BLE device. Second, the presented performance metrics show that not only does the system's size influences the performance of the learning algorithm. Additionally, the creation of a deterministic learning setup creates a significant overhead which has an impact on the efficiency of the learning algorithm, since we have to repeat queries and wait for answers.

The contribution of this paper is threefold: First, we present our developed framework that enables learning of BLE protocol implementations of peripheral devices. Second, we present the performed case study that evaluates our framework on real physical devices. The framework including the learned models is available **online**[1][22]. Third, we propose how our presented technique can be used to fingerprint black-box systems.

The paper is structured as follows. Section 2 discusses the used modeling formalism, active automata learning, and the BLE protocol. In Section 3, we propose our learning architecture, followed by the performed evaluation based on this framework in Section 4. Section 5 discusses related work and Section 6 concludes the paper.

---

[1] https://github.com/apferscher/ble-learning

2   Preliminaries

2.1   Mealy Machines

Mealy machines represent a neat modeling formalism for systems that create observable outputs after an input execution, i.e., reactive systems. Moreover, many state-of-the-art automata learning algorithms and frameworks [18,20] support Mealy machines. A Mealy machine is a finite state machine, where the states are connected via transitions that are labeled with input actions and the corresponding observable outputs. Starting from an initial state, input sequences can be executed and the corresponding output sequence is returned. Definition 1 formally defines Mealy machines.

Definition 1 (Mealy machine). *A Mealy machine is a 6-tuple* $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ *where*

- Q *is the finite set of states*
- q_0 *is the initial state*
- I *is the finite set of inputs*
- O *is the finite set of outputs*
- $\delta : Q \times I \to Q$ *is the state-transition function*
- $\lambda : Q \times I \to O$ *is the output function*

To ensure learnability, we require $\mathcal{M}$ to be deterministic and input-enabled. Hence, δ and λ are total functions. Let S be the set of observable sequences, where a sequence $s \in S$ consists of consecutive input/output pairs $(i_1, o_1), \ldots, (i_i, o_i), \ldots, (i_n, o_n)$ with $i_i \in I$, $o_i \in O$, $i \leq n$ and $n \in \mathbb{N}$ defining the length of the sequence. We define $s_I \in I^*$ as the corresponding input sequence of s, and $s_O \in O^*$ maps to the output sequence. We extend δ and λ for sequences. The state transition function $\delta^* : Q \times I^* \to Q$ gives the reached state after the execution of the input sequence and the output function $\lambda^* : Q \times I^* \to O^*$ returns the observed output sequence. We define two Mealy machines $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ and $\mathcal{M}' = \langle Q', q_0', I, O, \delta', \lambda' \rangle$ as equal if $\forall s_i \in I^* : \lambda^*(q_0, s_i) = \lambda'^*(q_0', s_i)$, i.e. the execution of all input sequences lead to equal output sequences.

2.2   Active Automata Learning

In automata learning, we learn a behavioral model of a system based on a set of execution traces. Depending on the generation of these traces, we distinguish between two techniques: *passive* and *active* learning. Passive techniques reconstruct the behavioral model from a given set of traces, e.g., log files. Consequently, the learned model can only be as expressive as the provided traces. Active techniques, instead, actively query the system under learning (SUL). Hence, actively learned models are more likely to cover rare events that cannot be observed from ordinary system monitoring.

Many current active learning algorithms build upon the L^* algorithm proposed by Angulin [7]. The original algorithm learns the minimal deterministic

finite automaton (DFA) of a regular language. Angluin's seminal work introduces the minimally adequate teacher (MAT) framework, which comprises two members: the *learner* and the *teacher*. The learner constructs a DFA by questioning the teacher, who has knowledge about the SUL. The MAT framework distinguishes between *membership* and *equivalence* queries. Using membership queries, the learner asks if a word is part of the language, which can be either answered with *yes* or *no* by the teacher. Based on these answers, the learner constructs an initial behavioral model. The constructed hypothesis is then provided to the teacher in order to ask if the DFA conforms to the SUL, i.e. the learner queries equivalence. The teacher answers to equivalence queries either with a counterexample that shows non-conformance between the hypothesis and the SUL or by responding *yes* to affirm conformance. In the case that a counterexample is responded, the learner uses this counterexample to pose new membership queries and construct a new hypothesis. This procedure is repeated until a conforming hypothesis is proposed.

The L^* algorithm has been extended to learn Mealy machines of reactive systems [19,21,27]. To learn Mealy machines, membership queries are replaced by output queries. For this, the learner asks for the output sequence on a given sequence of inputs. We assume that the teacher has access to the SUL in order to execute inputs and observe outputs.

In practice, we cannot assume a *perfect* teacher who provides the shortest counterexample that shows non-conformance between the hypothesis and the SUL. To overcome this problem, we use conformance testing to substitute equivalence queries. For this, we need to define a conformance relation between the hypothesis and the SUL based on testing. Tretmans [33] introduces an implementation relation $\mathcal{I}$ **imp** $\mathcal{S}$, which defines conformance between an implementation $\mathcal{I}$ and a specification $\mathcal{S}$. In model-based testing, $\mathcal{I}$ would be a black-box system and $\mathcal{S}$ a formal specification in terms of a model, e.g., a Mealy machine. Furthermore, he denotes that $\mathcal{I}$ **passes** t if the execution of the test t on $\mathcal{I}$ leads to the expected results. Based on a test suite $T_\mathcal{S}$ that adequately represents the specification $\mathcal{S}$, Tretmans defines the conformance relation as follows.

