Mälardalen University Press Dissertations
No. 445

FORMAL METHODS-BASED SECURITY TESTING UTILIZING THREAT MODELING, AUTOMATA LEARNING, AND MODEL CHECKING

Stefan Marksteiner

2025

School of Innovation, Design and Engineering

ISBN 978-91-7485-726-9
ISSN 1651-4238
Printed by E-Print AB, Stockholm, Sweden

Abstract

This thesis strives towards finding more efficient methods of automating security test case generation, which are currently in a state of infancy for automotive systems, in both white and, especially, black box settings. The thesis focuses on communication protocols used in vehicular systems and we base our research on formal methods. The rationale is their rigor, as they are based on sound logical principles, and their potential for efficiency gains, since formally defined systems can be more easily analyzed algorithmically and, therefore, tested automatically. Our contributions include:

- Methods for deriving automata:
  - We provide a method to automatically obtain behavioral models in the form of state machines of communication protocol implementations in real-world settings using automata learning.
  - We demonstrate a method to derive compound protocol state machines, i.e., state machines representing systems that communicate via more than one protocol at the same time.
- Methods for checking automata:
  - We provide a means to automatically check these state machines for their compliance with a specification (e.g., from a standard, like ISO/IEC 14443-3).
  - We provide a scheme, Context-based Proposition Maps (CPMs), to augment the state machines with propositions (i.e., attributes that can be checked).
  - We define generic Linear Temporal Logic (LTL)-based properties to recognize cybersecurity-related specification violations.
  - We provide a method to model-check inferred state machines utilizing the Rebeca modeling language providing a formally defined template.

- Methods to facilitate test case generation:
  - We present a technique to automatically derive test cases to demonstrate deviations identified in a state machine on the actual system.
  - We also present a method to create abstract cybersecurity test-case specifications from semi-formal threat models using attack trees.
  - We provide a method for utilizing Large Language Models (LLMs) to derive test cases from threat models and inferred state machines.
  - We present a method utilizing LLMs to derive security properties from threat models to model-check implemented state machines, determining the consistency of designs' threat models and implementations' state machines.

Sammanfattning

Denna avhandling strävar mot att hitta mer effektiva metoder för att automatisera testfallsgenerering i både white- och black box-scenarier och vårt fokus ligger på kommunikationsprotokoll som används i fordonssystem. Den huvudsakliga ansatsen är att använda modellbaserade metoder. Vi presenterar en praktisk metod för att automatiskt erhålla beteendemodeller i form av tillståndsmaskiner för implementeringar av kommunikationsprotokoll med hjälp av automatinlärning. Vi presenterar också ett sätt att automatiskt kontrollera dessa beteendemodeller för att se om de överensstämmer med en specifikation (t.ex. från en standard). Vi presenterar vidare en teknik för att automatiskt härleda testfall för att demonstrera ev. upptäckta avvikelser i modellen på det faktiska systemet. Vi presenterar också en metod för att skapa abstrakta testfallsspecifikationer för cybersäkerhet från semiformella hotmodeller med hjälp av attackträd.

”Gedanken ohne Inhalt sind leer,
Anschauungen ohne Begriffe sind blind.”
Immanuel Kant (1724 - 1804)

Acknowledgments

I like to use this space to thank this thesis' advisors Marjan Sirjani and Mikael Sjödin for spending their precious time and giving me guidance in research directions and writing, which brought me forward in both professional and scientific manners. Furthermore, I like to thank my employer at the AVL List Gmbh, foremost my former and current supervisors Horst Pflügl and Kieran McAleer for giving me the freedom to do the necessary research, without losing the vision for the application and the practical use of it. I also want to thank my son, Benjamin Marksteiner, just for being there and brining joy (and also action) to my life. Most of all, I like to thank my wife Astrid Marksteiner for going this journey with me, even if it is not easy alongside manyfold job duties and a small kid. No acknowledgement could make it up with her support.

Stefan Marksteiner, October 2025, Graz, Austria

This research received funding 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 agreements #867558 (TRUSTED) and #880852 (LEARNTWINS), from the European Union under the Horizon Europe programme under grant agreement #101168438 (INTACT), and within the CHIPS Joint Undertaking (JU) under grant agreements #876038 (InSecTT) and #101007350 (AIDOaRt). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Sweden, Spain, Italy, France, Portugal, Ireland, Finland, Slovenia, Poland, Netherlands, Turkey. The document reflects only the author's view and the Commission is not responsible for any use that may be made of the information it contains. We further acknowledge the support of the Swedish Knowledge Foundation (KKS) via the industrial doctoral school RELIANT under grant #20220130.

Main Thesis Supervisor:
Marjan Sirjani, Mälardalen University, Västerås, Sweden
Co-supervisor:
Mikael Sjödin, Mälardalen University, Västerås, Sweden

Public defense Nov. 28, 2025, Mälardalen University, Västerås, Sweden

Public Defense External Reviewer (Opponent):
Mariëlle Stolinga, University of Twente, Netherlands
Public Defense Examination Committee:
David Sands, Chamlers University of Technology, Sweden
Mattias Nyberg, KTH Royal Institute of Technology, Sweden
Mahsa Varshosaz, IT University of Copenhagen, Denmark

Based on the Licentiate Thesis
Model-Driven Security Test Case Generation Using Threat Modeling and Automata Learning
Public defense Apr. 25, 2024, Mälardalen University, Västerås, Sweden
examined by
Mohammed Reza Mousavi, King's College London, UK (opponent)
Bengt Jonsson, University of Uppsala, Sweden
Martin Törngren, KTH Royal Institute of Technology, Sweden

List of Publications

Papers Included in the PhD Thesis[1]

Paper I: *A Systematic Approach to Automotive Security*, Masoud Ebrahimi, **Stefan Marksteiner**, Dejan Ničković, Roderick Bloem, David Schögler, Philipp Eisner, Samuel Sprung, Thomas Schober, Sebastian Chlup, Christoph Schmittner, and Sandra König. In *Formal MethodsLecture Notes in Computer Science*, vol. 14000. Cham: Springer International Publishing, 2023. DOI: 10.1007/978-3-031-27481-7_34. [1]

Paper II: *Approaches For Automating Cybersecurity Testing Of Connected Vehicles*, **Stefan Marksteiner**, Peter Priller and Markus Wolf. In *Intelligent Secure Trustable Things*, M. Karner, J. Peltola, M. Jerne, L. Kulas, and P. Priller, Eds., in Studies in Computational Intelligence. Springer Nature, 2023. DOI: 10.1007/978-3-031-54049-3_13. [2]

Paper III: *Using Automata Learning for Compliance Evaluation of Communication Protocols on an NFC Handshake Example*, **Stefan Marksteiner**, Marjan Sirjani and Mikael Sjödin. In *Engineering of Computer-Based Systems Conference 2023*, J. Kofroň, T. Margaria, and C. Seceleanu, Eds., in Lecture Notes in Computer Science, vol. 14390. Cham: Springer Nature Switzerland, 2023. DOI: 10.1007/978-3-031-49252-5_13. [3]

Paper IV: *From TARA to Test: Automated Automotive Cybersecurity Test Generation Out of Threat Modeling*, **Stefan Marksteiner**, Christoph Schmittner, Korbinian Christl, Dejan Ničković, Mikael Sjödin, and Marjan Sirjani. In *Proceedings of the 7th ACM Computer Science in Cars Symposium (CSCS '23)*.

