International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-025-00797-y

GENERAL

Special Issue: ECBS 2023

Learning single and compound-protocol automata and checking behavioral equivalences

Stefan Marksteiner[1,2] · David Schögler[3] · Marjan Sirjani[2] · Mikael Sjödin[2]

Accepted: 4 April 2025

Abstract

This paper presents a method and a practical implementation that complements traditional conformance testing. We infer a Mealy state machine of the system-under-test using active automata learning. This automaton is checked for bisimulation with a specification automaton modeled after the standard, which provides a strong verdict of conformance or nonconformance. We further present a method to learn models of multiple communication protocols running on the same device using a dispatcher system in conjunction with the same automata learning algorithms. We subsequently use similar checking methods to compare it with separately learned models. This allows for determining whether there is some interference or interaction between those protocols. In the practical execution of the system, we concentrate on lower levels of the Near-Field Communication (NFC, ISO/IEC 14443-3) and the Bluetooth Low-Energy (BLE) protocols. As a by-product, we share some observations of the performance of different learning algorithms and calibrations in the specific setting of ISO/IEC 14443-3, which is the difficulty to learn models of systems that a) consist of two very similar structures and b) timeout very frequently, as well as the role of conformance testing for compound models and speed optimizations for time-sensitive protocols.

Keywords NFC · BLE · Automata learning · Protocol compliance · Bisimulation · Formal methods

1 Introduction

This article is based on a paper [30], in which we describe an approach for evaluating the compliance of Near-Field Communications (NFC)-based chip systems with the ISO/IEC 14443-3 NFC handshake protocol [20] using automata learning and equivalence checking. In this paper, we presented a tool chain that is easy to use; both the learning and the equivalence checking can run fully automatic. A complete protocol implementation automaton of the system-under-test (SUT), in the context of this paper called system-under-learning (SUL), compared with a specification automaton modeled after the protocol's standard, provides a complement to conformance testing.

In this paper, we expand the scope of our previous work by learning a compound-protocol automaton of multiple protocols running on the same device. In particular, we learn Bluetooth Low-Energy (BLE) [6] alongside with NFC. We also learn separate models of both protocols from the same device and subsequently compare them with the compound automaton. The individually learned automata are used as a specification with which we compare the behavior of the learned compound-protocol automaton using the same technique as described above. By comparing the compound automaton with the separately learned, we can check if the two protocols influence each other, whether intentionally or not, potentially uncovering unintended system states. This determines if device behaves differently when we stimulate both protocols. In theory, the approach can also be used to learn compound-protocol automata of even higher numbers of protocols. In practice, however, inherent complexity makes it more unlikely to yield different results and the practical feasibility is limited by timing constraints that are aggravated with more complex stacking of inputs from different protocols. The remainder of this paper is structured as follows.

✉ S. Marksteiner
stefan.marksteiner@avl.com

D. Schögler
david.schoegler@avl.com

M. Sirjani
marjan.sirjani@mdu.se

M. Sjödin
mikael.sjodin@mdu.se

1 AVL List GmbH, Graz, Austria

2 Mälardalen University, Västerås, Sweden

3 AVL DiTest GmbH, Graz, Austria

Published online: 15 April 2025

First, we provide its motivation and contribution. Section 2 gives an overview of basic concepts in this paper, including a formal definition of bisimulation for Mealy machines as used in this paper. Sections 3.1, 3.2, and 3.3 describe the developed interfaces for automata learning of NFC systems, BLE systems, and compound-protocol learning, respectively. Sections 4.1, 4.2, and 4.3 describe the learning setups including a comparison of different algorithms and calibrations to be most suitable for the specifics of the NFC handshake protocol, while Sect. 5 describes the methods for conformance checking and the respective protocols' specifics in this regard. Section 6 shows real-world results, while Sect. 7 compares them to the works of others. Section 8, eventually, concludes the paper and gives and outlook on future work.

1.1 Motivation

Both the NFC and BLE protocol are widely adopted protocols in a broad variety of different systems. NFC is used in often security-critical chip systems like banking cards, passports, access systems, etc. BLE is used in potentially privacy- and security-sensitive applications like healthcare, fitness, audio, and car access systems.

While there are many works about security weaknesses in NFC (e.g., [24, 49]), also specifically regarding the ISO/IEC 14443-3 handshake (e.g., [15, 29]), and BLE (e.g., [4]), there are few works on comprehensive testing (see Sect. 7). Assuring the correctness of the system is a principal step in the quest to trustworthy systems. As a specific application, we aim for a strong verdict of ISO compliance for NFC systems. There is, to the best of our knowledge, no comprehensive work regarding assessment of the handshake protocols, which is the fundament of secure protocols built atop. To make this verdict more scalable than manual modeling, yet strongly verified, we choose automata learning to automatically infer a formal model of the implementations under scrutiny. For the actual compliance checking, we use bisimulation and trace equivalence checks against a specification automaton from the ISO/IEC 14443-3 standard (a rationale is given in Sect. 2.2).

1.2 Contribution

Overall, this paper is on the interface between communications protocols, embedded systems, and formal methods. This work provides the following contributions for people with scholarly or applied interest in this approach of compliance checking:

- Insights regarding the specifics of learning an NFC implementation using (active) automata learning,
- An evaluation on the performance of different learning algorithms in systems containing two structures that are very similar to each other,
- Developing an NFC interface for a learning system,
- Utilizing bisimulation and trace equivalence in a combined approach for automated compliance checking,
- A novel approach for learning compound automata of implementations of multiple protocols,
- A method to compare this compound automaton with the individually protocol implementations using different methods, uncovering any potential cross-influences of multiple protocols running on the same device.

We saw the NFC handshake to be specific in two aspects: a) it consists of two parts that are very similar and hard to distinguish for Learners and b) the vast majority of outputs from a SUL are timeouts. This has severe impact on the learning where we examined different algorithms and configurations. The maximum word length has an impact on correctly inferring an automaton: too short yields incomplete automata, too long seemed to have a negative performance impact. Surprisingly the L* algorithm [3] with Rivest/Schapire (LSR) closure [41] surpassed more modern ones in learning performance. For discovering deviations from the standard, the minimum word length was found to have an impact. Here, the TTT algorithm [22] performed best, also followed by LSR. We further created a concrete hardware/software interface using a Proxmark device and an abstraction layer for NFC systems. We also integrated bisimulation and trace equivalence checking into the learning tool chain, which enables completely automated compliance checking with counterexamples in the case of deviations from the standard. Lastly, we developed an approach to learn a compound automaton of multiple protocols using a specialized SUL class working as a dispatcher. We subsequently use similar equivalence checking techniques to uncover any interferences among these protocols by comparing the compound automaton with standalone automata of the protocols (see Sect. 5.2).

2 Preliminaries

This section outlines the theoretical fundamentals of state machines and automata learning in the context of this paper and describes the used framework and the basics and characteristics of the scrutinized protocol.

2.1 State machines

A state machine (or automaton) is a fundamental concept in computer science. One of the most widely used flavors of state machines are Mealy machines, which describe a system as a set of states and functions of resulting state changes (transitions) and outputs for a given input in a certain state [33]. More formally, a Mealy machine can be defined as $M = (Q, \Sigma, \Omega, \delta, \lambda, q_0)$, with Q being the set of states, Σ the

input alphabet,[1] Ω the output alphabet (that may or may not be identical to the input alphabet), δ the transition function ($\delta : Q \times \Sigma \rightarrow Q$), λ the output function ($\lambda : Q \times \Sigma \rightarrow \Omega$), and q_0 the initial state. The transition and output functions might be merged ($\delta \times \lambda : Q \times \Sigma \rightarrow Q \times \Omega$). An even simpler type of automaton is a deterministic finite acceptor (DFA) [32]. It lacks of an output (i.e., no Ω and no λ), but instead it has a set of accepted finishing states F, which are deemed as valid final states for an input word (i.e., sequence of input symbols), resulting in a definition of $D = (Q, \Sigma, \delta, q_0, F)$. The purpose is to define an automaton that is capable of deciding if an input word is a valid part of a language. A special subset are combination lock automata (with the same properties) but the additional constraint that an invalid symbol in an input sequence would set the state machine immediately back into the initial state [35].

2.2 Transitions, equivalence, and preorder

An element of the combined transition/output function can be defined as 4-tuple ($\langle p, q, \sigma, \omega \rangle$) with $p \in Q$ as origin state of the transition, $q \in Q$ as destination state, $\sigma \in \Sigma$ as input symbol, and $\omega \in \Omega$ as output symbol. Generally, to conform to a standard, a system must display the behavior defined in that standard. The ISO 14443-3 standard [20] describes the states of the NFC handshake with their respective expected input and result. That means one can derive an automaton from this specification. The problem of determining NFC standard compliance can therefore be seen as comparing two (finite) automata. There is a spectrum of equivalences between Labeled Transition Systems (LTS) including automata. For being compliant with a standard, not necessarily every state and transition must be identical as long as the behavior of the system is the same. There might be learned automata that deviate from the standard automaton and still be compliant, e.g., if they are not minimal (the smallest automaton to implement a desired behavior). There are some efficient algorithms for automata minimization, e.g., by Hopcroft [18], and by Paige and Tarjan [37]. Figure 1 shows a very simple example of a three-state automaton and its behavior-equivalent (minimal) two-state counterpart.

To compare this type of equivalence between two LTS, LTS_1 and LTS_2, usually (various degrees of) simulation, bisimulation (noted as $LTS_1 \sim LTS_2$), and trace equivalence are used. Simulation (as used in the simulation preorder, see below) means that one automaton can completely reproduce the behavior of the other; for the bisimulation, this relation becomes bidirectional (i.e., functional). Because the states have to be directly related, bisimulation is a stronger relation than mutual simulation. Even if it is not common, there are cases where two LTS can simulate each other, but are still not bisimilar [42]. Trace equivalence compares the respective input/output sequences of automata (see below). Just unidirectional simulation alone is not sufficient, as this would only indicate the presence *or* absence of a certain behavior with respect to the specification, while the standard compliance mandates both. Bisimilarity of two transition systems is originally defined for labeled transition systems (LTS), defined as $LTS = (Q, Act, \rightarrow, I, AP, L)$, with Q being the set of states, Act a set of actions, $\rightarrow$ a transition function, I the set of initial states ($I \subseteq Q$), AP a set of atomic propositions, and L a labeling function.

