# Black-box protocol testing using Rebeca and Automata Learning*

Stefan Marksteiner[1,2][0000−0001−8556−1541] and Mikael Sjödin[2][0000−0001−7586−0409]

[1] AVL List Gmbh, Graz, Austria stefan.marksteiner@avl.com
[2] Mälardalen University, Västerås, Sweden
{stefan.marksteiner,mikael.sjodin}@mdu.se

**Abstract.** Industrial and critical infrastructure devices should be scrutinized with rigorous methods for inconsistencies with a specification. At the same time, this specification should also be correct, otherwise the specification conformance is of little value. On the example of eMRTDs (electronic Machine-Readable Travel Documents) we demonstrate an approach that combines model-checking a specification for correctness in terms of security with learning an implementation model using automata learning. Once the specification is modeled, we automatically mine a model of the implementation and check the model for compliance with the verified specification using simulation and trace preorder. Underspecification of the standard is in this setting modeled as non-deterministic behavior, so one of the possibilities has to simulate the implementation in order for the latter to be compliant. We also present a working tool chain realizing this method. When adopting the tool chain accordingly, the method might be used in practice for checking the correctness of any reactive system.

**Keywords:** Automata Learning · Rebeca · Compliance Checking · Model Checking · Formal Methods · NFC · eMRTD · Afra

## 1 Introduction

### 1.1 Motivation

Electronic Machine Readable Documents (eMRTDs) are critical infrastructure and should therefore be correct and secure systems. This means that they should be scrutinized with rigorous methods for inconsistencies with a specification. At the same time, this specification should also be correct, otherwise the specification conformance is of little value. We therefore strive for a methodology to automatically checking both a specification for its correctness and an implementation to be compliant to the former. In practice we present a practical approach to connect model checking for a correct specification for eMRTD communication (via

* The authors want to thank Marjan Sirjani, the receiver of this Festschrift and PhD advisor of the main author for teaching Rebeca and helping with the first steps of modeling the system described in this paper. Congratulations to the jubilee!

Near-Field Communications – NFC) with automata learning to mine a model of an implementation to check its conformance with the verified model. We thereby emphasize on security properties (i.e., protected information may only be read with proper authentication, etc.). With appropriate adapter classes, the method might be used for checking the correctness of many reactive systems. Having the interaction of two reactive systems (an eMRTD and a reader device) as target of examination, we use the Rebeca modeling language [28] (particularly *Core Rebeca*) to create a checkable specification model, since modeling these kind of systems is the very purpose of Rebeca. The latter, in conjunction with its Java-like syntax makes the modeling process fairly easy (compared to the decription syntax of other model checking systems) and, therefore, well-maintainable.

### 1.2 Contribution

This paper combines formal methods with systems engineering and testing to create a tool chain for checking implementations for their correctness and security. Our main contributions are:

- An approach for combining model-checking a specification for correctness with learning an implementation model
- An automated tool chain for the complete process, once a specification is modeled
- A verified specification model for eMRTDs

We use three formal methods: automata learning, equivalence checking (particularly simulation and trace preorder), and model checking. We use these methods in an automated tool chain and apply it to a practical use case, namely checking eMRTDs for their specification conformance and verifying the specification for security properties. Relying on Rebeca to model the standard, we produce a more secure (assured by model checking) and maintainable (through the traits of Rebeca) specification model to be used for checking the behavioral correctness of mined implementation models.

### 1.3 Approach

Starting from existing work on learning a behavioral model using automata learning and comparing it with a (partial) specification [19, 20], we use Rebeca to create a partial model of the International Civil Aviation Organization's (ICAO) Doc 9303 part 9 standard [25], which was done by hand in the contributions mentioned before. This document defines the structure of an eMRTD (including mandatory and optional elements, like stored document and personal data, biometrics, etc.) and how to access this data via the NFC protocol (ISO/IEC 14443-4 [10]) and standardized integrated-circuit interfaces (ISO/IEC 7816-4 [11]). It is important to note that despite using the standard that defines the data structure for eMRTDs, we actually model the behavior of inter-reacting systems: one hosting and one accessing the data structures defined in the standard.

We already outlined the specifics in another paper [20]. Rebeca's integration environment (Afra) comes with a specific model checker (Modere) [28]. This allows to verify the model for properties using Linear Temporal Logic (LTL) or Computational Tree Logic (CTL). The checker also creates a state space that represents the model (based on two communicating reactive systems). On the other hand we use active automata learning with the Learnlib library [13] to mine Mealy machine models of eMRTD implementations (i.e., the electronic representations of passports). We use a self-written converter to transform the Rebeca state space model into a Mealy-styled LTS (see Section 4.3). We can then check whether the learned implementation model is included (using simulation or trace preorder – see Section 4) in the verified specification. We use the MCRL2 toolset's [4] *ltscompare* tool to perform this analysis. If both the specification is successfully verified and the implementation is inside the specified behavior (i.e., preorder is successful), we can claim that the examined system is assured to fulfill the verified properties. We modeled security properties (e.g., authentication before access to sensitive data – see Section 4.2) and implemented this into a tool-supported process (see Figure 1).

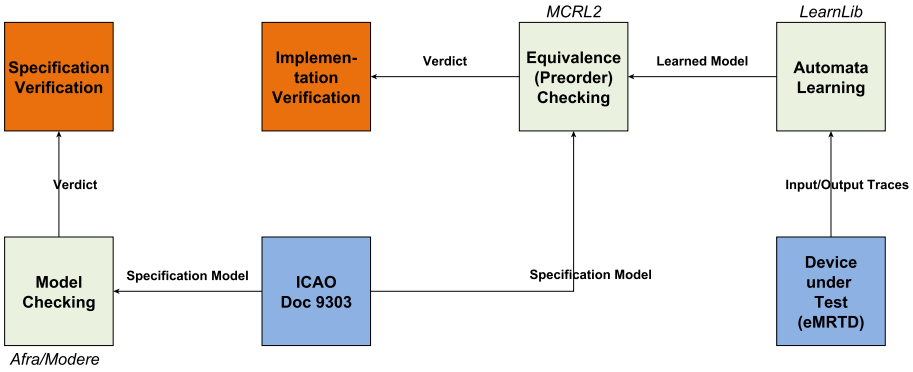


**Fig. 1.** Overview of the approach. Green are processes, blue are external inputs, and amber are outputs (i.e., results), the arrow labels are artifacts (input artifacts for arrows from blue to green boxes, output artifacts from green to amber boxes. The italic labels next to the green boxes denote the used tool sets/frameworks.

### 1.4 Limitations

Due to a lack of an available implementation, the authentication method is limited to Basic Access Control (BAC), while the other standardized methods, namely Password Authenticated Connection Establishment (PACE) and Terminal Authentication (TA) are not included in the models.

## 2 Preliminaries

Here we give a brief overview of some fundamental concepts used in this paper, as well as some basic descriptions of used tools and methods. We also give some definitions to well-known concepts to show our interpretation and avoid ambiguities.

