Formal Methods in System Design (2022) 61:35–62
https://doi.org/10.1007/s10703-023-00425-y

ORIGINAL ARTICLE

Fingerprinting and analysis of Bluetooth devices with automata learning

Andrea Pferscher[1] · Bernhard K. Aichernig[1]

Received: 1 April 2022 / Accepted: 16 April 2023 / Published online: 22 May 2023

Abstract
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 only the size of the system limits the applicability of automata learning. Also, the interaction with the system under learning creates a major bottleneck that is rarely discussed. In this article, 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 six investigated BLE devices. Furthermore, we extended the learning technique to learn security critical behavior, e.g., key-exchange procedures for encrypted communication. The learned models depict several behavioral differences and inconsistencies to the BLE specification. This shows that automata learning can be used for fingerprinting black-box devices, i.e., characterizing systems via their specific learned models. Moreover, learning revealed a crashing scenario for one 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 4.7 billion Bluetooth devices are shipped annually and that the number will grow to seven billion by 2026 [1]. This growth mainly refers to the increase of peripheral

✉ Andrea Pferscher
apfersch@ist.tugraz.at

Bernhard K. Aichernig
aichernig@ist.tugraz.at

[1] Institute of Software Technology, Graz University of Technology, Inffeldgasse 16b/II, 8010 Graz, Austria

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, especially, since the insight into IoT components is frequently limited. For example, Texas Instruments [2] motivates in a technical report that wired communication in a car can be replaced by BLE. Considering that automotive components are developed by many different suppliers, the used BLE chip and, more likely, the installed firmware version might be unknown. Facing such challenges, the system under test must be considered 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. [3] successfully used a generic model of the BLE protocol to detect security vulnerabilities of BLE devices via model-based fuzzing. However, they state that the creation of such a comprehensive model was challenging since the BLE protocol is underspecified.

To overcome the possibly tedious and error-prone process of model creation, learning-based testing techniques have been proposed [4]. Learning-based testing applies automata learning algorithms to automatically infer a behavioral model of a black-box system. The learned model could then be used for further verification and testing.

Existing work [5–10] applied learning-based testing to create behavioral models of communication protocols like TLS, TCP or SSH. The learned models show behavioral inconsistencies to the specification and security vulnerabilities. In the literature, this technique is also known as protocol state fuzzing. Motivated by promising results of protocol state fuzzing, various automata learning algorithms have been proposed to extend learning for more complex system properties like timed [11, 12] or stochastic behavior [13]. However, few evaluations of these algorithms on implementations on physical devices exist.

In this article, we present a case study that applies automata learning on BLE devices. Figure 1 illustrates the basic concept presented in this article. Our objective is to learn the behavioral model of the BLE protocol implementation on a physical device. 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

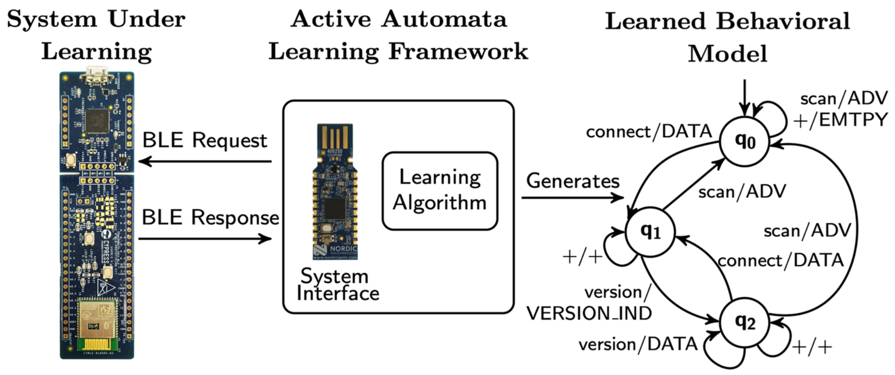

Fig. 1 Automata learning framework for the inference of behavioral models of BLE devices. The model is generated via an active interaction through a system interface that includes an additional BLE device for communication

techniques. The learning algorithm actively queries the System Under Learning (SUL). To enable an active interaction with the SUL, we propose a system interface that communicates via BLE with the SUL. For this, we include in our interface an additional BLE device that enables the sending of custom BLE packets. The system interface also allows handling encrypted communication. Furthermore, we adapt the learning algorithm considering practical challenges that occur in learning real network components.

In our case study, we present our results on learning six different BLE devices. For all six devices, we learn the behavior during the BLE connection establishment. In addition, for three out of these six devices, we learn a model of the BLE pairing process. The pairing process includes security critical behavior like the exchange of keys to establish an encrypted communication. Based on our results, we discuss three 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 fingerprint black-box systems. Second, the presented performance metrics show that not only does the system's size influences the performance of the learning algorithm, but also 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. Third, learning reveals robustness issues of the investigated devices. We observe that devices might crash upon unexpected input sequences.

The paper includes the following contributions:

1. A learning framework that enables learning of BLE protocol implementations of peripheral devices. The framework including the learned models are available online [14].
2. An extension of the learning framework that enables the learning of the key-exchange protocol during the BLE pairing procedure.
3. A case study that evaluates our framework on real physical devices.
4. A model-based manual analysis that checks if the devices conform to the BLE specification.
5. A sequence found by learning that crashes a BLE device.
6. A sequence that allows fingerprinting the investigated BLE devices.

One future purpose of our work is to develop model-based testing techniques for BLE using our learned models. Following the model-based fuzzing technique by Garbelini et al. [3], in the next step, our learning framework could facilitate such a black-box fuzzer by automatically learning the model. In our follow-up work [15], we have already applied the learning framework that we propose in this article to generate a stateful black fuzzer for BLE. Our learning-based fuzzing technique revealed inconsistencies to the BLE specification and robustness issues in the investigated BLE devices.

This article is an extended version of our conference paper "Fingerprinting Bluetooth Low Energy Devices via Active Automata Learning" [16] presented at *Formal Methods - 24th International Symposium*. Additional content and contributions in this journal article include the extension of our learning framework for the BLE pairing procedure. For this, we create an advanced learning interface that enables the establishment of encrypted communication with the peripheral. Further extensions, include an advanced caching technique that deals with non-deterministic behavior to increase the robustness of our proposed learning technique. Furthermore, we consider an additional BLE device in the case study and the identification of a crashing sequence for a BLE device.

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

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 [17, 18] 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 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
- $\delta : Q \times I \rightarrow Q$ is the state-transition function
- $\lambda : Q \times I \rightarrow O$ 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 $(i_1, o_1), \ldots, (i_j, o_j), \ldots, (i_n, o_n)$ with $i_j \in I$, $o_j \in O$, $j \leq n$ 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 $\delta^* : Q \times I^* \rightarrow Q$ gives the reached state after the execution of the input sequence and the output function $\lambda^* : Q \times I^* \rightarrow O^*$ 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 equivalent if there exists no $s_I \in I^*$ such that $\lambda^*(q_0, s_I) \neq \lambda'^*(q'_0, s_I)$, i.e., the execution of all input sequences leads to the same 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. Behavior that is not covered in the data set, can only be approximated by generalizations. Hence, incomplete data presents a problem for passive learning since the generalization for unusual behavior could be inadequate. For example, ordinary log files might not cover the behavior of unusual input sequences. In our previous work [19], we showed that passive learning requires a significantly larger data set than active learning to cover the same behavior with randomly generated input sequences. Therefore, the attempt to include rare behavior via random sequences requires a large number of samples. Active techniques, instead, actively query the SUL. As a result, 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 [20]. 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 we illustrate in Fig. 2. The framework comprises two members: the *learner* and the *teacher*. The learner constructs a DFA by questioning the teacher, who knows 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 to ask if the DFA conforms to the SUL, i.e., the learner queries equivalence. The teacher answers 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 returned, the learner uses this counterexample to pose new membership queries and construct a new hypothesis. This iterative procedure is repeated until a conforming hypothesis is proposed.

The L^* algorithm has been extended to learn Mealy machines of reactive systems [21–23]. Figure 2 includes the adaptions for learning Mealy machines, where membership queries are replaced by output queries. For this, the learner asks for the output sequence produced by a given input sequence. We assume that the teacher has access to the SUL to execute inputs and observe outputs. Furthermore, Angluin's L^* algorithm requires the SUL to be resettable to an initial state.