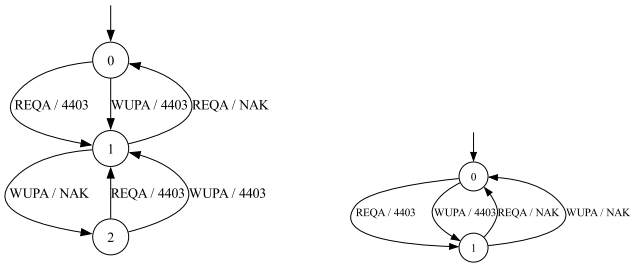

Fig. 1 Example for a partial automaton and its minimal counterpart

Definition 1 (Bisimilarity)
Bisimilarity of two LTS ($LTS_1 \sim LTS_2$) is defined as exhibiting a binary relation $R \subseteq Q \times Q$ such that [5]:

A) $\forall s_1 \in I_1 \exists s_2 \in I_2 \cdot (s_1, s_2) \in R$ and $\forall s_2 \in I_2 (\exists s_1 \in I_1 \cdot (s_1, s_2) \in R)$.
B) for all $(s_1, s_2) \in R$, there must hold
   1) $L_1(s_1) = L_2(s_2)$,
   2) if $s_1\prime \in Post(s_1)$ then there exists $s_2\prime \in Post(s_2)$ with $(s_1\prime, s_2\prime) \in R$,
   3) if $s_2\prime \in Post(s_2)$ then there exists $s_1\prime \in Post(s_1)$ with $(s_1\prime, s_2\prime) \in R$.

Condition A of Definition 1 means that all initial states must be related, while Condition B means that for all related states the labels must be equal (1) and their successor states must be related (2–3). Formally, the succession ($Post$) is defined as $Post(q, \alpha) = \{q\prime \in Q | q \xrightarrow{\alpha} q\prime\}$ and $Post(q) = \bigcup_{\alpha \in Act} Post(q, \alpha)$, meaning the union of all action successions, which again are again the result the transition function with a defined action and state as input [5]. As this is recursive, a relation of the initial states implies that all successor states are related. Since all reachable states are (direct or indirect) successor states of the initial states, this definition encompasses the complete LTS. We interpret Mealy machines as LTS using the output functions as labeling functions for transitions and the input symbols as actions, similar to [46]. Based on this, we define Mealy bisimilarity ($M_1 \sim M_2$) for our purpose as follows:

[1] It is common to use ε to denote empty sets in this context.

Definition 2 (Mealy bisimilarity)
Bisimilarity of two Mealy machines ($M_1 \sim M_2$) is defined as exhibiting a binary relation $R \subseteq Q \times Q$ such that

A) $q_{0_1} \in Q_1, q_{0_2} \in Q_2 \cdot (q_{0_1}, q_{0_2}) \in R$,
B) for all $q_1 \in Q_1, q_2 \in Q_2 \cdot (q_1, q_2) \in R$, there must hold
   1) $\sigma \in \Sigma \cdot \lambda_1(q_1, \sigma) = \lambda_2(q_2, \sigma)$,
   2) $\sigma \in \Sigma \cdot$ if $q_1\prime = Post(q_1, \sigma)$ then there exists $q_2\prime = Post(q_2, \sigma)$ with $(q_1\prime, q_2\prime) \in R$,
   3) $\sigma \in \Sigma \cdot$ if $q_2\prime = Post(q_2, \sigma)$ then there exists $q_1\prime = Post(q_1, \sigma)$ with $(q_1\prime, q_2\prime) \in R$.

As the transition function is dependent on the input, we define $Post(q, \sigma) = \delta(q, \sigma)$, which is essentially the same as for LTS brought into the notation of Sect. 2.1, with the constraint that the same input on a pair of related states must lead to another pair of related states. There are some different bisimulation types that differentiate by the handling of unobservable (internal) transitions (ordinarily labeled as τ transitions), e.g., strong and weak bisimulation, and branching bisimulation to give a few examples [5]. This distinction is, however, theoretical in the context of this paper. The reason is that we intend to compare a specification, which consists of an automaton that does not contain any τ transitions, with an implementation that is externally (black box) learned, rendering τs unobservable. Therefore, two automata without any τs are compared directly, which makes this distinction not applicable. More precisely, from a device perspective, the type of bisimulation equivalence cannot be determined, as the SULs are black boxes, which means that internal state changes (commonly denoted as τ) are not visible. From a model perspective, the chosen comparison implies strong bisimulation, i.e., the initial state is related (formally, $q_{0_{M_l}} = q_{0_{M_s}}$) and all subsequent states are related as well (formally, $Q = Q_{M_l} = Q_{M_s}; n = |Q|; \forall n \in Q | q_{nM_l} = q_{n_{M_s}}$). We, however, use a τ function for hiding (see Sect. 2.3) to restrict the input alphabet of a compound-protocol automaton to one of its single component automata and compare it to a separately learned version (see Sect. 5.2) with weak bisimulation.

Trace equivalence, on the other hand, means that two transitions systems produce the same traces for each same input.

Definition 3 (Trace equivalence)
Trace equivalence of two LTS ($LTS_1 \sim LTS_2$) is defined as $Traces(LTS_1) = Traces(LTS_2)$.

In an LTS, a trace is a sequence of labels of a path (or path fragment) $\pi = q_0, q_1, \ldots \; | trace(\pi) = L(q_0), L(q_1), \ldots$ [5]. Since in a Mealy machine we use the output function as a labeling function, traces are sequences of outputs $trace(\pi) = \lambda(q_0, \sigma_0), \lambda(q_1, \sigma_1), \ldots$ As only the input and output are directly observable (not the states themselves), we use an input/output as a notion in the form $\langle \sigma_0/\omega_0 \rangle, \langle \sigma_1/\omega_1 \rangle, \ldots$, with $\sigma \in \Sigma$ and $\omega \in \Omega$.

Both bisimulation and trace equivalence might be in principal capable of comparing a specification with an implementation automaton for determining the standard compliance. Both determining bisimulation and trace equivalence are problems to be solved efficiently [2, 19, 37]. In any case, bisimulation implies trace equivalence ($LTS_1 \sim LTS_2 \implies Traces(LTS_1) = Traces(LTS_2)$), but is finer than the latter [5]. For the purpose of this paper, we consider two automata equivalent if they are trace *or* bisimulation equivalent. In practice, we have obtained positive results with both bisimulation and trace equivalence (see Sect. 5.1).

Simulation preorder means that only one system has to be able to simulate the other. Informally, this means that the behavior of a system LTS_1 has to be included in another system LTS_2, but the latter might display additional behavior not included in the former. Formally, it is defined as follows.

Definition 4 (Simulation preorder)
Simulation preorder of two LTS ($LTS_1 \preceq LTS_2$) is defined as exhibiting a binary relation $R \subseteq Q \times Q$ such that [5]:

A) $\forall s_1 \in I_1 \cdot (\exists s_2 \in I_2 \cdot (s_1, s_2) \in R)$,
B) for all $(s_1, s_2) \in R$, there must hold
   1) $L_1(s_1) = L_2(s_2)$,
   2) if $s_1\prime \in Post(s_1)$ then there exists $s_2\prime \in Post(s_2)$ with $(s_1\prime, s_2\prime) \in R$.

This basically means a not (necessarily) symmetric bisimulation relation. Analogously, this means for the trace preorder that the set of traces of LTS_1 has to be included in that of LTS_2, which might or might not contain additional traces.

Definition 5 (Trace preorder)
Trace preorder of two LTS is defined as $Traces(LTS_1) \subseteq Traces(LTS_2)$.

2.3 Hiding operation

We can use hiding to restrict a compound-protocol automaton to transitions from a certain part of its input alphabet (particularly, one of the protocol's input alphabet). Groote and Mousavi [14] define a hiding operator (τ_I) that removes any actions in certain set of actions (I) and labels them as unobservable (τ) transitions. For Mealy machines, we use input symbols as (part of) label sets for the hiding operator. This alters the combined output and transition function as follows:

$$I \subseteq \Sigma | \tau_I(\delta \times \lambda) : Q \times \Sigma \begin{cases} Q \times \Omega & \text{if } \sigma \in \Sigma \setminus I, \\ Q \times \tau & \text{if } \sigma \in I. \end{cases}$$

The result is a *subtraction* of the automata parts included in the set I from the observable behavior, which we can use for comparison with weak bisimulation. This is also similar to the *restriction* operator from Hoare ($\restriction$), only that this operator is defined for traces where parts of a trace included in a given subset are ignored [16].

2.4 Automata learning

The classical method of actively learning automata of systems, was outlined in Angluin's pivotal work known as the L* algorithm [3]. This work uses a *minimally adequate Teacher* that has (theoretically) perfect knowledge of the SUL behind a *Teacher* and is allowed to answer to kinds of questions:

– *Membership queries* and
– *Equivalence queries*.