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

Informally, $\mathcal{I}$ conforms to $\mathcal{S}$, if $\mathcal{I}$ passes all test cases. We apply this conformance relation for conformance testing during learning. In learning, we try to verify if the learned hypothesis $\mathcal{H}$ conforms to the black-box SUL $\mathcal{I}$, i.e., if the relation $\mathcal{H}$ **imp** $\mathcal{I}$ is satisfied. Furthermore, we assume that $\mathcal{I}$ can be represented by the modeling formalism of $\mathcal{H}$. Based on the definition of equivalence of Mealy machines, Tappler [29] stresses that $\mathcal{I}$ **imp** $\mathcal{H} \Leftrightarrow \mathcal{H}$ **imp** $\mathcal{I}$ holds. Therefore, we can define the conformance relation for learning Mealy machines based on a test suite $T \subseteq I^*$ as follows.

$$\mathcal{H} \textbf{ imp } \mathcal{I} \Leftrightarrow \forall t \in T : \lambda_\mathcal{H}^*(q_0^\mathcal{H}, t) = \lambda_\mathcal{I}^*(q_0^\mathcal{I}, t) \tag{2}$$
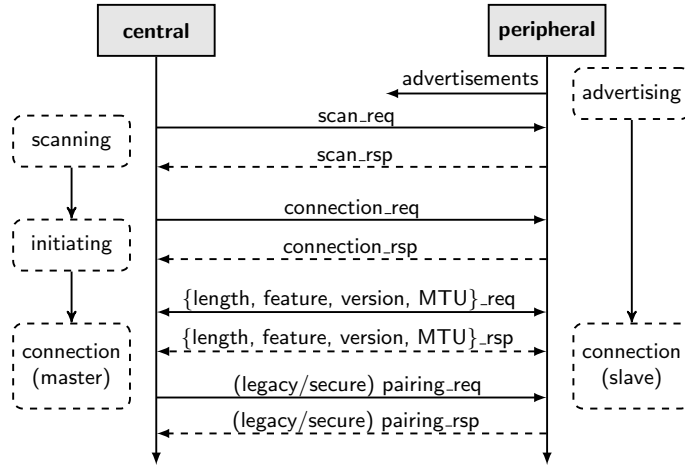
Fig. 1. Communication between a BLE central and peripheral to establish connection. The sequence diagram is adapted from [17].

2.3   Bluetooth Low Energy

The BLE protocol is a lightweight alternative to the classic Bluetooth protocol, specially designed to provide a low-energy alternative for IoT devices. The Bluetooth specification [10] defines the connection protocol between two BLE devices according to different layers of the BLE protocol stack. Based on the work of Garbelini et al. [17], Figure 1 shows the initial communication messages of two connecting BLE devices on a more abstracted level. We distinguish between the *peripheral* and the *central* device. In the remainder of this paper, we refer to the central device simply as *central* and to the peripheral device as *peripheral*. The peripheral sends advertisements to show that it is available for connection with a central. According to the BLE specification, the peripheral is in the *advertising* state. If the central scans for advertising devices it is in the *scanning* state. For this, the central sends a scan request (scan_req) to the peripheral, which response with a scan response (scan_rsp). In the next step, the central changes from the *scanning* to the *initiating* state by sending the connection request (connection_req). If the peripheral answers with a connection response (connection_rsp), the peripheral and central enter the *connection* state. The BLE specification defines now the central as *master* and the peripheral as *slave*. After the connection, the negotiation on communication parameters starts. Both the central and peripheral can request features or send control packages. These request and control packages include maximum package length, maximum transmission unit (MTU), BLE version, and feature exchanges. As noted by Garbelini et al. [17], the order of the feature requests is not defined in the BLE specification and can differ for each device. After this parameter negotiation, the pairing procedure starts by sending a pairing request (pairing_req) from the central to the peripheral, answered by a pairing response (pairing_rsp). The BLE protocol distinguishes two pairing procedures: *legacy* and *secure* pairing. In the remainder of this paper, we will only consider *secure* pairing requests.
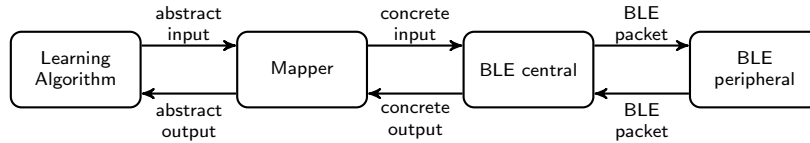
Fig. 2. Similar to the interface of Tappler et al. [31] we create a learning architecture to execute abstract queries on the BLE peripheral.

3   Learning Setup

Our objective is to learn the behavioral model of the BLE protocol implemented by the peripheral device. The learning setup is based on active automata learning, assuming that unusual input sequences reveal characteristic behavior that enables fingerprinting. According to Section 2.3, we can model the BLE protocol as a reactive system. Tappler et al. [31] propose a learning setup for network components. Following a similar architecture, we propose a general learning framework for the BLE protocol. Figure 2 depicts the four components of the learning interface: (1) learning algorithm, (2) mapper, (3) BLE central and (4) BLE peripheral.

(1) The applied *learning algorithm* is an improved variant of the L^* algorithm. Since L^* is based on an exhaustive input exploration in each state, we assume that it is beneficial for fingerprinting. Rivest and Schapire [24] proposed the improved L^* version that contains an advanced counterexample processing. This improvement might reduce the number of required output queries. Considering that the BLE setup is based on Python, we aim at a consistent learning framework integration. At present, AALPY [20] is a novel active learning library that is also written in Python. AALPY implements state-of-the-art learning algorithms and conformance testing techniques, including the improved L^* variant that is considered here. Since the framework implements equivalence queries via conformance testing, we assume that the conformance relation defined in Equation 2 holds. To create a sufficient test suite, we combine random testing with state coverage. The applied test-case generation technique generates for each state in the hypothesis n_{test} input traces. The generated input traces of length n_{len} comprise the input prefixes to the currently considered state concatenated with a random input sequence.