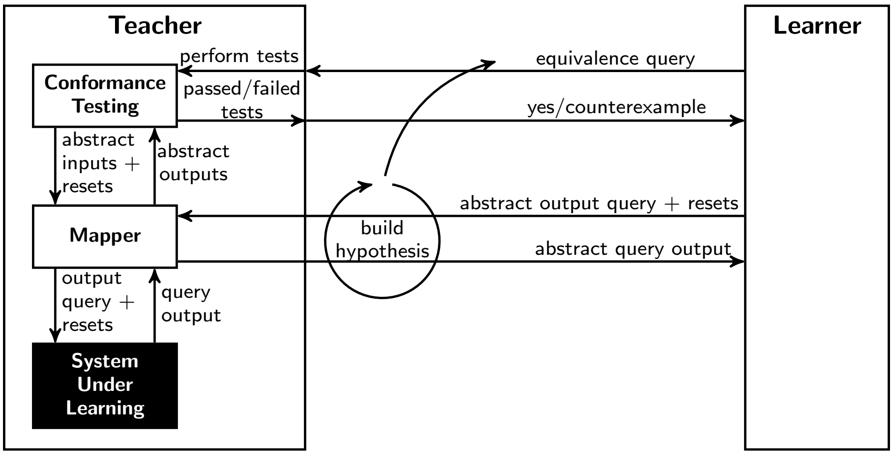

Fig. 2 An adapted version of the Minimally Adequate Teacher (MAT) framework proposed by Angluin [20]. The adaptions include the learning of Mealy machines, the replacement of the equivalence oracle by conformance testing, and the concept of abstraction by a mapper component. This figure has been adapted from the illustration proposed by Aichernig et al. [4]

In practice, we cannot assume a *perfect* teacher who provides the counterexample that shows non-conformance between the hypothesis and the SUL. To overcome this problem, we use conformance testing to substitute equivalence queries. According to the definition of Lee and Yannakakis [24], conformance testing assesses if an implementation conforms to a specification. They assume that the specification is a Mealy machine and the implementation is a black box where outputs to the corresponding input sequences are observable. For learning, we want to test if our learned model correctly defines the behavior of our SUL. For this, we test the conformance between the SUL and the provided hypothesis.

We assume that the behavior of the SUL can be represented by a Mealy machine. For this, we can define the conformance relation based on the equivalence of Mealy machines. However, since the final number of states of the SUL is unknown, we can only approximate conformance by a set of finite input/output sequences. For this, we say that the learned hypothesis $\mathcal{H} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ conforms to the SUL $\mathcal{I} = \langle Q', q'_0, I', O', \delta', \lambda' \rangle$ if for a finite set of input sequences $\mathcal{S}_I$ the following relation holds.

$$\forall s_I \in \mathcal{S}_I \, : \, \lambda^*(q_0, s_I) = \lambda'^*(q'_0, s_I) \tag{1}$$

The goal in conformance testing during learning is to find an input sequence that violates this conformance relation. In the case a counterexample is found, the teacher provides such an input sequence to the learner as a counterexample to the conformance between the hypothesis and the SUL. The learner uses this counterexample to refine the hypothesis by performing further output queries. The refinement of the hypothesis is repeated until no counterexample to the conformance between the hypothesis and the SUL can be found.

The complexity of automata learning depends on the number of states and the considered input alphabet. Especially, in the learning of communication protocols, considering all possible inputs would make learning infeasible. Cho et al. [25] introduce the concept of abstraction to make the learning of communication protocols feasible. For this, instead of considering a large set of inputs, a smaller set of abstract inputs and outputs is considered for learning. Hence, the learned model represents an abstraction of the SUL. Aarts et al. [26] present the concept of abstraction with the introduction of a mapper component. The purpose of the mapper is to translate the input and output actions respectively. For this, abstract inputs for learning are translated by the mapper into concrete inputs that can be executed on the SUL. For outputs, the mapper translates the received concrete outputs into abstract outputs. Figure 2 illustrates the placement of the mapper in the MAT framework.

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 [27] defines the connection and pairing protocol between two BLE devices according to different layers of the BLE protocol stack. Figure 3 shows the initial communication messages of two BLE devices that first establish a connection and then exchange parameters to establish an encrypted communication. We distinguish between the *peripheral* and the *central* device. An example of a central device would be a smartphone that wants to connect with a peripheral device, e.g., a smart watch. In the remainder of this article, 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 in the *scanning* state. For this, the central

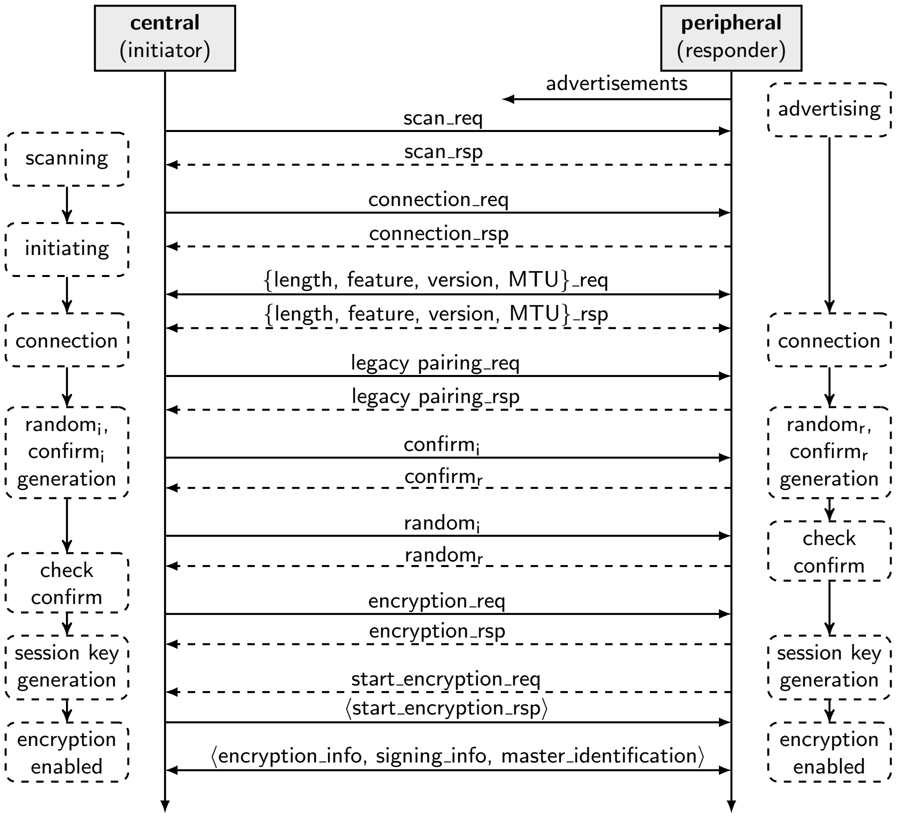

Fig. 3 Communication between a BLE central and peripheral device to establish a connection. After the connection is established, the pairing procedure starts. In the legacy pairing method, the central and peripheral exchange values to generate a session key that is used for an encrypted communication (indicated by ⟨...⟩). The sequence diagram is taken from Garbelini et al. [3] and extended by the message sequence for the legacy pairing procedure taken from the BLE specification [27]

sends a scan request (scan_req) to the peripheral, which responses 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. After the connection, the negotiation on communication parameters starts. Both, the central and peripheral can request features or send control packets. These request and control packets include maximum packet length, maximum transmission unit (MTU), BLE version, and feature exchanges. As noted by Garbelini et al. [3], the order of the feature requests is not defined in the BLE specification and can differ for each device.

After the parameter negotiation, the central initiates the pairing procedure by sending a pairing request to peripheral, which is answered by a pairing response. The BLE protocol distinguishes two pairing methods: *legacy* and *secure* pairing. The difference between the two pairing methods is in the generation of encryption keys. *Legacy* pairing uses priorly exchanged parameters to create a session key, whereas *secure* pairing requires a public/private key exchange to establish a secure encrypted connection. In the remainder of this article, we will only consider *legacy* pairing, since this pairing mode was implemented by all investigated devices.

The *legacy* pairing procedure starts with sending the corresponding request (legacy_pairing_req). The BLE specification [27] refers to the device that sends the pairing request as *initiator* and to the recipient as *responder*. In our case, the central is always the initiator and the peripheral the responder. If the responder accepts to pair, a pairing response (legacy_pairing_rsp) is provided. Then both parties generate a random number and a confirm value, where the calculation of the confirm value takes connection parameters and the random value into account. First, both parties distribute the confirm value. Next, the initiator sends its generated random value. The responder checks if the confirm and random value of the initiator match. If the values match, the responder returns its random value and the initiator checks whether the values match. Afterwards, the initiator shares its part of the encryption key and the initialization vector by forwarding an encryption request (encryption_req). The responder first sends the corresponding response (encryption_rsp) and then a request to start the encryption (start_encryption_req). The initiator uses the received key parts and transfers an encrypted response (start_encryption_rsp) to the encryption start request. For encryption AES-CCM [28] is used. After the responder receives the encrypted start_encryption_rsp, the central and peripheral communication is encrypted. An established communication can be terminated via a termination indication (termination_ind) and an exchanged encryption key can be renewed by the encryption pause procedure pause_encryrption_req.

