Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning

Andrea Pferscher(✉) and Bernhard K. Aichernig

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

Abstract. Fuzzing (aka fuzz testing) shows promising results in security testing. The advantage of fuzzing is the relatively simple applicability compared to comprehensive manual security analysis. However, the effectiveness of black-box fuzzing is hard to judge since the internal structure of the system under test is unknown. Hence, in-depth behavior might not be covered by fuzzing. This paper aims at overcoming the limitations of black-box fuzzing. We present a stateful black-box fuzzing technique that uses a behavioral model of the system under test. Instead of manually creating the model, we apply active automata learning to automatically infer the model. Our framework generates a test suite for fuzzing that includes valid and invalid inputs. The goal is to explore unexpected behavior. For this, we test for conformance between the learned model and the system under test. Additionally, we analyze behavioral differences using the learned state information. In a case study, we evaluate implementations of the Bluetooth Low Energy (BLE) protocol on physical devices. The results reveal security and dependability issues in the tested devices leading to crashes of four out of six devices.

Keywords: Automata learning · Fuzz testing · Model-based fuzzing · Bluetooth Low Energy

1 Introduction

The Internet of Things (IoT) connects billions of devices, and the number of connected devices will increase with the pervasion of new communication protocols. One popular protocol for short-range communication is Bluetooth. The introduction of Bluetooth Low Energy (BLE) made Bluetooth also available for low-energy devices in the IoT. Nowadays, manufacturers advertise BLE as a key communication technology that could make wired communication obsolete in some applications. For example, Texas Instruments [21] motivates the use of BLE chips for the automotive industry. They suggest that BLE can replace wires that connect sensors in a vehicle. Further automotive applications include, e.g., the use of the smartphone as a "virtual" key. These proposals stress the need for thorough testing techniques to ensure the safety and security of the user.

J. V. Deshmukh et al. (Eds.): NFM 2022, LNCS 13260, pp. 373–392, 2022.
https://doi.org/10.1007/978-3-031-06773-0_20

Fuzzing (aka fuzz testing) is a security and robustness testing technique. Fuzzing aims to reveal unexpected behavior, e.g. crashes. For this, fuzzing executes a large number of randomly generated test cases that include invalid or unusual inputs. One problem in fuzzing is the definition of a termination criterion for testing. In a white- or gray-box setting, coverage measurements, e.g. code coverage, create the possibility to define termination criteria. However, assuming a black-box environment hampers coverage measurements. One solution to obtain behavioral coverage for black-box fuzzing is the extension of fuzzing by model-based testing. Garbelini et al. [17] used model-based fuzzing to reveal security issues in BLE devices. They manually created a generic model based on the BLE specification. However, the manual creation of a model can be an error-prone process and additionally requires the ongoing effort of keeping the model up-to-date.

Instead of manual modeling, automata learning automatically creates behavioral models of black-box components from observed system behavior. In practice, automata learning has successfully been applied to show flaws in communication protocols like (D)TLS [14,30], TCP [13], SSH [15], or MQTT [37]. These learning applications deduce behavioral inconsistencies by comparing learned models against the specification.

In this paper, we present a stateful black-box fuzzer that tests BLE devices. For this, we combine automata learning and fuzzing. In preliminary work, Aichernig et al. [2] proposed learning-based fuzzing for MQTT servers. In contrast to their work, we do not learn one generic model for all devices, but rather base our proposed fuzzing technique on individual learned models for every BLE device. This is motivated by the observation in our previous work [26], where BLE devices behaved differently. The learned models show that functionalities might not be available or only after a specific message sequence. Hence, learning a generic model from one device is not feasible. Unlike Aichernig et al. [2], we do not only test one specific input. Instead we fuzz several packets from different layers of the BLE stack. Additionally, we extended the learning-based fuzzing framework with a counterexample analysis technique that automatically investigates unexpected behavior. Our results show that all BLE devices contain behavioral inconsistencies, security, or robustness issues. As a result, we were able to crash four out of the six investigated devices.

The contribution of this paper is threefold. First, we propose a stateful black-box fuzzing framework for fuzzing BLE devices. Second, we present the conducted case study that is based on six BLE devices. Last, we provide the code of the learning-based fuzzing framework **online**[1] [25].

The paper is structured as follows. Section 2 discusses the needed background. In Sect. 3, we introduce the developed learning-based fuzzing framework. Section 4 presents the case study on learning-based fuzzing of the BLE devices. We discuss related work in Sect. 5, and conclude our work with a discussion and an outlook on future work in Sect. 6.

[1] https://git.ist.tugraz.at/apferscher/ble-fuzzing.

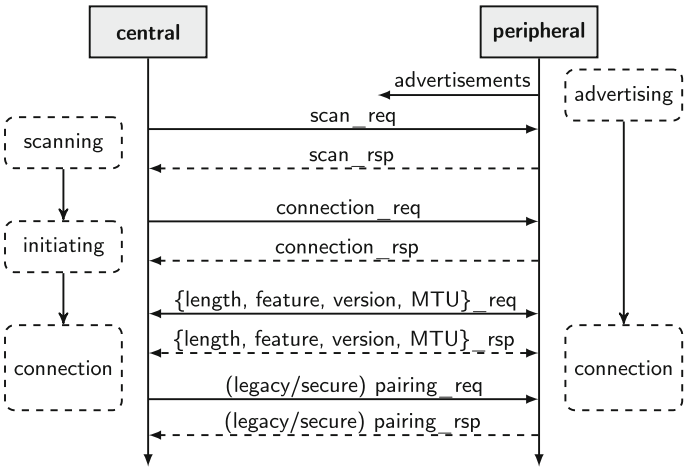

Fig. 1. Bluetooth sequence diagram including the connection procedure and the start of the pairing procedure. The figure is taken from our previous work [26].

2 Preliminaries

2.1 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is part of the Bluetooth specification [6] since version 4.2 and enables communication via Bluetooth also for low energy devices. Compared to the Bluetooth classic, BLE builds upon a different protocol stack.