### 2.1 Labeled Transition Systems and Mealy Machines

There are some basic approaches to model reactive systems. We use two of them, particularly Labeled Transitions Systems (LTS) and Mealy Machines. Transition systems describe a system's behavior in a graph-based manner by defining a set of states the system is in and a set of transitions that realize changes of these states, normally denoted by actions (i.e., inputs) and a transition function, along with initial states and atomic propositions (i.e., properties of the system in a certain state). An LTS contains also a labelling function that assigns actions to transitions. Formally, an LTS is 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, $AP$ a set of atomic propositions and $L$ a labelling function [2]. Finite State Machines (FSMs), also called automata, are similar to LTS, but their number of states and transitions is finite (a restriction not applicable to LTS), often used in a deterministic version, so called deterministic finite automata (or acceptor - DFA). This means that every input must have exactly one result in each state whereas, LTS do not have to be deterministic and can be seen as non-deterministic automata [8]. To model real-world, reactive systems, automata types that provide input and output are used. The two most common are Mealy [21] and Moore machines [23], where the difference lies in the output being produced by transitions (Mealy) or by states (Moore). For easier access to learning algorithms, we use Mealy machines. In a Mealy Machine, each input from a set (the alphabet) must be matched with a transition (i.e., change to a certain state, which can also be the original one) and an output. A Mealy Machine is defined as $M = (Q, \Sigma, \Omega, \delta, \lambda, q_0)$, with $Q$ being the set of states, $\Sigma$ the input alphabet, $\Omega$ the output alphabet (that may or may not be identical to the input alphabet), $\delta$ the transition function ($\delta : Q \times \Sigma \rightarrow Q$), $\lambda$ the output function ($\lambda : Q \times \Sigma \rightarrow \Omega$) – or a merger of both functions ($Q \times \Sigma \rightarrow Q \times \Omega$) – and $q_0$ the initial state. The transitions can be viewed also as tuples $\langle p, q, \sigma, \omega \rangle$ with $p, q \in Q$, $\sigma \in \Sigma$, and $\omega \in \Omega$ as elements of the combined input/transition function. State machines can be viewed as LTS by interpreting input/output pairs as labels of an LTS [29].

### 2.2 Types of Equivalence

To check the conformity of a system with a standard, we look for standard conform behavior of the system. The idea is to compare the behavior of a system with a specification. Using formal methods, we use behavioral equivalence checks of a learned (see Section 2.3) model of the implemented system with a correct

(see Section 2.5) specification model. Since we treat our models as LTS, we concentrate on equivalences for LTS. There are different types of formally defined equivalences, of which we use simulation and trace preorder, as well as bisimulation and trace equivalence [2]. The difference between preorder and equivalence relations is that preorder is reflexive and transitive, whereas equivalence is reflexive, transitive and symmetric (i.e., an equivalence is a symmetric preorder) [5]. Formally defined:

**Definition 1 (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 [2]:*

*A)* $\forall s_1 \in I_1 \cdot (\exists s_2 \in I_2 \cdot (s_1, s_2) \in R)$.
*B) for all* $(s_1, s_2) \in R$ *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* $(s_1\prime, s_2\prime) \in R$

Where $Post$ is the set of successor states of another state $Post(s) = \bigcup_{\alpha \in ACT} Post(s, \alpha)$ and $Post(s, \alpha) = \{s\prime \in Q | s \rightarrow s\prime\}$ [2]. For preorder, the respective relation may be unidirectional, whereas for equivalence, it is bidirectional. That means that when comparing two LTS ($LTS_1$ and $LTS_2$) with simulation preorder ($LTS_1 \preceq LTS_2$), $LTS_2$ has to simulate every behavior of $LTS_1$, but not vice versa; with bisimulation equivalence ($LTS_1 \sim LTS_2$) $LTS_1$ has to simulate $LTS_2$'s behavior and vice versa. For preorder, 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. Additionally there is trace preorder, which mandates that the set of traces of $LTS_1$ has to be included in the one of $LTS_2$, which might or might not contain additional traces:

**Definition 2 (Trace preorder).**
$Traces(LTS_1) \subseteq Traces(LTS_2)$

For bidirectional relations, there are equivalence relations with the same principles as preorder, namely bisimilarity and trace equivalence.

**Definition 3 (Bisimilarity).** *of two LTS ($LTS_1 \sim LTS_2$) has a binary relation $R \subseteq QxQ$, such that [2]:*

*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* $(s_1, s_2) \in R$ *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* $(s_1\prime, s_2\prime) \in R$

Trace equivalence is the symmetric version of trace preorder, which means that two transitions systems produce the same traces for each same input.

**Definition 4 (Trace equivalence).** $Traces(LTS_1) = Traces(LTS_2)$

### 2.3 Automata Learning

(Active) Automata Learning is a method of deriving a system model by querying a system with input data. Originally, it was described by Angluin in her work on learning regular sets [1], where she introduces the Learner-Teacher-Framework. In this framework, a learning system might ask a teacher two kinds of questions about the scrutinized system (System-under-learning – SUL):

- *Membership queries* and
- *Equivalence queries.*

Thereby, it is assumed that the teacher possesses a correct automaton of the SUL. The naming stems from the original purpose of learning Deterministic Finite Acceptors (DFA), a state machine type that describe regular languages. The DFA does or does not accept (hence acceptor) arbitrary sequences of symbols from a specific alphabet by deciding if the sequence (i.e., word) is a well-formed part of the respective language. Therefore, membership queries denote such input words, where the teacher answers whether or not they are accepted. The learner uses the respective output in a systematic way to infer a state machine. This also works for real-world reactive systems, but instead of generating queries to learn a DFA, the target is usually to learn a Mealy or Moore type automaton. Given the nature of these, the answer to membership queries is not yes or no, but rather the output of the automaton according to the output function, i.e. a query consists of an input word $W_\sigma$ that consists of symbols from the input alphabet ($\sigma \in \Sigma | W_\sigma = \langle \sigma_1, \sigma_2..\sigma_n \rangle$) and delivers an output word $W_\omega$ consisting of symbols from the output alphabet ($\omega \in \Omega | W_\omega = \langle \omega_1, \omega_2..\omega_n \rangle$) according to the output function $\lambda$ (simultaneously traversing through the sates according to $\delta$). If there is enough data to construct a state machine, the learner might ask the teacher whether the constructed state machine (hypothesis) corresponds to the actual system. This type of question is called equivalence query. The teacher answers with yes, if the hypothesis is correct (i.e., the hypothesis automaton is equivalent to the SUL automaton). Otherwise, the answer is a counterexample in form of an input word and the respective output word from the SUL automaton, that deviates from the hypothesis automaton's output word. Since the original L* algorithm, many improvements in learning methodologies have been developed, most notably the closure strategy of Rivest and Schapire [26]. More recent improvements include the replacement of the originally used observation tables by tree structures that represent distinctive features between states and allow for more efficient membership query generation. Notable algorithms using trees include Kearns-Vazirani (KV) [15], Direct Hypothesis Construction (DHC) [22], TTT [12], and L# [30]. When learning real-world systems, the assumption of possessing a correct SUL automaton is not feasible, especially for black-box learning settings (which is one of the main use cases for automata learning). Therefore, generally equivalence queries are replaced by conformance tests, i.e.

a *sufficient*[3] amount of (potentially long) queries after a certain strategies, e.g., random walks [18].