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. Following related protocol state fuzzing techniques [5–10], we assume that unusual input sequences that are executed during active learning test the robustness of the SUL. Additionally, we aim to reveal characteristic behavior that enables fingerprinting of the peripheral. According to Sect. 2.3, we can model the BLE protocol as a reactive system.

Our objective is to learn the behavior during the connection and pairing procedure as depicted in Fig. 3. However, the first experiments indicate that especially the pairing procedure leads more frequently to non-deterministic behavior which hampers the learnability of BLE devices. Therefore, we separated the learning of the connection and pairing procedure. Our first learning setup considers inputs that are required to establish a connection until a pairing request initiates the pairing procedure. Related to Fig. 3, this connection procedure includes the first four request/response steps. The second learning setup considers only the inputs required to establish an encrypted communication, i.e., the request/response steps of the pairing procedure, including the pairing request. Hence, both models contain the pairing request.

Even though the considered inputs are different, we apply the same learning architecture for both learning setups. Figure 4 depicts our learning architecture which is based on the architecture for network protocols proposed by Tappler et al. [8]. We extended their proposed learning architecture with an additional learning interface. This learning interface should enable the robust learning of a behavioral model. This additional layer ensures an reliable reset and query execution on the SUL. Fig. 4 includes the five components of

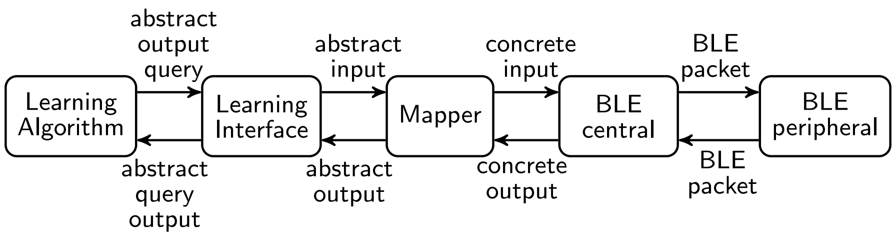

Fig. 4 We extended the learning architecture of Tappler et al. [8] with a learning interface that enables the robust learning of an abstracted model from a BLE device

the learning interface: learning algorithm (Sect. 3.1), learning interface (Sect. 3.2), mapper (Sect. 3.3), BLE central (Sect. 3.4), and BLE peripheral (Sect. 3.5). In the following sections, we describe each component of the applied learning architecture.

3.1 Learning algorithm

The applied *learning algorithm* is an improved variant of the L^* algorithm for Mealy machines [21–23]. Since L^* is based on an exhaustive input exploration in each state, we assume that it is beneficial for performing a behavioral analysis and fingerprinting. Rivest and Schapire [29] proposed the improved L^* version that contains advanced counterexample processing. This advanced counterexample processing reduces the number of required output queries in case a counterexample to the conformance between the SUL and provided hypothesis is found. Especially for long counterexamples, this improved L^* version decreases the number of required queries.

Since Python enables the usage of convenient libraries for the composition of BLE packets, we aim at a consistent learning framework integration. At present, AALpy [18] is a novel active learning library 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 Eq. 1 holds. To create a sufficient test suite, we combine random testing with state coverage. For this, the generated set of input sequences accesses each state in the learned hypothesis. In each state, we then execute a random set of inputs. 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. Since the final number of states of the minimal Mealy machine that represents the SUL is unknown, the parameters n_{test} and n_{len} can only be approximated. The approximation is a trade-off between sufficient conformance testing to find a counterexample and an efficient runtime of the learning algorithm.

3.2 Learning interface

The applied L^*-based learning algorithm requires the system to be resettable and to behave deterministically. These requirements hamper the straightforward application of learning physical devices via a wireless network. To overcome these issues, we introduce an additional layer that ensures that the SUL is reliably reset and that the observed non-deterministic behavior is resolved.

3.2.1 Guarantee reliable resets

The learning algorithm expects that every performed query is executed from an initial state. Hence, we require that the SUL can be reset to an initial state. Lee and Yannakakis [24] defined a reset to be *reliable* if this reset action transfers the system to the initial state independent from the current state. We assume that a hard reset reliable resets the peripheral device. Since a hard reset after each query would make active learning a tedious process, we assume that the central can reset the device via BLE messages. For this, we assume that a scan_req or a termination_req resets the peripheral in any state to the advertising state as shown in Fig. 3. Depending on which part of the protocol we aim to learn, i.e., the connection or the pairing procedure, we consider different initial states. The initial state for the connection procedure is the advertising state. For the pairing procedure, we assume that the connection is established and all required parameters are negotiated such that the pairing procedure can be initiated.

The learning library AALpy can perform resetting actions before and after the output query execution. Like in other automata learning libraries [30], we denote the method that is called *before* executing the output query as `pre` and the method *after* the output query as `post`. In any case, we want to terminate a possibly established connection after the execution of an output query. For this, we perform a termination request in the `post` method. The termination request is specific BLE packet that indicates the device that sends this indication wants to terminate the connection. To ensure a proper reset before executing the output query, a scan request is performed in the `pre` method which checks if the device distributes advertisements.

For learning the pairing procedure, we extend the `pre` and `post` method. In the `pre` method, we establish a valid connection, which includes the first three steps of Fig. 3. After this procedure, the peripheral should be ready to accept a pairing request from the central. In the `post` method, we initiate the pause encryption procedure if encryption is enabled. This procedure indicates that the encryption key shall be changed. Afterward, we terminate the connection as described before.

We assume that this reset procedure reliably resets the SUL to the assumed initial state if the peripheral responds to our resetting BLE messages. If do not receive a response, we repeat this resetting procedure n_{error} times. For example, we repeat the `pre` method as long as we found advertisements sent by the peripheral. After n_{error} repetitions, we abort the learning procedure.

Another problem that hampers a reset is that some devices stop sending advertisements. This could be the case if we send a large number of unexpected BLE packets to the device, even if we are not connected. In active automata learning this might be the case, e.g., if we test conformance. For this, we use the resetting procedure also to establish and terminate a valid connection before we execute an output query. This should avoid the peripheral from running into a timeout and stop sending advertisements.

3.2.2 Handling non-determinism

Our learning interface also has to deal with non-deterministic behavior. In general, we assume that the BLE devices behave deterministically. However, the influence of environmental conditions on learning a wireless communication protocol can introduce non-determinism. We might experience lost packets or delayed responses. Packet loss is critical for packets that are necessary to establish a connection. For example, the loss of a connection request packet prohibits the establishment of a connection between the two BLE devices. As a result, all responses to further requests are answered differently than for a valid connection. The same problem arises for BLE packets that arrive delayed. For example, consider the following output query.

$$\mathsf{scan\_req} \cdot \mathsf{connection\_req} \cdot \mathsf{feature\_rsp} \cdot \mathsf{pairing\_req}$$

The expected query output would be the following.

$$\mathsf{ADV} \cdot \mathsf{FEATURE\_REQ} \cdot \mathsf{DATA} \cdot \mathsf{PAIRING\_RSP}$$

However, if the peripheral device receives the feature response delayed or if the response got lost, the peripheral might reject the pairing request. Hence, the received query output would look different. In this case, a possible example would be the following output sequence.

$$\mathsf{ADV} \cdot \mathsf{FEATURE\_REQ} \cdot \mathsf{DATA} \cdot \mathsf{FAILED}$$

To deal with non-deterministic behavior that occurs during the execution of output queries, we repeat output queries. However, repeating every output query several times would make learning very inefficient. For this, we introduce an enhanced strategy that aims at the saving of queries. Under the assumption that packet loss or delayed messages occur rarely, we repeat queries only if we observe non-deterministic behavior. For this, our learning interface utilizes the caching strategy of the used learning library AALpy. AALpy provides a tree structure that collects the performed inputs and the corresponding observed outputs. Every observed output on the SUL is checked against the stored output in the cache. We start to collect possible outputs for a node in the tree only in the case that the outputs do not match. After the collection of n_{cache} outputs for that node, we select the most frequently observed output. If the output changes, the cache gets updated. Note that the update might violate the consistency of the data structure for learning. However, the performed counterexample processing proposed by Rivest and Schapire [29] during the conformance test takes care of any inconsistencies. The majority-based update is only done once. Afterward, if we observe non-determinism, we simply repeat the output query. Again, we define an upper limit for a repeating non-deterministic behavior by a maximum of n_{nondet} query executions.