Figure 1 shows the sequence chart of the connection procedure between two BLE devices. We distinguish between two roles of the devices: the central and the peripheral device. In the remaining of the paper, we refer to the central and peripheral devices as *central* and *peripheral*. The central initiates the connection, whereas the peripheral is available for the establishment of a connection. For example, a central would be a smartphone that connects to a peripheral like a smartwatch. The peripheral is initially in an *advertising* state sending advertisements. The central starts in the *scanning* state by scanning for advertisements via performing a scan_req. The peripheral's response either contains an advertisement or a scan response, both are referred to in Fig. 1 as scan_rsp. Then the central enters the *initiating* state by sending a connection_req that is answered by a connection_rsp. Next, a negotiation phase starts, where parameters like maximum transmission unit (MTU) or Bluetooth version are agreed upon. Note that also the peripheral may request parameters from the central, which should be answered. The BLE specification does not specify which parameters must be negotiated. Hence, the establishment of a connection might be different for every BLE device. Afterward, the devices are connected and the pairing procedure can start. We distinguish between *legacy* and *secure* pairing, which differ in the encryption key-exchange procedure. A connection can be terminated by sending an additional scan_req or by a termination indication (termination_ind).

2.2 Mealy Machine

Mealy machines are finite state machines including state transitions that are labeled with input/output action pairs. Therefore, Mealy machines represent a modeling formalism for reactive systems. We define a Mealy machine as 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, and $\lambda : Q \times I \to O$ is the output function.

We assume that δ and λ are total functions, i.e., $\mathcal{M}$ is input enabled and deterministic. Let $s \in (I \times O)^*$ be a sequence of alternating inputs and outputs, where $s_I \in I^*$ is the corresponding input sequence and $s_O \in O^*$ the output sequence. We denote the empty sequence as ϵ and upgrade a single element to a sequence of size one. We write $s \cdot s'$ for the concatenation of two sequences s and s'. We extend δ and λ for sequences. Let $\delta^* : Q \times I^* \to Q$ be a function that returns the target state after executing an input sequence from a source state. Similarly, we define $\lambda^* : Q \times I^* \to O^*$ which returns the corresponding output sequence that is observable on executing an input sequence starting at a given state. Let S_I be the set of all possible input sequences in $\mathcal{M}$. We denote that 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 behavioral equivalent if $\forall\, s_I \in S_I : \lambda^*(q_0, s_I) = \lambda'^*(q'_0, s_I)$. Consequently, a counterexample to conformance between $\mathcal{M}$ and $\mathcal{M}'$ is an input sequence $s_I \in S_I$ where $\lambda^*(q_0, s_I) \neq \lambda'^*(q'_0, s_I)$.

2.3 Automata Learning

Automata learning creates behavioral models of black-box systems using system observations. In automata learning, we distinguish between two main directions: passive and active learning. Passive learning infers a behavioral model from a given data set, e.g., log files. Therefore, the quality of the learned behavioral model depends on the provided data. Active learning creates a behavioral model by actively querying the System Under Learning (SUL). For this, we require an interface to the SUL that enables query execution and observation of outputs.

Many state-of-the-art learning algorithms build upon the L^* algorithm proposed by Dana Angluin [3]. Her seminal work introduces the minimally adequate teacher framework that comprises two members: the learner and the teacher. The learner's objective is to create a minimal Deterministic Finite Automaton (DFA) that represents a hidden regular language. The teacher knows this regular language. To learn the DFA, the learner asks the teacher two different types of queries. First, the learner asks if a word is part of the language. We denote such queries as membership queries. Based on the answers to the membership queries, the learner creates an initial hypothesis. This hypothesis is then provided to the teacher. Since the learner asks for equivalence between the hypothesis and the SUL, the second query type is named equivalence queries. The teacher answers equivalence queries either by confirming that the provided hypothesis conforms to the regular language or by returning a counterexample that shows the non-conformance between the hypothesis and the SUL. In

the case of non-conformance, the learner creates new membership queries based on the counterexample and then proposes a new hypothesis. This procedure repeats until a conforming hypothesis is provided. The L^* algorithm has been extended for various behavioral system types. Shahbaz and Groz [34] propose Angluin-style learning for reactive systems, where the behavior is formalized by Mealy machines. For this, the learner asks output queries instead of membership queries. Output queries include an input sequence, and the teacher responds with the corresponding output sequence. Since the efficiency of L^*-based algorithms depends on the considered alphabet, active learning does not scale well for systems with large input and output space. To overcome this issue, Aarts et al. [1] introduce a mapper component that enables learning with an abstracted alphabet.

The assumption of a teacher with a perfect equivalence oracle is not practical. Therefore, conformance testing implements the equivalence oracle. For this, a conformance relation must be defined. Assuming that an implementation represents a hidden Mealy machine $\mathcal{I}$, Tretmans [38] defines the implementation relation $\mathcal{I}$ **imp** $\mathcal{S}$ between an implementation $\mathcal{I}$ and a Mealy machine specification $\mathcal{S}$. We denote conformance between $\mathcal{I}$ and $\mathcal{S}$ based on the behavioral equivalence of Mealy machines. In automata learning, conformance testing aims to test that a proposed hypothesis $\mathcal{H}$ conforms to a black-box implementation $\mathcal{I}$, i.e., testing that $\mathcal{H}$ **imp** $\mathcal{I}$ is satisfied. Since the behavioral conformance between two Mealy machines is based on equivalence, $\mathcal{I}$ **imp** $\mathcal{H} \Leftrightarrow \mathcal{H}$ **imp** $\mathcal{I}$ holds. Assuming that a finite set of input sequences $S'_I \subseteq S_I$ adequately represents the behavior of $\mathcal{I}$, we can define the following conformance relation for learning:

$$\mathcal{H} \textbf{ imp } \mathcal{I} \Leftrightarrow \forall s'_I \in S'_I : \lambda^*_{\mathcal{H}}(q_0^{\mathcal{H}}, s'_I) = \lambda^*_{\mathcal{I}}(q_0^{\mathcal{I}}, s'_I). \tag{1}$$

2.4 Fuzzing

Fuzzing aims at finding unexpected behavior of the System Under Test (SUT) by executing a large number of tests. To trigger unexpected behavior, executed tests not only contain valid inputs, but also invalid or unusual inputs. The first fuzzing framework was introduced by Miller et al. [22] to test UNIX utilities.

We categorize fuzzing based on three access levels to the SUT: white, gray, and black box. In white-box fuzzing access to the code is given. White-box techniques, e.g. SAGE [19], apply symbolic execution to generate inputs that also execute in-depth behavior. Gray-box fuzzer, e.g. american fuzzing loop (AFL) [40], are based on instrumented binary code, which enables reasoning about covered behavior. In black-box fuzzing, no access to the system's code is assumed. Böhme et al. [7] distinguish between mutational and generational black-box techniques. Mutational fuzzers generate random inputs by modifying an initial input, e.g. via bit-flipping. Generational fuzzers require a priori knowledge about the input structure of the SUT, e.g., the packet structure of the tested protocol.