Learning physical devices via a wireless network connection introduces problems that hamper the straightforward application of the learning algorithm. We observe two main problems: package loss and non-deterministic behavior. Both problems required adaptions of the AALPY framework. Package loss might be critical for packages that are necessary to establish a connection. To overcome unexpected connection losses, we assume that the scanning and connection requests are always answered by corresponding responses of the peripheral. If we do not receive such a response, we assume that the request was lost and report a connection error. In the case of a connection error, we repeat the performed output query. To guarantee termination, the query is only repeated up to n_{error} times. After n_{error} repetitions, we abort the learning procedure.

We pursue an akin procedure for non-deterministic behavior. Non-determinism might occur due to the loss or delay of responses. In Section 4, we discuss further causes of non-deterministic behavior that we experienced during learning. If we observe non-determinism, we repeat the output query. Again we define an upper limit for a repeating non-deterministic behavior by a maximum of n_{nondet} query executions.

The applied learning algorithm requires that the SUL is resettable since it is expected that every output query is executed from the initial state of the SUL. The learning library AALPY can perform resetting actions before and after the output query execution. We denote the method that is called *before* executing the output query as `pre` and the method *after* the output query as `post`. We assume that the peripheral can be reset by the central by sending a `scan_req`. To ensure a proper reset before executing the output query, a scan request is performed in the `pre` method.

Besides the reset, we have to consider that some peripherals might enter a standby state in which they stop advertising. This could be the case, e.g., if the peripheral does not receive any expected commands from the central after a certain amount of time. The main problem of a peripheral entering the standby state is that the central might not be able to bring back the peripheral to the advertising state. To prevent the peripheral from entering the standby state, we send keep-alive messages in the `pre` and `post` method. These keep-alive messages include a connection request followed by a scan request. To ensure a proper state before executing the output query, we check for connection errors during the keep-alive messages as previously described.

(2) The *mapper* component serves as an abstraction mechanism. Considering a more universal input and output alphabet, we learn a behavioral model on a more abstract level. The learning algorithm, therefore, generates output queries that comprise abstract input sequences. The mapper receives these abstract inputs and translates them to concrete inputs that can be executed by the central. After the central received a concrete input action, the central returns the corresponding concrete output. This concrete output is then taken by the mapper and translated to a more abstract output that is used by the learning algorithm to construct the hypothesis.

The abstracted input alphabet to learn the behavior of the BLE protocol implementations is defined by $I^{\mathcal{A}} = \{\mathsf{scan\_req}, \mathsf{connection\_req}, \mathsf{length\_req}, \mathsf{length\_rsp}, \mathsf{feature\_req}, \mathsf{feature\_rsp}, \mathsf{version\_req}, \mathsf{mtu\_req}, \mathsf{pairing\_req}\}$. The abstract inputs of $I^{\mathcal{A}}$ are then translated to concrete BLE packages that can be sent by the central to the peripheral. For example, the abstract input $\mathsf{length\_req}$ is translated to a BLE control package including a corresponding valid command of the BLE protocol stack. For the construction of the BLE packages we use the Python library SCAPY [26]. In SCAPY syntax the BLE package for the $\mathsf{length\_req}$ can be defined as $\mathsf{BTLE/BTLE\_DATA/BTLE\_CTRL/LL\_LENGTH\_REQ}(params)$.

Considering the input/output definition of reactive systems, it may be unusual to include responses in the input alphabet. For our setup, we included the feature and length response as inputs. In Section 2.3, we explained that after the

connection request of the central, also the peripheral might send control packages or feature requests. To explore more behavior of the peripheral, we have to reply to received requests from the peripheral. In a learning setup, the inputs feature_rsp and length_rsp are responses from the central to received outputs from the peripheral that contain requests. For learning an expressive behavioral model, we consider responses to feature and length requests, i.e. feature_rsp and length_rsp, as additional inputs.

Regarding translation of outputs, the mapper returns the received BLE packages conforming to the SCAPY syntax. One exception applies to the response on scan_req, where two possible valid responses are mapped to one scan response (ADV). In the BLE protocol it is possible that one input might lead to multiple responses that are distributed via individual BLE packages. For the creation of a single output, the mapper collects several responses in a set. The collected outputs in the set are then concatenated in alphabetical order to one output string. This creates deterministic behavior, even though packages might be received in a a different order. We repeat the collection of BLE package responses at least $n_{\min}^{\mathrm{rsp}}$ times. If after $n_{\min}^{\mathrm{rsp}}$ responses no convincing response has been returned, we continue listening for responses. We define a response as *convincing*, if the received package contains more than a BLE data package, i.e. BTLE/BTLE_DATA. However, the maximum number of listening attempts is limited by $n_{\max}^{\mathrm{rsp}}$. If we do not receive any BLE package after $n_{\max}^{\mathrm{rsp}}$, the mapper returns the empty output which is denoted by the string EMPTY. As previously mentioned, the assumption of an empty response is not valid for scan and connection requests. In the case of $n_{\max}^{\mathrm{rsp}}$ empty responses, we perform the described connection-error handling.

(3) The *BLE central* component comprises the adapter implementation and the physical central device. We use the Nordic nRF52840 USB dongle as central. Our learning setup requires to stepwise send BLE packages to the peripheral device. For this, our implementation follows the setup proposed by Garbelini et al. [17]. We use their provided firmware for the Nordic nRF52840 System on a Chip (SoC) and adapted their driver implementation to perform single steps of the BLE protocol.

(4) The *BLE peripheral* represents the black-box device that we want to learn, i.e., the SUL. We assume that the peripheral is advertising and only interacts with our central device. For learning, we require that the peripheral is resettable and that the reset can be initiated by the central. After a reset, the peripheral should be again in the advertising state.

4   Evaluation

We evaluated the proposed automata learning setup for the BLE protocol in a case study consisting of five different BLE devices. The learning framework is available **online**[2][22]. The repository contains the source code for the BLE learning framework, the firmware for the Nordic nRF52840 Dongle and Nordic nRF52840 Development Kit, the learned automata, and the learning results.