3.3 Mapper

The *mapper* component serves as an abstraction mechanism, since considering all possible BLE packets for learning would not be feasible. Therefore, we use a generic input and output alphabet to learn a behavioral model on a more abstract level. Following Fig. 4, the learning algorithm generates output queries that comprise abstract input sequences. The learning interface receives these abstract input sequences and forwards the single abstract inputs to the mapper. The mapper then 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 interface to perform the corresponding actions for robust learning. The processed abstract output sequence is then used by the learning algorithm to construct and test the hypothesis.

The abstracted input alphabet for learning the connection procedure is defined by $I_C^{\mathcal{A}} = \{\mathsf{scan\_req}, \mathsf{connection\_req}, \mathsf{length\_req}, \mathsf{length\_rsp}, \mathsf{feature\_req}, \mathsf{feature\_rsp}, \mathsf{version\_req}, \mathsf{mtu\_req}, \mathsf{legacy\_pairing\_req}\}$ and for the pairing procedure $I_P^{\mathcal{A}} = \{\mathsf{legacy\_pairing\_req}, \mathsf{confirm}, \mathsf{random}, \mathsf{encryption\_req}, \mathsf{start\_encryption\_rsp}\}$. 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 Sect. 2.3, we explained that after the connection request of the central, also the peripheral might send control packets 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, length_rsp, and start_encryption_rsp are responses from the central that we consider as additional inputs.

The abstract inputs of $I_C^{\mathcal{A}}$ and $I_P^{\mathcal{A}}$ are then translated to concrete BLE packets that can be sent by the central to the peripheral. For example, the abstract input length_req is translated to a BLE control packet including a corresponding valid command of the BLE protocol stack. For the construction of the BLE packets, we use the Python library Scapy [31]. In Scapy syntax, the BLE packet for the length_req can be defined as $\mathsf{BTLE/BTLE\_DATA/BTLE\_CTRL/LL\_LENGTH\_REQ}(\mathit{max\_tx\_bytes}, \ldots)$, where *max_tx_bytes* is a field that is concretized by the mapper component. To concretize fields, we mainly select values from a set of preset values defined by Scapy. These preset values conform to standard values that enable the establishment of a connection.

For the translation of outputs, the mapper receives a list of concrete BLE packets from the central device. The central provides a list of packets since the peripheral device answers with multiple BLE packets to a single BLE request. The set of received packets is parsed using the Scapy library. In the following example, we receive for the sent feature_rsp a list of different BLE packets.

$$\begin{aligned}
\mathit{req} &= \mathsf{BTLE/BTLE\_DATA/BTLE\_CTRL/LL\_FEATURE\_RSP}\\
\mathit{rsp} &= \{\mathsf{BTLE/BTLE\_DATA},\\
&\quad \mathsf{BTLE/BTLE\_DATA/BTLE\_CTRL/LL\_LENGTH\_REQ},\\
&\quad \mathsf{BTLE/BTLE\_DATA},\\
&\quad \mathsf{BTLE/BTLE\_DATA},\\
&\quad \mathsf{BTLE/BTLE\_DATA/L2CAP\_Hdr/ATT\_Hdr/ATT\_Exchange\_MTU\_Request},\\
&\quad \mathsf{BTLE/BTLE\_DATA},\\
&\quad \mathsf{BTLE/BTLE\_DATA}\}
\end{aligned}$$

For simplicity, the example hides concrete field values and illustrates the received BLE packets conforming to the Scapy syntax. This list of output packets will be merged into a single output by concatenating the packets in alphabetical order to one output string. This creates deterministic behavior, even though packets might be received in a different order. For the example above, the output would be ATT_Exchange_MTU_Request, ATT_Hdr, BTLE, BTLE_CTRL, BTLE_DATA, L2CAP_Hdr, LL_LENGTH_REQ. For the abstraction of the BLE packets, we use the naming provided by Scapy. One exception applies to the response on scan_req, where two possible valid responses are mapped to one scan response (ADV). If the central device returns an empty list of BLE packets, the mapper returns the empty output which is denoted by the string EMPTY.

In the pairing procedure, the central and the peripheral device exchange key information over a multistage response/request dialog. Since the concrete values of the packets in this key-exchange procedure depend on the previously exchanged packets, we cannot use preset values from Scapy for the concretization. Also randomly guessing the concrete values for the key exchange would not be feasible to successfully establish an encrypted connection. Otherwise, the key-exchange procedure would be insecure, since the keys do not depend on randomness or could be brute-forced. Due to this key exchange, we require the mapper to be stateful. For this, the mapper stores and collects messages that are later required to establish an encrypted communication, e.g., received parts of the key information. Furthermore, the mapper memorizes if the encryption is enabled. In the case of encrypted communication, the mapper encrypts and decrypts transmitted messages.

3.4 BLE central

The *BLE central* component comprises the adapter implementation and the physical central device. We use the Nordic nRF52840 USB dongle and the Nordic nRF52840 Development Kit as central. Our learning setup requires sending BLE packets stepwise to the peripheral device. For this, our implementation follows the setup proposed by Garbelini et al. [3]. We use their provided firmware for the Nordic nRF52840 devices and adapted their driver implementation to perform single steps of the BLE protocol.

The central device receives from the mapper a concrete BLE packet, which is then transmitted to the peripheral device. Then the central device checks for responses if the peripheral responds to the transmitted packet. As mentioned in the previous section, the peripheral can respond with several packets. For this, the central listens $n^{\mathrm{rsp}}_{\mathrm{min}}$ times for any responses. If after $n^{\mathrm{rsp}}_{\mathrm{min}}$ responses no convincing response has been returned, we continue listening for responses. We define a response as *convincing* if the received packet contains more than an empty BLE data packet, i.e., BTLE/BTLE_DATA. However, the maximum number of listening attempts is limited by $n^{\mathrm{rsp}}_{\mathrm{max}}$. The selection of the parameters $n^{\mathrm{rsp}}_{\mathrm{min}}$ and $n^{\mathrm{rsp}}_{\mathrm{max}}$ depends on the environmental conditions in which the experiment is executed. For example, we need to consider the response time and distance of the SUL.

3.5 BLE peripheral

The *BLE peripheral* represents the black-box device that we want to learn, i.e., the SUL. We assume that the peripheral advertises 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 six different BLE devices. The learning framework is available online [14]. 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.

Table 1 Evaluated BLE devices

Manufacturer (Board)	SoC	Application
Texas Instruments (LAUNCHXL-CC2640R2)	CC2640R2	CC2640R2 LaunchPad
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 BLE devices

Table 1 lists the six investigated BLE devices. In the remainder of this section, we refer to the BLE devices by their System on a Chip (SoC) identifiers. For the case study, we considered devices from different manufacturers. Some of the devices were already included in the case study of Garbelini et al. [3]. We extended the collection with well-known boards, e.g., the Raspberry Pi 4. Since one of our objectives is to identify the SoC based on the observed behavior, we included different SoCs from one manufacturer. All evaluated SoCs support the Bluetooth v5.0 standard [27]. To enable BLE communication, we deployed and ran an exemplary 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.

4.2 BLE Learning

Our learning framework is built upon Python 3.9.0. For our learning setup, we used the Python learning library AALpy [18] (version 1.0.1). For the composition of the BLE packets, we used a modified version of the Python library Scapy [31] (version 2.4.4). The used modifications are available starting from Scapy v2.4.5. As BLE central device, we used the Nordic nRF52840 Dongle and the Nordic nRF52840 Development Kit. The deployed firmware for the USB dongle was taken from the SweynTooth repository [32].

As explained in the previous section, we required a special learning interface to learn the communication protocol implemented on a physical device. For this, we extended some components of the AALpy framework to enable robust automata learning. Our learning interface modified the implementation of the conformance testing technique and the used caching mechanism. These modifications of our framework handled connection errors and non-deterministic outputs according to our explanation in Sect. 3. To enable an efficient but also robust learning environment, we set the maximum number of consecutive connection errors to $n_{\text{error}} = 20$, the size of the non-deterministic cache to $n_{\text{cache}} = 20$, and the number of consecutive non-deterministic output queries to $n_{\text{nondet}} = 20$. These numbers were high enough to recover from faults but low enough to detect early that a device stopped responding.

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 $n_{\text{len}} = 10$. These numbers were set according to the abstract input alphabet size, which included nine different inputs. We stress that the primary focus of this article was an initial exploration of the state space and to fingerprint 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 the conformance of the model to the SUL is required.