The membership queries are used to determine if a certain word is part of the accepted language of the automaton, or, in the case of Mealy machines, which output word will result of a specific input word. These words are noted in an observation table that will be made *closed* and *consistent*. The observation table consists of suffix-closed columns (E) and prefix-closed rows. The rows are intersected in short prefixes (S) and long prefixes ($S.\Sigma$). The short prefixes initially only contain the empty prefix (λ), while the long ones and the columns contain the members of the input alphabet. The table is filled with the respective outputs of prefixes concatenated with suffixes ($S.E$ or $S.\Sigma.E$). The table closed if for every long prefix row, there is a short prefix row with the same content ($\forall s.\sigma \in S.\Sigma \exists s \in S : s.\sigma = s$). The table is consistent if for any two equal short prefix rows, the long prefix rows beginning with these short prefixes are also equal ($\forall s, s\prime \in S \forall a \in \Sigma : s = s\prime \rightarrow s.a = s\prime.a$). A complete, closed, and consistent table can be used to infer a state machine (set of states Q consists of all *distinct* short prefixes, the transition function is derived by following the suffixes). Even though this algorithm was initially defined for DFAs, it has been adapted to other types of state machines (e.g., Mealy or Moore machines) [25]. Alternatively, some algorithms use a discrimination tree that uses inputs as intermediate nodes, states as leaf nodes, and outputs as branch labels, with a similar method of inferring an automaton. One of these algorithms, TTT [22], is deemed currently the most efficient [47]. Other widely used algorithms include a modified version of the original L* with a counterexample handling strategy by Rivest and Schapire [41], or the tree-based Direct Hypothesis Construction (DHC) [34] and Kearns–Vazirani (KV) [28] algorithms.

Once this is performed, the resulting automaton is presented to the Teacher, which is called equivalence query. The Teacher either acknowledges the correctness of the automaton or provides a counterexample. The latter is incorporated into the observation table or discrimination tree and the learning steps described above are repeated until the model is correct. To allow for learning black box systems, the equivalence queries in practice often consist of a sufficient set of conformance tests instead of a Teacher with perfect knowledge [38]. Originally for Deterministic Finite Automata, this learning method could be used to learn Mealy Machines [44]. This preferred for learning black box reactive systems (e.g., cyber-physical systems), as modeling these as Mealy is comparatively simple.

2.5 Abstraction

Ordinarily, when applying automata learning to real-world systems, the input and output spaces are very large. To reduce the alphabets' cardinalities to a manageable amount, an abstraction function (∇) is employed, that transforms the concrete inputs (I) and outputs (O) to symbolic alphabets (Σ and Ω) using equivalence classes. Of all possible combinations of data to be send, we therefore concentrate on relevant input for the purpose of compliance verification. In the following, we present some rationale for the chosen degree of abstraction through the input and output alphabets. These alphabets' symbols are abstracted and concretized via an according adapter class that translates symbols to data to be send (see Sect. 3.1.2).

2.6 LearnLib

To utilize automata learning, we use a widely adopted Java library called LearnLib [23]. This library provides a variety of learning algorithms (L* and variants thereof, KV, DHC, and TTT), as well as various strategies for membership and equivalence testing (e.g., conformance testing like random words, random walk, etc.). The library provides Java classes for instantiating these algorithms and interfaces SULs. The interface classes further allow for defining the input alphabets that the algorithm routines use to factor queries used to fill an observation table or tree. Depending on the used algorithms, the library is capable of inferring DFAs, NFAs (Nondeterministic Finite Acceptors), Mealy machines or VPDAs (Visibly Pushdown Automata).

2.7 Near field communication

Near Field Communication (NFC) is a standard for simple wireless communication between close coupled devices with relatively low data rates (106, 212, and 424 kbit/s). One distinctive characteristic of this standard (operating at 13.56 MHz center frequency) is that it, based on Radio-Frequency Identification (RFID), uses passive devices (proximity cards, PICCs) that receive power from an induction field from an active device (reader or proximity coupling

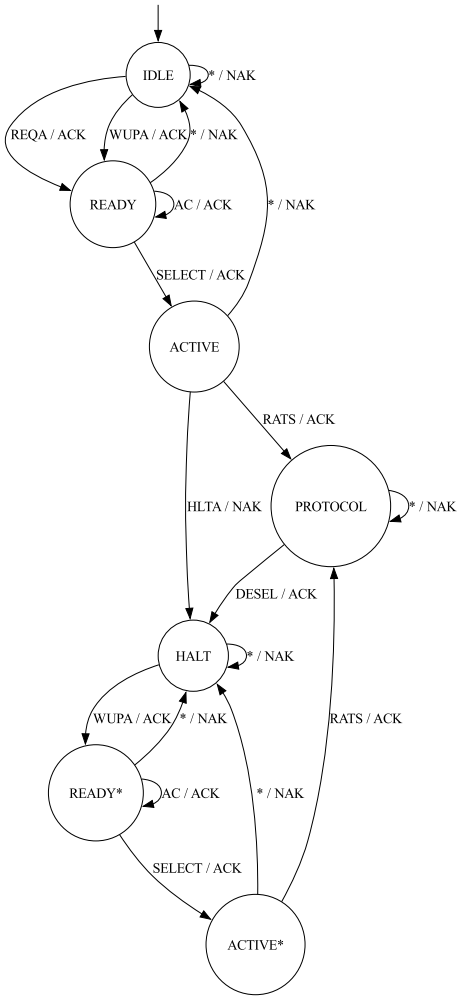

Fig. 2 NFC handshake automaton after ISO 14443-3 [20] augmented with abstract outputs. Note that star (*) as input means any symbol that is not explicitly stated in another outbound transition of the respective state

device PCD) that also serves as a field for data transmission. There are a couple of defined procedures that allow for operating proximity cards in the presence of other wireless objects in order to exchange data [21]. One standard particularly defines two handshake procedures based on cascade-based anticollision and card selection (called type A and type B), one of which NFC proximity cards must be compliant with [20]. This handshake is the particular target SUL of this paper, with the purpose of providing very strong evidence for compliance. Due to the proliferation and the nature of the given SUL, this paper concentrates on type A devices. Therefore, all statements on NFC and its handshake apply for type A only.

2.8 The NFC handshake automaton

ISO 14443-3 contains a state diagram that outlines the Type A handshake procedure for an NFC connection (see Fig. 2). This diagram is not a state machine of the types described in Sect. 2.1, for it lacks both output and final states. As we learn Mealy machines, we augmented it with abstract outputs (see Sects. 4.1.2 and 5.1) to get a machine of the same type. The goal of the handshake is to reach a defined state in which a higher layer protocol (e.g., as defined in ISO 14443-4 [21]) can be executed (the *PROTOCOL* state). The intended way described in the standard to reach this state is: when coming into an induction field and powering up, the passive NFC device enters the *IDLE* state. After receiving a wake-up (*WUPA*) or request (*REQA*) message, it enters the *READY* state. In this state, anti-collision (*AC*, remaining in that state) or card selection (*SELECT* going to the *ACTIVE* state) occurs. In the latter state, the card waits for a request to answer-to-select (*RATS*), which brings it into said *PROTOCOL* state. In all of these states, an unexpected input would return the system to the *IDLE* state, giving no answers (denoted as *NAK*). Based solely on ISO 14443-3 commands, the card should only leave this state after a *DESELECT* command, after which it enters the *HALT* state. Apart from a complete reset, it only leaves the *HALT* state after a wake-up (*WUPA*) signal (in contrast to the initial *IDLE* state, which also allows an *REQA* message). This brings it into the *READY** state, which again gets via a *SELECT* into the *ACTIVE** state that can be used to get to the *PROTOCOL* state again. The only difference between *READY* and *READY**, as well as *ACTIVE* and *ACTIVE**, states is that it comes from the *HALT* instead of *IDLE* state. Similar to the first part of the automaton, an unexpected answer brings the state back to *HALT* without an answer (*NAK*).

Apart from the commands stated above that are expected by a card in the respective state, every other (i.e., unexpected) command would reset the handshake if its not complete (i.e., wrong commands from *IDLE, READY,* and *ACTIVE* states would lead back to the *IDLE* state, while *HALT, READY**, and *ACTIVE** lead back to the *HALT* state, and unexpected commands in the *PROTOCOL* state let it remain in that state). Even though this behavior of falling back into a base state resembles a combination-lock automaton or generally an accepting automaton, we model the handshake as a Mealy machine for the following reasons:

a) As we observe a black box, input/output relations are easier to observe than not intrinsically defined accepting states.
b) The states are easier distinguishable: a variety of input symbols with the corresponding output may represent a broader signature than just if a state is accepting (apart from the transition to other states).
c) The output may processed at different level of abstraction (see Sect. 2.5).

There is also one specific feature to the NFC handshake protocol: unlike most communication protocols, an unexpected

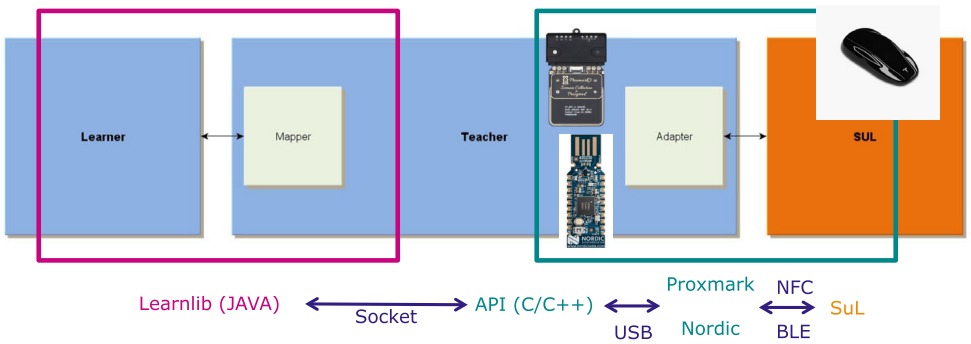

Fig. 3 Learner interface setup

or wrong input yields no output. This has an implication to learning, as a timeout will be interpreted as a general error message.

2.9 Bluetooth Low-Energy

Bluetooth Low-Energy (BLE) is a standard for mid-ranged wireless communications optimized for low power consumption (through long sleep phases and relatively small active periods). It operates at data rates up to 2 Mbps in the 2.4 GHz band (for devices supporting version > 5.0). It is divided in four layers: application, middleware, data link, and physical. In the data link and middleware layers, basic connection is used [6].

3 Learning interfaces

As a learner, we use the algorithm implementations in the Learnlib Java library (see Sect. 2.6). This section outlines the details for the setup for NFC, BLE, and compound learning, and the respective teacher and SUL interfaces.