---

[2] https://github.com/apferscher/ble-learning

Table 1. Evaluated BLE devices

Company (Board)	SoC	Application
Cypress (CY8CPROTO-063-BLE)	CYBLE-416045-02	Find Me Target
Nordic (decaWave DWM1001-DEV)	nRF52832	Nordic GATTS
Texas Instruments (LAUNCHXL-CC2640R2)	CC2640R2	Project Zero
Texas Instruments (LAUNCHXL-CC2650)	CC2650	Project Zero
Cypress (Raspberry Pi 4 Model B)	CYW43455	BlueZ GATT Server[3]

4.1 BLE Devices

Table 1 lists the five investigated BLE devices. In the remainder of this section, we refer to the BLE devices by their SoC identifiers. All evaluated SoCs support the Bluetooth v5.0 standard [10]. To enable a BLE communication, we deployed and ran an example of a BLE application on the SoC. The considered BLE applications were either already installed by the semiconductor manufacturer or taken from examples in the semiconductor's specific software development kits. In the case of the CYW43455 (Raspberry Pi), an example code from the internet was used.

4.2 BLE Learning

For our learning setup, we used the Python learning library AALpy [20] (version 1.0.1). For the composition of the BLE packages, we used a modified version of the Python library Scapy [26] (version 2.4.4). The used modifications are now available on Scapy v2.4.5. All experiments were performed with Python 3.9.0 on an Apple MacBook Pro 2019 with an Intel Quad-Core i5 operating at 2.4 GHz and with 8 GB RAM. As BLE central device, we used the Nordic nRF52840 Dongle. The deployed firmware for the USB dongle was taken from the SweynTooth repository [16].

Learning the communication protocol in use by interacting with a non-simulated physical device may cause unexpected behavior, e.g., the loss of transmitted packages. This erroneous behavior can cause missing responses or non-deterministic behavior. To adapt the AALpy framework for such a real-world setup, we modified the implementation of the equivalence oracle and the used caching mechanism. These modifications of our framework handle connection errors and non-deterministic outputs according to our explanation in Section 3. For this, we set the maximum number of consecutive connection errors $n_{\text{error}} = 20$ and the number of consecutive non-deterministic output queries to $n_{\text{nondet}} = 5$. Our experiments show that this parameters setup created a stable and fast learning setup.

For conformance testing, we copied the class `StatePrefixEqOracle` from AALpy and added our error handling behavior. The number of performed queries per state is set to $n_{\text{test}} = 10$ and the number of performed inputs per query is set to $n_{\text{len}} = 10$. We stress that the primary focus of this paper was

---

[3] `https://scribles.net/creating-ble-gatt-server-uart-service-on-raspberry-pi/`

Table 2. Learning results of four out of five evaluated BLE SoCs

	CYBLE-416045-02	nRF52832	CC2650	CYW43455
# States	3	5	5	11
Total Time in minutes (min)	25.86	151.47	49.63	209.43
Learning (min)	20.42	85.43	36.57	159.66
Conformance Checking (min)	5.44	66.04	13.06	49.77
# Output Queries	243	406	405	891
# Output Query Steps	729	1461	1458	4131
# Conformance Tests	30	50	50	111
# Conformance Test Steps	330	580	580	1403
# Connection Errors	555	913	910	2071
# Non-Deterministic Queries	0	1	0	0

to generate a fingerprint of the investigated BLE SoCs. Therefore, it was sufficient to perform a lower number of conformance tests. However, we recommend increasing the number of conformance tests if a more accurate statement about conformance of the model to the SUL is required.

In Section 3, we explained that a sent BLE message could lead to multiple responses. These responses can be distributed over several BLE packages. Hence, our central listens for a minimum number of responses $n_{\min}^{\mathrm{rsp}}$, but stops listening after $n_{\max}^{\mathrm{rsp}}$ attempts. For our learning setup, we set for all SoCs $n_{\min}^{\mathrm{rsp}} = 20$ and $n_{\max}^{\mathrm{rsp}} = 30$. Experiments during our evaluation show that this setup enables stable and fast learning for all SoCs. However, we decided to create a different parameter setup for the scan request. The parameter setup depends on the purpose of the request. We distinguish between two cases. In the first case, we perform the scan request to reset the SUL. On the one hand, we want to continue fast if we receive a response, therefore, $n_{\min}^{\mathrm{rsp}} = 5$. On the other hand, we want to be sure that the SUL is properly reset, therefore $n_{\max}^{\mathrm{rsp}} = 100$. The second case occurs during learning where the scan request is included as an input action in an output query. For this purpose, we decrease the parameters to $n_{\min}^{\mathrm{rsp}} = 5$ and $n_{\max}^{\mathrm{rsp}} = 20$, since the query is repeated in case of a missing response.

Table 2 shows learning results for four out of the five investigated SoCs. Results of CC2640R2 are not included, since we were not able to learn a deterministic model of CC2640R2 using the defined input alphabet. We discuss possible reasons for the non-deterministic behavior later. For all other SoCs, we learned a deterministic Mealy machine using the complete input alphabet.

We required for each SUL one learning round, i.e. we did not find a counterexample to conformance between the initially created hypothesis and the SUL. The learned behavioral models range from a simpler structure with only three states (CYBLE-416045-02) to more complex behavior that can be described by eleven states (CYW43455).

The learning of the largest model regarding the number of states (CYW43455) took approximately 3.5 hours, whereas the smallest model (CYBLE-416045-02) could be learned in less than half an hour. We observed that the total runtime for SoCs with a similar state space (nRF52832 and CC2650) significantly differs. The results presented in Table 2 show that learning the nRF52832 took

three times as long as learning the CC2650, where both learned models have five states. The difference in runtime indicates that the scalability of active automata learning does not merely depends on the input alphabet size and state space of the SUL. Rather, we assume that the overhead to create a deterministic learning setup, e.g. repeating queries or waiting for answers, also influences the efficiency of active automata learning.