In Sect. 3, we explained that a sent BLE message leads to multiple responses. These responses could be distributed over several BLE packets. Hence, our central listened for a minimum number of responses $n^{\mathrm{rsp}}_{\mathrm{min}}$ but stopped listening after $n^{\mathrm{rsp}}_{\mathrm{max}}$ attempts. For our learning setup, we set for five out of six SoCs $n^{\mathrm{rsp}}_{\mathrm{min}} = 10$ and $n^{\mathrm{rsp}}_{\mathrm{max}} = 20$. This setup enabled robust and fast learning for five SoCs, since none of the devices responded with more than ten different outputs, but was able to send a meaningful response within twenty listening attempts. For the sixth device, the nRF52832, we used $n^{\mathrm{rsp}}_{\mathrm{min}} = 20$ and $n^{\mathrm{rsp}}_{\mathrm{max}} = 30$ since our experiments show that this device requires more time to respond. Furthermore, we applied a different parameter setup for the scan request and termination indication that enables a fast, but decent reset. For this, we set $n^{\mathrm{rsp}}_{\mathrm{min}} = 5$ and $n^{\mathrm{rsp}}_{\mathrm{max}} = 50$. For the termination indication, we set $n^{\mathrm{rsp}}_{\mathrm{min}} = n^{\mathrm{rsp}}_{\mathrm{max}} = 1$. Note that the termination indication was not part of the abstract input alphabet. Hence, the purpose was not to capture the output behavior on this request but simply to reset the connection. This justified the low number of listening attempts for the output.

4.2.1 Connection procedure evaluation

All experiments for learning the connection procedure were performed on an Apple MacBook Pro 2019 with an Intel Quad-Core i5 operating at 2.4GHz and with 8GB RAM, running macOS Catalina (Version 10.15.7).

Table 2 shows the learning results for five out of the six investigated SoCs. For two devices, CC2652R1 and CYW43455, we excluded the scan and connection request from the input alphabet. For both devices, we observed that a scan request did not always trigger the expected reset of the connection. Therefore, we needed to check if a connection could be established. For this, we learned the behavior after the execution of a connection request. Table 2 does not include the results of CC2640R2, since we were not able to learn

Table 2 Learning results of five out of six evaluated BLE SoCs. The †-symbol indicates that device was reset to the state where a connection request was already performed. Consequently, no connections errors occur during learning for these devices

	CC2650	CC2652R1†	CYBLE-416045-02	CYW43455†	nRF52832
# States	5	4	3	16	5
Total Time in minutes (min)	23.61	5.57	12.00	65.12	126.24
Learning (min)	18.22	3.46	9.41	51.07	73.80
Conformance Checking (min)	5.39	2.11	2.59	14.05	52.44
# Output Queries	405	196	243	784	405
# Output Query Steps	1542	588	747	3136	1459
# Conformance Tests	59	44	32	164	50
# Conformance Test Steps	626	467	344	1958	580
# Connection Errors	526	–	292	–	459
# Non-Deterministic Outputs	5	1	0	3	1

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 needed only one learning round for each SUL, 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 16 states (CYW43455).

The learning of the largest model regarding the number of states (CYW43455) took a bit more than one hour, whereas the smallest model (CYBLE-416045-02) could be learned in twelve minutes. Even if the nRF52832 did not have the largest state space, the runtime was significantly higher compared to devices with the same state space (CC2650). The results presented in Table 2 show that learning the nRF52832 took more than five times as long as learning the CC2650. The difference in runtime occurred due to the extended waiting time for the nRF52832. This result indicates that the scalability of active automata learning did not only depend 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 influenced the efficiency of active automata learning.

Conforming to the state space, the number of performed output queries and steps increased. For the devices where we considered the connection input, also the number of connection errors seemed to align with the complexity of the behavioral model. This observation emphasized our assumption that message loss regularly occurs. This justified the overhead of a decent error-handling procedure to ensure that the SUL is adequately reset to the initial state before the output query is executed.

Figure 5 shows the learned model of the CC2650 and Fig. 6 of the nRF52832. To provide a clear and concise representation, we merged and simplified transitions. The + -symbol summarizes input and output labels, since depicting all labels for all nine considered inputs would make the models hardly readable. The unmodified learned models of all SoCs considered in this case study are available online [14]. The comparison between the learned models of the CC2650 (Fig. 5) and the nRF52832 (Fig. 6) shows that even

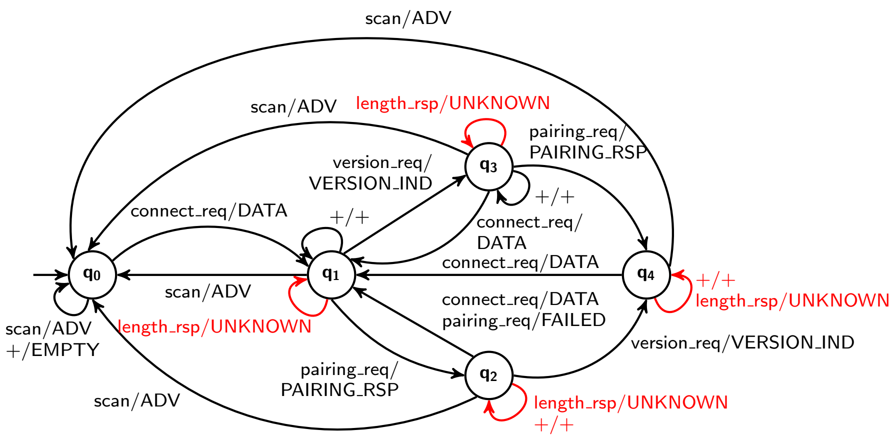

Fig. 5 Simplified learned model of the CC2650. Inputs are lowercased and outputs are capitalized. For a clear presentation, outputs are abbreviated and we highlight the behavior on some selected input/output labels. Other inputs and outputs are summarized by the +-symbol. The complete model is available online [14]

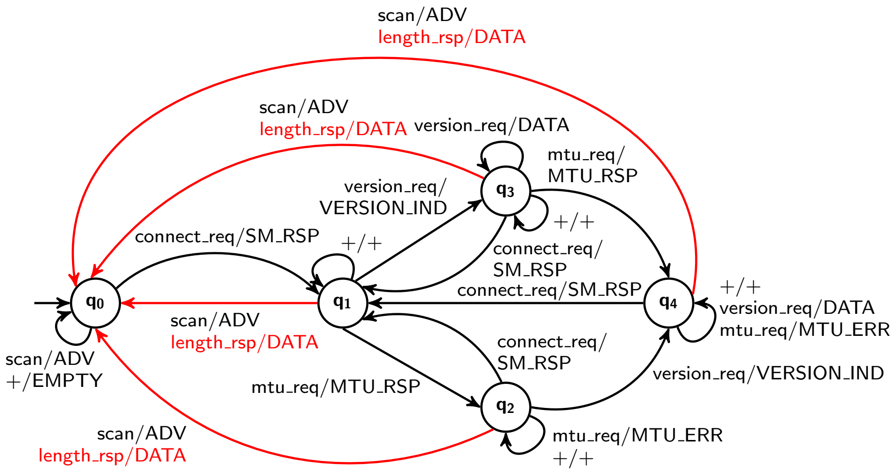

Fig. 6 Simplified learned model of the nRF52832. Labels are abbreviated and summarized. The complete model is available online [14]. Unlike the model of CC2650, presented in Fig. 5, the nRF52832 resets the connection after an unexpected length response

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 responded to an unrequested length response only with a BLE data packet and then completely reset the connection procedure. Therefore, executing an unexpected length response on the nRF52832 led to the initial state akin to the performance of a scan request. The CC2650, instead, reacted to an unrequested length response with a response containing the packet LL_UNKNOWN_RSP and remained in the same state.

Figure 7 illustrates the learned model of the CC2652R1. Note that the initial state q_0 describes the behavior after the peripheral received a connection request. Therefore, the model defines the behavior of the parameter negotiation including the initiation of the pairing procedure. In our recent work [15], we already reported that the learned model

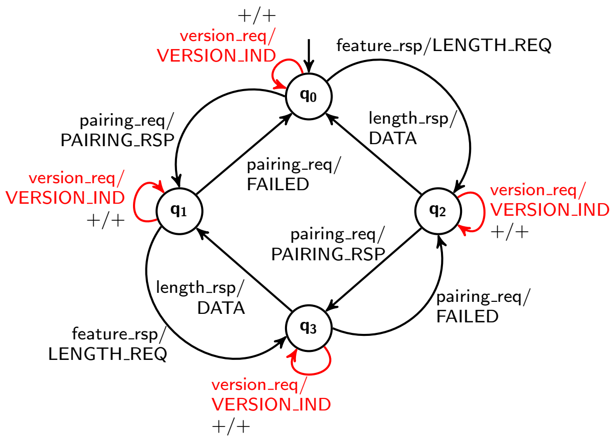

Fig. 7 Model learned of CC2652R1. For clarity, some transitions are not displayed. The complete model is available online [14]

indicates inconsistency with the BLE specification. According to the BLE specification [27], a version indication should be only answered once. As indicated by the red transitions, the CC2652R1 always responded with a version indication.