[1] The included papers have been reformatted to comply with the thesis layout.

New York, NY, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3631204.3631864. [4]

Paper V: *Automated Passport Control: Mining and Checking Models of Machine Readable Travel Documents*, **Stefan Marksteiner**, Marjan Sirjani, and Mikael Sjödin. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, in ARES '24. New York, NY, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3664476.3670454 [5]

Paper VI: *Black-box protocol testing using Rebeca and Automata Learning*, **Stefan Marksteiner** and Mikael Sjödin. In *Rebeca for Actor Analysis in Action*, E. A. Lee, M. R. Mousavi, C. Talcott, Eds., in Lecture Notes in Computer Science, vol. 15560. Cham: Springer International Publishing, 2025. DOI: 10.1016/0890-5401(87)90052-6. [6]

Paper VII: *Learning Single and Compound-protocol Automata and Checking Behavioral Equivalences*, **Stefan Marksteiner**, David Schögler, Marjan Sirjani, and Mikael Sjödin. In *International Journal on Software Tools for Technology Transfer*. Cham: Springer International Publishing, 2025. DOI: 10.1007/s10009-025-00797-y. [7]

Paper VIII: *STAF: Leveraging LLMs for Automated Attack Tree-Based Security Test Generation*, Tanmay Kuhle, **Stefan Marksteiner**, Jose Alguindigue, Hannes Fuchs, Sebastian Fischmeister and Apurva Narayan. *ESCAR Europe, 2025*. Paper accepted, not yet published. Preprint available on arXiv. DOI: 10.48550/arXiv.2509.20190. [8]

Paper IX: *Learn, Check, Test - Security Test Generation Utilizing Automata Learning and Model Checking*, **Stefan Marksteiner**, Marjan Sirjani, and Mikael Sjödin. In *Computers & Security*. Elsevier, 2025. Paper submitted, not yet accepted. Preprint available on arXiv. DOI: 10.48550/arXiv.2509.22215. [9]

Additional Peer-reviewed Publications Related to the Thesis[2]

Paper 1: *Wireless Security in Vehicular Ad Hoc Networks: A Survey*, Thomas Blazek, Fjolla Ademaj, **Stefan Marksteiner**, Peter Priller, Hans-Peter Bernhard. In SAE International Journal of Connected and Autonomous Vehicles, vol. 6, no. 2, 2022. DOI: 10.4271/12-06-02-0011. [10]

Paper 2: *A Global Survey of Standardisation and Industry Practices of Automotive Cybersecurity Validation & Verification Testing Processes and Tools*, Andrew Roberts, **Stefan Marksteiner**, Mujdat Soyturk, Berkay Yaman, Yi Yang. In SAE International Journal of Connected and Autonomous Vehicles, vol. 7, no. 2, 2023.DOI: 10.4271/12-07-02-0013. [11]

Paper 3: *"Zeroth-order optimization attacks on deep reinforcement learning-based lane changing algorithms for autonomous vehicles*, Dayu Zhang, Nasser Azad, Sebastian Fischmeister, **Stefan Marksteiner**. In Proceedings of the 20th international conference on informatics in control, automation and robotics - volume 1: ICINCO, SciTePress / INSTICC, 2023.
DOI: 10.5220/0012187700003543. [12]

Paper 4: *AVATAR: Autonomous Vehicle Assessment through Testing of Adversarial Patches in Real-time*, Abhijith Sharma, Apurva Narayan, Nasser Lashgarian Azad, Sebastian Fischmeister, and **Stefan Marksteiner**. In IEEE Transactions on Intelligent Vehicles, pp. 1–14, 2024.
DOI: 10.1109/TIV.2024.3449830. [13]

Paper 5: *Actors for Timing Analysis of Distributed Redundant Controllers*, Marjan Sirjani, Edward A. Lee, Zahra Moezkarimi, Bahman Pourvatan, Bjarne Johansson, **Stefan Marksteiner**, Alessandro Papadopoulos. In *Concurrent Programming, Open Systems and Formal Methods*, J. Meseguer, C. A. Varela, N. Venkatasubramanian, Eds., in Lecture Notes in Computer Science, vol. 16120. Cham: Springer International Publishing, 2026. DOI: 10.1007/978-3-032-05291-9_8. [14]

[2]These papers are not included in this thesis.

Contents

I

Thesis

Chapter 1

Introduction

This thesis concentrates on using formal methods for assuring correctness (with regards to a formal specification) and security (regarding properties and behavior) of systems with a focus on the automotive domain. Formal methods are used, because their roots in strong logical concepts, which makes them intrinsically sound and their ability to be efficiently processed by computing systems, which makes them help increasing the degree of automation of test systems. To utilize these methodologies in practice, some problems have to be solved, that in turn lead to solving some scientific problems that lay the fundament for this thesis.

A special emphasis is given to verifying the correctness of communication protocol implementations in system components (e.g., NFC interfaces in key fobs, Bluetooth interfaces in infotainment units, or the Unified Diagnostics Services protocol in automotive control units). and to verification from an architectural threat modeling perspective. Regulation R.155 [15] from the United Nations Economic Commission for Europe (UNECE)[1] mandates not only the introduction of a cybersecurity management system (CSMS) and according security measures for automotive systems, but also evidence of their appropriateness and effectiveness, which is to be demonstrated by testing. This requires an amount of testing of the cybersecurity of vehicle systems that is not to be covered with manual testing alone. Therefore, automated methodologies for automotive cybersecurity testing are needed. Furthermore, the testing should work in both white box and black box settings. The automation for both types, but especially for the latter is still in its infancy. White box methods do

[1] https://unece.org

not work in black box settings (for the lack of information), however black box methods are not efficient in white box settings (for not using available information). Therefore we use white box methods to cover the security of all aspects of a system in the most thorough way. We also use black box methods for two reasons: a) provide an attacker's view, and b) because of the lack of access to source code or other system component internals. The latter is due to long supply chains, as well as the unwillingness of suppliers to disclose system details.