Conforming to the state space, the number of performed output queries and steps increases. Rather unexpected, also the number of connection errors seems to align with the complexity of the behavioral model. Therefore, we assume that message loss regularly occurs in our learning setup. The comparison between the number of performed output queries, including conformance tests, and the observed connection errors show that more connection errors occur than output queries are performed. Since an output query would have been repeated after a connection error, we assume that we observe more connection errors in the resetting procedure. This creates our conjecture that a decent error-handling resetting procedure is required to ensure that the SUL is reset to the initial state before the output query is executed. Furthermore, we observe fewer connection errors and non-determinism during the output queries. Hence, we assume that our proposed learning setup appropriately resets the SUL.

Figure 3 shows the learned model of the nRF52832 and Fig. 4 of the CC2650. To provide a clear and concise representation, we merged and simplified transitions. The unmodified learned models of all SoCs considered in this case study are available online[4]. The comparison between the learned models of the nRF52832 (Fig. 3) and the CC2650 (Fig. 4) shows that even models with the same number of states describe different BLE protocol stack implementations. We highlighted in red for both models the transitions that show a different behavior on the input length_rsp. The nRF52832 responds to an unrequested length response only with a BLE data package and then completely resets the connection procedure. Therefore, executing an unexpected length response on the nRF52832 leads to the initial state akin to the performance of a scan request. The CC2650, instead, reacts to an unrequested length response with a response containing the package LL_UNKNOWN_RSP and remains in the same state.

Using the learning setup of Section 3, we could not learn the CC2640R2. Independent from the adaption of our error handling parameters, we always observed non-deterministic behavior. More interestingly, the non-deterministic behavior could repeatedly be observed on the following output query.

$$\text{connection\_req} \cdot \text{pairing\_req} \cdot \text{length\_rsp} \cdot \text{length\_req} \cdot \text{feature\_req}$$

In earlier stages of the learning procedure, we observed the following output sequence after the execution of the inputs.

LL_LENGTH_REQ·SM_PAIRING_RSP·BTLE_DATA·LL_LENGTH_RSP·LL_FEATURE_RSP

Later in learning, we never again received any feature response for the input feature_req if we executed this output query. The observed outputs always cor-

---

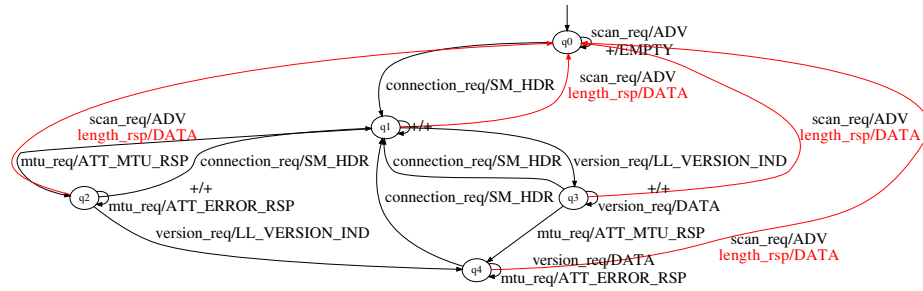[4] https://github.com/apferscher/ble-learning

Fig. 3. Simplified learned model of the nRF52832. Inputs are lowercased and outputs are capitalized. For a clear presentation, received outputs are abbreviated, and input and outputs are summarized by the +-symbol.
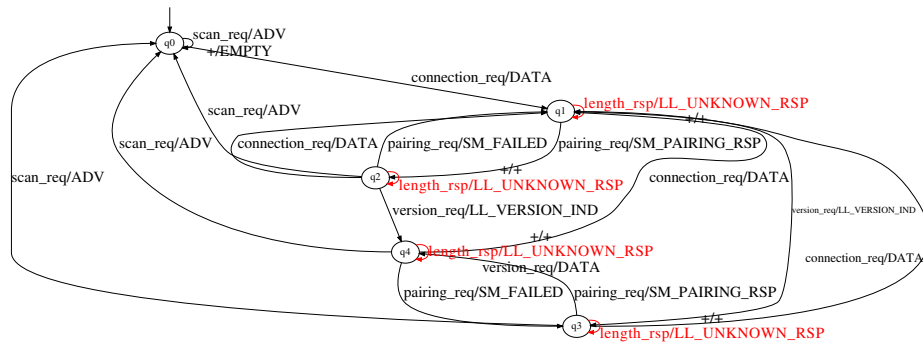

Fig. 4. Simplified learned model of the CC2650.

Table 3. The non-deterministic behavior of the CC2640R2 BLE SoC disabled learning considering the entire input alphabet. The table shows the results of learning with a reduced input alphabet.

	no pairing_req	no length_req	no feature_req
# States	6	11	11
Total Time (min)	54.17	88.76	87.15
Learning Time (min)	40.83	61.57	60.28
Conformance Checking Time (min)	13.34	27.19	26.87
# Output Queries	390	705	704
# Output Query Steps	1499	3141	3136
# Conformance Tests	61	110	110
# Conformance Test Steps	710	1370	1370
# Connection Errors	978	1657	1628
# Non-Deterministic Queries	7	1	0

responded to the following sequence.

LL_LENGTH_REQ · SM_PAIRING_RSP · BTLE_DATA · LL_LENGTH_RSP · <u>BTLE_DATA</u>

If we remove one of the inputs pairing_req, length_req or feature_req, our learning setup successfully learned a deterministic model. Table 3 shows the learning results for the CC2640R2 with the adapted input alphabets. Compared to the results in Table 2, we observe more non-deterministic behavior, which led to repetitions of output queries.

4.3   BLE Fingerprinting

The comparison of the learned models shows that all investigated SoCs behave differently. Therefore, it is possible to uniquely identify the SoC. The advantage of active automata learning, especially using L^*-based algorithms, is that every input is queried in each state to uniquely identify a state of the model. The collected query information can then be used to fingerprint the system. A closer look at the models shows that even short input sequences sufficiently fingerprint the SoC.