3.1 NFC interface

The learner is configured as outlined in Sect. 4.1 using an adapter class (cf. Sect. 3.1.2). To interact with the NFC SUL, a Proxmark RFID/NFC device (see Sect. 3.1.1) is used that works with an adapter written in C++ (see Sect. 3.1.2). Figure 3 provides an overview of the setup.

3.1.1 Learner interface device

The interface with an NFC SUL is established via Proxmark3, which is a pocket-size NFC device capable of acting as an NFC reader (PCD) or tag (PICC), as well as sniffing device [13]. Proxmark3 can be controlled from a PC, as well as allowing firmware updates. Thus, it allows us to construct the NFC frames needed for learning and establishing a connection to the learning library via a software adapter (see Sect. 3.1.2).

3.1.2 Adapter class

The Java learner communicates with the SUL via a distinctive class that handles input, reset, etc. The actual access to the NFC interface runs over a C++ program, running on a PC, based on a provided application that comes with the Proxmark device. As this application is open source, it was possible to modify it in order to adapt it for learning. The main interface to the Java-based Learner is a Socket connection that take symbols from the Learner (see Sect. 2.5) and concretizes them by translating the symbols into valid NFC frames utilizing functions from the *SendCommand* and *WaitForResponse* families. These functions send and receive, respectively, command data (i.e., concrete inputs, symbol for symbol) to the Proxmark device where the firmware translates it into frames and sends them to the SUL, and proceeds vice versa for the response. This, however, turned out to create an error prone bottleneck at the connection between the PC application and the Proxmark device running over USB. Due to round-trip times and timeouts, the learning was slowed down and occasional nondeterministic behavior was introduced, which jeopardized the learning process and made it necessary to repeat the latter (depending on the scrutinized system, multiple times, which hindered the overall learning greatly). Therefore, the Learner was reimplemented to send bulk inputs (i.e., send complete input words instead of single symbols), which improved the throughput significantly and solved nondeterminism.

3.1.3 Firmware modifications

In order to be able to transfer traces wordwise instead of symbolwise, significant modifications of the device's firmware were necessary. The standard interface of the device is designed for sending a single packet at one time (via a provided application on a PC) and delivering the answer back to the application via a USB interface. This introduces latency, which through the sheer amount of symbols sent in the learning process, has a significant performance impact. To reach the device's firmware with multiple symbols at once, we

modulate the desired inputs into one sent message in Type–Length–Value (TLV) format (implemented types are with or without CRC and a specialized type for SELECT sequences) and modify the main routine of the running firmware to execute a custom function if a certain flag is set. This custom function deserializes the sent commands and sends them to the NFC SUL. Answers are modulated into an answer packet in length–value format, followed by subsequent answer messages containing precise logging and timestamps, if used. As NFC is a protocol that works with relatively low round-trip times and timeouts these modifications, eliminating a great portion of the latency times of frequently used USB connections, it boosted the performance of the learning using different learning algorithms significantly (for a performance evaluation, see Sect. 4.1.1).

3.2 BLE interface

For BLE, we use the same learner as for NFC (based on LearnLib), but with some distinct SUL classes for Bluetooth. Particularly, we developed distinct classes for Input and Output packets to configure the sent data [43]. We use an identical socket construction like in the NFC interface to communicate with the adapter class that handles the actual BLE connection (see Fig. 3).

3.2.1 Learner interface device

For learning BLE systems, we used a Nordic nRF52840, which is a multiprotocol SoC that supports most of the defined hardware functionalities in the Bluetooth Core Specifications 5.3 [40]. It comes with the ability to update the and upload custom firmware. This allows for making the necessary modifications for BLE learning, including the ability to craft link layer packets, which is not possible on off-the-shelf adapters.

3.2.2 Adapter class

The BLE adapter is written in Python 3.10 and makes use of a modified version of Scapy 2.4.5.[2] We use Scapy to generate packets and to parse packets received by the firmware. It supports most of the specified packets by default, but was missing some of the newer packets and some fields were not updated with the changes of Bluetooth 5.0 [6].

The communication with the firmware is split into two kinds of message:

– Packet transmitted or received
– Commands to change connection parameter

[2] https://scapy.net/.

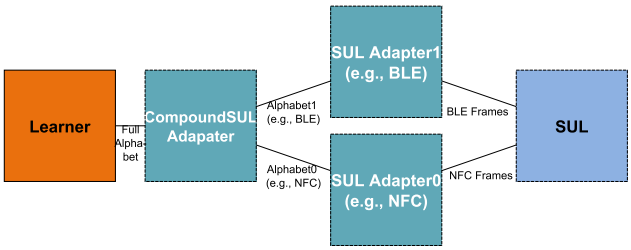

Fig. 4 Compound protocol interface setup. Amber is the Learner class, turquoise the SUL adapter classes, and blue the actual SUL

The Interface will communicate with the firmware in case packets are sent and received, or if the Interface wants to change some aspects of the firmware, like physical connection parameters of the BLE link. The communication with the learner works in a similar manner. The Learner will query the Interface with a trace of packets. In this setting, generating packets that depend on previously transmitted packets is challenging, as the trace generated by the learner may not include these prerequisite packets. To address this issue, we decided that the default value of a field is either the lowest value in a range or zero. This decision has been made to improve predictability and prevent nondeterminism. The learning cache uses a prefix tree (trie) data structure for caching and error correction. All the queries to the system will be saved in this trie. This helps prevent requiring the system with duplicate traces for the equivalence oracle and allows the system to resume learning.

3.2.3 Firmware modifications

In BLE, the controller and link layer is separated from the host via a Host Controller Interface (HCI) from the Host System. The controller is closed source. An of-the-shelf BLE controller would restrict the packets we are able to send and remove the ability to introduce faulty packets. Therefore we needed a custom controller and Host Controller Interface (HCI) that does not have these limitations. Based on the Firmware for SweynTooth [12], we reimplemented the BLE controller to have an open access to all link layer functions necessary for proper learning.

3.3 Compound protocol interface

To learn multiple protocols at once, we created an adapter that can act as a SUL interface to the learner and serves as a dispatcher to actual SUL classes for different single-protocol SULs. It receives the respective input alphabet and labels each – when receiving an input symbol it dispatches it to the respective protocol SUL (see Fig. 4).

Table 1 Runtime (minutes) per algorithm and maximum word length

Max. word length	Algorithm						
	L*-C	L*-RS	DHC	KV-L	KV-B	TTT-L	TTT-B
10	5.92	**5.05**	6.00	4.38	4.38	5.45	5.37
20	**20.08**	9.34	10.93	12.24	11.65	7.66	7.40
30	41.90	12.92	**9.82**	**12.19**	**11.47**	**10.67**	10.04
40	68.17	8.54	11.16	15.56	12.89	10.87	**9.49**
50	34.75	7.87	11.02	15.60	12.53	11.29	9.91
60	77.33	17.15	12.98	17.16	13.37	13.04	10.85
70	134.65	11.34	14.46	17.68	14.81	13.06	11.32

3.3.1 Adapter class

The class caring about addressing multiple SULs (*compoundSUL*) at once is actually a container. It provides the same functionalities as other SUL classes, but can have other SULs as child SULs. Technically, each added SUL is equipped with an identifier (sequential number) as a prefix and its input alphabet is added to that of the compoundSUL (more formally, $I_{Mult} = \bigcup_{j \in n} I_j$ with j being identifiers of child SULs and $I_{Mult} = \varepsilon$ for a childless compoundSUL). Additionally, the identifier of a child SUL is added as a prefix to each element of its input alphabet upon adding it to the compoundSUL alphabet ($i \in I_j \mid i = j\_i$). The learner uses the joint alphabet including the prefixes as input symbols (as the compoundSUL presents the full joint alphabet as possible input). When receiving an input trace from the learner, the compoundSUL sequentially hands each input symbol to the child SUL with the respective index and removes the prefix (sending each consecutive sequence of inputs for the same child SUL as a bulk sequence) and queues the output. When the complete trace has been processed by the child SULs, it hands the concatenated output trace back to the learner. This makes the internal structure of the compoundSUL (or even the fact that it is a composite SUL) completely opaque to the learner.

4 Learning protocols

In this section we discuss the characteristics of learning communication protocols, particularly NFC and BLE, as well as of learning compound-protocol automata.

4.1 Learning NFC

One distinctive attribute of ISO14443-3 with respect to learning is that it specifies to not give an answer on unexpected (i.e., not according to the standards specification) input. Ordinarily, the result of such a undefined input is to drop back to a defined (specifically, the IDLE or HALT) state. In this sense, the NFC handshake resembles a combination lock automaton. A positive output, on the other hand, ordinarily consists of a standardized status code or information that is needed for the next phase of the handshake, e.g., parts of a card's unique identifier (UID). The non-answer to undefined input is a characteristic feature of the NFC standard. This directly affects the learning because it yields many identical answers and efficient timeout handling is essential. It is therefore necessary to evaluate different state-of-the-art learning algorithms for their specific fitness (see Sect. 4.1.1), as well as determining the optimal parameter set (Sect. 4.1.1). We scrutinize the main algorithms supported by Learnlib: classical L*, L* with Rivest/Schapire counterexample handling, DHC, KV and TTT – the latter two with linear search (L) and binary search (B) counterexample analysis.