Formal methods has been used in engineering complex systems, e.g., in the automotive and aerospace domains [16]. Despite the effort they normally come with, their founding on strong mathematical principles has three main advantages: a very structured approach that provides a high level of comprehensiveness, well-reasoned verdicts, and a high degree of automation capability (once a model has been derived). This is a more rigorous and thorough approach than, for instance, conformance testing based on combinatorial algorithms, since the complete behavior a system is scrutinized systematically. Also, hypothetical behavior can easily be introduced by slightly altering the model, which allows for examining otherwise invisible corner cases and possible mitigations along with non-deterministic behavior. There are different approaches to formal modeling practically in use in industry: models based on state machines have a long tradition of analyzing systems' correctness , on the other hand architectural threat models [17] in the form of graphs with clear semantics and first order logic-based threat rule sets are used in the software industry for quite some time and have become very widely adopted as part of a Threat Analysis and Risk Assessment (TARA) in the automotive industry [18]. Threat models analyze a system design based on data flows between its architecture's components [17].

Regardless of these beneficial traits, the methods mentioned above are not only ordinarily very labor-intense, but also hard to apply on black-box components, as without access to internal information it is not trivial to obtain a state machine model (also called automaton) in an automated way. Even if such an automaton is present, reliable methods and rules to check it for correctness in terms of security have to be in place – none such is so far formulated in general, therefore sets of these rules and methods have to be established for each use case individually. Lastly, it is an open question how white box and black box modeling methods could support each other. An architectural threat model from a TARA is usually (manually) generated at design time and is based on assumptions that are often tacitly made when using modeling components. We therefore strive to investigate, if it is possible to check whether made assump-

tions actually hold in the implementation. If they do not, the model may be inaccurate and the following security analysis unreliable. Concentrating on external interfaces of system components, automatically derived state machine models of communication protocols are investigated. One further question is also which algorithm and parameter configuration is most suitable for inferring models of implementations of a specific protocol.

This thesis therefore provides an approach to generate security test cases from an architectural threat model in white box settings, concentrating on the question how to provide a formal translation from threat modeling results to actions in test cases utilizing attack trees [19, 20] and labelled transition systems (LTS) [21]. For black box settings, this thesis investigates inferring a behavioral model in form of a state machine. To derive such a model, the L^* algorithm by Angluin [22] and variations thereof [23], as well as more modern algorithms (e.g. [24]) are examined. Automatic model derivation is beneficial because not only manual modeling in general is very resource-intense, but also hard to perform in black box settings. On an example case of the Near-field Communication's (NFC) handshake protocol (ISO 144443-3 [25]), for which a learning interface working in real-world environments is provided, the different algorithm and parameter sets were investigated, giving details (efficacy and performance) for automata learning in special cases. The thesis also works out the necessary level of abstraction in order to investigate the possibilities of implementing this kind of learning in a practical (i.e., real-world) setup. Using these learned models, a behavior-based black box compliance checking method using bisimulation or trace equivalence is provided[2]. The comparison object for these equivalences is an automaton modeled after the respective specification or standard, the learned implementation should comply to. For the sake of practical use we also investigate the possibility of learning models of systems that contain more than one protocol. In practice, we encountered systems that not only display more than one communication protocol interface, but also the usage of these protocols in an intertwined manner. We show this in practice on the example of a car access system that uses Bluetooth Low-Energy (BLE) [26] for control signals and simultaneously NFC as an out-of-band method for key exchange in the key paring process. We compare these compound models with separately learned automata (in the example case a pure NFC and a pure BLE automaton) to spot differences and, therefore, uncover behavior that only appears when using both protocols. To provide a more targeted approach to security test case generation, we also directly examine the model for patterns of

[2] In black box settings with a low number of states (¡15), as encountered in practice during this work, the difference in both performance and strength of provided guarantees is negligible

undesired behavior using model checking. We therefore augment the learned Mealy-type automata with atomic propositions using an adaptable rule set and check for certain properties in linear temporal logic (LTL). For instance, we check whether a read operation in a protocol is only possible when the system has successfully completed an authentication process before. A concrete example on a electronically machine-readable travel document (eMRTD – i.e., a passport) is that an NFC *secure select file* command must only be successful iff an authentication process (e.g., a BAC) has successfully been executed before *and* no deauthentication (by, e.g., selecting a different application) has occurred. The same applies for the automotive Unified Diagnostic Service (UDS) protocol: for certain (protected) memory areas, a *read* command must only be successful iff there was a successful security access procedure. The threat model-based test case generation is a novel approach, while the method of combining automata learning and bisimulation checking is rarely used in general, and not at all for behavior-based black box implementation compliance checking. Nevertheless, it provides a more rigorous way of compliance checking due to its more exhaustive way of model building. So far, little comprehensive work for automata learning in different settings has been performed (see Section 5), for which this thesis also provides a contribution. Lastly, the thesis sketches some notions of using feedback from the behavioral state machine checking to the architectural threat models. The latter are ordinarily generated manually during design phase and based on assumptions about the modeled components (e.g., a modeled component complying to a certain standard or featuring a distinct property). These assumptions can be checked in later phases by checking the state machines of the implemented systems if these assumptions hold (e.g., if the implemented component actually complies to the standard, or a security guarantee as an attribute of a component in the model). This allows for statements about the accuracy of the threat model from checking the behavioral state machine models. Another novel approach within this work is the learning of compound automata. So far, joint automata have been created by calculating product automata, but not by explicitly learning automata of two systems (i.e., protocols) at once. This allows for the analysis of behavior that does not appear in one of these systems only, which is not possible by creating such a combination merely by calculating. The novelty of the approach of model checking the automata lies in automated conversion of learned automata and its automatic augmentation with atomic properties to allow for the applicability of the process. Furthermore, the analysis of assumptions about components in threat models, as mentioned above, become more targeted when connecting component attributes with LTL properties in the model checker.

1.1 Motivation: Industrial Problems Leading to Scientific Problems

This section contains an outline of what practical problems motivate the research goals (Section 3.1) and solutions to scientific problems (Section 4) in this thesis. It also serves the purpose of giving a context how the problem fields for the research goals are related, targeting to give a better understanding of the overall picture of the research. The practical overall goal of this thesis is facilitating automated cybersecurity testing of vehicular systems. Analyzing contemporary state-of-the-art automotive cybersecurity engineering processes (most prominently ISO/SAE 21434:2021 [27]), several gaps that hinder the efficiency of testing were identified. A standard cybersecurity engineering process is aligned with the general automotive engineering processes. One proliferated example is Automotive Spice, which is roughly defined along four major activities [28]:

- Threat Analysis and Risk Assessment (TARA – accompanying the system design as part of a cybersecurity security requirements elicitation process) to analyze potential weaknesses and threats in the design and assess their severity during the design phase.