### 2.4 LearnLib

Learnlib [13] is arguably the most widely used library for automata learning (however, there are others, e.g., AALpy [24] or Libalf [3]). Written in Java, It features the most used automata learning algorithms (L*, Rivest-Schapire, AAAR, ADT, KV, DHC and TTT) and an addon L# implementation is available [17]. Also, it contains classes for conformance testing strategies (complete depth-bounded exploration, random words, random walk, W-method, Wp-method) and interfaces for providing connectors to SULs. It further contains AutomataLib, which contains tools for automata analysis and manipulation (e.g., minimizing automata).

### 2.5 Model Checking

Model checking is an automated methodology that (efficiently) explores all states (i.e., system scenarios) of (state-based) system model. Traversing through the states, it can check if certain system properties are satisfied in a certain state based on a sound fundament of graph theory, data structures, and logic [2]. The checkable properties can be stated in different kinds of logic like Linear Temporal Logic (LTL) or Computation Tree Logic (CTL), or the branching-time logic CTL* that encompasses both of the former. For its availability in the used model checker, we concentrate on LTL formulas. These are propositional logic [27] formulas with temporal modalities. Those modalities are

- *always* ($\Box$): the proposition must hold in any state
- *eventually* ($\Diamond$): the proposition must hold in some subsequent state (could hold before)
- *next* ($\bigcirc$): the proposition must hold in the immediately subsequent state and
- *until* ($\mathcal{U}$): the proposition $A_1$ must hold until another defined proposition $A_2$ occurs ($A_1 \mathcal{U} A_2$).

The respective proposition is embedded in a propositional formula. LTL formulas can also be nested. This allows for describing state conditions that must and must not occur in any state-based model. We use model checking in LTL for verifying the security properties (particularly authentication) of a specification model.

### 2.6 Rebeca

Rebeca is a modeling language that can be used to model reactive systems with a Java-like syntax [28]. It possesses its own modeling IDE (Afra [16]), which

[3] What is sufficient heavily depends on the specific use case and cannot be determined generally.

has also a built-in model checker (Modere [14]) that uses LTL statements for checking Rebeca models. A Rebeca model mainly consists of *reactive classes*, which model the behavior of a specific actor. These classes can have (internal) functions and (externally callable) message servers. Both allow local variables and basic statements like arithmetic and logic operations, assignments, conditionals, comparisons, casting, and instance operators that work like in the Java programming language. Additionally, it has non-deterministic assignment operator to model behavior that may take one of multiple paths. Additionally, a reactive class can have state variables that are maintained in any state of the model. These state variables can also be checked with Modere. Instances of a class are called reactive actors or Rebecs. Each class can have a list of known Rebecs with which its instances can interact by sending messages to its message servers. A class has also a message queue of defined size that holds (and sequences) messages for its specific server functions. This queue is also used for checking purposes like deadlock detection – if the system reaches a state where the message queue of the Rebec that is to take action at that point is empty, the system stalls in a deadlock. For model checking, a property file is defined that contains property definitions (i.e. atomic propositions that are statements formed from state variables of Rebecs), assertions (simple logic formulas that are always checked in any model checker execution) and LTL formulas (that can be executed one-by-one). The model checker also creates a state space of all visited states in an XML format, which is also convertible to the Graphviz format. The model checker thereby creates a state for every execution of a message server. This means that for a model of two interacting reactive systems, the resulting state machine shows mutual calling of the two Rebecs, with the possibility of a Rebec also calls its own message server (i.e., a self-loop). The state variables referenced in the property file are included as atomic propositions of the sate (they show up in the state if they are true and do not show up if they are false). Local functions and variables are not part of LTS generated from the state space.

### 2.7 Near Field Communication

Near Field Communication (NFC) is a wireless communication standard for passive (powerless), small embedded devices such as Radio-Frequency Identification (RFID) and chip cards (also known as smart cards). A proximity coupling device (PCD) creates an induction field that powers up a proximity integrated circuit card (PICC) and modulates the communication signals onto the induction field for transfer between PCD and PICC. ISO/IEC 14443-4 [10] defines the messages types for data transmission (information or $I$ blocks), signaling (supervisory or $S$ blocks), and acknowledgements (receive-ready or $R$ blocks), along with protocol mechanisms like block numbering, chaining, error correction, etc.

### 2.8 Integrated Circuit Access

ISO/IEC 7816-4 [11] defines data structures for transmission (both wired and wireless) to and from integrated circuit cards, including PICCs in the NFC pro-

tocol. Potential defined operations are data access, reading and writing data, as well as administrative and security functions, including authentication. For data access, PICCs are usually segmented into different applications (comparable to directories in a file system) that can be accessed via *Dedicated Files (DFs)*. The actual data resides in *Elementary Files (EFs)*. Both are usually accessed through a *SELECT* command. Once (potentially a DF and) an EF is (successfully) selected it can be manipulated via *READ*, *WRITE* and similary commands. Since the access to certain data should be protected, also the *GETCHALLENGE* and *AUTHENTICATE* commands are defined. The former is to initiate an authentication process, while the second concludes it. How that authentication works in particular is subject to the respective application and out of scope of the standard, usually some cryptographic operation based on a (symmetric or asymmetric) secret is conducted on the value obtained with the GETCHALLENGE command and returned in the AUTHENTICATE command. It is also expected that, after the authentication process, the commands for file selection and manipulation are secured (i.e., usually encrypted). It is also common that a successful authentication is tied to the application (i.e., DF) and could also differ on different applications on the same PICC.