4.1.1 Comparing learning algorithms and calibrations

All of the algorithms can be parameterized regarding the membership and equivalence queries. The former are mainly defined via the minimum and maximum word length, while the equivalence queries (lack of a *perfect Teacher*) are determined by the method and number of conformance tests. Generally speaking, a too short (maximum) word length results in an incomplete learning (which, if the implementation is correct, should contain seven states). The maximum length, however, has a different impact on the performance for observation and tree-based algorithms: table-based are quicker with a short maximum word length, whereas for tree-based ones there seems to be a break-even point between many sent words and many sent symbols in our specific setting. Table 1 shows a comparison of the runtime of different algorithms with different maximum word lengths (in bold the respective algorithm's shortest runtime that learned the correct 7-state model). Some of the nonsteadiness in the results can be explained by the fact that some calibrations with shorter word lengths required more equivalence queries and, thus, refinement procedures. Table 2 shows the results with the best performing (correct) run of the respective algorithm. This, however, only covers the performance of learning a correct

Table 2 Performance evaluation of different algorithms for a compliant system with their respective fastest calibration in the given setting

Algorithm	L*-C (20)	L*-RS (10)	DHC (30)	KV-L (30)	KV-B (30)	TTT-L (30)	TTT-B (40)
States	7	7	7	7	7	7	7
Runtime (min)	20.08	5.05	9.82	12.19	11.47	10.67	9.49
Words	1137	282	539	496	451	468	382
Symbols	10,192	2588	5124	7932	7607	6628	6213
EQs	2	3	2	5	5	4	4

Table 3 Performance evaluation of different algorithms for a noncompliant system with their respective fastest calibration in the given setting

Algorithm	L*-RS	DHC	KV-L	KV-B	TTT-L	TTT-B
Min Length	20	20	10	10	10	10
Runtime (min)	309.81	328.83	520.34	423.27	277.67	131.43
Words	575	855	952	679	688	616
Symbols	14,637	15,262	23,867	19,241	13,353	11,769
Eqs	5	3	6	6	5	5

implementation. The opposite side, discovering a bug, shows a different picture. We therefore used a SUL with a slightly deviating behavior (see Sect. 6.2). This system is much more error-prone, needing significantly higher timeout values, resulting in higher overall runtimes. One key property in this case seems to be the minimum word length. Some of the algorithms require a lower minimum word length to discover than others. This has a significant impact with the special setting of getting relatively many timeouts, which is greatly aggravated by the necessary long timeout periods. With a minimum word length of 10 symbols, again the original L* with the Rivest/Schapire closing strategy was performing quickest, but discovered only 7 out of 10 states of the deviating implementation. DHC yielded a similar result. Both needed a word length of 20 to discover the actual uncompliant model, which was significantly less efficient in terms of runtime. The TTT and KV algorithms needed a minimum length of 10, however, with quite some deviation in efficiency. While TTT was the best performing algorithm to learn the SUL's actual behavior model, KV was performing the worst. The runtimes roughly correspond with the amount of sent symbols, in this case a very long timeout has to be set to avoid nondeterminism. The classical L* is not in the list, as the algorithm crashed after more than 24 hours of runtime. Table 3 provides an overview of minimum word lengths, runtime, words, symbols, and equivalence queries. Lower minimum word lengths yielded false negatives (i.e., the result showed a correct model with the deviation not discovered).

4.1.2 Input and output alphabets

For the input alphabet, we use that needed for successfully establishing a handshake (cf. Fig. 2), according to the state diagram for Type-A cards in the ISO 14443-3 standard [20]:

- Wake-UP command, Type A (WUPA)
- Request command, Type A (REQA)
- Anticollision (AC)
- Select command, Type A (SELECT)
- Halt command, Type A (HLTA)
- Request for answer to select (RATS)
- Deselect (DESEL)

The last two commands are actually defined in the ISO 14443-4 standard [21]. However, as the handshake's purpose is to enter and leave the protocol state, they are included in the 14443-3 state diagram and, consequentially, in our compliance verification.

In general, the output alphabet does not need to be defined beforehand. It simply consists of all output symbols observed by the Learner in a learning run. The Learner can derive the output alphabet implicitly. This means that if a system behaved nondeterministically, the output alphabet could vary – although when learning Mealy machines, which are deterministic by definition, nondeterminism would jeopardize the Learner. The output alphabet has obviously to be defined (in the abstraction layer) when abstracting the output. Therefore, using raw output has the benefit of not having to define the alphabet beforehand. The raw method has one drawback: there are cards that use a random UID (specifically, this behavior was observed in passports). Every anticollision (*AC*) and *SELECT* yields a different output, which introduces nondeterministic behavior. This is not a problem with abstract

output, as the concrete answer is abstracted away. We therefore tried a heavily abstracted output consisting of only two symbols, namely *ACK* for a (positive) answer and *NAK* for a timeout, which in this case means a negative answer (see Sect. 2.7). This solves the problem, but degrades the performance of the Learner, since states are harder to distinguish if the possible outputs are limited to two (aggravated by the similar behavior of certain states – see Sect. 2.8). This idea was therefore forfeited in favor of raw output for the learning. We still maintained this higher abstraction for the equivalence checking (see Sect. 5.1 for the reasoning). Raw output, however, retains this problematic nondeterminism. We therefore introduce a caching strategy to cope with this issue. Whenever a valid (partial) UID is received as an answer to an anticollision or select input symbol, we put it on one of two caches (one for partial UIDs from *AC* and one for full ones from *SELECT* sequences). The Learner will subsequently only be confronted with the respective top entries of these caches. We therefore abstract away the randomness of the UID by replacing it with an actual but fixed one. This keeps the learning deterministic while saving the other learned UIDs for analysis, if needed.

4.1.3 Labeling and simplification

An implementation that conforms to the standard will automatically labeled correctly, as the labeling function follows a standards-conform handshake trace:

a) label the initial state with *IDLE*,
b) from that point, find the state, where the transition with *REQA* as an input and a positive acknowledgment as an output ends and label it as *READY*,
c) from that point, find the endpoint of a positively acknowledged *SELECT* transition and label it as *ACTIVE*,
d) from that point, find the endpoint of a positively acknowledged *RATS* transition and label it as *PROTOCOL*,
e) from that point, find the endpoint of a positively acknowledged *DESELECT* transition and label it as *HALT,*
f) from that point, find the endpoint of a positively acknowledged *WUPA* transition and label it as *READY**,
g) from that point, find the endpoint of a positively acknowledged *SELECT* transition and label it as *ACTIVE**.

If the labeling algorithm fails or there are additional states (which are out of the labeling algorithm's scope), this is an indicator for the learned implementation's noncompliance with the ISO 14443-3 standard (given that only the messages defined in that standard are used as an input alphabet – see Sect. 4.1.2).

To simplify the state diagram for better readability and analysis, we cluster the transitions of each states for output/target tuples and label the input for that mostly traveled tuple with a star (∗). Normally, that is the group of transitions that mark an unexpected input and transitions back to the IDLE or HALT state. This reduces the diagram significantly. Therefore, in those simplified diagrams, all inputs not marked explicitly in a state can be subsumed under the respective star (∗) transition.

4.2 Learning BLE

Like many network protocols, but in contrast to NFC, BLE is time sensitive with regard to answers to requests within a reasonable time frame. In practice, we observed the importance of timing within BLE through the necessity of swift message processing to generate inputs quick enough for learning before timeouts occur. In order to yield good results, many optimizations were necessary that front-load operations from the learner into the adapter including caching functionalities to achieve the speed needed for proper communications. Tests with this optimized BLE adapter yielded a model with 33 states – but only four output symbols (*LL_VERSION_IND, LL_UNKNOWN_RSP(code=12), ATT_Exchange_MTU_Response,* and *ATT_Error_Response*). The relatively high number of states (given such few input symbols) can be explained by a narrow margin of time windows to hit. This results in different states to be detected upon timeout occurrences. When using the BLE learner without optimizations (which is a necessity for compound-protocol learning), the automaton becomes much simpler (three states). With this configuration, only two packets (maximum, for *ATT_Exchange_MTU_REQ* only one) can be sent in time before the connection times out and we end in a unresponsive sink state (while such restrictions do not apply for the optimized version). This means that we get only one non-timed-out answer per trace, greatly simplifying the automaton (see Fig. 6). This observation is specific to the used SUL, namely a Tesla key fob (see Sect. 6.2). The optimized learner, configured with more input symbols, yielded different results with other systems [11], which also applies to an updated version of the compound learner (see Sect. 6.2.1).

4.2.1 Input and output alphabets

To both reduce the resulting model's complexity and the effort of learning, we do not use connection requests as a dedicated input, but as part of the reset procedure. Therefore, our learner is intrinsically built to connect with the SUL before using any explicit input symbols. Also, for the sake of simplicity, we use a small set of low-level messages that are used to establish BLE connections. This reduced set consists of

– LL_VERSION_IND
– LL_FEATURE_RSP and

– ATT_EXCHANGE_MTU_REQ

Like in NFC, the output consists on the system's reaction to the input symbols. We therefore do not need to explicitly define an output alphabet. This is arguably only a small subset of the outputs defined in the standard. The reason is not only the small input alphabet, but also timing issues. Due to creating the necessary interlace between NFC and BLE symbols for learning compound-protocol automata, we were forced to deactivate some efficiency and caching procedures, which results in a lower overall performance. This is because the front-loaded optimizations directly in the adapter cannot be transferred to the learner, which would be necessary to allow for the intertwining of symbols from different protocols. For the sake of comparability, we used the low-performing methodology to learn the BLE-only automaton as well.

4.3 Learning compound-protocol automata

For compound protocols, we use a container SUL that deserializes and dispatches SULs to child SULs that handle the interface to the protocols present in the compound SUL (cf. Sect. 3.3 and Fig. 4). The actual protocol SULs are invisible to the learner.

4.3.1 Abstraction

The abstraction layer for compound SUL containers mainly consists of handling the dispatcher. The container can be equipped with one or more protocol SULs (which we do with NFC and BLE SULs – see Sect. 6.2.1). On adding a protocol SUL, its respective input alphabet will be added to the compound's (which is just ε for an empty compound SUL) with an identifier as a prefix. When the learner sends a trace, it will be deserialized and distributed to its child SULs according to the respective identifiers. The answers will be assembled again in the same sequence and sent back to learner as output word.

4.3.2 Input and output alphabets

We used the input alphabets for NFC and BLE described above. It was necessary to deactivate some performance features for each protocol to successfully intertwine the alphabets.