- Implement the system using security goals and claims drawn from the TARA mitigating the found risks and therefore implementing a secure design (in cybersecurity implementation process).

- Validate and verify the system security measurements' effectiveness, providing evidence and arguments for the implemented system's cybersecurity (as risk treatment verification and validation processes).

- Repeat the actions above during the rest of the system's life cycle after the start of production (SOP) with both updates of the system (functional and non-functional) and of the threat landscape (e.g., discovery of new vulnerabilities).

We set a focus on correct systems, which means that system is faithful to its requirements [29]. To provide both more efficient and more rigorous testing of system correctness structuring and automation is desirable to be applied. Setting an emphasis on cybersecurity, we focus on deviations from a correct system (i.e., flaws) that impact the system's cybersecurity. One of potential fields

identified is to combine threat modeling done in the TARA with testing by automatically deriving test cases from the former. This enhances efficiency by doing two necessary things at once (TARA and test case definition) and enhances testing rigor by directing the testing to the very security measures derived from the security goals drawn from the TARA. To gain this ability, formalized test cases descriptions[3] has to be connected to the results of a TARA, constituting a research problem resulting in research goal 1 (RG1 – Section 3.1.1). Although TARA is conducted during design time (without an implementation available), it is still possible to create the necessary test case descriptions, as those can be made technology-agnostic and formed into practically executable test cases once an implementation is available (see Section 5.3). Also, it is quite usual in the automotive industry that an Original Equipment Manufacturer (OEM – in the automotive context mostly a car maker) integrates components from suppliers without getting access to their internals (i.e., not getting source code, internal specifications, etc.). This creates the needs to test the correctness of these systems in a black box setting. One scientific problem this requirement opens up is a means to automatically obtain a formalized description of the behavior of a (sub-)system's implementation to have an object to automatically analyze, which eventually lead to research goal 2 (RG2 – Section 3.1.2) efficient ways to create a state machine model from a black box system real-world settings. The other side of the medal is to have a means to actually check that model for its correctness and security. This relays to the research problem of how to check the behavior of a model against a given specification, yielding research goal 3 (RG3 – Section 3.1.3). Both the solution for RG1 and for RG3 have been created after industrial needs which is underpinned by patents that have been filed as Austrian patent applications No. A50667/2023 (pending) and A50660/2023 (granted [31]), respectively.

Furthermore, we use the learned models for targeted checking of security properties. We augment the models with atomic propositions and derive LTL formulas for property checking (e.g., is the authentication mechanism secure) by protocol analysis, generalization, and from threat model attributes. In the latter case, found property checking violations can be fed back into threat modeling: assumptions made in the threat model design through set attributes then do not hold in the implementation (as they are falsified by checking).

Future work includes methods to use learned models for fuzz testing and derive features to check from threat modeling in order to check modeled assumptions more rigorously. As this kind of test automation usually iterate over

[3] These formalized descriptions are outside of this thesis and have been defined by the authors in previous work [30].

the complete life cycle the correctness of the implementation also feeds back to the threat modeling, by check if the assumptions made in the design phase hold in the implementation, which influences future iterations in the life cycle. Fig-

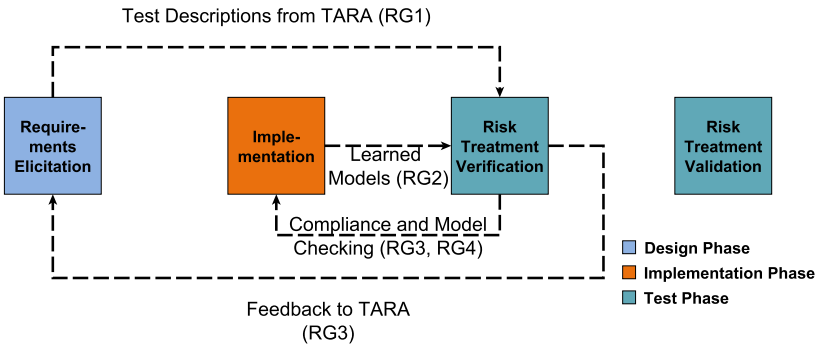

Figure 1.1: Research goals in relation to the Security Engineering Process after Automotive Spice [28]. The boxes represent the processes of the cybersecurity engineering process group, while the arrows represent improvement from this thesis, leading to the research questions in Section 3.1. Note that Risk Treatment Validation is outside the scope of this thesis.

ure 1.1 gives an overview of the automotive cybersecurity process group [28] and the practical implications of its automation leading to the research goals of this thesis; the boxes represent the processes of the cybersecurity engineering process group, while the arrows represent practical improvements as presented in this thesis (see above), leading to the research goals.

Methodological Toolset

As a summary, the research goals target towards creating a set of formal methods to derive cybersecurity test cases from live, real-world systems. We leverage automata learning to infer formal models of systems under test and propose two methods to use the models for test case generation:

- Compliance checking
- Model checking

The compliance checking approach encompasses (manually) modeling specification automata (from international standards and/or vendor specifications)

and using behavioral equivalence to determine the compliance of the learned system model with the specification automaton. Model checking involves augmenting the learned model with propositions and then use logical formulas, derived from system analysis and generalization, as well as from threat models. We then check if the implementation of the system violates any of the specified properties.

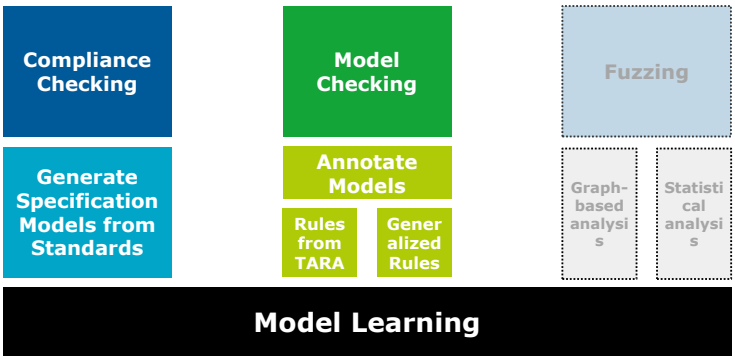

Figure 1.2: Overview of the methodologies to be researched in this thesis with model learning as a fundament and three different approaches to generate security tests from models, including intermediate goals. Fuzzing is in light colors, as it is part of an overall concept, but out of this thesis' scope.