The answer to any request contains the (encrypted or unencrypted) return data and a (always unencrypted status code, consisting of two bytes. In our work with eMRTDs, we have learned (and modeled) the following status codes as answers to specific queries:

- *9000* - OK
- *6300* - No information given (seen at authentication attempts with wrong credentials)
- *6700* - Error with no information given (when trying to perform write operations without authentication)
- *6982* - Security status not satisfied (i.e., lack of authentication)
- *6985* - Conditions of use not satisfied (when trying to authenticate without an application selected)
- *6986* - Command not allowed (when trying to read without a file selected)
- *6988* - Insecure messaging DOs (when encrypting data with a wrong key)
- *6A82* - File not found
- *6D00* - Instruction code not supported or invalid (when sending malformed commands)

### 2.9 Electronically Machine-Readable Travel Documents

Electronically Machine-Readable Travel Documents (eMRTDs) refer to the data stored on passport integrated circuits, accessible via NFC. The data structure is standardized in ICAO Doc 9303 part 9 [25]. It defines four applications: eMRTD, travel records, visa records, and additional biometrics. They are grouped into *LDS1* (only the eMRTD application) and *LDS2* (all other applications). Only the first is mandatory. Since we found only LDS1 on examined passports, we concentrate on this group. In the common area of the device (i.e., the area

without selecting an application), the standard defines the following files to be (mandatorily or optionally) present:

- *Attributes/Info (ATTR/INFO)*: containing the card capabilities (only mandatory if LDS2 is present).
- *Directory (DIR)*: containing a list of supported applications on the device (only mandatory if LDS2 is present).
- *Card Access (CA)*: containing security infos required for PACE authentication (only mandatory if PACE is implemented).
- *Card Security (CS)*: containing chip and terminal authentication (only mandatory if PACE with chip authentication mapping is implemented).
- *Common (EF.COM)*: containing metadata (version, encoding, etc.) of the application
- *Data Group 1 (EF.DG 1)*: containing the machine readable zone (mandatory).
- *Data Group 2 (EF.DG 2)*: containing the holder's face image (mandatory).
- *Data Group 3 (EF.DG 3)*: containing the holder's fingerprints image (optional).
- *Data Group 4 (EF.DG 4)*: containing the holder's iris image (optional).
- *Data Group 5 (EF.DG 5)*: containing holders displayed portrait(s) (optional).
- *Data Group 6 (EF.DG 6)*: is reserved for future use (optional).
- *Data Group 7 (EF.DG 7)*: containing the holder's displayed signature (optional).
- *Data Group 8 (EF.DG 8)*: containing data features (optional).
- *Data Group 9 (EF.DG 9)*: containing structure features (optional).
- *Data Group 10 (EF.DG10)*: containing substance features (optional).
- *Data Group 11 (EF.DG11)*: containing additional personal details (e.g., localized name, place-of-birth – optional).
- *Data Group 12 (EF.DG12)*: containing additional document details (e.g., issuing authority, date-of-issue – optional).
- *Data Group 13 (EF.DG13)*: containing optional details (optional).
- *Data Group 14 (EF.DG14)*: containing data elements (only mandatory if PACE is implemented).
- *Data Group 15 (EF.DG15)*: containing the public key info for active authentication (only mandatory if active authentication is implemented).
- *Data Group 16 (EF.DG16)*: containing persons to notify (optional).
- *Document Security Object (EF.SOD)*: containing hash values of the data group for integrity checking (mandatory).
- *Country Verifying Certification Authorities (EF.CVCA)*: containing public keys of CVCA for terminal authentication (mandatory).
- Key files for authentication.

The elementary files inside applications need, according to the standard, authentication. The LDS1 authentication needs Basic Access Control (BAC) or the newer Password Authenticated Connection Establishment (PACE), and data groups 3 and 4 additionally needs Terminal Authentication (which is, in contrast

to the other methods, based on a public key infrastructure). LDS2 applications (travel records, visa records, additional biometrics) need PACE and Terminal Authentication. All of these methods are defined in the standard. For the lack of a usable implementation of others authentication methods, we are limited to BAC. This method performs cryptographic operations based on a challenge with the passport number, expiration date, and the owner's date of birth as key material. This information is readable on the main page of the passport, but the accessible information is basically just an electronic version of the former (except for DG 3 and 4, which requires additional authentication). The purpose is not to completely shield this information, but to prevent bystanders of a person to obtain its personal information by just reading it out from the passport. In contrast to LDS2, the elementary files in the LDS1 application are defined to be read-only. Note that, due to many optional elements, the standard is underspecified. This means that more than one implementation is correct, which makes the standard non-deterministic from a modeling perspective.

## 3 Learning

We use Learnlib (see Section 2.4) to create a setup to learn NFC-based eMRTDs. This section covers the details on the interface, abstraction layer and alphabets. To learn the models we relied on the TTT algorithm and random walks as conformance testing algorithm. We learn the behavior of an eMRTD device with regard to the ICAO Doc 9303 part 9 (file structure and access rules), ISO/IEC 7816-4 (file access and security commands), and ISO/IEC 14443-4 (command and data transmission) standards. The learning setup is based on a previous paper [20], which we extend with the aspect of checking the specification model.

### 3.1 NFC Interface

To interact with eMRTDs, we created an NFC interface for Learnlib [19]. We connect the developed SUL class via a Socket to an API (based on C++) that operates a Proxmark3 device [7], running a custom firmware enhanced to support automata learning.

### 3.2 Abstraction

As usual with learning real-world systems, we use an abstraction layer to limit the potentially very large input space. This means that we reduce the input alphabet to sensible commands targeting the data structures from the ICAO standard. Since this includes secure communications, we use the commands (except for authentication-related) in encrypted and unencrypted versions. We also abstract the output to the status code, since it does not contain data and is also always unencrypted. This avoids non-determinism.