5 Equivalence checking

We mainly use the two mentioned methods of compliance checking (bisimulation and trace equivalence) to check the compliance of an implementation with a system specification. Additionally, for our proposed method of multiprotocol learning, we can also use the method set to detect deviations of a compound-protocol automaton from its separately learned single-protocol counterparts, indicating cross-protocol influences.

5.1 Compliance evaluation

Proving or disproving compliance needs a verdict if a potential deviation from the standard violates the (weak) bisimulation relation. We use mCRL2 with the Aldebaran (.aut) format for bisimilarity and trace equivalence checking (as described in Sect. 2.2) [9]. As the Learnlib toolset provides the possibility to store the learned automata in a couple of formats, including Aldebaran, setting up the tool chain is easy, even though some reengineering was necessary. Learnlib's standard function for exporting in the Aldebaran format does not include outputs. We therefore rewrote this function to use the transition's in the label of an LTS as well. mCRL2 comes with a model comparison tool that uses, among others, the algorithm of Jansen et al. [26] for bisimilarity checking. For NFC, we therefore simply model the specification in form of the handshake diagram (see Fig. 2) as an LTS with the corresponding Mealy's input and output as a label in the Aldebaran format and use the mCRL2 tool to compare it to automata of learned implementations.

5.1.1 NFC specifics

In NFC, the models of SULs could look very different, even if the behavior is equal. Due to different UIDs, the outputs to legit AC and SELECT commands would ordinarily differ between any two NFC cards. Also most other outputs might differ slightly. For example, we observed some cards to respond to select with *4800*, others with *4400*. We therefore use the higher abstraction level as described above and use only NAK and ACK as output, circumventing this problem. This way, inequalities as detected by the tool indicate noncompliance to the ISO 14443-3 standard of the scrutinized implementation. A trace of the noncompliant state/transition is trivial to extract from the automaton (see the example in Sect. 6.2). If that trace is executed on the SUL and actually behaves like predicted in the model, we have found the actual specification violation in the real system, disproving the compliance.

5.1.2 BLE specifics

One specific of the BLE handshake is that the first request always has to be a connection request in order to leave the initial state. To boost efficiency, we included this request in the reset procedure. Therefore, a connection request is implicit

in our model and included as a distinct input symbol. Also, like many other communication protocols but in contrast to NFC, timing has an influence on the states. Sessions time out after a specific period. The impact of this circumstance is that (with our learner's used timing) the SUL's state machine enters an unresponsive sink state after two to four sent messages, if the timeout is not reset through a message.

5.2 Compound protocol comparison

It is one of the main motivations of this paper to determine if different protocols running on the same device influence each other. To determine this, the described equivalence checking methods compare a compound-protocol automaton with the separately learned automata of each individual protocol in two different ways: a) based on preorder and b) based on hiding. The *preorder* method (see Sect. 2.2), checks for each of the single protocol automata if it is *included* in the compound automaton using simulation or trace preorder. Included in this context means that the including automaton can simulate the complete behavior of the included one, but not (necessarily) vice versa. The hiding method creates multiple automata (one for each protocol) from the compound one, by hiding (see Sect. 2.3) transitions from the alphabets of all other protocols (i.e., replacing them with τ transitions). It then checks the equivalence of each automaton generated this way with its separately learned single-protocol counterpart using weak trace equivalence or weak bisimulation. Each of the methods generates counterexamples on deviations. If counterexamples are found, they indicate that the joint behavior is different. This then suggests that an interference between these protocols on the device has occurred or an application on the examined systems uses both protocols in conjunction. However, the checks based on hiding are stronger – as preorder just checks if the compound automaton can simulate the single-protocol, it does not discover extra behavior. Using both methods in concatenation can automatically determine if a deviation indicates lack of or extra behavior: if the preorder check fails, the compound automaton does not fully simulate the single protocol; if it succeeds but the hiding-based (weak bisimulation or weak trace) equivalence checks fail, it provides extra behavior; if both succeed, the single protocol is exactly reproduced in the compound automaton.

6 Evaluation

In this section we briefly outline the achieved results with the described tool chain. We used several different NFC card systems for testing, which are described below. All of these systems have shown to be conform to the ISO14443-3 standard, except for the Tesla key fob. This key fob was also the main system examined using the dual learning approach, for it displays both protocols (NFC and BLE) which are used for its proper functioning: while open/close signals come via BLE, NFC is used as an out-of-band method during the pairing process for exchanging key material to secure the BLE connection.

6.1 Test cards, credit cards, and passports

We used five different NFC test cards by NXP (part of an experimental car access system) to develop and configure the Learner. Furthermore, we used two different banking cards, a Visa and a Mastercard debit. All of these cards conform to the standard, with only minor differences. One of these differences is replying with different ATQA to REQA/WUPA messages with 4400 and 4800, respectively. Overall, the results with these cards are very similar. Figure 5 shows an example of a learned automaton (left). We also examined two different passports from European Union countries: one German and one Austrian. The main noticeable difference (at ISO 14443-3 levlel) to other systems is that these systems answer to AC and SELECT inputs with randomly generated (parts of) UIDs. This implements a privacy feature to make passports less traceable. Without accessing the personal data stored on the device, the passport should not be attributable. This, however, requires authentication. We also scrutinized the upper-layer passport protocol in another publication [31].

6.2 Tesla key fob

Apart from significantly slower answers than for the other devices, which required adapting the timeouts to avoid nondeterministic behavior, the learned automaton slightly differs when learned with the TTT algorithm. Figure 5 (right) shows a model of a Tesla car key fob learned with TTT. The (unnamed) states 3, 4, and 6 are very similar to the HALT, READY*, and ACTIVE* states, respectively. Apart from the entry points (HALTA from the ACTIVE state for the first and DESEL from the PROTOCOL state, respectively), these two structures are identical and, in the reference model, those two transitions lead to the same state. However, the ACTIVE* transition allows for issuing a DESELECT command that actually returns a value (i.e., an ACK in the higher abstraction), which does not correspond to the standard. The mCRL2 comparison tool rightfully identifies this model not to be bisimilar and trace equivalent with the specification. Using the according option, the tool also provided a counterexample in the form of the trace (⟨*REQA/ACK*⟩, ⟨*SELECT/ACK*⟩, ⟨*RATS/ACK*⟩, ⟨*DESEL/ACK*⟩, ⟨*WUPA/ACK*⟩, ⟨*SELECT/ ACK*⟩, ⟨*DESEL /ACK*⟩). According to the specification, the last label should be ⟨DESEL/NAK⟩. We also learned a BLE automaton with a reduced set used for comparison with the compound-protocol

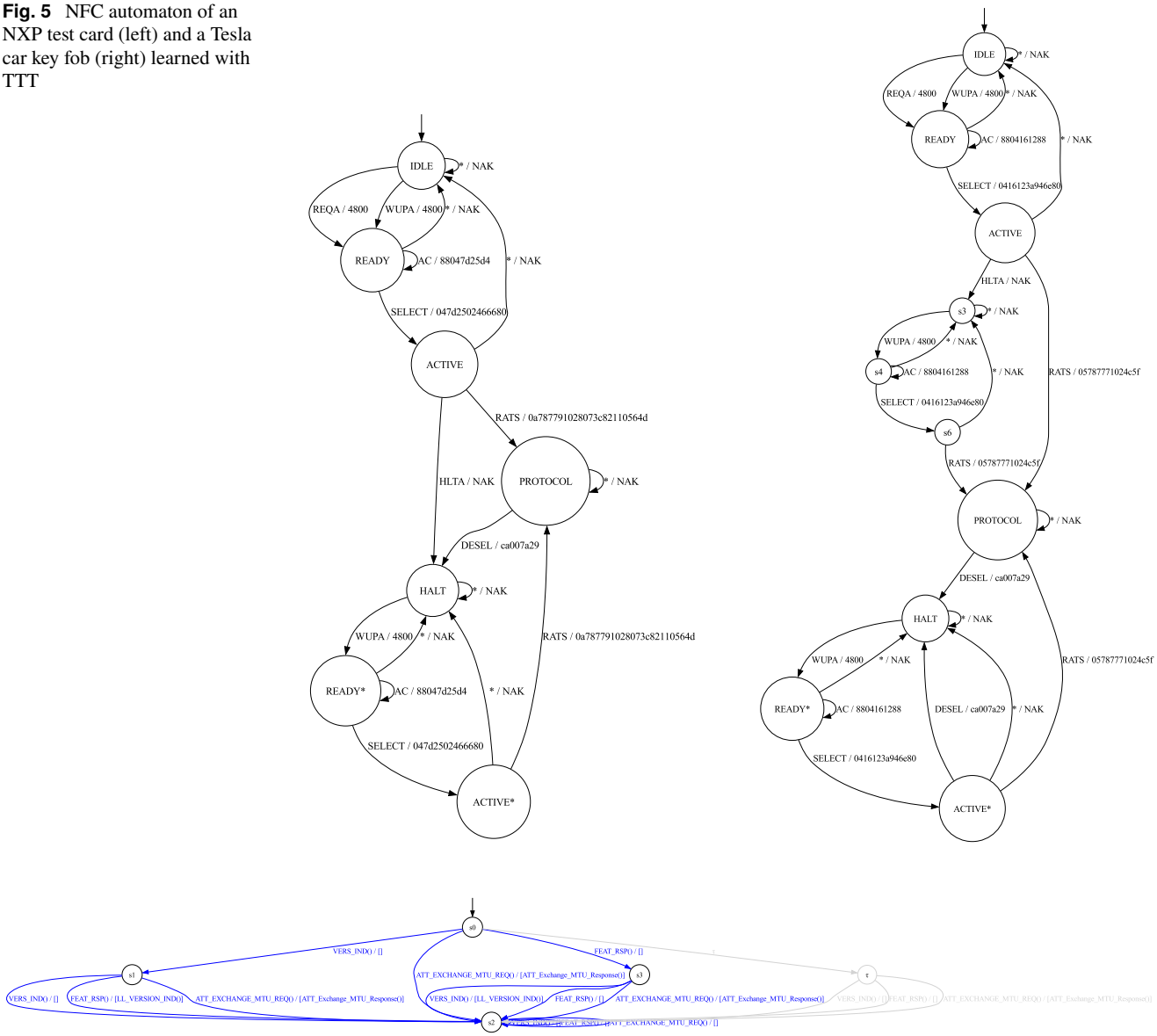

Fig. 6 Automaton of a Tesla key fob's BLE interface, learned with TTT without BLE optimizations (in blue), added (in light gray) is the additional state class when learning a compound model (see Sect. 6.2.2) (Color figure online) (Zoomable figure online)

automata (see also Sect. 4.2). Figure 6 shows this automaton, including the deviations from the compound automaton.