Using the learning setup of Sect. 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.

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

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

$$\mathsf{LL\_LENGTH\_REQ} \cdot \mathsf{SM\_PAIRING\_RSP} \cdot \mathsf{BTLE\_DATA} \cdot \mathsf{LL\_LENGTH\_RSP} \cdot \underline{\mathsf{LL\_FEATURE\_RSP}}$$

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 corresponded to the following sequence.

$$\mathsf{LL\_LENGTH\_REQ} \cdot \mathsf{SM\_PAIRING\_RSP} \cdot \mathsf{BTLE\_DATA} \cdot \mathsf{LL\_LENGTH\_RSP} \cdot \underline{\mathsf{BTLE\_DATA}}$$

We assume that the execution of the pairing request changed the internal behavior of the SUL. Hence, after the establishment of a certain number of pairing requests, the device failed to respond to provided requests. If we removed 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 observed more non-deterministic behavior than for the other devices, which led to repetitions of output queries.

4.2.2 Pairing procedure evaluation

All experiments for learning the pairing procedure were performed on an HP EliteBook 840 G2 with an Intel i5-5200 operating at 2.2 GHz and with 16GB RAM, running Ubuntu 20.04.2 LTS. We required a Linux-based operating system since we used the security

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)	26.40	47.57	40.29
Learning Time (min)	16.94	30.73	28.29
Conformance Checking Time (min)	9.46	16.84	11.70
# Output Queries	384	705	704
# Output Query Steps	1474	3143	3143
# Conformance Tests	61	115	111
# Conformance Test Steps	712	1406	1371
# Connection Errors	449	822	821
# Non-Deterministic Outputs	1	10	2

manager interface provided by SweynTooth [32]. This library creates valid field values for establishing encrypted communication. The provided module is implemented in C/C++ and uses the BlueZ library, which is a Bluetooth stack implementation for Linux.

We slightly adapted the parameter configuration for learning the pairing procedure. Since we observed for the devices more non-deterministic behavior, we set $n_{\text{error}} = 5$, $n_{\text{cache}} = 3$, and the number of consecutive non-deterministic output queries to $n_{\text{nondet}} = 3$. These numbers are lower than for learning the connection procedure, but in the case of a repeated connection error or non-deterministic error, we did not immediately abort the learning procedure. Instead, the learning framework requested to hard reset the physical device. After the user performed a hard reset the learning procedure continued.

Table 4 presents the learning results of the pairing procedure. For this evaluation, we selected three out of the six devices, since these three devices supported the legacy pairing procedure and acted reasonably reliable. The learned models have between six and eleven states and the learning procedure took between 0.9h and 5.2h. Compared to the learning of the connection procedure, we observed significantly more non-deterministic outputs. We also extended Table 4 by the number of cached values that were changed during learning and the number of performed hard resets.

The results show that the CC2640R2 and the CC2650 required hard resets, but the reasons for the hard reset were different. In the case of the CC2640R2, we observed that the SoC stopped accepting pairing requests after a certain amount of exchanged messages. Hence, repeated non-deterministic errors occur. More interestingly, the CC2650 required a hard reset since the device stopped responding to any request. The following sequence leads to the crash of the device:

$$\mathsf{connection\_req} \cdot \mathsf{pairing\_req} \cdot \mathsf{confirm} \cdot \mathsf{random} \cdot \\ \mathsf{encryption\_req} \cdot \langle \mathsf{pause\_encryption\_req} \rangle \cdot \mathsf{terminate\_ind}$$

After the encryption request, the peripheral expected an encrypted response to the sent encrypted start request. However, our device sent a different unexpected encrypted message. After performing the reset, the CC2650 does not return to the advertising state. This shows that already the execution of an unexpected input sequence can trigger faulty behavior.

Table 4 Learning results of three out of six evaluated BLE SoCs

	CC2640R2	CC2650	CYW43455
# States	11	10	6
Total Time (min)	133.01	312.37	52.72
Learning Time (min)	116.95	201.34	38.83
Conformance Checking Time (min)	16.06	111.03	13.89
# Output Queries	487	453	223
# Output Query Steps	3142	2869	1012
# Conformance Tests	110	100	60
# Conformance Test Steps	273	601	287
# Non-Deterministic Outputs	133	80	29
# Cache Updates	1	3	0
# Hard Resets	6	11	0

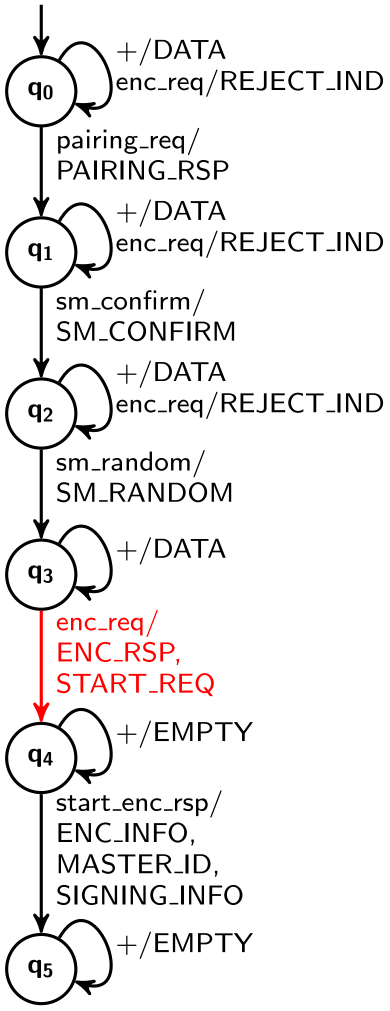

Fig. 8 Model of CYW43455 pairing procedure. Labels are abbreviated and summarized. A complete model is available online [14]

Figure 8 presents the model of the pairing procedure of the CYW43455. We see that the message sequence conforms to the sequence chart shown in Fig. 3. Encryption is enabled via the red transition between states q_3 and q_4. Afterward, only encrypted messages can be distributed. The self-loops for every state show that an unexpected input does not cancel the pairing procedure.

Figure 9 describes the behavior of the CC2640R2. Compared to Fig. 8, we see that unexpected inputs might revert the pairing procedure to previous states. The encryption is enabled after the transition from states q_3 and q_4 is performed. In contrast to the behavior of CYW43455, we still could initiate further pairing procedures after keys have been already exchanged. Our presented learning results show that behavioral differences also occur during the pairing procedure and that the models also enable to fingerprint the tested devices.

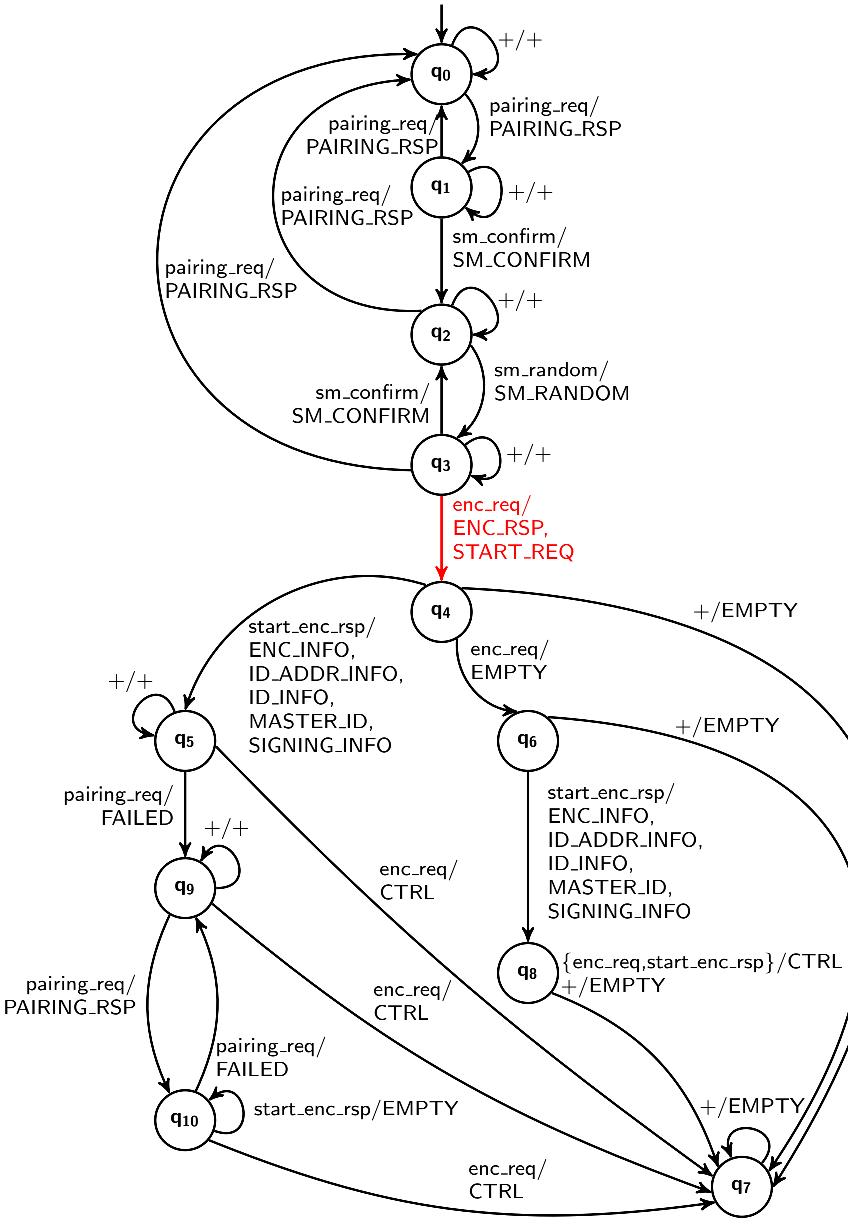

Fig. 9 Model learned of CC2640R2 pairing procedure. For clarity, some transitions are not displayed. The complete model is available online [14]