1.2 Thesis Outline

This thesis consists of two parts: Part I provides a coat for the research, namely the necessary preliminaries, the aim of the research, its contribution and comparison with existing work. Part II consists of the research papers constituting this thesis. The remainder of Part I is organized as follows: Chapter 2 contains the background, Chapter 3 gives and overview of the research goals and methods, Chapter 4 outlines the research contributions including a short description of the included publications, Chapter 5 outlines related work and Chapter 6 concludes the thesis and gives an outlook to future work.

Chapter 2

Background and Preliminaries

This section very briefly explains some basic concepts that are used in this thesis. Other related work and alternative approaches towards reaching the research goal are outlined in Section 5. The usage of this background research this thesis builds on is as follows: threat modeling, attack trees, labeled transition systems (LTS), and Formalized Attack Languages in RG1 (Section 3.1.1), Mealy Machines and Automata Learning in RG2 (Section 3.1.2), Mealy Machines and Behavioral Equivalences in RG3 (Section 3.1.3).

2.1 Threat Modeling

Threat modeling is a systematic, semi-formal approach to scrutinize systems for potential threats and pitfalls. Threat modeling ordinarily requires two components [17]. One is a structured representation of the considered system (i.e., a system model), containing all information necessary to determine potential threats, as well as assessing their impact and likelihood of occurrence. A commonly used form of representation in sophisticated tools for threat modeling are data flow or architecture diagrams (see Figure 2.1 for a simple example) . The second component is the actual threat model. This model consists of a set of rules that determine which potential threat would occur if two components in the system model are connected in a certain way (see Listing 2.1 for a simple rule). These rule sets can, depending on the application domain, become very

complex, with the goal being to scrutinize the considered system very comprehensively and structured. The given example architecture and rule would yield a threat of *Introduction of Malicious Software* to be applicable to the system. This threat list will then ordinarily be subject to a risk assessment[1] to prioritize the threats. These further lead to security goals and, eventually, security requirements for the system design and implementation. Sophisticated threat models contain a considerable amount of domain knowledge and are usually created by groups of security experts in the respective domain. This thesis uses threat models as a basis to create test cases in a structured way (see Section 4.1).

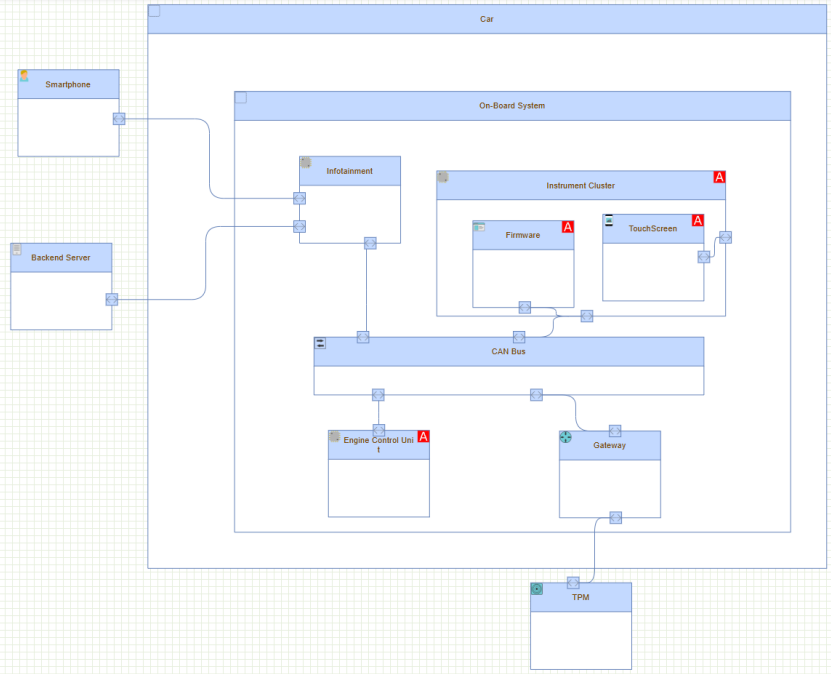

Figure 2.1: Simple example architecture diagram for threat modeling.

[1]Therefore in the automotive industry often the term TARA (Threat Analysis and Risk Assessment) is used.

```
CONNECTOR {
    SOURCE ELEMENT {
        REQUIRES CAPABILITY "Control" >= "true"
    } &
    TARGET ELEMENT {
        PROVIDES CAPABILITY "Control" := "true"
    }
}
```

Listing 2.1: Example Threat Rule

2.2 Attack Trees

Attack trees display relations, interdependencies and hierarchies of threats and vulnerabilities [19, 20]. The advantage of this form of representation is the ability to display different paths towards a certain objective i.e., to show different opportunities to concatenate attacks in order to exploit a certain vulnerability from a distinct starting point (mostly an interface accessible from the outside). This way, attack trees are a capable tool for assessing how combined attacks that exploit a complete set of threats impact the overall attack surface and success likelihood [32]. Figure 2.2 shows an attack tree for the example architecture outlined in Section 2.1. The displayed information on top represents the visible dashboard information (speed and RPM gauge, warning lamps, etc.) running on the instrument cluster, which in turn is susceptible to spoofing attacks, since it is connected to an unsecured CAN bus. Following the rightmost path, the CAN bus can be accessed via the Infotainment system, which again will be accessed through a controlled Smarthpone. One of the paths inside the given tree represents a sequence of threats from an exposed interface to the defined goal (in this case manipulating the dashboard). In this thesis, attack trees stemming from threat models are the origin for a method to automatically derive technology-agnostic security test scenarios to provide evidence for the correct functioning of implemented security measures (see Section 4.1).

2.3 Labeled Transition Systems

A principal notation for formal representations of systems used in this thesis are transition systems (TS) and labeled transition systems (LTS). A TS is defined

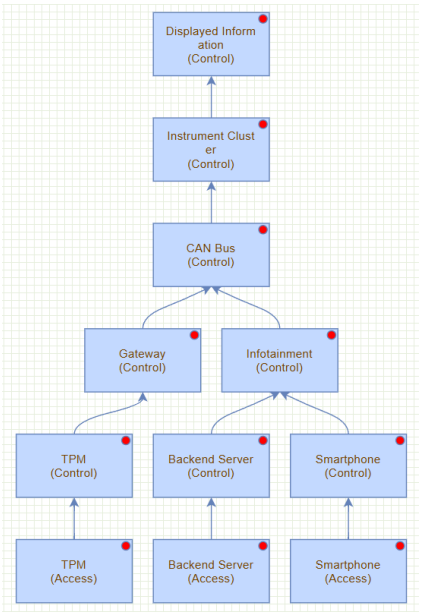

Figure 2.2: Example attack tree for the architecture shown in Figure 2.1 with exposed interfaces as sources and manipulating the dashboard (displayed information on the instrument cluster) as a target.

as a set of states (Q), a transition relation ($\rightarrow \in Q \times Q$, with $q, q\prime \in Q; q \rightarrow q\prime$), and an initial state (q_0). An LTS additionally possesses a set of labels (Σ), such that each transition is named with a label σ in Σ ($q, q\prime \in Q, \sigma \in \Sigma; q \xrightarrow{\sigma} q\prime$) [21]. LTS can describe the behavior of systems and mechanisms at different levels. This thesis uses LTS for a translation mechanism from attack trees to attack descriptions in a specifically designed attack description language (see Section 4.1).