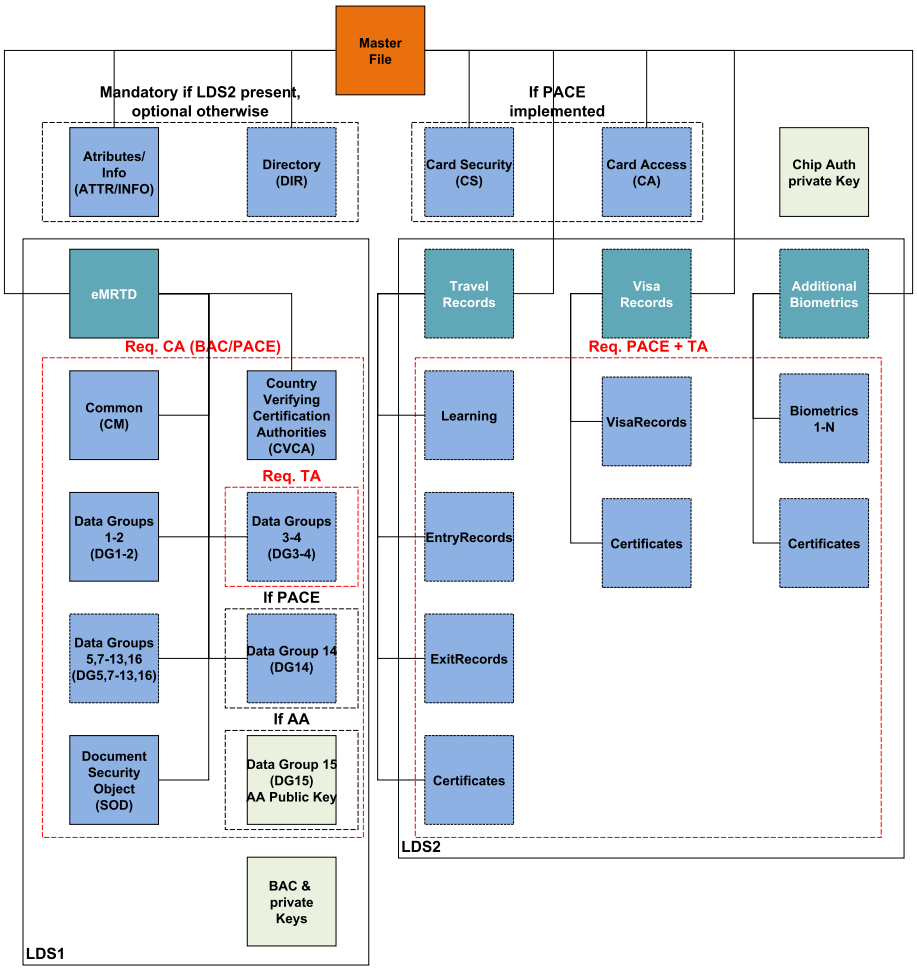


**Fig. 2.** Logical Data Structure of Machine Readable Travel Documents from [20]. Amber is the master file (MF), Cyan are dedicated files (DF), Blue are Elementary Files (EF), and Green are key files. Solid frames means mandatory files, dashed ones optional files. Solid boxes donate the LDS contexts, dashed black boxes requirements, and dashed red boxes necessary authentication.

### 3.3 Input Alphabet

The input alphabet consist of the data structure from ICAO Doc 9303 part 9 accessed with ISO/IEC 7816-4 commands over the ISO/IEC 14443-4 protocol. Concretely, we use the select DF, select EF, GETCHALLENGE, EXTERNAL AUTHENTICATION, READ BINARY, and UPDATE BINARY commands. Limiting to LDS1 (i.e., the eMRTD application), the full alphabet consists of SELECT DF LDS1 (SEL_DF.LDS1) and SELECT EF for the files mentioned (e.g., SEL_EF.CM) in Section 2.9, as well as READ BINARY to perform a read operation on selected files. Each of these inputs is used in a plain, unencrypted and a secure, encrypted version (e.g., SSEL_EF.CM for securely accessing the Common file). Additionally, we use a BAC symbol that issues the correct sequence (consisting of a GETCHALLENGE and an EXTERNAL AUTHENTICATION command based on the former) for performing this type of authentication. This also yields a session key that is used for performing the encryption operations in secure commands.

**Output Alphabets** The output alphabet does not have to be defined beforehand, but is discovered in-situ by the received answers. However, since we are not interested in the actual data, but rather want to prevent non-determinism (which the learner requires for proper functioning), we abstract the answers by using just the status code as an output. This is enough information to create a model for checking the security properties, mainly to check the proper authentication of data access. This eases also handling the answers of secure commands, as otherwise their data would need to be decrypted first. Using always-unencrypted status codes only, this is not needed.

### 3.4 Labeling and Simplification

Based on the (by the standard) known status codes, we can issue a simple labeling of the states, that also correspond to atomic propositions if the learned Mealy Machine is seen as an LTS [20]:

- From the initial state, follow the select EF for CardAccess transition
- If output yields 9000, label this state as *EF*
- From the inital state, select the DF for LDS1.eMRTD tansition
- If output yields 9000, label this state as *DF*
- From DF, follow the BAC transition
- If output yields 9000, lable this state as *DF—AUTH*
- From DF—AUTH, follow the encrypted select Data Group 1 transition
- If output yields 9000, label this state as *DF—AUTH—EF*
- From the initial state, select the BAC transition
- If output yields 6985 label the state as *FAILAUTH*
- From FAILAUTH, follow the select LDS1.eMRTD transition
- If output yields 9000 label the state as *FAILAUTH—DF*
- From EF, select the BAC transition

- If output yields 6985 label the state as *FAILAUTH—EF*
- From the DF—AUTH—EF, select the unencrypted READ BINARY transition
- If output yields 6982 label the state as *DEAUTH*

The propositions are: *EF* for a selected elementary file, *DF* for a selected dedicated file (i.e. the LDS1.eMRTD application), AUTH for a successful authentication (i.e., BAC), FAILAUTH for a failed authentication, *DEAUTH* for a revoked authentication.

## 4 Compliance Evaluation

To determine an implementation's compliance with the standards, we compare a learned implementation model from previous works [20] with a specification model derived from the standards (see Section 4.1). As the ICAO standard is underspecified, the results of access operations on (dedicated or elementary) files may have more than one legit result. This means that we cannot model a properly defined Mealy Machine from the standard (since it is non-deterministic), but rather a Mealy-styled LTS (or pseudo-Mealy), with more than one transition target and/or output label for a given input in a given state. However, since we make the comparison operations on LTS with the input/output pair being a combined label, the restriction to a deterministic model does not apply in practice. Due to multiple legit (i.e., standard-compliant) transitions for the same state/input pair, we cannot check for full equivalence. The learned model is a proper (and, thus, deterministic) Mealy Machine (otherwise the learner would crash for failing to handle non-deterministic behavior). Therefore, our learned model can always only cover one (of potentially multiple) state/input transitions. This makes it very likely that the specification displays extra behavior in comparison. For this reason we use preorder instead of simulation for compliance checking – the (learned) implementation model's behavior should stay inside the boundaries of the (modeled and checked) specification behavior.

### 4.1 Specification Model

We model the eMRTD specification as outlined in Section 2.9 in Rebeca by defining two reactive classes: a PCD class that is instantiated with a Rebec called *reader* and a PICC class that is instantiated with a Rebec called *PP* (short for passport). Each of the inputs (e.g., SELECT DF LDS1) is a message server of the PICC, while the respective answers (OK or respective error codes) are message servers of the PCD. Inside the PICC servers, we define the behavior according to the standard given the state variables (e.g., return OK for a secure select of a present file in an authenticated state, while returning an error in an unauthenticated). Listing 1.1 gives an example of the secure SELECT EF CA

and CVCA command[4]. It checks (via state variables) if there was a successful authentication. If that is the case it sets the EF selected state variable and calls the reader message server for ok (the integer parameter identifies the secure SELECT EF CM as originator of the call). Otherwise, it calls the message server for a missing authentication (since the secure command is out-of-context outside of the LDS1 application) or a not found error if the file is not present (which is selected by a non-deterministic assignment). This includes the optionality of files having multiple possible answer paths even if the state (determined by the set state variables) is the same.

**Listing 1.1.** Exemplary message server for the secure SELECT EF CM command

```
msgsrv SSEL_EF_CA_CVCA(){
    boolean present=?(true,false);
    if(!AUTH) {
        if(present)     reader.FailSec();
        else reader.NoFind();
    }
    else {
        EF=true;
        reader.OK(6001);
    }
}
```

The reader possesses a message server for every (standard-defined) answer code and a reset server that resets intermediately used state variables and is called at the beginning. Except for the message server for an OK message, every answer server and the reset function has local integer variable that is non-deterministically filled with an identifier and an according switch/case statement to determine the function to call next (the default is an error function that should never be reached, which is checked as an assertion by the model checker). This way, all of the possible combinations for inputs are invoked by the model checker. The message server for OK is different, as it sets some intermediate state variables that are used by the model checker to determine whether some specific operations are successfully executed. These intermediate state variables are reset by the reset message server. Listing 1.2 gives an example for the OK and reset servers (note that the list of called functions is arbitrarily truncated for the sake of brevity.