4.3 BLE fingerprinting

The comparison of the learned models of the connection procedure shows that all investigated SoCs behave differently. Therefore, it is possible to fingerprint 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 connection procedure models shows that even short input sequences sufficiently fingerprint the SoC.

Even if random testing might be faster in the generation of a fingerprinting sequence for the investigated case study subjects. Still there exists many advantages in learning the behavioral model of the BLE device for fingerprinting. The learned behavioral models support the explainability of the found fingerprinting sequence. For example, the models illustrate in which state the models differ. An additional advantage of learning the model is that new fingerprinting sequences can be retrieved offline. For example, if we want to extend our set of investigated devices, we only learn the model of the new device. We can then check if the fingerprinting sequence is still valid to characterize uniquely all investigated devices. If not, we can use the learned models to generate a new fingerprinting sequence. This can be done offline, i.e., no active interaction with the BLE devices is required.

Lee and Yannakakis [24] discuss the conformance testing problem, also referred to as the fault-detection problem. To test the conformance between two systems, they define a so-called *checking sequence* which is an input sequence that generates a different output sequence on both systems. In fingerprinting, we aim at generating such a checking sequence for a set of systems, where the output sequence should be unique for every device.

To generate such a checking sequence, we utilized the data structure that the learner generated during learning to build the hypothesis. Table 5 shows the observable outputs for each input after performing the initial connection request, i.e., the table shows the outputs that identify the state for the corresponding SoC. We determined 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 5 potential output combinations that depict the fingerprint of an 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 investigated SoCs. Under the consideration that a scan request resets the SoC, we define the fingerprinting sequence for the six SoCs output query as follows:

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

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

$$\mathsf{ADV} \cdot \mathsf{SM\_HDR} \cdot \mathsf{LL\_UNKNOWN\_RSP} \cdot \mathsf{ADV} \cdot \mathsf{BTLE\_DATA} \cdot \mathsf{LL\_VERSION\_IND},$$

whereas the sequence for the CC2650 is

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

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 for every SoC should be learned to identify a possibly extended set of input sequences for fingerprinting. However, we recommend replacing the manual analysis with an automatic conformance testing technique between the models akin to Lee and Yannakakis [24] or Tappler et al. [8].

Table 5 The investigated SoCs can be identified by only a single model state that is reached after performing an initial connection request

	feature_rsp	version_req	length_req	length_rsp
CC2640R2	BTLE_DATA	**BTLE_DATA**	**LL_LENGTH_RSP**	**BTLE_DATA**
CC2650	BTLE_DATA	**LL_VERSION_IND**	**LL_UNKNOWN_RSP**	**LL_UNKNOWN_RSP**
CC2652R1	**LL_LENGTH_REQ**	LL_VERSION_IND	LL_LENGTH_RS	BTLE_DATA
CYBLE-416045-02	**LL_REJECT_IND**	LL_VERSION_IND	LL_UNKNOWN_RSP	LL_UNKNOWN_RSP
CYW43455	**ATT_MTU_REQ**	LL_VERSION_IND	LL_LENGTH_RS	LL_REJECT_IND
nRF52832	**LL_UNKNOWN_RSP**	LL_VERSION_IND	LL_LENGTH_RS	BTLE_DATA

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

The bold outputs distinguish the different devices. We observe five different outputs when performing a feature response. To distinguish the sixth device, we could perform either a version request, a length request, or a length response

5 Related work

Celosia and Cunche [33] 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) or Security Manager (SM), 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. [34] 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. [8] 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 [5], TCP [6], SSH [7] or DTLS [10] 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. [9]. They detected security vulnerabilities in the 802.11 4-Way handshake protocol by testing Wi-Fi routers. Aichernig et al. [35] 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 [36] and biometric passports [37]. The proposed techniques use a USB-connected smart card reader to interact with the cards. Furthermore, Chalupar et al. [38] used Lego® to create an interface to learn the model of a smart card reader. In the context of protocol fuzzing, we showed in a follow-up work [15] of this article that the learned BLE models can be used to reveal robustness issues in BLE devices.

6 Conclusion

6.1 Summary

In this article, we presented a case study on learning-based testing of the BLE protocol. The case study aimed 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 six BLE devices. The results of the case study showed that the active learning of a behavioral model is possible in a practicable amount of time. However, our evaluation showed that extensions to state-of-the-art learning algorithms, such as including error-handling procedures, were required for successful model inference. By using learning-based testing, we revealed that one device crashes on the execution of an in-depth testing sequence. Furthermore, 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.

6.2 Discussion

We successfully applied active automata learning to reverse engineer the behavioral models of BLE devices. We experienced challenges in creating a reliable and general learning framework to learn a behavioral model of a wireless protocol implemented on a physical device. To learn deterministic models, we needed to repeat executions on the SUL. Especially, the required guarantee of a reliable reset created issues. Another possibility to overcome this issue would have been to use a resetless learning algorithm as proposed by Rivest and Schapire [29]. However, packet loss and delayed responses still present a problem in such algorithms, since we might assume that we are in the wrong state. Hence, these algorithms would also require countermeasures against non-deterministic observations. The advantage is that BLE interface creation only needs to be done once. Our proposed framework, which is also publicly available [14], 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 depended on the efficiency of the underlying learning algorithm but also on the overhead due to SUL interaction. However, once this interface was created, active automata learning successfully revealed a robustness issue in a tested BLE device. 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 [3]. Compared to the BLE fingerprinting technique of Celosia and Cunche [33], 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.

6.3 Future work

Our first investigation of the security-critical behavior of BLE devices could successfully reveal robustness issues via active automata learning. Despite the found robustness issue, the learned models do not show any security vulnerabilities. However, for future work, we plan to consider additional functionality of the BLE protocol stack, e.g., the secure pairing procedure. Considering the public/private key-exchange procedure might reveal further security issues.

Our proposed method was inspired by the work of Garbelini et al. [3] since their presented fuzz-testing technique demonstrated that model-based testing can be applied 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. In recent work [15], we extended our proposed learning framework for learning-based fuzzing of the BLE protocol. For this, we used our learned models of the connection procedure to generate test cases for fuzzing. Our proposed technique successfully revealed several issues in the implementation of the BLE protocol. We are currently working on extending this technique for learning-based fuzzing the BLE pairing procedure.

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 to the MQTT protocol [39]. Following a similar idea, we could learn a non-deterministic model of the BLE protocol.

Acknowledgements 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.

Funding Open access funding provided by Graz University of Technology.

Declarations

Conflict of interest The datasets generated during and/or analysed during the current study are available in the https://github.com/apferscher/ble-learning repository.