2.4 Formalized Attack Languages

Domain-specific languages (DSLs) are computer programming language of limited expressiveness focused on a particular domain [33]. That means that they should be just expressive enough to model any necessary features and conditions of the respective domain and lean enough for domain experts to be

easily read and communicated about. Besides they ordinarily have formal syntax and semantics in the sense that a state machine (particularly a deterministic finite acceptor – DFA) built for one particular language should be able to decide if a word or a statement is a well-formed statement in that language. A DSL (see Section 5.3 is used as a means for describing attacks in a technology-agnostic manner as part of test case generation (see Section 4.1).

2.5 Mealy Machines

Mealy machines are a specific form of state machines (or automata), which are a fundamental concept in computer science. Similar to LTS, Mealy machines provide a formal notation for systems' behaviors. The main difference is, that a Mealy machine provides an output for any input, which makes them an adequate representation for real-world cyber-physical systems. The definition of Mealy machines reads $M = (Q, \Sigma, \Omega, \delta, \lambda, q_0)$, with Q being the set of states, Σ the input alphabet, Ω the output alphabet (that may or may not identical to the input alphabet), δ the transition function ($\delta : Q \times \Sigma \to Q$), λ the output function ($\lambda : Q \times \Sigma \to \Omega$), and q_0 the initial state [34]. The transition and output functions might be merged ($Q \times \Sigma \to Q \times \Omega$). This thesis uses Mealy machines to represent learnt system behavior through observation of inputs and outputs in automata learning (see Section 4.2).

2.6 Automata Learning

Active automata learning is a method of actively querying systems and noting the output to a given respective input. This allows for inferring behavioral models of black-box systems. The classic method of automata learning, called the L* algorithm, uses the concept of the *minimally adequate Teacher* [35]. This teacher has (theoretically) perfect knowledge of system to learn and can answer two kinds of questions:

- Membership queries and
- Equivalence queries.

The membership queries' answers are denoted in an observation table, that eventually allows for trying to infer an automaton. The equivalence queries determine if the inferred automaton is correct. Lacking a teacher with perfect

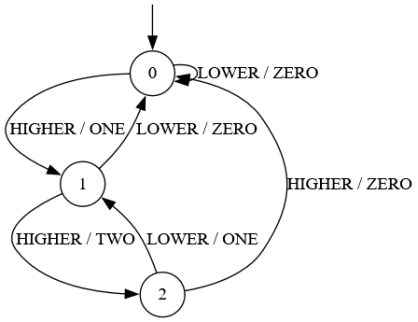

Figure 2.3: Exemplary simple state machine.

system knowledge in a black-box situation, the equivalence queries are ordinarily replaced by traditional conformance testing. More modern algorithms (like TTT [36]) rely on discrimination trees instead of observation tables to be more efficient [37]. To exemplify the difference between table and tree-based learning algorithms, Figure 2.3 shows a simple automaton with three states and two possible inputs (higher and lower). Higher increases the number of the state and Lower decreases it. If the state is zero, it cannot be decreased (remains zero) and if it is two, an increase flips it over to zero again. The output of a transition is always the target state number. Table 2.1 shows the final observation table for that automaton derived with the L* algorithm. The columns represent suffixes. The upper part or the table rows are the short prefixes, with each indicating a state. The lower part are longer prefixes that are discovered along the learning process. Each value of the long prefix part must be present in the short prefixes (*closedness*) and they must not contradict each other (*consistency*). If the table is not closed, lines from the long prefix section must be included into the short prefixes (indicating new states). If the table is not consistent, a new suffix (i.e., a new column) must be introduced, distinguishing the contradicting prefixes from each other. As the short prefixes alone denot the states, the long prefixes can be seen as overhead in the table. In contrast, tree-based algorithms concentrate on state-distinguishing features. In a tree-based structure (the discrimination tree), they determine the distinguishing set of inputs that would clearly discriminate a state by the respective output. Figure 2.4 shows the discrimination tree for the example above. It can be clearly seen that the input HIGH distinguishes all three states (in state 0, it yields ONE, in state 1 TWO, and in state 2 ZERO as an output). This is not the

	HIGHER
ϵ	ONE
HIGHER	TWO
HIGHER_HIGHER	ZERO
LOWER	ONE
HIGHER_LOWER	ONE
HIGHER_HIGHER_HIGHER	ONE
HIGHER_HIGHER_LOWER	TWO

Table 2.1: Final observation table for the automaton displayed in Figure 2.3 by the L* algorithm. Rows are (short and long) prefixes, columns are suffixes. It reads as prefix plus suffix equals result in the respective cell. E.g., only HIGHER (ϵ+HIGHER) yields ONE, three times HIGHER yields ZERO.

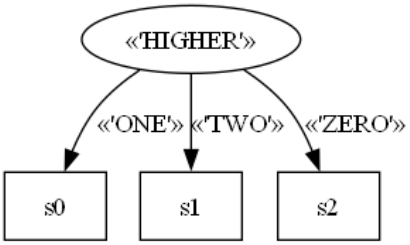

Figure 2.4: Final discrimination tree for the automaton in Figure 2.3 by the TTT algorithm. One input (HIGHER) discriminates all states by giving a different output in the respective state.

case for LOWER, as it yields ZERO in both states 0 and 1. In this example, the tree version basically reduces the overhead of the longer prefixes compared to the observation table. This thesis uses both traditional and modern types of automata learning to infer behavioral component models (see Section 4.2).