The main problem in black-box fuzzing is the assurance of sufficient test coverage. To overcome this problem, Aichernig et al. [2] presented a generational black-box fuzzer that is based on active automata learning. Figure 2 illustrates

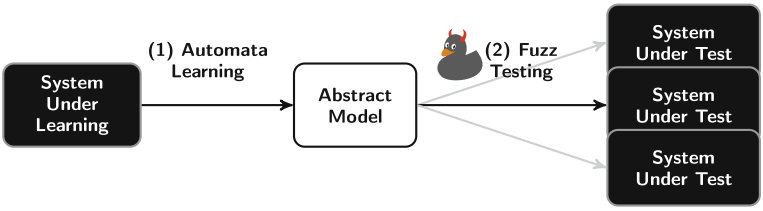

Fig. 2. Concept of learning-based fuzzing technique proposed by Aichernig et al. [2].

the proposed two-step procedure of learning-based fuzzing. First, Aichernig et al. learn a behavioral model of the SUL. Since learning would not be feasible using all possible inputs, an abstracted alphabet is considered. The learned abstract model is then used for model-based fuzzing of other SUTs which represent different implementations of the SUL. The learned model is on a more abstract level than the SUTs. Therefore, Aichernig et al. extend the model-based testing technique with a so-called fuzzing mapper. The fuzzing mapper generates fuzzing inputs by concretizing the abstract inputs to invalid and valid inputs. The stateful fuzzer then identifies behavioral differences between the model and the SUTs.

3 Methodology

In the following, we present a fuzzing framework that combines automata learning and fuzzing to create a stateful black-box fuzzing technique. Figure 3 illustrates our proposed learning-based fuzzing framework. The framework consists of three components: the system interface, the active automata learning component, and the stateful fuzzer. The presented technique is a two-step procedure. First, we use active automata learning to generate a behavioral model of the SUT. Second, we fuzz the SUT based on the learned model. In the following sections, we describe the details of each component.

3.1 System Interface

The system interface comprises the SUT and an adapter that enables communication to the SUT. We assume that the SUT is a reactive black-box system, where we can execute inputs and observe outputs. Furthermore, we require the SUT to be resettable via inputs sent by the adapter.

Our testing targets are BLE devices. To communicate with a black-box BLE device, we require another BLE device as part of our adapter component. The device used in the adapter is controlled by us and enables the transmission of manually crafted BLE packets to the SUT. In the context of Fig. 1, the adapter device represents the central, whereas the SUT acts as peripheral. Hence, the SUT initially distributes BLE advertisements. The used learning algorithm and conformance testing technique require that the SUT can be reset to the initial

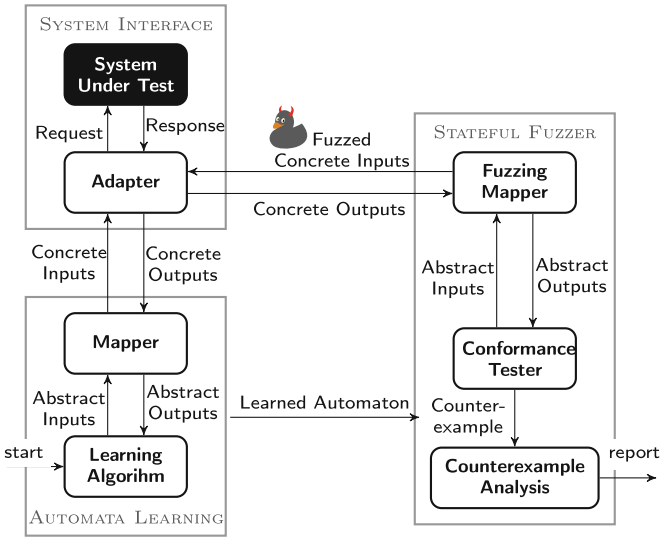

Fig. 3. The proposed framework for learning-based fuzzing of BLE devices comprises three components: the system interface, the learning component, and the fuzzer.

advertising state. We assume that the BLE connection can be terminated by performing termination_ind or scan_req. After the connection is terminated, we expect that the peripheral again enters the advertising state. A reset via BLE messages is assumed to be equal to a hard reset, e.g., via pressing the reset button on the device.

Garbelini et al. [17] provide a firmware for the central that enables the transmission of manually crafted BLE packets. Using Python and the library Scapy [31], we can draft BLE packets and send them via the central to the peripheral, i.e. the SUT. Vice versa, we parse received packets from the peripheral by the central again using Scapy.

3.2 Automata Learning

The automata learning component implements a framework that deduces a behavioral model from the observations returned by the system interface. Since we require for fuzzing that the behavior in every state for all inputs is defined, we apply an L^*-based automata learning algorithm. We applied an improved variant of the L^* algorithm proposed by Shahbaz and Groz [34] that learns a Mealy machine of a reactive system.

The L^*-based learning algorithm requires the SUL to be deterministic and resettable. Due to the wireless setup, we have to take care of non-deterministic behavior and a sufficient reset procedure. For this, we use an updated version of the learning framework for BLE devices presented in our previous work [26]. The updates include a different reset implementation and a more fault-tolerant han-

dling of non-deterministic behavior. The SUT must be reset after each performed query by the learning algorithm. Since a hard reset would make learning a tedious process, we perform a termination_ind via the peripheral. To ensure that the SUT is properly reset, we search for advertisements via a scan_req before executing a query. Hence, we can control the reset completely remotely via our adapter component. Furthermore, we have to consider non-determinism. Even though, we assume that the SUL behaves deterministically, non-determinism is still possible due to the wireless learning interface. Examples for non-deterministic behavior could be the lost or delayed messages either by the adapter or the SUL. In case of non-determinism, we repeat the query and apply a majority selection to select the most commonly observed value.

Similar to our previous technique [26], we consider an abstracted input alphabet which makes learning feasible within an adequate time. Figure 3 shows the mapper component that translates abstract inputs of the learning algorithm to concrete inputs that are executable on the SUL. Based on Fig. 1, we consider the following abstract input alphabet $I_A = \{$scan_req, connect_req, length_req, length_rsp, feature_req, feature_rsp, version_req, mtu_req, pairing_req$\}$. The concretization of these abstract inputs is based on values, which we assume to be valid BLE packets that lead to similar responses each time they are sent. The concrete values are mainly based on preset values provided by Scapy. For example, the mapper translates the abstract input version_req to BTLE($access\_addr$)/BTLE_DATA()/BTLE_CTRL()/LL_VERSION_IND($version$), where the fields $access\_addr$ and $version$ are defined by the mapper. The mapper then forwards the concrete packet to the system interface and waits for a response. The received concrete packets from the system interface are then translated by the mapper to an abstract output. The output abstraction removes field values from the packets and considers only the packet type name provided by Scapy.