References

1. Bluetooth SIG: Market update. https://www.bluetooth.com/2022-market-update/. Accessed: 2022-10-20
2. Le KT (2021) Bluetooth Low Energy and the automotive transformation. https://www.ti.com/lit/wp/sway008/sway008.pdf. Accessed: 29 Dec 2021
3. Garbelini ME, Wang C, Chattopadhyay S, Sun S, Kurniawan E (2020) SweynTooth: Unleashing mayhem over Bluetooth Low Energy. In: Gavrilovska, A., Zadok, E. (eds.) 2020 USENIX Annual Technical Conference, USENIX ATC 2020, pp. 911–925. USENIX Association, Virtual. https://www.usenix.org/conference/atc20/presentation/garbelini
4. Aichernig BK, Mostowski W, Mousavi MR, Tappler M, Taromirad M (2018) 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, Revised Papers. Lecture Notes in Computer Science, vol. 11026, pp. 74–100. Springer, Dagstuhl Castle, Germany. https://doi.org/10.1007/978-3-319-96562-8_3
5. de Ruiter J, Poll E (2015) Protocol state fuzzing of TLS implementations. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, pp. 193–206. USENIX Association, Washington, D.C., USA. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter
6. Fiterau-Brostean P, Janssen R, Vaandrager FW (2016) Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 454–471. Springer, Toronto, ON, Canada. https://doi.org/10.1007/978-3-319-41540-6_25
7. Fiterau-Brostean P, Lenaerts T, Poll E, de Ruiter J, Vaandrager FW, Verleg P (2017) 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, pp. 142–151. ACM, Santa Barbara, CA, USA. https://doi.org/10.1145/3092282.3092289

8. Tappler M, Aichernig BK, Bloem R (2017) Model-based testing IoT communication via active automata learning. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017, pp. 276–287. IEEE Computer Society, Tokyo, Japan. https://doi.org/10.1109/ICST.2017.32
9. Stone CM, Chothia T, de Ruiter J (2018) 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, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11098, pp. 325–345. Springer, Barcelona, Spain. https://doi.org/10.1007/978-3-319-99073-6_16
10. Fiterau-Brostean P, Jonsson B, Merget R, de Ruiter J, Sagonas K, Somorovsky J (2020) Analysis of DTLS implementations using protocol state fuzzing. In: Capkun, S., Roesner, F. (eds.) 29th USENIX Security Symposium, USENIX Security 2020, pp. 2523–2540. USENIX Association, Virtual Event. https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean
11. Tappler M, Aichernig BK, Larsen KG, Lorber F (2019) 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, Proceedings. Lecture Notes in Computer Science, vol. 11750, pp. 216–235. Springer, Amsterdam, The Netherlands. https://doi.org/10.1007/978-3-030-29662-9_13
12. Aichernig BK, Pferscher A, Tappler M (2020) From passive to active: Learning timed automata efficiently. In: Lee, R., Jha, S., Mavridou, A. (eds.) NASA Formal Methods - 12th International Symposium, NFM 2020, Proceedings. Lecture Notes in Computer Science, vol. 12229, pp. 1–19. Springer, Moffett Field, CA, USA. https://doi.org/10.1007/978-3-030-55754-6_1
13. Tappler M, Aichernig BK, Bacci G, Eichlseder M, Larsen KG (2019) 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, Proceedings. Lecture Notes in Computer Science, vol. 11800, pp. 651–669. Springer, Porto, Portugal. https://doi.org/10.1007/978-3-030-30942-8_38
14. Pferscher A Fingerprinting Bluetooth Low Energy via active automata learning. https://github.com/apferscher/ble-learning. Accessed 31 Mar 2022
15. Pferscher A, Aichernig BK (2022) Stateful black-box fuzzing of Bluetooth devices using automata learning. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13260, pp. 373–392. Springer, Pasadena, CA, USA. https://doi.org/10.1007/978-3-031-06773-0_20
16. Pferscher A, Aichernig BK (2021) 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, Proceedings. Lecture Notes in Computer Science, vol. 13047, pp. 524–542. Springer, Virtual Event. https://doi.org/10.1007/978-3-030-90870-6_28
17. Isberner M, Howar F, Steffen B (2015) 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, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 487–495. Springer, San Francisco, CA, USA. https://doi.org/10.1007/978-3-319-21690-4_32
18. Muškardin E, Aichernig BK, Pill I, Pferscher A, Tappler M (2022) AALpy: an active automata learning library. Innov Syst Softw Eng 18(3):417–426. https://doi.org/10.1007/s11334-022-00449-3
19. Aichernig BK, Muskardin E, Pferscher A (2022) Active vs. passive: A comparison of automata learning paradigms for network protocols. Comput Res Repos **abs/2209.14031** 2209.14031. https://doi.org/10.48550/arXiv.2209.14031
20. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106. https://doi.org/10.1016/0890-5401(87)90052-6
21. Margaria T, Niese O, Raffelt H, Steffen B (2004) Efficient test-based model generation for legacy reactive systems. In: Ninth IEEE International High-Level Design Validation and Test Workshop 2004, 2004, pp. 95–100. IEEE Computer Society, Sonoma Valley, CA, USA. https://doi.org/10.1109/HLDVT.2004.1431246. https://ieeexplore.ieee.org/xpl/conhome/9785/proceeding
22. Niese O (2003) An integrated approach to testing complex systems. PhD thesis, Technical University of Dortmund, Germany. https://d-nb.info/969717474/34
23. Shahbaz M, Groz R (2009) Inferring Mealy machines. In: Cavalcanti, A., Dams, D. (eds.) FM 2009, Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 207–222. Springer, Eindhoven, The Netherlands. https://doi.org/10.1007/978-3-642-05089-3_14
24. Lee D, Yannakakis M (1996) Principles and methods of testing finite state machines-a survey. Proc IEEE 84(8):1090–1123. https://doi.org/10.1109/5.533956

25. Cho CY, Babic D, Shin ECR, Song D (2010) Inference and analysis of formal models of botnet command and control protocols. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010, pp. 426–439. ACM, Chicago, Illinois, USA. https://doi.org/10.1145/1866307.1866355
26. Aarts F, Jonsson B, Uijen J, Vaandrager FW (2015) Generating models of infinite-state communication protocols using regular inference with abstraction. Form Meth Syst Design 46(1):1–41. https://doi.org/10.1007/s10703-014-0216-x
27. Bluetooth SIG: Bluetooth core specification v5.3. Standard, Bluetooth SIG (2021). https://www.bluetooth.com/specifications/specs/core-specification-5-3/
28. Murphy S (1999) The advanced encryption standard (AES). Inf Secur Tech Rep 4(4):12–17. https://doi.org/10.1016/S1363-4127(99)80083-1
29. Rivest RL, Schapire RE (1993) Inference of finite automata using homing sequences. Inf Comput 103(2):299–347. https://doi.org/10.1006/inco.1993.1021
30. Howar F, Isberner M, Merten M, Steffen B (2012) LearnLib tutorial: From finite automata to register interface programs. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I. Lecture Notes in Computer Science, vol. 7609, pp. 587–590. Springer, Heraklion, Crete, Greece. https://doi.org/10.1007/978-3-642-34026-0_43
31. S, R.R., R R, Moharir M, G S (2018) Scapy - a powerful interactive packet manipulation program. In: 2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS), pp. 1–5 . https://doi.org/10.1109/ICNEWS.2018.8903954
32. Garbelini ME, Wang C, Chattopadhyay S, Sun S, Kurniawan E SweynTooth - Unleashing Mayhem over Bluetooth Low Energy. https://github.com/Matheus-Garbelini/sweyntooth_bluetooth_low_energy_attacks. Accessed: 2021-05-05
33. Celosia G, Cunche M (2019) 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, pp. 24–31. ACM, London, UK. https://doi.org/10.1145/3338507.3358617
34. Argyros G, Stais I, Jana S, Keromytis AD, Kiayias A (2016) 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, pp. 1690–1701. ACM, Vienna, Austria. https://doi.org/10.1145/2976749.2978383
35. Aichernig BK, Burghard C, Korosec, R (2019) Learning-based testing of an industrial measurement device. In: Badger, J.M., Rozier, K.Y. (eds.) NASA Formal Methods - 11th International Symposium, NFM 2019, Proceedings. Lecture Notes in Computer Science, vol. 11460, pp. 1–18. Springer, Houston, TX, USA. https://doi.org/10.1007/978-3-030-20652-9_1
36. Aarts F, de Ruiter J, Poll E (2013) Formal models of bank cards for free. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, pp. 461–468. IEEE Computer Society, Luxembourg, Luxembourg. https://doi.org/10.1109/ICSTW.2013.60
37. Aarts F, Schmaltz J, Vaandrager FW (2010) 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, Proceedings, Part I. Lecture Notes in Computer Science, vol. 6415, pp. 673–686. Springer, Heraklion, Crete, Greece. https://doi.org/10.1007/978-3-642-16558-0_54
38. Chalupar G, Peherstorfer S, Poll E, de Ruiter J (2014) Automated reverse engineering using Lego®. In: Bratus, S., Lindner, F.F. (eds.) 8th USENIX Workshop on Offensive Technologies, WOOT '14. USENIX Association, San Diego,CA, USA. https://www.usenix.org/conference/woot14/workshop-program/presentation/chalupar
39. Pferscher A, Aichernig BK (2020) 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, Proceedings. Lecture Notes in Computer Science, vol. 12543, pp. 52–69. Springer, Naples, Italy. https://doi.org/10.1007/978-3-030-64881-7_4