2.7 Behavioral Equivalences

LTS and automata (particularly of the Mealy type used in this thesis) can be compared for their equivalence. In particular for the purpose of this thesis, an equivalent behavior is important. That means that two automata do not necessarily have to be identical (i.e. all states and transitions being identical), but

merely the same input has to yield the same output. This equivalence can be evaluated by trace equivalence (i.e., assessing the same output from the same input) or various degrees of bisimulation [2]. For Mealy machines, bisimulation can be defined (with Q_1 and Q_2 being two compared Mealy machines as defined in Section 2.5) as [2]:

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 $q_1 \in Q_1, q_2 \in Q_2 \cdot (q_1, q_2) \in R$ must hold

   1) $\sigma \in \Sigma \cdot \lambda_1(q_1, \sigma) = \lambda_2(q_2, \sigma)$
   2) if $q_1\prime \in Post(q_1)$ then there exists $q_2\prime \in Post(q_2)$ with $(q_1\prime, q_2\prime) \in R$
   3) if $q_2\prime \in Post(q_2)$ then there exists $q_1\prime \in Post(q_1)$ with $(q_1\prime, q_2\prime) \in R$

This thesis uses behavioral equivalence for compliance checking (see Section 4.3).

2.8 Linear Temporal Logic

Linear Temporal Logic (LTL) is an extension of propositional logic [38] that allows for expressing logical temporal[2] modalities [40]. Therefore, LTL extends propositional logic with to following modalities:

- *always* ($\Box$, or G for Globally): the proposition must hold in any subsequent state
- *eventually* ($\Diamond$, or F for Finally): the proposition must hold in some (i.e., any arbitrary) subsequent state (may or may not hold before)
- *next* ($\bigcirc$, or X for neXt): the proposition must hold in the immediately subsequent state and
- *until* ($\mathcal{U}$ or U): the proposition A_1 must hold until another defined proposition A_2 occurs ($A_1 \mathcal{U} A_2$). A_1 may or may not hold *after* A_2 has occurred. In any case, A_2 has to occur at some point.

[2] In this context, *temporal* is not to be confused with *timed*. It has been formulated by Pnueli in 1977 [39]. LTL only allows statements about the modality and succession order of events to occur, not about a duration of any kind.

For an example $\Box(\neg A)$ means that atomic proposition A must never become true in the entire system. It is important to note, that all modal operators are to be seen from the perspective of the state, they are evaluated in, hence *globally* counts the current and subsequent states. For instance, $\Box(B \rightarrow \Box(\neg A))$, means that A must not become true after B occurs, since the expression after the error is only evaluated once B has become true. If there is no expression before the modal operator, it automatically counts for the initial state, which for *globally* means that the following expression counts for the entire system. $\Diamond(B)$ means that proposition B must become true in some state (not specifying which). $A\mathcal{B}$ means that, starting from the initial state, proposition A must remain true until B becomes true, not specifying what happens after. $A\mathcal{U}B$ means that proposition A must be true until proposition B becomes true. A may become false at the very moment B becomes true. It does not say what happens with A after. If A should stay false ever after, that property must be appended, which results in $A\mathcal{U}B \wedge G(B \rightarrow G(\neg A))$ If the property should be relaxed to allowing B to never occur, this property must be appended, leading to $(A\mathcal{U}B \wedge \Box(B \rightarrow \Box(\neg A)))||\Box(\neg B)$.

2.9 Model Checking

Model checking is a technique that is built on a sound fundament of graph theory, data structures, and logic to evalutate systems for defined properties to hold. [40]. It explores the complete state space of a model (e.g., an LTS). In each state, it evaluates the given set of properties to hold. These properties are often expressed as formulas in modal logic like Linear Temporal Logic (LTL), Computation Tree Logic (CTL), or the branching-time logic (CTL*). It therefore finds any undesired conditions a system can be brought into, wich is expressable in such a logic. For instance, if a resource must only be readable if authenticated, we can define a *READ* and an *AUTH* proposition and check for the (LTL) property $\Box\neg(READ \wedge \neg AUTH)$, meaning *globally*, there must not be any state where *READ* is true and *AUTH* is false. If a model checker traverses through a system and encounters any state where this property does not hold, it reports a violation. In this case, it had found a system state, where the resource is readable without being authenticated. An efficient way to perform this is automata-theoretic model checking, where the model checker negates the property and creates a (Büchi) automaton from it. The property is satisfied if the cross product of this negative automaton and the examined model automaton have only the empty word as accepted language[41].

Chapter 3

Research Overview

Formalized methods can be useful to improve structure and reproducibility of both black box and white box verification of the correctness and security of systems. This also increases their comprehensiveness and efficiency. The overall objective of this thesis is to investigate more comprehensive and efficient methods for verification through testing. This thesis therefore proposes a structured and automated way to model-based testing in order to achieve this objective at an architectural and a component level. Ultimately, the thesis proposes methods for assuring the cybersecurity of and enhancing the trust in systems with a special emphasis on communication protocols used in vehicular systems. To achieve that, the thesis is to provide a set of formal methods that facilitates the automation of test generation from threat models, as well as automatically checking implementations for specification compliance (which again requires an automatic method to derive behavioral component models). As the component behavior has to work black box, because of the reasons stated above, the model generation concentrates on outside interfaces of that component, which is generally a specific implementation of a (standardized or proprietary) communication protocol. Based on our literature survey of previous approaches in these areas, we decided to concentrate on Automata Learning [37] to infer Mealy-type state machines of the behavior of implementations of communication protocols. Then we used trace and bisimulation equivalence checking for compliance checks, as well as for LTL-based model checking [40].

Overall Objective: Facilitate automated security test generation from architectural threat models, as well as implementation/specification compliance checking by employing automata learning and model checking techniques.

3.1 Research Goals

The overall objective stated above can be divided in several sub goals each of which solves a different scientific problem. The main motivation is to likewise utilize design-phase artifacts (threat models) and implementations (utilizing formal methods) to derive tests for verification. To utilize threat models, we use approaches to directly create formal descriptions of attacks from threat models, and utilize LLMs to generate both executable test scripts in Python, and LTL-based security properties that can be used for model checking. We therefore provide a means to automatically create security tests from the same source as the requirements, both gaining efficiency through synergies and accuracy through synthesis of requirements and verification. This part (research goal RG1) fulfills the the first part of the overall objective. The second part of the objective targets black-box checking implementations for their security and compliance with specifications. We aim for automatically obtain suitable models (Mealy machines, which consider input and output of a reactive system) of these implementations (RG2) and checking them for full equivalence with its intended behavior (defined in a specification - research goal RG3) and for specific security properties (through model checking - research goal RG4). Therefore, the fulfillment all four research goals meets the overall objective. As a summary, the four resulting research goals are to:

RG1 Derive test cases, formal descriptions of cyberattacks, and security properties from a system's threat model.

RG2 Automatically obtain state machines of communication protocols from black box scenarios that can be used for correctness and security analysis.

RG3 Facilitate behavioral equivalence as a method for compliance checking of a learned implementation to a given specification (e.g., a standard).

RG4 Assemble a model suitable to be processed by a model checker, including appropriate checking rules.