3.3 Stateful Fuzzer

The stateful fuzzer is the third component in our learning-based fuzzing framework and executes the last step in our fuzzing technique. Our proposed stateful fuzzer aims to find and analyze counterexamples to the conformance between the provided hypothesis and the SUT. The stateful fuzzer takes the learned automaton as input and has access to the interface of the SUT.

In contrast to other model-based fuzzers [2,17], our technique does not require a generic behavioral model of the SUT. According to Garbelini et al. [17], the manual creation of a generic BLE model is tedious due to the underspecified connection procedure in the BLE specification [6]. Furthermore, we cannot follow the learning-based fuzzing technique proposed by Aichernig et al. [2], since there does not exist any SUL that implements a generic model of the BLE protocol. The results of previous work [26] show that the learned model differs for every BLE device. Differences arise due to limitations of functionality, e.g. no support for BLE pairing, or functionality that is only available after a certain input sequence.

Our fuzzing technique is based on model-based testing and the conformance relation used in the fuzzing component follows the one during active automata

learning. However, we need to adapt this conformance relation, since our fuzzing technique aims at testing if the provided SUT $\mathcal{I}$ implements the behavior defined by the provided automaton $\mathcal{H}$. Therefore, we test if $\mathcal{I}$ **imp** $\mathcal{H}$ holds. The provided automaton $\mathcal{H}$ specifies the behavior on an abstract level. Due to the abstraction, we define conformance based on the abstract input and output alphabet. Let T_A, where $T_A \subseteq I_A^*$, be a finite set of abstract input sequences. For this, we denote conformance between the provided automaton $\mathcal{H}$ and the SUT $\mathcal{I}$ for fuzzing as follows:

$$\mathcal{I} \textbf{ imp } \mathcal{H} \Leftrightarrow \forall t_A \in T_A : \lambda_{\mathcal{I}}^*(q_0^{\mathcal{I}}, t_A) = \lambda_{\mathcal{H}}^*(q_0^{\mathcal{H}}, t_A). \tag{2}$$

Note that λ^* returns a sequence of abstract outputs for the provided abstract input sequence. Let I^* be the set of possible concrete input sequences. The stateful fuzzer aims at finding a concrete input sequence $t \in I^*$ that shows for the corresponding abstract input sequence $t_A \in I_A^*$ that $\lambda_{\mathcal{I}}^*(q_0^{\mathcal{I}}, t_A) \neq \lambda_{\mathcal{H}}^*(q_0^{\mathcal{H}}, t_A)$ is fulfilled. In the remainder of this work, we denote $t \in I^*$ as test sequence and the corresponding $t_A \in I_A^*$ as abstract test sequence.

We generate abstract test sequences that consist of three parts $p \cdot f \cdot s$, where $p \in I_A^*$ is the prefix of the sequence, $f \in I_A$ is a fuzzing input, and $s \in I_A^*$ is the suffix of the test sequence. The prefix p represents an *access sequence* to a state in the behavioral model. The *access sequence* is an abstract test sequence that defines the shortest sequence to reach a state in the behavioral model starting from the initial state, where the access sequence for the initial state is the empty sequence ϵ. We can guarantee state coverage for fuzzing by generating for every access sequence corresponding test sequences. The fuzzing input f is a randomly selected abstract input that is later concretized by fuzzing techniques. The suffix s is a sequence of randomly selected inputs.

The fuzzing mapper translates the abstract test sequence to a concrete test sequence. The generation of concrete inputs differs for the three parts of the test sequence. The prefix p and the suffix s correspond to valid BLE packets similar to the translation during learning. The fuzzing input f is differently generated. The fuzzing mapper selects concrete fuzzed values for fields in the BLE packet based on given value ranges. For every packet, we fuzz exactly one field and if the packet has several fields randomly one field is chosen. The selection of the concrete value is based on randomness. For some fields, a set of possible values is given, whereas others are limited by minimum and maximum values. Additionally, for fields that are limited by an upper and lower bound, the selection of boundary values is preferred. For example, if the abstract input is a connect_req, then the concrete BLE packet in Scapy syntax is BTLE() / BTLE_ADV(...) / BTLE_CONNECT_REQ(*interval*, *timeout*, ...). The fuzzing mapper concretizes the fields and chooses exactly one field to be fuzzed. For example, the mapper selects to fuzz the field *timeout*. Next, the fuzzer randomly picks a value between 0 and $2^{16} - 1$, since the BLE specification considers two bytes for the *timeout* field. The fuzzing mapper similarly translates all other fields as in the learning phase. Note that the fuzzed fields might be invalid according to the BLE specification. Considering the given example,

the BLE specification defines the supervision timeout to be within 100 ms and 32 s, which corresponds to *timeout* values between 10 and 3 200.

We check after each executed input on the SUT $\mathcal{I}$ if the received output deviates from the defined output in the hypothesis $\mathcal{H}$. If this is the case, we stop the execution of the test sequence and truncate the test sequence after the first non-corresponding output. The counterexample to the conformance between $\mathcal{I}$ and $\mathcal{H}$ is then provided to the counterexample analysis component. Before we start the analysis, we try to reproduce the found counterexample. To avoid the reporting of counterexamples due to connection errors and non-deterministic behavior, we require to observe the found counterexample again within n_{cex} attempts.

If we found a reproducible counterexample, we perform the counterexample analysis. The counterexample analysis examines unexpected state transitions revealed by the fuzzing input. Based on the W-Method [9], we use the *characterization set* to calculate possible different state transitions between $\mathcal{I}$ and $\mathcal{H}$. The characterization set contains input sequences that generate a unique set of output sequences for every state. By the execution of input sequences of the characterization set, we aim to identify if an unexpected output leads to a different state. Since the characterization set might change for the extended fuzzing input alphabet, we extend the characterization set always by the input alphabet. The advantage of performing an L^*-based learning algorithm in advance is that the characterization set can be automatically derived from the data structures used during learning. Note that this counterexample analysis only hints at a possible target state. For example, a BLE connection might terminate on an invalid request. In this case, we would observe a transition to the initial state. To check the actual state equivalence, a more comprehensive conformance test would be required. The counterexample analysis is also limited by n_{cex} repetitions in the case of connection errors or non-deterministic behavior.

To make conformance testing feasible, we limit the size of T_A by $n_{\text{fuzz}} \in \mathbb{N}$ and the size of the suffix s for each trace by $n_{\text{suffix}} \in \mathbb{N}$. All executed test sequences, including the counterexample analysis, are stated in a final report that is generated after the conformance test. In the case that the SUT crashes, the report includes all executed traces up to the crash.