6.2.1 Compound protocol automaton

We also learned a compound automaton of the Tesla key fob (since it supports both NFC and BLE). We used the same input alphabets as for the NFC and BLE learning, respectively. The result is a 20-state automaton that combines both protocols (see Fig. 7). We subsequently used the equivalence checks described in Sect. 5.2. For NFC, we found some deviations with both checking methods because in the compound setting the learner failed to spot the noncompliance with the ISO/IEC standard (see above in Sect. 6.2). We subsequently compared it with the standard's specification automaton (cf. Sect. 5.1) and both methods rendered it to be included or (weakly) equivalent, respectively. An interesting detail is that in a failed learning attempt with an early error in BLE (trace #233, that produced a nondeterminism in trace #1391), the learning took a different path, which resulted in the discovery of the noncompliant transition. For BLE, the preorder checks were positive, meaning that the compound automaton includes the behavior of the stand-alone BLE automaton. However, the hiding-based method yielded that they are not equivalent. This means that there is extra BLE behavior by using NFC input in parallel.

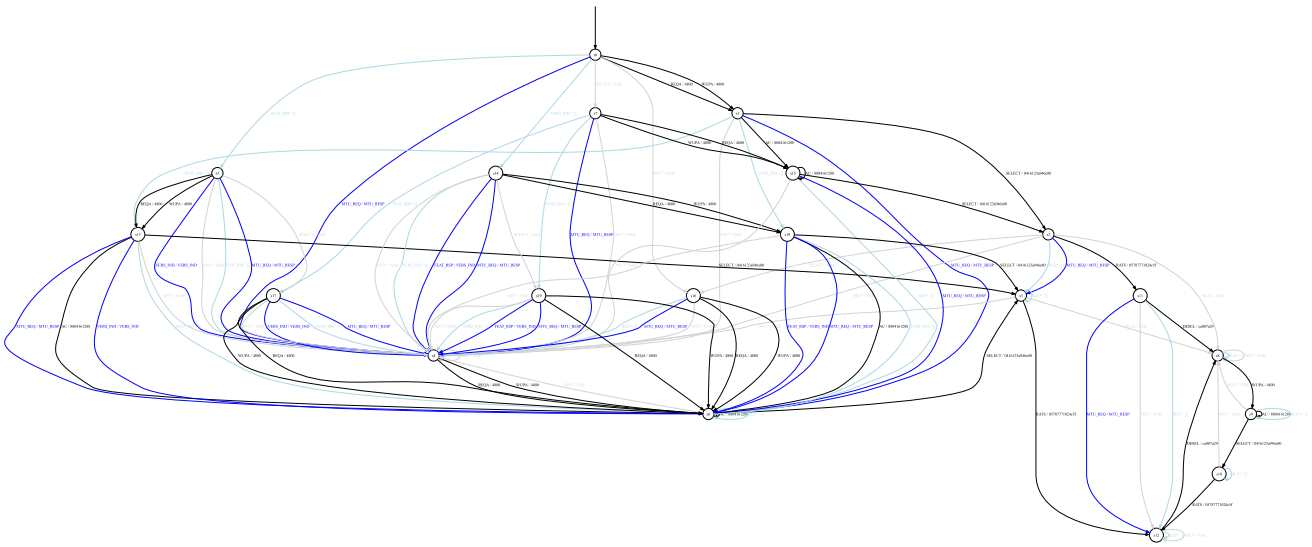

Fig. 7 Compound NFC/BLE Tesla key fob model learned with TTT. Transitions in black are NFC and those in blue are BLE, pale lines are without output (i.e., timeouts). We also used NFC* and BLE* for *all other* BLE and NFC inputs for better readability (Color figure online) (Zoomable figure online)

6.2.2 Compound protocol deviation analysis

We therefore manually analyzed the compound automaton with hidden NFC further. First, we built equivalence classes of states according to labels of outgoing transitions (which we name λ analysis after the labeling function). This yielded 4 equivalence classes (i.e., four different types of states, which corresponds to the stand-alone BLE model). Then, we checked the transitions for all states, if the respective target state is in the same equivalence class as the respective state in the BLE-only model (δ analysis, named after the transition function). With this, we identified four states (s2, s11, s15, and s16 in Fig. 7) that differ, i.e., their successor states do not belong to the same label equivalence classes as in the stand-alone BLE model (Fig. 6). This means that these four states form their own class and are not mergeable with the others – hence the two models cannot be equivalent. This class is characterized by a) having the output functions as s0 in BLE model, but b) all have the transition functions just as s1, s2, and s3 (they are not distinguishable, as all transitions from these classes lead to s2), and c) they are only reachable via τ transitions. This indicates that this behavior is NFC-induced, since the states are unreachable from BLE transitions alone (black in Fig. 7). An analysis of the traces yielded that all of them undergo NFC connections with timeouts, procrastinating the connection. This leads to a very simple explanation: the NFC transitions use up a good amount of the timer that checks for BLE connection timeouts. That way, they deliver an output from the s0 class (just as its first BLE input symbol), but then immediately jump to the BLE s2 class without giving s1 or s3 output because the timer runs out and s2 is a sink state that does not give BLE output (BLE s2 is the timed-out state – see the τ state in Fig. 6). As a result, we found some optimization potential in the learner code to quicken the send/receive process. This yielded a more complex model that is subject to future works.

7 Related work

There are other, partly theoretic, approaches of inferring a model using automata learning and comparing it with other automata using bisimulation algorithms. However, they target DFAs [10] or probabilistic transition systems (PTS) [17]. Neider et al. [36] present some significant theoretic fundamentals of using automata learning and bisimulation for different types of state machines, including Mealys. This work also contains the important observation that (generalized) Mealy machines are bisimilar if their underlying LTS are bisimilar. Tappler et al. [46] used a similar approach of viewing Mealy machines as LTS to compare automata regarding their bisimilarity. Similarly, bisimulation checking was also used to verify a model inferred from an embedded control software [45]. There is also previous work on using automata learning for inferring models of BLE systems, with the main target of fuzzing [39]. Another approach that targets compliance checking based on model checking that also sequences the subprotocol of BLE is from Karim et al. [27]. There is also work on learning NFC card models [1], which concentrates on the upper layer (ISO/IEC 14443-4) protocol, dodging the specific challenges of the handshake protocol.

Also there is no mentioning of automatic compliance checking in this approach. To the best of our knowledge, there is no comprehensive approach for compliance verification of the ISO/IEC 14443-3 and also no work on compound-protocol learning.

8 Conclusion

In this paper, we demonstrated the usage of automata learning to infer models of SULs and evaluate their compliance with the ISO 14443-3 protocol by checking their bisimilarity with a specification. We described a learning interface setup, showed practical results and made interesting observations on the impact of the protocol specifics on learning algorithms' performance. We further demonstrated the practical ability to learn a compound automaton of two protocols running on the same device (particularly using BLE symbols in the input alphabet) using a dispatcher SUL adapter. We then used similar techniques to determine differences between the sum (a compound automaton) and its parts (separately learned automata): preorder checks if the parts are included in the compound automaton and dissecting the compound automaton and comparing it with the parts. The results showed the compound and separately learned automata to be very similar. However, the added complexity made it more difficult to learn. The same learner configuration failed to spot the NFC nonconformance of the Tesla and showed a standard-conformant system instead. We also found extra states for BLE in the compound automaton: preorder yielded a positive result, while the hiding-base weak equivalence demonstrated that the automata are not equal. This means that those two checks combined (or two vice versa executed preorder checks) can determine if deviations are additional or missing behavior. The hiding-based method eases the root cause analysis what causes these deviations.

8.1 Discussion

Examining pure NFC, we found little differences between the SULs – all examined systems but the Tesla key fob were compliant to ISO/IEC 14443-3. However, the scrutinized NFC handshake protocol has two characteristics that are distinct from other communications protocols: a) it does not send an answer on unexpected input and b) the automaton has two almost identical parts that pose challenges in learning. Supposedly these characteristics are responsible for the somewhat surprising finding that the L* algorithm with the Rivest/ Schapire improvement surpasses more modern tree-based algorithms for correct systems. Still, TTT performed best in finding a noncompliant system and the minimum word length has an impact on the ability to find noncompliances. This helps learning similar structures. When looking at compound automata, we saw that two things were crucial – conformance testing and timing. Since the input alphabets of compound automata were bigger and combined words more complex, the possibility of missing subtle deviations like in the Tesla key fob is bigger. This induced the need of more conformance testing. Also, for time-sensitive (like most network) protocols it was important to optimize speed to avoid extra behavior stemming from timeouts. Since timing also played a role when learning deterministic machines, corner cases were important. We easily got a deterministic result if a session timeout was far away or long passed. But if input symbols were (coincidentally) sent near the timeout, they partially yielded a different result even in the same sequence (producing a nondeterminism): one in-time and one timed-out. This issue might be overcome by using timed automata, but this is a complex solution.

8.2 Outlook

Compliance checking is only a first step towards assuring correctness and, subsequently, cybersecurity for NFC systems. Concretely, further research directions include test case generation using model checking and targeting upper layer protocols (partly addressed in [31], but to be further extended). The compound learning yielded promising results, but their significance can be improved with speed optimizations in the learner as mentioned above. To overcome problems with timing-induced nondeterminism, we would need the ability to recognize time properties. Timed automata, however, are hard to create [7]. To avoid this burden, while still considering timing, Mealy machines with one [48] or multiple timers [8] could be used. This could contribute towards a general solution of learning system models with multiple protocols.