In our BLE learning setup, we noticed that for each learned model, an initial connection request leads to a new state. Table 4 shows the observable outputs for each input after performing the initial connection request connect_req, i.e., the table shows the outputs that identify the state for the corresponding SoC. We determine that the set of observable outputs after an initial connection request is different for every SoC.

A closer look at the observable outputs shows that a combination of only two observable outputs is enough to identify the SoC. We highlight in Table 4 two possible output combinations that depict the fingerprint of a SoC. We note that also other output combinations are possible. We can now use the corresponding inputs to generate a single output query that uniquely identifies one of our

Table 4. The investigated SoCs can be identified by only a single model state that is reached after performing an initial connection request. The columns of the table present the outputs that are observed when the input (row) is executed in the connection state. The observable outputs show that only two inputs are required to distinguish the SoCs.

	CYBLE-416045-02	nRF52832	CC2640R2	CC2650	CYW43455
scan_req	ADV	ADV	ADV	ADV	ADV
connect_req	BTLE_DATA	SM_HDR/RAW	LL_LENGTH_REQ	BTLE_DATA	LL_FEATURE_REQ
length_req	LL_UNKNOWN_RSP	LL_LENGTH_RSP	LL_LENGTH_RSP	LL_UNKNOWN_RSP	LL_LENGTH_RSP
length_rsp	LL_UNKNOWN_RSP	BTLE_DATA	BTLE_DATA	LL_UNKNOWN_RSP	LL_REJECT_RSP
feature_req	LL_FEATURE_RSP	LL_FEATURE_RSP	LL_FEATURE_RSP	LL_FEATURE_RSP	LL_FEATURE_RSP
feature_rsp	**LL_REJECT_RSP**	**LL_UNKNOWN_RSP**	**BTLE_DATA**	**BTLE_DATA**	**LL_LENGTH_REQ**
version_req	**LL_VERSION_IND**	**LL_VERSION_IND**	**BTLE_DATA**	**LL_VERSION_IND**	**LL_VERSION_IND**
mtu_req	ATT_MTU_RSP	ATT_MTU_RSP	ATT_MTU_RSP	ATT_MTU_RSP	BTLE_DATA
pairing_req	SM_FAILED	SM_PAIRING	SM_PAIRING	SM_PAIRING	BTLE_DATA

investigated SoCs. Under the consideration that a scan request resets the SoC, we define the fingerprinting for the five SoCs output query as follows.

$$\text{scan\_req} \cdot \text{connection\_req} \cdot \text{feature\_rsp} \cdot \text{scan\_req} \cdot \text{connection\_req} \cdot \text{version\_req}$$

The execution of this output query leads to a different observed output sequence for each of the five investigated SoCs. For example, the corresponding output sequence for the nRF52832 is

$$\text{ADV} \cdot \text{SM\_HDR} \cdot \text{LL\_UNKNOWN\_RSP} \cdot \text{ADV} \cdot \text{SM\_HDR} \cdot \text{LL\_VERSION\_IND},$$

whereas the sequence for the CC2650 is

$$\text{ADV} \cdot \text{BTLE\_DATA} \cdot \text{BTLE\_DATA} \cdot \text{ADV} \cdot \text{BTLE\_DATA} \cdot \text{LL\_VERSION\_IND}.$$

The proposed manual analysis serves as a proof of concept that active automata learning can be used for fingerprinting BLE SoCs. Obviously, the found input sequences for fingerprinting are only valid for the given SoCs. For other SoCs, a new model should be learned to identify a possibly extended set of input sequences for fingerprinting. We note that this fingerprinting sequence could also be found rather fast by random test execution. The advantage of using automata learning for fingerprinting is that the models only have to be created once. Based on these behavioral models, we could create new fingerprinting sequences if we consider further SoCs. For this, is not required to test the prior investigated SoCs. However, we recommend replacing the manual analysis with an automatic conformance testing technique between the models akin to Tappler et al. [31].

5   Related Work

Celosia and Cunche [11] also investigated fingerprinting BLE devices, however, their proposed methodology is based on the Generic Attribute Profile (GATT), whereas our technique also operates on different layers, e.g. the Link Layer (LL), of the BLE protocol stack. Their proposed fingerprinting method is based on a large dataset containing information that can be obtained from the GATT profile, like services and characteristics.

Argyros et al. [8] discuss the combination of active automata learning and differential testing to fingerprint the SULs. They propose a framework where they first learn symbolic finite automata of different implementations and then automatically analyze differences between the learned models. They evaluated their technique on implementations of TCP, web application firewalls, and web browsers. A similar technique was proposed by Tappler et al. [31] investigating the Message Queuing Telemetry Transport (MQTT) protocol. However, their motivation was not to fingerprint MQTT brokers, but rather test for inconsistencies between the learned models. These found inconsistencies show discrepancies to the MQTT specification. Following an akin idea, but motivated by security testing, several communication protocols like TLS [25], TCP [13], SSH [15] or DTLS [14] have been learning-based tested. In the literature, these techniques are denoted as protocol state fuzzing. To the best of our knowledge, none of these techniques interacted with an implementation on an external physical device, but rather interacted via localhost or virtual connections with the SULs.

One protocol state fuzzing technique on physical devices was proposed by Stone et al. [28]. They detected security vulnerabilities in the 802.11 4-Way handshake protocol by testing Wi-Fi routers. Aichernig et al. [3] propose an industrial application for learning-based testing of measurement devices in the automotive industry. Both case studies emphasize our observation that non-deterministic behavior hampers the inference of behavioral models via active automata learning. Other physical devices that have been learned are bank cards [1] and biometric passports [2]. The proposed techniques use a USB-connected smart card reader to interact with the cards. Furthermore, Chalupar et al. [12] used Lego® to create an interface to learn the model of a smart card reader.