4 Evaluation

We evaluated our learning-based fuzzing technique on six different BLE devices. In the following, we present the practical setup for learning and fuzzing the BLE devices. Furthermore, we discuss the issues found by learning-based fuzzing. Our learning-based fuzzing framework, implemented in Python 3.9, and the learned automata are available **online**[2] [25]. We ran all experiments on an Apple MacBook Pro 2019 with an Intel Quad-Core i5 (2.4 GHz) and 8 GB RAM.

[2] https://git.ist.tugraz.at/apferscher/ble-fuzzing.

Table 1. Investigated BLE devices

Manufacturer (Board)	SoC	Application
Texas Instruments (LAUNCHXL-CC2640R2)	CC2640R2F	Project Zero
Texas Instruments (LAUNCHXL-CC2650)	CC2650	Project Zero
Texas Instruments (LAUNCHXL-CC26X2R1)	CC2652R1	Project Zero
Cypress (CY8CPROTO-063-BLE)	CYBLE-416045-02	Find Me Target
Cypress (Raspberry Pi 4 Model B)	CYW43455	bluetoothctl
Nordic (decaWave DWM1001-DEV)	nRF52832	Nordic GATTS

4.1 General Setup

Table 1 lists the six evaluated BLE devices. Our evaluation includes five devices that were already considered in previous work [26] and an additional device from Texas Instruments (CC2652R1). All of the selected devices implement the BLE 5 standard. The selection involves devices from different manufacturers that were also part of the case study by Garbelini et al. [17]. We extended our selection to popular boards, e.g., the Raspberry Pi 4. Furthermore, we aimed to identify behavioral differences between boards of the same manufacturer. All devices run an example application that sends BLE advertisements and allows a connection with the central. We refer to the BLE devices by their System on Chip (SoC) name. As central for learning and fuzzing, we used the Nordic nRF52840 Dongle and the Nordic nRF52840 Development Kit. We flashed both devices with custom firmware provided by Garbelini et al. [17].

To learn behavioral models of the BLE devices, we followed the learning setup presented in our previous work [26]. We used an adapted version of the learning library AALpy [23] (v1.1.5) which implements Rivest and Shapire's [28] improved L^* for Mealy machines. The learning library was extended by a method to calculate the characterization set, which is now included in v1.1.7. For the creation of BLE packets, we used an adapted version of the library Scapy [31] (v2.4.4), where the used updates are available in v2.4.5.

Similar to our previous work [26], we adapted the considered input alphabet for the CC2640R2F to learn deterministic behavioral models, since the SoC behaves non-deterministically on some input sequences. We learned three different deterministic models of the CC2640R2 using a decreased input alphabet. The first variation considers the abstracted input alphabet I_A, introduced in Sect. 3.2, without pairing_req, the second without feature_req, and the third without length_req. For fuzzing, we separately tested each behavioral model against the SoC CC2640R2 with the corresponding reduced input alphabet.

For CC2651R1 and CYW43455, we required a different learning setup since the consecutive performing of connection_req led disproportionately often to connection errors. For these SoCs, we established a connection before executing a test sequence. Considering Fig. 1, we started learning in the initiating phase of the central after the connection_req. After executing the test sequence, a termination indication was performed to cancel the connection. Furthermore,

Table 2. Overview on fuzzing results. The *-symbol denotes that learning and fuzzing starts after the connection_req. Two SoCs crash before executing 1 000 queries.

SoC	States	Fuzzing rounds	Crashes	Queries	CEX
CC2640R2F (no pairing_req)	6	4	3	1 280	27
CC2640R2F (no feature_req)	11	5	5	928	50
CC2640R2F (no length_req)	11	5	5	767	39
CC2650	5	4	3	1 375	28
CC2652R1*	4	5	5 (6)	919	39
CYBLE-416045-02	3	2	1	1 413	38
CYW43455*	16	1	0	2 652	197
nRF52832	5	1	0	2 258	113

we decrease the learning alphabet to $I'_A = \{\mathsf{length\_req}, \mathsf{length\_rsp}, \mathsf{feature\_req}, \mathsf{feature\_rsp}, \mathsf{version\_req}, \mathsf{mtu\_req}, \mathsf{pairing\_req}\}$. Hence, we solely learned for these devices the behavior during the parameter negotiation phase until the initiation of the pairing procedure.

For the conformance check during fuzzing, we define the minimum number of generated test sequences to $n_{\text{fuzz}} = 1\,000$. Since we want to create a stateful fuzzer, we defined the actual number of performed tests depending on the number of states. Let $n \in \mathbb{N}$ be the number of states of the provided learned model, then the number of generated test sequences per state is defined as follows $\lceil \frac{n_{\text{fuzz}}}{n} \rceil$. In previous work [26], we observed that the SUT might behave non-deterministically due to lost or delayed packets. Additionally, we check if a valid connection can be established before executing a test sequence. If not, we note down a connection error. We also require an error-handling behavior for the conformance testing. For each performed query, we set the maximum number of non-deterministic errors $n_{\text{nondet}} = 20$ and connection errors $n_{\text{errors}} = 20$. In case the BLE device crashed due to the execution of a fuzzed input, the conformance check stops after observing n_{errors} connection errors. For the counterexample analysis, the maximum attempts to reproduce the counterexample is $n_{\text{cex}} = 5$.

4.2 Fuzzing Results

Table 2 shows the learning-based fuzzing results for the investigated BLE SoCs. For every SoC, we list the number of states of the learned Mealy machine. The learned behavioral models of the SoCs that we already considered in the previous case study [26] did not change except for CYW43455. For CYW43455, we updated the BlueZ version and used a different example application. Due to the update, the behavior on the connection_req changes, since consecutive connection_req lead more frequently to connection errors.

The *Fuzzing Rounds* indicate the number of performed conformance testing attempts performed by our stateful fuzzer. The stateful fuzzer aims to execute $\lceil \frac{1\,000}{n} \rceil \cdot n$ test sequences, where n is the number of states. However, four out of the six investigated SoCs crashed during the execution of our fuzzing technique.

In the case of a crash, we identify the cause of the crash. For example, whether a BLE packet with a fuzzed field causes the crash. In Sect. 4.3, we provide examples for fuzzed fields that led to a crash. If there exists such a field, we exclude the fuzzing of it in the next fuzzing execution. If the cause for crashing is not obvious, we repeat the stateful fuzzing without any changes a second time.