Acknowledgements We like to thank the reviewers of this paper, as well as the reviewers of the original conference paper, for their valuable insights that significantly helped improving this paper's quality.

Funding information Open access funding provided by Mälardalen University. This research received funding within the CHIPS Joint Undertaking (JU) under grant agreements No. 876038 (project InSecTT) and 101007350 (project AIDOaRt) and from the program "ICT of the Future" of the Austrian Research Promotion Agency (FFG) and the Austrian Ministry for Transport, Innovation and Technology under grant agreement No. 880852 (project LEARNTWINS). The JU receives support from the European Union's Horizon 2020 research and innovation program and Austria, Sweden, Spain, Italy, France, Portugal, Ireland, Finland, Slovenia, Poland, Netherlands, Turkey. We further acknowledge the support of the Swedish Knowledge Foundation via the industrial doctoral school RELIANT, grant No. 20220130. The document reflects only the authors' views and the Commission is not responsible for any use that may be made of the information it contains.

References

1. Aarts, F., De Ruiter, J., Poll, E.: Formal models of bank cards for free. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 461–468 (2013)
2. Aceto, L., Ingolfsdottir, A., Srba, J.: The algorithmics of bisimilarity. In: Advanced Topics in Bisimulation and Coinduction, pp. 100–172. Cambridge University Press, Cambridge (2011)
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
4. Antonioli, D., Tippenhauer, N.O., Rasmussen, K., Payer, M.: BLURtooth: exploiting cross-transport key derivation in Bluetooth classic and Bluetooth low energy. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security. ASIA CCS'22, pp. 196–207. Association for Computing Machinery, New York (2022)
5. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
6. Bluetooth SIG: Bluetooth Specification. Core Specification v5.4, Bluetooth SIG (2023)
7. Brouwer, R.: Learning Timed Mealy Machines of the Physical Processes of an Industrial Control System for Anomaly-Based Attack Detection. Master's thesis, University of Twente, Twente, the Netherlands (2020)
8. Bruyère, V., Garhewal, B., Pérez, G.A., Staquet, G., Vaandrager, F.W.: Active Learning of Mealy Machines with Timers. Preprint (2024). arXiv:2403.02019
9. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, pp. 21–39. Springer, Cham (2019)
10. Chen, Y.F., Hong, C.D., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: 2017 Formal Methods in Computer Aided Design (FMCAD), pp. 76–83 (2017)
11. Ebrahimi, M., Marksteiner, S., Ničković, D., Bloem, R., Schögler, D., Eisner, P., Sprung, S., Schober, T., Chlup, S., Schmittner, C., König, S.: A systematic approach to automotive security. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) Formal Methods. Lecture Notes in Computer Science, vol. 14000, pp. 598–609. Springer, Cham (2023)
12. Garbelini, M.E., Wang, C., Chattopadhyay, S., Sun, S., Kurniawan, E.: SweynTooth: unleashing mayhem over Bluetooth low energy. In: Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference. USENIX ATC'20, pp. 911–925. USENIX Association, USA (2020)
13. Garcia, F.D., de Koning Gans, G., Verdult, R.: Tutorial: Proxmark, the Swiss army knife for rfid security research: Tutorial at 8th workshop on RFID security and privacy (RFIDSec 2012). Tech. Rep., Radboud University Nijmegen, ICIS, Nijmegen (2012)
14. Groote, J.F., Mousavi, M.R.: Modelling and Analysis of Communicating Systems. MIT Press, Cambridge (2023)
15. Hancke, G.: Practical attacks on proximity identification systems. In: 2006 IEEE Symposium on Security and Privacy (S&P'06), pp. 6–333 (2006)
16. Hoare, C.A.R.: Communicating Sequential Processes, vol. 178. Prentice-Hall, Englewood Cliffs (1985)
17. Hong, C.D., Lin, A.W., Majumdar, R., Rümmer, P.: Probabilistic bisimulation for parameterized systems. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. Lecture Notes in Computer Science, pp. 455–474. Springer, Cham (2019)
18. Hopcroft, J.: An $n \log l$ algorithm for minimizing states in a finite automaton. In: Kohavi, Z., Paz, A. (eds.) Theory of Machines and Computations, pp. 189–196. Academic Press, San Diego (1971)
19. Hopcroft, J.E., Karp, R.M.: A Linear Algorithm for Testing Equivalence of Finite Automata. Tech. Rep., Cornell University (1971)
20. International Organization for Standardization: Cards and security devices for personal identification – Contactless proximity objects – Part 3: Initialization and anticollision. ISO/IEC Standard "14443-3", International Organization for Standardization (2018)
21. International Organization for Standardization: Cards and security devices for personal identification – Contactless proximity objects – Part 4: Transmission protocol. ISO/IEC Standard "14443-4", International Organization for Standardization (2018)
22. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification. Lecture Notes in Computer Science, pp. 307–322. Springer, Cham (2014)
23. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. Lecture Notes in Computer Science, pp. 487–495. Springer, Cham (2015)
24. Issovits, W., Hutter, M.: Weaknesses of the ISO/IEC 14443 protocol regarding relay attacks. In: 2011 IEEE International Conference on RFID-Technologies and Applications, pp. 335–342 (2011)
25. Jacobs, B., Silva, A.: Automata learning: a categorical perspective. In: van Breugel, F., Kashefi, E., Palamidessi, C., Rutten, J. (eds.) Horizons of the Mind. A Tribute to Prakash Panangaden: Essays Dedicated to Prakash Panangaden on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, pp. 384–406. Springer, Cham (2014)
26. Jansen, D.N., Groote, J.F., Keiren, J.J.A., Wijs, A.: An $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, pp. 3–20. Springer, Cham (2020)
27. Karim, I., Ishtiaq, A.A., Hussain, S.R., Bertino, E.: BLEDiff: scalable and property-agnostic noncompliance checking for BLE implementations. In: 2023 IEEE Symposium on Security and Privacy (SP), pp. 3209–3227. IEEE Computer Society (2023)
28. Kearns, M.J., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
29. Maass, M., Müller, U., Schons, T., Wegemer, D., Schulz, M.: NFCGate: an NFC relay application for Android. In: Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks. WiSec'15, pp. 1–2. Association for Computing Machinery, New York (2015)
30. Marksteiner, S., Sirjani, M., Sjödin, M.: Using automata learning for compliance evaluation of communication protocols on an NFC handshake example. In: Kofroň, J., Margaria, T., Seceleanu, C. (eds.) Engineering of Computer-Based Systems. Lecture Notes in Computer Science, vol. 14390, pp. 170–190. Springer, Switzerland (2023)
31. Marksteiner, S., Sirjani, M., Sjödin, M.: Automated passport control: mining and checking models of machine readable travel documents. In: Proceedings of the 19th International Conference on Availability, Reliability and Security. ARES'24, pp. 1–8. Association for Computing Machinery, New York (2024)

32. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. **5**(4), 115–133 (1943)
33. Mealy, G.H.: A method for synthesizing sequential circuits. Bell Syst. Tech. J. **34**(5), 1045–1079 (1955)
34. Merten, M., Howar, F., Steffen, B., Margaria, T.: Automata learning with on-the-fly direct hypothesis construction. In: Hähnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation, vol. 336, pp. 248–260. Springer, Berlin (2012)
35. Moore, E.F.: Gedanken-experiments on sequential machines. In: Automata Studies, AM-34, vol. 34, pp. 129–154. Princeton University Press, Princeton (1956)
36. Neider, D., Smetsers, R., Vaandrager, F., Kuppens, H.: Benchmarks for automata learning and conformance testing. In: Margaria, T., Graf, S., Larsen, K.G. (eds.) Models, Mindsets, Meta: The What, the How, and the Why Not? Lecture Notes in Computer Science, pp. 390–416. Springer, Cham (2019)
37. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)
38. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) Formal Methods for Protocol Engineering and Distributed Systems. IFIP Advances in Information and Communication Technology, pp. 225–240. Springer, Boston (1999)
39. Pferscher, A., Aichernig, B.K.: Stateful black-box fuzzing of Bluetooth devices using automata learning. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods. Lecture Notes in Computer Science, vol. 13260, pp. 373–392. Springer, Cham (2022)
40. Ribas Sobreviela, J.: Bluetooth Low Energy Based on the nRF52840 USB Dongle. Bachelor thesis, Universitat Politècnica de Catalunya (2019)
41. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. **103**(2), 299–347 (1993)
42. Sangiorgi, D.: On the origins of bisimulation and coinduction. ACM Trans. Program. Lang. Syst. **31**(4), 15:1–15:41 (2009)
43. Schögler, D.: An Automata Learning Framework for Bluetooth Low Energy. Master's thesis, Graz University of Technology, Graz, Austria (2023)
44. Shahbaz, M., Groz, R.: Inferring Mealy machines. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009: Formal Methods. Lecture Notes in Computer Science, pp. 207–222. Springer, Berlin (2009)
45. Smeenk, W., Moerman, J., Vaandrager, F., Jansen, D.N.: Applying automata learning to embedded control software. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) Formal Methods and Software Engineering, pp. 67–83. Springer, Cham (2015)
46. 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), pp. 276–287 (2017)
47. Vaandrager, F.: Model learning. Commun. ACM **60**(2), 86–95 (2017)
48. Vaandrager, F., Ebrahimi, M., Bloem, R.: Learning Mealy machines with one timer. Inf. Comput. **295**, 105013 (2023)
49. Vila, J., Rodríguez, R.J.: Practical experiences on NFC relay attacks with Android. In: Mangard, S., Schaumont, P. (eds.) Radio Frequency Identification. Lecture Notes in Computer Science, pp. 87–103. Springer, Cham (2015)