6   Conclusion

Summary. In this paper, we presented a case study on learning-based testing of the BLE protocol. The aim of this case study was to evaluate learning-based testing in a practical setup. For this, we proposed a general learning architecture for BLE devices. The proposed architecture enabled the inference of a model that describes the behavior of a BLE protocol implementation. We evaluated our presented learning framework in a case study consisting of five BLE devices. The results of the case study show that the active learning of a behavioral model is possible in a practicable amount of time. However, our evaluation showed that adaptions to state-of-the-art learning algorithms, such as including error-handling procedures, were required for successful model inference. The learned models depicted that implementations of the BLE stack vary significantly from device to device. This observation confirmed our hypothesis that active automata learning enables fingerprinting of black-box systems.

Discussion. We successfully applied active automata learning to reverse engineer the behavioral models of BLE devices. Despite the challenges in creating a reliable and general learning framework to learn a physical device, the BLE interface creation only needs to be done once. Our proposed framework, which is

also publicly available [22], can now be used for learning the behavioral models of many BLE devices. Our presented learning results show that in practice the scalability of active automata learning not only depends on the efficiency of the underlying learning algorithm but also on the overhead due to SUL interaction. All of the learned models show behavioral differences in the BLE protocol stack implementations. Therefore, we can use active automata learning to fingerprint the underlying SoC of a black-box BLE device. The possibility to fingerprint the BLE could be a possible security issue, since it enables an attacker to exploit specific vulnerabilities, e.g. from a BLE vulnerability collection like SWEYNTOOTH [17]. Compared to the BLE fingerprinting technique of Celosia and Cunche [11], our proposed technique is data and time efficient. Instead of collecting 13 000 data records over five months, we can learn the models within hours.

Future work. To the best of our knowledge, the learned models do not show any security vulnerabilities. However, for future work, we plan to consider further levels of the BLE protocol stack, e.g., the encryption-key exchange in the pairing procedure. Considering these levels of the BLE stack might reveal security issues. Related work [25,13,28,14] has shown that automata learning can successfully be used to detected security vulnerabilities. Therefore, learning the security-critical behavior of the BLE protocol might be interesting for further security analysis and testing.

Our proposed method was inspired by the work of Garbelini et al. [17], since their presented fuzz-testing technique demonstrated that model-based testing is applicable to BLE devices. Instead of creating the model manually, we showed that learning a behavioral model of the BLE protocol implemented on a physical device is possible. For future work, it would be interesting to use our learned models to generate test cases for fuzzing. We are currently working on extending our proposed learning framework for learning-based fuzzing of the BLE protocol. For this, we follow a similar technique that we proposed on fuzzing the MQTT protocol via active automata learning [5].

We find that the non-deterministic behavior of the BLE devices hampered the learning of deterministic models. Instead of workarounds to overcome non-deterministic behavior, we could learn a non-deterministic model. We already applied non-deterministic learning on the MQTT protocol [23]. Following a similar idea, we could learn a non-deterministic model of the BLE protocol.

Acknowledgment

References

1. Aarts, F., de Ruiter, J., Poll, E.: Formal models of bank cards for free. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013. pp. 461–468. IEEE Computer Society (2013). https://doi.org/10.1109/ICSTW.2013.60, `https://doi.org/10.1109/ICSTW.2013.60`

2. Aarts, F., Schmaltz, J., Vaandrager, F.W.: Inference and abstraction of the biometric passport. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I. Lecture Notes in Computer Science, vol. 6415, pp. 673–686. Springer (2010). https://doi.org/10.1007/978-3-642-16558-0_54, `https://doi.org/10.1007/978-3-642-16558-0_54`

3. Aichernig, B.K., Burghard, C., Korosec, R.: Learning-based testing of an industrial measurement device. In: Badger, J.M., Rozier, K.Y. (eds.) NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11460, pp. 1–18. Springer (2019). https://doi.org/10.1007/978-3-030-20652-9_1, `https://doi.org/10.1007/978-3-030-20652-9_1`

4. Aichernig, B.K., Mostowski, W., Mousavi, M.R., Tappler, M., Taromirad, M.: Model learning and model-based testing. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers. Lecture Notes in Computer Science, vol. 11026, pp. 74–100. Springer (2018). https://doi.org/10.1007/978-3-319-96562-8_3, `https://doi.org/10.1007/978-3-319-96562-8_3`

5. Aichernig, B.K., Muskardin, E., Pferscher, A.: Learning-based fuzzing of IoT message brokers. In: 14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021. pp. 47–58. IEEE (2021). https://doi.org/10.1109/ICST49551.2021.00017, `https://doi.org/10.1109/ICST49551.2021.00017`

6. Aichernig, B.K., Pferscher, A., Tappler, M.: From passive to active: Learning timed automata efficiently. In: Lee, R., Jha, S., Mavridou, A. (eds.) NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12229, pp. 1–19. Springer (2020). https://doi.org/10.1007/978-3-030-55754-6_1, `https://doi.org/10.1007/978-3-030-55754-6_1`

7. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6

8. Argyros, G., Stais, I., Jana, S., Keromytis, A.D., Kiayias, A.: Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1690–1701. ACM (2016). https://doi.org/10.1145/2976749.2978383, `https://doi.org/10.1145/2976749.2978383`

9. Bluetooth SIG: Market update. `https://www.bluetooth.com/wp-content/uploads/2021/01/2021-Bluetooth_Market_Update.pdf`, Accessed: 2021-05-06

10. Bluetooth SIG: Bluetooth core specification v5.2. Standard (2019), `https://www.bluetooth.com/specifications/specs/core-specification/`

11. Celosia, G., Cunche, M.: Fingerprinting Bluetooth-Low-Energy devices based on the generic attribute profile. In: Liu, P., Zhang, Y. (eds.) Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, IoT S&P@CCS 2019, London, UK, November 15, 2019. pp. 24–31. ACM (2019). https://doi.org/10.1145/3338507.3358617, `https://doi.org/10.1145/3338507.3358617`