Looking at the number of crashes reported in Table 2, we not only see that four out of six SoCs crash, but, more seriously, two SoCs (CC2640R2F and CC2652R1) crash on every execution. Hence, we could not execute at least 1 000 fuzzed test sequences without crashing the devices. For the CC2652R1, we recognized an additional crash during the learning setup.

The column *Queries* reports the number of performed test sequences on the SoC during fuzzing. This number also includes the executions for the repetition of counterexamples and the following state analysis. The column *CEX* shows the number of found counterexamples to the conformance between the learned model and the SoC. Note that the number of counterexamples does not conclude that the SoC behaves erroneously. Instead, a high number of counterexamples more likely indicates that we observe more countermeasures against invalid inputs. In case of crashes, we take the number of performed tests and counterexamples from the fuzzing execution that executed the most test sequences.

The execution of conformance testing including the counterexample analysis took on average 5.6 h for non-crashing runs. However, this average runtime does not include the runtime of the nRF52832, since it has an extraordinary high runtime of 42.2 h. This observation conforms to the learning results we obtained in previous work [26] where the interaction with nRF52832 was more time-consuming than with other devices. We detect crashes within 12.6 min and 22.2 h. We assume that there is a high potential for optimization of the time to detect crashes due to the immediately performed counterexample analysis and the high number of accepted connection errors n_{errors}.

4.3 Bug Hunt

Table 3 presents the found vulnerabilities, and anomalies to the BLE specification [6] or compared to other devices. We found four different crash scenarios denoted by an identifier (ID) starting with a "C". Furthermore, we present two anomalies, A1 and A2, to the BLE specification and another two, A3 and A4, that shows a unique behavior compared to all other devices. The last finding of our paper reveals a security vulnerability (identified by V1) which allows a reduction of the key's entropy during the pairing procedure.

Connection crashes (C1-C4). All three investigated SoCs from Texas Instruments crashed due to performing connection requests (C1-C4). The crash C1 requires no input modification. Instead, a sequence of valid inputs crashes the CC2651R1. During learning, we observed that the CC2651R1 crashes on a sequence of non-fuzzed connection_req. For example, the execution of the following sequence of valid inputs leads to a crash on the CC2651R1:

$$\mathsf{scan\_req} \cdot \mathsf{connection\_req} \cdot \mathsf{connection\_req}.$$

Table 3. List of found crashes and anomalies. The identifiers (IDs) of crashes start with a "C", behavioral anomalies with an "A", and other vulnerabilities with a "V".

ID	Issue	SoCs
C1	Crash on consecutive connection_req	CC2652R1
C2	Crash on connection_req(*interval*)	CC2640R2F, CC2650, CYBLE-416045-02
C3	Crash on connection_req(*timeout*)	CC2640R2F, CC2650
C4	Crash on connection_req(*latency*)	CC2640R2F, CC2650
A1	Multiple responses to version_req	CC2652R1
A2	Accepting pairing_req($max\_kex\_size :> 16$)	CYW43455
A3	Connection termination on length_rsp	nRF52832
A4	Unknown behavior on length_{req, rsp}($max\_\{tx, rx\}\_bytes$)	CC2652R1
V1	Key size reduction on pairing_req($max\_kex\_size : [7, 16]$)	All devices (except CYBLE-416045-02)

With the support of Texas Instruments, we identified the origin of this issue in the installed application software. The running application stops sending advertisements after two consecutive connections. Additionally, the connection cannot be reset, since no further scan_req are accepted. Hence, the device is inaccessible.

The crashes C2-C4 were caused by fuzzed fields of the connection_req. The fields that crashed the devices CC2640R2F and CC2650 were *latency*, *timeout*, and *interval*. Invalid values of the field interval (C1) also crashed one BLE device of Cypress (CYBLE-416045-02). We assume that issues relate to CVE-2019-19193 which has been reported by Garbelini et al. [17]. According to Garbelini et al., this issue has been fixed by the manufacturers.

Multiple Answers to Version Requests (A1). Figure 4 illustrates a simplified learned model of the CC2651R1. This model shows that every version_req is always answered by a version indication. The BLE specification [6] defines that an already answered version_ind should not be answered again.

Anomalies in Length Requests and Responses. A comparison of the learned models shows that the nRF52832 is the only SoC that terminates the BLE connection and returns to the initial state if an unexpected length_rsp is performed. We observed that behavior even though the length_rsp did not contain any fuzzed fields. Furthermore, our counterexample analysis revealed an anomalous behavior for the CC2651R1 (A4). We trigger the anomaly by performing a length_rsp or length_req, where we fuzzed the fields max_tx_bytes or max_rx_bytes. After this, we execute a non-fuzzed mtu_req or pairing_req. Executing this sequence, the CC2651R1 enters a state, where only empty BLE data packets are received for all inputs except those that reset the connection. A4 also violates the BLE specification, since none of the further requests is appropriately answered. Furthermore, this unknown state cannot be exited by performing another valid length_req.

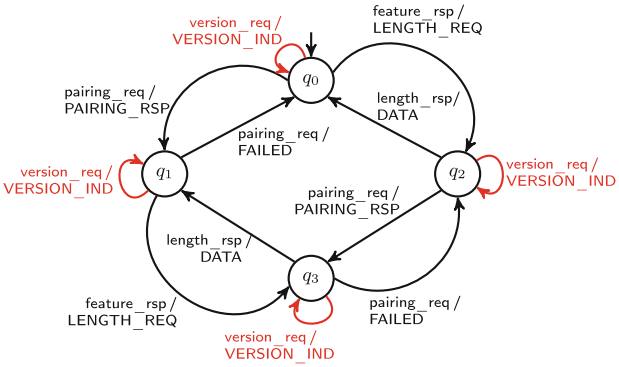

Fig. 4. Model learned of CC2652R1. For clarity, some transitions are not displayed. The complete model is available online [25].

Key Size Acceptance in Pairing Request (V1, A2). We see for all tested SoCs, except for the CYLBE-416045-02, that a reduction of the maximum key size during the pairing request is possible (V1). The test fails on the CYLBE-416045-02 since the SoC does not accept any pairing_req. By performing a pairing_req, the requesting party proposes some parameters for the pairing, e.g., the maximum key size of the long-termkey (LTK) that is later used to distribute session keys for a secure connection. The BLE specification defines that the key size field needs to be within 7 and 16 bytes. Downgrade attacks, e.g. the KNOB attack [4], show that accepting low key sizes decreases the entropy of the LTK and, therefore, enables brute-forcing of the used key. All devices, except CYLBE-416045-02, accept a key with an entropy of 7 bytes. Additionally, fuzzing the accepted key sizes shows that CYW43455 accepts pairing_req that contains maximum key sizes greater than 16 bytes.