**Listing 1.2.** Exemplary message server for the OK answer followed by a reset.

```
[...]
msgsrv reset(){
    OK=false;
    rdBinOK=false;
```

[4] The command is the same for the optional CA file in the common area and the mandatory CVCA file in the LDS1 application area. The difference is just whether the DF LDS1 is selected or not.

```
    srdBinOK=false;
    ERROR = false;
    sselEFOK=false;
    int data =
       ?(1001,2001,2002,2003,3001,4001,6001,6002,6003,7001);
    switch(data){
        case 1001: PP.SEL_DF_LDS1(); break;
        case 2001: PP.SEL_EF_CA_CVCA(); break;
        case 2002: PP.SEL_EF_CM(); break;
        case 2003: PP.SEL_EF_CS_SOD(); break;
        case 3001: PP.RD_BIN(); break;
        case 4001: PP.BAC(); break;
        case 6001: PP.SSEL_EF_CA_CVCA(); break;
        case 6002: PP.SSEL_EF_CM(); break;
        case 6003: PP.SSEL_EF_CS_SOD(); break;
        case 7001: PP.SRD_BIN(); break;
        default: self.ERR();
    }
}


msgsrv OK(int data){
  switch(data){
        case 1001: break;
        case 2001: break;
        case 2002: break;
        case 2003: break;
        case 3001: rdBinOK=true; break;
        case 4001: break;
        case 6001: sselEFOK=true; break;
        case 6002: sselEFOK=true; break;
        case 6003: sselEFOK=true; break;
        case 7001: srdBinOK=true; break;
        default: self.ERR();
    }
  OK=true;
  reset();
}
[...]
```

Figure 3 shows a simplified state model of a small part of the specified standard, including just selecting the CM elementary file, the LDS1 application and an authentication. The full standard contains far too many states (134) to display here.

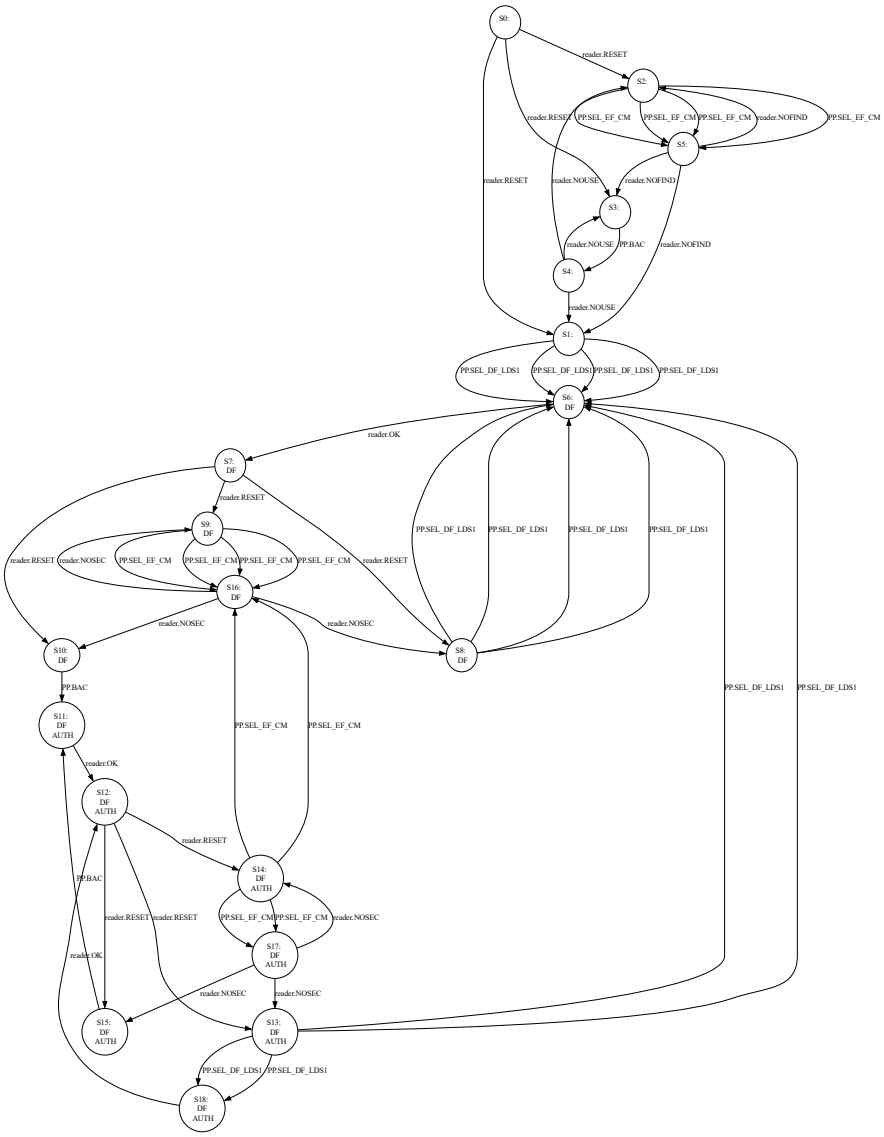


**Fig. 3.** Example of a simple Rebeca model. It only covers selecting one EF (CM), one DF (LDS1) and a BAC authentication command.

### 4.2 Model Checking

Before using the specification model for compliance checking, we assure its correctness with regard to some basic security properties, as stated below, using model checking. For checking the model, we defined six different rules in LTS that assure the correctness of the security properties of the modeled specification. In particular these rules are:
NoXStates: $\Box(\neg\ (\neg \text{DF} \wedge \text{AUTH}))$;
PlainRead: $\Box(\neg\ (\text{READOK} \wedge (\neg \text{EF} \vee \text{DF})))$;
ReadFollowsSelect: $(\neg \text{READOK}\ \mathcal{U}\ \text{EF}) \vee \Box(\neg \text{READOK})$;
SecureRead: $\Box(\neg\ (\text{SREADOK} \wedge \neg(\text{DF} \wedge \text{AUTH} \wedge \text{EF})))$;
SecureSelect: $\Box(\neg\ (\text{SSELEFOK} \wedge \neg(\text{DF} \wedge \text{AUTH})))$;
SecureReadFollowsSecureSelect: $(\neg \text{SREADOK}\ \mathcal{U}\ \text{SSELEFOK}) \vee \Box(\neg \text{SREADOK})$;
Apart from obvious atomic propsitions like EF, AUTH, and DF, we use READOK, SREADOK, and SSELEFOK, which stand for successful operations and are only set for one state before being unset in the reset message server. NoXStates contains a check for illegal states where authentication is true without an LDS selected (which should not happen because of lacking context for authentication). Plainread says that a (plain EF) READ should not be successful without selected EF or with selected DF (in the latter case, only secure READs should be successful), ReadFollowsSelect sayst that a READ should only be successful when a file is selected or not successful at all. SecureRead says that a secure READ should only successful in an authenticated, DF and EF selected state, SecureSelect says that a successful secure SELECT should only occur in an authenticated and DF-selected state, while SecureReadFollowsSecureSelect says that a secure READ can only occur after a secure SELECT (implying authentication).