12. Chalupar, G., Peherstorfer, S., Poll, E., de Ruiter, J.: Automated reverse engineering using Lego®. In: Bratus, S., Lindner, F.F. (eds.) 8th USENIX Workshop on Offensive Technologies, WOOT '14, San Diego, CA, USA, August 19, 2014. USENIX Association (2014), `https://www.usenix.org/conference/woot14/workshop-program/presentation/chalupar`

13. Fiterau-Brostean, P., Janssen, R., Vaandrager, F.W.: Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 454–471. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_25, `https://doi.org/10.1007/978-3-319-41540-6_25`

14. Fiterau-Brostean, P., Jonsson, B., Merget, R., de Ruiter, J., Sagonas, K., Somorovsky, J.: Analysis of DTLS implementations using protocol state fuzzing. In: Capkun, S., Roesner, F. (eds.) 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020. pp. 2523–2540. USENIX Association (2020), `https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean`

15. Fiterau-Brostean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F.W., Verleg, P.: Model learning and model checking of SSH implementations. In: Erdogmus, H., Havelund, K. (eds.) Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017. pp. 142–151. ACM (2017). https://doi.org/10.1145/3092282.3092289, `https://doi.org/10.1145/3092282.3092289`

16. Garbelini, M.E., Wang, C., Chattopadhyay, S., Sun, S., Kurniawan, E.: Sweyn-Tooth - Unleashing mayhem over Bluetooth Low Energy. `https://github.com/Matheus-Garbelini/sweyntooth_bluetooth_low_energy_attacks`, Accessed: 2021-05-05

17. Garbelini, M.E., Wang, C., Chattopadhyay, S., Sun, S., Kurniawan, E.: Sweyntooth: Unleashing mayhem over Bluetooth Low Energy. In: Gavrilovska, A., Zadok, E. (eds.) 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020. pp. 911–925. USENIX Association (2020), `https://www.usenix.org/conference/atc20/presentation/garbelini`

18. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib - A framework for active automata learning. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 487–495. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_32, `https://doi.org/10.1007/978-3-319-21690-4_32`

19. Margaria, T., Niese, O., Raffelt, H., Steffen, B.: 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. pp. 95–100. IEEE Computer Society (2004). https://doi.org/10.1109/HLDVT.2004.1431246, `https://ieeexplore.ieee.org/xpl/conhome/9785/proceeding`

20. Muškardin, E., Aichernig, B.K., Pill, I., Pferscher, A., Tappler, M.: AALpy: An active automata learning library. In: Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, Australia, October 18-22, 2021, Proceedings. Lecture Notes in Computer Science, Springer (2021)

21. Niese, O.: An integrated approach to testing complex systems. Ph.D. thesis, Technical University of Dortmund, Germany (2003), `https://d-nb.info/969717474/34`

22. Pferscher, A.: Fingerprinting Bluetooth Low Energy via active automata learning. `https://github.com/apferscher/ble-learning`, Accessed: 2021-05-10

23. Pferscher, A., Aichernig, B.K.: Learning abstracted non-deterministic finite state machines. In: Casola, V., Benedictis, A.D., Rak, M. (eds.) Testing Software and Systems - 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9-11, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12543, pp. 52–69. Springer (2020). https://doi.org/10.1007/978-3-030-64881-7_4, `https://doi.org/10.1007/978-3-030-64881-7_4`

24. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. **103**(2), 299–347 (1993). https://doi.org/10.1006/inco.1993.1021, `https://doi.org/10.1006/inco.1993.1021`

25. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. pp. 193–206. USENIX Association (2015), `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter`

26. S, R.R., R, R., Moharir, M., G, S.: Scapy - a powerful interactive packet manipulation program. In: 2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS). pp. 1–5 (2018). https://doi.org/10.1109/ICNEWS.2018.8903954

27. Shahbaz, M., Groz, R.: Inferring Mealy machines. In: Cavalcanti, A., Dams, D. (eds.) FM 2009, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 207–222. Springer (2009). https://doi.org/10.1007/978-3-642-05089-3_14, `https://doi.org/10.1007/978-3-642-05089-3`

28. Stone, C.M., Chothia, T., de Ruiter, J.: Extending automated protocol state learning for the 802.11 4-way handshake. In: López, J., Zhou, J., Soriano, M. (eds.) Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11098, pp. 325–345. Springer (2018). https://doi.org/10.1007/978-3-319-99073-6_16, `https://doi.org/10.1007/978-3-319-99073-6_16`

29. Tappler, M.: Learning-Based Testing in Networked Environments in the Presence of Timed and Stochastic Behaviour. Ph.D. thesis, TU Graz (2019), `https://mtappler.files.wordpress.com/2019/12/thesis.pdf`

30. Tappler, M., Aichernig, B.K., Bacci, G., Eichlseder, M., Larsen, K.G.: L^*-based learning of Markov decision processes. In: ter Beek, M.H., McIver, A., Oliveira,

J.N. (eds.) Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11800, pp. 651–669. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_38, `https://doi.org/10.1007/978-3-030-30942-8_38`

31. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing IoT communication via active automata learning. In: ICST 2017, Tokyo, Japan, March 13-17, 2017. pp. 276–287. IEEE (2017). https://doi.org/10.1109/ICST.2017.32, `https://doi.org/10.1109/ICST.2017.32`

32. Tappler, M., Aichernig, B.K., Larsen, K.G., Lorber, F.: Time to learn - learning timed automata from tests. In: André, É., Stoelinga, M. (eds.) Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11750, pp. 216–235. Springer (2019). https://doi.org/10.1007/978-3-030-29662-9_13, `https://doi.org/10.1007/978-3-030-29662-9_13`

33. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4949, pp. 1–38. Springer (2008). https://doi.org/10.1007/978-3-540-78917-8_1, `https://doi.org/10.1007/978-3-540-78917-8_1`