5 Related Work

In practice, protocol state fuzzing proved itself a useful tool to reveal security issues and behavioral anomalies of communication protocols, e.g. TCP [13], SSH [15], TLS [30], DTLS [14], 802.11 4-way handshake [36], MQTT [37], OpenVPN [11], and QUIC [27]. In the literature, this technique is also known as learning-based testing where the SUT is tested via active automata learning. Hence, our learning-based fuzzer also performs protocol state fuzzing during learning the behavioral models of the BLE devices.

Several black-box fuzzers for network protocols exist, e.g. *boofuzz* (former Sulley) [24] or *GitLab Protocol Fuzzer Community Edition* (former Peach) [18]. They require user-defined input generators and guidance to in-depth paths. To enable stateful black-box fuzzing model-based techniques have been proposed. *SNOOZE* [5] is a model-based fuzzer for network protocols. However, input generators, as well as the model, must be manually crafted. Another model-based fuzzer for telecommunication protocols is *T-Fuzz* [20]. T-Fuzz extracts the

required model via static analysis during compile time. Therefore, this model-based fuzzer is only applicable in this special environment.

Instead of creating the model manually, learning-based fuzzing techniques automatically infer a behavioral model. Based on passive learning, Comparetti et al. [10] extract a model from given input data. The generated model can then be used as input for the black-box fuzzer Peach [18]. Doupé et al. [12] also present a black-box fuzzer for web applications, where they generate the model via crawling the tested web application. However, passive learning can cover only behavior that is provided by the given input data. Aichernig et al. [2] presented a learning-based fuzzing technique for MQTT servers based on active automata learning. Different from our technique, they learned the model of one SUL, which they assume contained the most conforming behavior. This model is then used to fuzz other implementations of the tested system. The assumption of a generic model might hamper the applicability of learning-based fuzzing. Our results show that such an approach would not be feasible for BLE devices. In contrast to our technique, they fuzzed only one specific input field type and based their conformance tests on random traces which does not provide any guarantees about in-depth state coverage.

Bluetooth attacks and vulnerability collections like *BlueBorne* [32], *BLEEDINGBIT* [33], *KNOB* [4], *BLESA* [39], *Frankenstein* [29], *SweynTooth* [17] and *BrakTooth* [16] reveal severe issues in the Bluetooth protocol. Ruge et al. [29] detected issues concerning Bluetooth classic and BLE. They propose a framework that fuzzes an emulated chip firmware. Since no over-the-air communication is required the time efficiency of fuzzing can be significantly improved. However, preparing the firmware for emulation is tedious. The motivation for our work originates from the fuzzing framework proposed by Garbelini et al. [17]. Instead of providing a handcrafted general model, we extend the model-based fuzzing framework by automata learning. This allows us to create more BLE device-specific input sequences. Furthermore, behavioral differences become visible through the learned individual models. Additionally, we extend our fuzzer by a counterexample analysis tool that reports unknown state transitions or states.

6 Conclusion

6.1 Summary

We presented a learning-based fuzzing technique for BLE devices. Our proposed method is based on a black-box assumption. To achieve in-depth testing, we require a behavioral model of the SUT. Instead of manually crafting the model, we used automata learning to automatically generate the model. Using the learned model, we created a stateful black-box fuzzer. Furthermore, we extended our fuzzer with a counterexample analysis tool that examines unknown behavior. Our evaluation revealed anomalies and security issues in the tested BLE devices. Additionally, our fuzzer crashed four out of six devices.

6.2 Discussion

The missing ability to measure coverage limits the applicability of black-box fuzzing. To overcome this limitation, black-box fuzzing extended by model-based testing techniques shows promising results [17,20]. However, manually crafting models might be an error-prone process. In learning-based fuzzing, we extend fuzzing by automata learning to automatically create behavioral models. Still, our previous work [26] showed that the creation of a fault-tolerant interface for learning a remote physical device might not be straightforward. Nevertheless, this work has to be only done once. Our work indicates that the learning interface can then be easily extended to a stateful black-box fuzzer. The availability of distinct behavioral models enables checking for behavioral differences. Furthermore, we can automatically analyze found counterexamples.

6.3 Future Work

In future work, we propose advancements for learning as well as for fuzzing. For learning, we want to consider other modeling formalisms. Evaluating communication protocols shows that non-deterministic behavior hampers deterministic learning. Modeling this non-deterministic behavior might also hint at faulty behavior. Additionally, the work of Garbelini et al. [17] revealed several vulnerabilities during the pairing procedure of the BLE protocol. For this, we will extend our learning framework to deduce behavioral models of the pairing procedure. For fuzzing, we plan to adapt our generation of fuzzed inputs with a search-based technique. By defining a reward function for test sequences, we might cover more error-handling behavior of the SUT. Regarding our proposed counterexample analysis, we saw that fuzzed inputs revealed not yet observed behavior. Smeters et al. [35] used fuzzing as an equivalence oracle during learning. Following an akin idea, we can extend our learned models by the information that we already gained during the counterexample analysis. With this, we can generate models that also formalize error-handling behavior.

Acknowledgement. This work is funded by the TU Graz LEAD project *Dependable Internet of Things in Adverse Environments*, by the *LearnTwins* project (No 880852) from the Austrian Research Promotion Agency (FFG), and by *AIDOaRt* project (grant agreement No 101007350) from the ECSEL Joint Undertaking (JU). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Sweden, Austria, Czech Republic, Finland, France, Italy, and Spain. We would like to thank Maximilian Schuh for providing support for the BLE devices and the authors of the SweynTooth paper for creating an open-source BLE interface. Furthermore, we thank the anonymous reviewers for their useful remarks.