### 4.3 Converting the Specification Model

A full state space export (see Section 4.2) comes in an XML format, that can be converted into the Graphviz (DOT) format. However, displaying two communicating reactive systems (see Section 3), the format is not compatible with the learned implementation automata, which are Mealy Machines. The Mealy Machines combine the two systems into a single combined state, where the two communication directions are visible in the input/output dualism of the transitions.

We therefore wrote an automated conversion tool that removes the states and transitions concerning the reset function and the respective states for unsetting the successful operation propositions (READOK, SREADOK, and SSELEFOK), as those are just used for checking the model (if certain operations succeed only if certain conditions are met) and should not show up in the specification automaton. Then, it determines the mergable states by first building equivalence classes based on the states' propositions and remove redundant ones. In our case, it is possible to form equivalence classes for merging states this way, because the Rebeca model sets the states' propositions according to system properties relevant for the comparison with a learner, reducing the possible set of remaining

ones to the relevant ones, i.e., the set of possible equivalence classes is predefined by the possible combinations of propositions in the model. This was also the motivation of writing a dedicated converter instead of relying on established toolsets. Lastly, for each equivalence class, it replaces reader/PP state pairs with single states and take the transitions from reader to PP as *input* part and from PP to reader as *output* part of the resulting combined transition label. This eventually creates a Mealy-styled LTS with the transitions labeled with input and output. The difference to an actual Mealy Machine is, again, that the we can have more than one transition for the same input in the same state (making the LTS effectively a non-deterministic pseudo-Mealy Machine). This is, however, not a concern for the compliance checking procedure, as we treat the learned implementation (a Mealy Machine) and the specification automaton (a pseudo-Mealy Machine) as LTS with the input/output pairs as labels. This makes the automata comparable by diverse kinds of equivalence relations using standard methods.

Figure 4 shows a comparison of the specification pseudo-Mealy Machine with the learned passport model. For readability, just to demonstrate the concept, we shrinked the showed model the operations SELECT EF CM, SELECT DF LDS1 and BAC. The evaluation, however, covers the full standard specification (see Section 5).

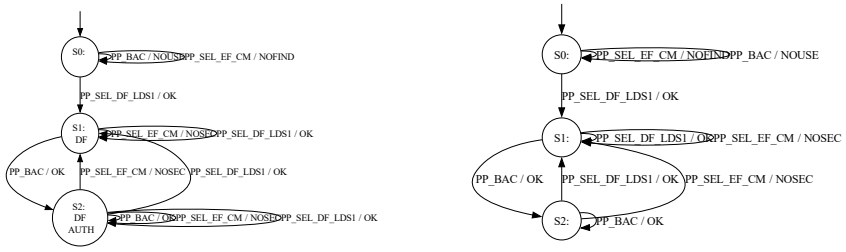


**Fig. 4.** Simplified example of a Rebeca model converted in a non-deterministic Mealy-style LTS compared to a learned MRTD Mealy model. Note that the difference lies in additional transitions in the Rebeca version, modeling optional behavior. The Rebeca model shows the same behavior as the one in Figure 3.

## 5 Evaluation

In this section we briefly outline the achieved results with the described methods. We used the described model-checked ICAO standard's specification model and used trace and simulation preorder to check a learned model of an Austrian passport for its compliance with the specification model. We obtained the learned

model in Graphviz format from our Learnlib-based NFC learner (see a labeled, simplified version in Figure 5), created the specification model using Rebeca in the Afra IDE and verified it with the Modere model checker. We then converted the verified model into a pseudo-Mealy Machine in Graphviz format (see Figure 6 for a simplified, consolidated version). The full pseudo-Mealy has 6 states and 662 transitions, whereas the original state model from Rebeca has 134 states and over 5300 transitions. This shows the significant decrease of complexity through the Rebeca state model's conversation into a Mealy-styled LTS. The learned model has 6 states and 342 transitions. We converted both Graphviz models into the Aldebaran format and used it as an input for mCRL2's [4] ltsccompare tool, with which we performed simulation and trace preorder checking. The result showed a positive check, i.e. it showed that a simulation preorder relation between the learned model and the specification exists. Hence, the learned model behavior is included in the behavior of the specification. We were therefore able to demonstrate the passport's compliance with the ICAO standard and, through the model checking, we were able to demonstrate the security of the standard specification (in terms of the data being securely stored on the eMRTD device).

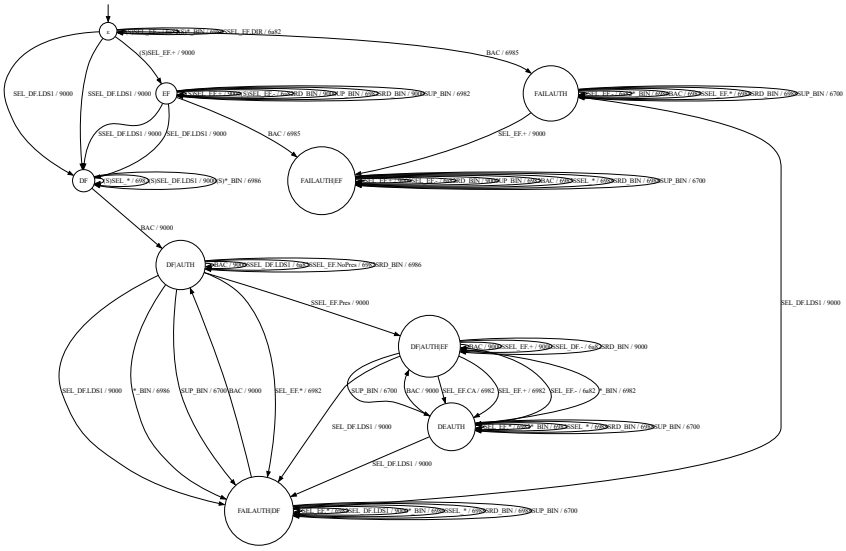


**Fig. 5.** Labeled model of an Austrian passport learned with TTT from [20].

## 6 Related Work

There are approaches for combining automata learning and bisimulation algorithms [6, 9]. However, to the best of our knowledge, there are no approaches for

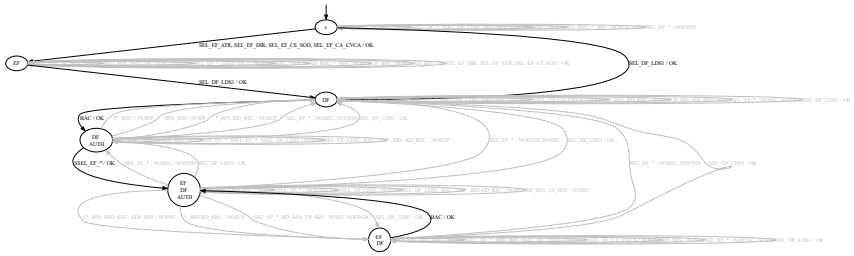


**Fig. 6.** Example of the modeled ICAO specification. Please note that we simplified the transitions because of the complexity. Solid lines are transitions towards more elevated access rights, light lines are transitions leading towards the same or lower access privileges.

combining model checking, learning and preorder checking as ours. We funded this approach on our method on partly specifying the ICAO eMRTD specification [20] and used the learned model from that work. However, in the former approach we manually modeled the specification pseudo-Mealy Machine using pure graphical notation (i.e, we hand-modeled it in the graphviz format) and did not use a modeling language nor model checking. Using Rebeca and Modere in the present work, we expanded the approach by formally verifying the specification model and, through the automatic conversion to the specification LTS, eliminated sources of error in the specification modeling. We also used the approach of using equivalence checking (bisimulation and trace equivalence) with NFC before, particularly for an automatic compliance checker for the ISO/IEC 14443-3 (the NFC handshake) protocol [19]).

## 7 Conclusion

In this paper, we demonstrated an approach and a tool chain of automata learning to infer models of systems under test and evaluate their compliance with a specification model in Rebeca, that is formally checked for security properties. We used this approach in practice to automatically mine a Mealy model of an eMTRD device (an Austrian passport). We further created a Rebeca model of the ICAO Doc 9303 part 9 standard and verfied it for security properties (particularly, proper authentication for access to restricted files). We then used the mCRL2 toolset to check the compliance of the mined model with the specification model using simulation and trace preorder, for which we converted the specification model into a Mealy-styled LTS. This way we could show the compliance of the SUT with the verfied standard.

### 7.1 Discussion

We have limited the current approach LDS1 application of eMRTDs. Furthermore, we were unable to learn another passport (particularly a newer German

one), because it uses the more recent PACE authentication for which we do not have a working implementation. Nevertheless, the approach is scalable and can be expanded to cover these areas. Furthermore, the process is transferrable to other systems, mainly depending on the availibility of a learner.

### 7.2 Outlook

To generalize the approach, we plan to use the vice-versa method, namely converting learned Mealy machines into Rebeca code. This way we can mine models of arbitrary systems automatically, which is very tedious to do manually. Once having these models in Rebeca, we can use Modere for checking them for security and other correctness properties.

## References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (Nov 1987). https://doi.org/10.1016/0890-5401(87)90052-6
2. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (Apr 2008)
3. Bollig, B., Katoen, J.P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: Libalf: The Automata Learning Framework. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 360–364. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6˙32
4. 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. pp. 21–39. Lecture Notes in Computer Science, Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1˙2
5. Champarnaud, J.M., Coulon, F.: NFA reduction algorithms by means of regular inequalities. Theoretical Computer Science **327**(3), 241–253 (Nov 2004). https://doi.org/10.1016/j.tcs.2004.02.048
6. 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 (Oct 2017). https://doi.org/10.23919/FMCAD.2017.8102244
7. 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)
8. Groote, J.F., Vaandrager, F.: Structured operational semantics and bisimulation as a congruence. Information and Computation **100**(2), 202–260 (Oct 1992). https://doi.org/10.1016/0890-5401(92)90013-6
9. Hong, C.D., Lin, A.W., Majumdar, R., Rümmer, P.: Probabilistic Bisimulation for Parameterized Systems. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 455–474. Lecture Notes in Computer Science, Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4˙27

10. 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)
11. International Organization for Standardization: Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange (2020)
12. 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. pp. 307–322. Lecture Notes in Computer Science, Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3"˙26
13. Isberner, M., Howar, F., Steffen, B.: The Open-Source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 487–495. Lecture Notes in Computer Science, Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4"˙32
14. Jaghoori, M.M., Movaghar, A., Sirjani, M.: Modere: The model-checking engine of Rebeca. In: Proceedings of the 2006 ACM Symposium on Applied Computing. pp. 1810–1815. SAC '06, Association for Computing Machinery, New York, NY, USA (Apr 2006). https://doi.org/10.1145/1141277.1141704
15. Kearns, M.J., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press (Aug 1994)
16. Khamespanah, E., Sirjani, M., Khosravi, R.: Afra: An Eclipse-Based Tool with Extensible Architecture for Modeling and Model Checking of Rebeca Family Models. In: Hojjat, H., Ábrahám, E. (eds.) Fundamentals of Software Engineering. pp. 72–87. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-42441-0˙6
17. Kruger, L., Junges, S., Rot, J.: State Matching and Multiple References in Adaptive Active Automata Learning. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods. pp. 267–284. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-71162-6˙14
18. Lee, D., Sabnani, K., Kristol, D., Paul, S.: Conformance testing of protocols specified as communicating finite state machines-a guided random walk based approach. IEEE Transactions on Communications **44**(5), 631–640 (May 1996). https://doi.org/10.1109/26.494307
19. 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 Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-49252-5˙13
20. 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. pp. 1–8. ARES '24, Association for Computing Machinery, New York, NY, USA (Jul 2024). https://doi.org/10.1145/3664476.3670454
21. Mealy, G.H.: A method for synthesizing sequential circuits. The Bell System Technical Journal **34**(5), 1045–1079 (Sep 1955). https://doi.org/10.1002/j.1538-7305.1955.tb03788.x
22. 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 Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34781-8˙19
23. Moore, E.F.: Gedanken-Experiments on Sequential Machines. In: Automata Studies, AM-34, vol. 34, pp. 129–154. Princeton University Press (1956). https://doi.org/10.1515/9781400882618-006
24. Muškardin, E., Aichernig, B.K., Pill, I., Pferscher, A., Tappler, M.: AALpy: An active automata learning library. Innovations in Systems and Software Engineering **18**(3), 417–426 (Sep 2022). https://doi.org/10.1007/s11334-022-00449-3
25. Organization, I.C.A.: Machine Readable Travel Documents – Part 9: Deployment of Biometric Identification and Electronic Storage of Data in MRTDs (Eigth Edition) (2021)
26. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. In: Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing. pp. 411–420. STOC '89, Association for Computing Machinery, New York, NY, USA (Feb 1989). https://doi.org/10.1145/73007.73047
27. Russell, B.: Mathematical Logic as Based on the Theory of Types. American Journal of Mathematics **30**(3), 222–262 (1908). https://doi.org/10.2307/2369948
28. Sirjani, M.: Rebeca: Theory, Applications, and Tools. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures. Lecture Notes in Computer Science, vol. 4709, pp. 102–126. Springer (2006). https://doi.org/10.1007/978-3-540-74792-5˙5
29. 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 (Mar 2017). https://doi.org/10.1109/ICST.2017.32
30. Vaandrager, F., Garhewal, B., Rot, J., Wißmann, T.: A New Approach for Active Automata Learning Based on Apartness. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 223–243. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9˙12