References

1. Aarts, F., Jonsson, B., Uijen, J., Vaandrager, F.W.: Generating models of infinite-state communication protocols using regular inference with abstraction. Formal Methods Syst. Des. **46**(1), 1–41 (2015). https://doi.org/10.1007/s10703-014-0216-x

2. Aichernig, B.K., Muškardin, 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, 12–16 April 2021, pp. 47–58. IEEE (2021). https://doi.org/10.1109/ICST49551.2021.00017
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
4. Antonioli, D., Tippenhauer, N.O., Rasmussen, K.: Key negotiation downgrade attacks on Bluetooth and Bluetooth Low Energy. ACM Trans. Priv. Secur. **23**(3), 14:1–14:28 (2020). https://doi.org/10.1145/3394497
5. Banks, G., Cova, M., Felmetsger, V., Almeroth, K.C., Kemmerer, R.A., Vigna, G.: SNOOZE: Toward a stateful network protocol fuzzer. In: Katsikas, S.K., López, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, 30 August–2 September 2006, Proceedings. Lecture Notes in Computer Science, vol. 4176, pp. 343–358. Springer (2006). https://doi.org/10.1007/11836810_25
6. Bluetooth SIG: Bluetooth core specification v5.3. Standard (2021). https://www.bluetooth.com/specifications/specs/core-specification/
7. Böhme, M., Cadar, C., Roychoudhury, A.: Fuzzing: Challenges and reflections. IEEE Softw. **38**(3), 79–86 (2021). https://doi.org/10.1109/MS.2020.3016773
8. Capkun, S., Roesner, F. (eds.): 29th USENIX Security Symposium, USENIX Security 2020, 12–14 August 2020. USENIX Association (2020). https://www.usenix.org/conference/usenixsecurity20
9. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Software Eng. **4**(3), 178–187 (1978). https://doi.org/10.1109/TSE.1978.231496
10. Comparetti, P.M., Wondracek, G., Krügel, C., Kirda, E.: Prospex: Protocol specification extraction. In: 30th IEEE Symposium on Security and Privacy (S&P 2009), 17–20 May 2009, Oakland, California, USA, pp. 110–125. IEEE Computer Society (2009). https://doi.org/10.1109/SP.2009.14
11. Daniel, L., Poll, E., de Ruiter, J.: Inferring OpenVPN state machines using protocol state fuzzing. In: 2018 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2018, London, United Kingdom, 23–27 April 2018, pp. 11–19. IEEE (2018). https://doi.org/10.1109/EuroSPW.2018.00009
12. Doupé, A., Cavedon, L., Kruegel, C., Vigna, G.: Enemy of the state: A state-aware black-box web vulnerability scanner. In: Kohno, T. (ed.) Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, 8–10 August 2012, pp. 523–538. USENIX Association (2012). https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe
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
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 and Roesner [8], pp. 2523–2540. 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, 10–14 July 2017, pp. 142–151. ACM (2017). https://doi.org/10.1145/3092282.3092289
16. Garbelini, M.E., Chattopadhyay, S., Bedi, V., Sun, S., Kurniawan, E.: BRAKTOOTH: Causing havoc on Bluetooth link manager. https://asset-group.github.io/disclosures/braktooth/braktooth.pdf (2021). Accessed 8 Jan 2022
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, 15–17 July 2020, pp. 911–925. USENIX Association (2020). https://www.usenix.org/conference/atc20/presentation/garbelini
18. Gitlab.org: Gitlab protocol fuzzer community edition. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce. Accessed 8 Jan 2022
19. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: Whitebox fuzzing for security testing. ACM Queue **10**(1), 20 (2012). https://doi.org/10.1145/2090147.2094081
20. Johansson, W., Svensson, M., Larson, U.E., Almgren, M., Gulisano, V.: T-Fuzz: Model-based fuzzing for robustness testing of telecommunication protocols. In: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, 31 March 2014–4 April 2014, Cleveland, Ohio, USA, pp. 323–332. IEEE Computer Society (2014). https://doi.org/10.1109/ICST.2014.45
21. Le, K.T.: Bluetooth Low Energy and the automotive transformation. https://www.ti.com/lit/wp/sway008/sway008.pdf. Accessed 29 Dec 2021
22. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (1990). https://doi.org/10.1145/96267.96279
23. Muškardin, E., Aichernig, B.K., Pill, I., Pferscher, A., Tappler, M.: AALpy: An active automata learning library. Innovations Syst. Softw. Eng. (2022). https://doi.org/10.1007/s11334-022-00449-3
24. Pereyda, J.: boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz. Accessed 8 Jan 2022
25. Pferscher, A.: Stateful black-box fuzzing of BLE devices using automata learning. https://git.ist.tugraz.at/apferscher/ble-fuzzing. Accessed 9 Jan 2022
26. Pferscher, A., Aichernig, B.K.: Fingerprinting Bluetooth Low Energy devices via active automata learning. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) Formal Methods - 24th International Symposium, FM 2021, Virtual Event, 20–26 November 2021, Proceedings. Lecture Notes in Computer Science, vol. 13047, pp. 524–542. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_28
27. Rasool, A., Alpár, G., de Ruiter, J.: State machine inference of QUIC. CoRR abs/1903.04384 (2019). http://arxiv.org/abs/1903.04384
28. 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
29. Ruge, J., Classen, J., Gringoli, F., Hollick, M.: Frankenstein: Advanced wireless fuzzing to exploit new Bluetooth escalation targets. In: Capkun and Roesner [8], pp. 19–36. https://www.usenix.org/conference/usenixsecurity20/presentation/ruge
30. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, 12–14 August 2015, Washington, D.C., USA, pp. 193–206. USENIX Association (2015). https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

31. Rohith Raj, S., Rohith, R., Moharir, M., Shobha, G.: 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
32. Seri, B., Livne, A.: Exploiting BlueBorne in Linux-based IoT devices. Armis, Inc (2019). https://www.armis.com/research/blueborne/. Accessed 8 Jan 2022
33. Seri, B., Vishnepolsky, G., Zusman, D.: BLEEDINGBIT: The hidden attack surface within BLE chips. Armis, Inc (2019). https://www.armis.com/research/bleedingbit/. Accessed 8 Jan 2022
34. Shahbaz, M., Groz, R.: Inferring Mealy machines. In: Cavalcanti, A., Dams, D. (eds.) FM 2009, Eindhoven, The Netherlands, 2–6 November 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
35. Smetsers, R., Moerman, J., Janssen, M., Verwer, S.: Complementing model learning with mutation-based fuzzing. CoRR abs/1611.02429 (2016). http://arxiv.org/abs/1611.02429
36. 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, 3–7 September 2018, Barcelona, Spain, 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
37. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing IoT communication via active automata learning. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, 13–17 March 2017, Tokyo, Japan, pp. 276–287. IEEE Computer Society (2017). https://doi.org/10.1109/ICST.2017.32
38. 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
39. Wu, J., et al.: BLESA: Spoofing attacks against reconnections in Bluetooth Low Energy. In: Yarom, Y., Zennou, S. (eds.) 14th USENIX Workshop on Offensive Technologies, WOOT 2020, 11 August 2020. USENIX Association (2020). https://www.usenix.org/conference/woot20/presentation/wu
40. Zalewski, M.: American fuzzy lop. https://lcamtuf.coredump.cx/afl/ (2013). Accessed 2 Jan 2022