The attack descriptions (RG1) provide a test scenario in the form of an abstract attack description for the overall system. This scenario is derived from threat

modeling the architectural design using a rule set, that scrutinizes potential threats based on the architecture. The state models are inferred from implemented components (RG2) and the compliance checking of these state models (RG3) provide a verdict that verifies compliance to a specification (e.g., a standard). This compliance (RG3) is assumed beforehand in the design phase during the architectural threat modeling (RG1 - that lead to the attack descriptions). A counterexample regarding the compliance provides (currently manually) input to the rule set for the threat modeling and therefore potentially leads to a different outcome of the threat modeling process. RG3 therefore provides an iterative feedback loop from the implementation back to the design phase and the threat model-based test generation in RG1. This also indirectly brings the standards specification into the threat modeling rule set. RG4 sets the scope more directly on security issues. We apply model checking to directly check the models from RG2 for security properties by converting the models into a modeling language and augmenting them with propositions. The security properties are derived by generalizing (so these properties can applied to a broad variety of different protocols), creating protocol-specific properties, and generating them from threat models (part of RG1). Figure 3.1 provides an overview of the research goals in the context of an exemplary automotive cybersecurity testing process. In this figure, amber denotes artifacts, blue denotes activities, and cyan denotes specification inputs. The arrows denote inputs and outputs, with the dashed input denotes a process including output. The research goals are marked with the dashed red boxes.

3.1.1 RG1: Threat-Model-Based Test Generation

A prominent example of model-based security analysis is the threat analysis and risk assessment (TARA) process widely used in the automotive industry [18]. It uses a threat modeling, based on an architectural design model, to identify threats and prioritize them in order to derive security goals and requirements, which ultimately results in security measures to be implemented in the architecture and components. Some kind of assessment in the fashion of a TARA (although not necessarily the exact same) is even prescribed by the automotive admission process in the UNECE region and the only recognized international standard for implementing a cybersecurity management system [15, 27]. Both admission and standard also mandate to verify cybersecurity measures by testing. In order to create these tests in an efficient manner, another goal of this thesis is to automatically derive test cases from the models made in the design phase to use it later after implementation to verify the

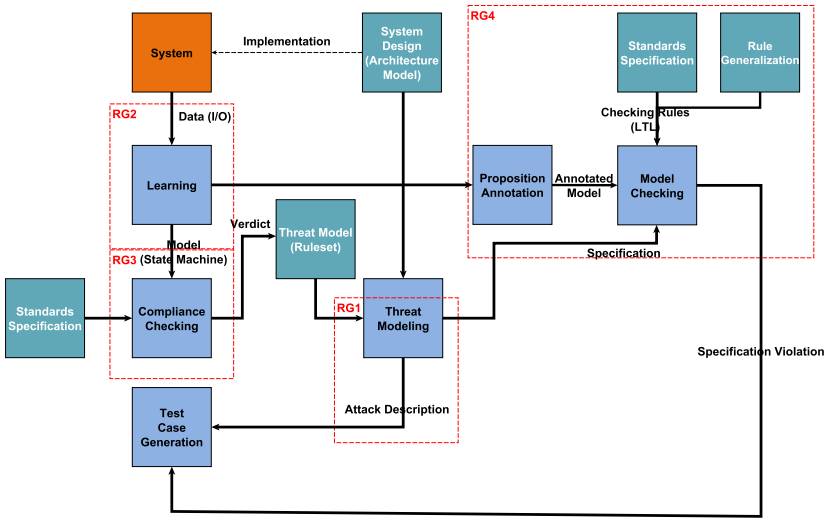

Figure 3.1: Positioning of the research goals in a structured testing process. Amber denotes artifacts, blue denotes activities, and cyan denotes specification inputs. The arrows denote inputs and outputs, with the dashed input denotes a process including output. The research goals are marked with the dashed red boxes.

efficacy of the planned security measures, using a fixed rule set [4] and using Large-Language Models (LLMs) [8]. The TARA process also determines the verification and validation planning and methods. In this process we included learning-based component testing as presented in RG2 and RG3 [1]. On the other hand, during threat modeling certain assumptions about the model elements are made (e.g., it is assumed that a component's communication complies with a certain standard) [42, 17]. If these assumptions do not hold, the model becomes inaccurate. It is therefore beneficial for the model's accuracy that the component's behavior is checked against the assumptions. When these assumptions can be formulated into a specification, the respective component's behavior can be automatically checked to comply with that specification. This behavioral compliance checking is formulated in RG3. Furthermore, we utilize LLMs to derive security properties from these models that we can utilize to check derived models for (RG4) [8].

Contributing papers: Paper I, Paper II, Paper IV, Paper VIII.

RG1: Enable deriving formal descriptions of cyberattacks from a system's threat model.

3.1.2 RG2: Automated Derivation of State Machines

Formal models have been used very broadly in both research and industrial applications. It is, however, very tedious and costly to create suitable models for correctness and security analysis manually. Furthermore, in some industries like the automotive, the necessary information to manually creating models might not be present due to very long supply chains and/or non-disclosure. It is therefore beneficial to possess a method to automatically infer formal behavioral models (i.e., state machines) of systems under test in order to foster more rigorous analysis and verification processes. As these state machines have to be derived from black box systems (due to the reasons stated above), the interfaces to interact with these systems are their respective implementations of communications protocols. These implementations are the first entry point for adversaries through faults and vulnerabilities. Inferring state machines for correctness and security analysis (as well as test generation) is therefore a significant building block for security improvement. In the course of this, it should also be examined how effective Automata Learning is to infer state machines of industrial real-world communication systems. We also introduce an approach to learn *compound automata*. With a single learning process, we infer two different protocol running on the same machine (e.g., an NFC and a BLE interface of a Tesla key fob). We can then investigate for protocol interferences by comparing the compound automaton with separately learned single-protocol automata. In an automated process, attack descriptions derived from threat models (RG1) provide the V&V planning for components that should be examined, while the actual compliance checking refers to RG3 and security property checking to RG4, respectively.
Contributing papers: Paper I, Paper II, Paper III, Paper VII.

RG2: Investigate automated methods to automatically obtain state machines of communication protocols from industrial black box scenarios to use these state machines for correctness and security analysis.

3.1.3 RG3: Compliance Checking

In many contemporary industries one of the main means for collaborations along the supply chain is written specifications and standards. These include (semi-)formal specifications like development interface agreements (DIAs), specifications in *requests for quotations (RFQs)*[1], as well as international and (de facto-) industry standards. To deliver a correct system it is crucial to comply with the respective specification. Furthermore, deviations from standards (e.g., absent or wrong answers to requests, vendor-specific or even undocumented behavior, etc.) can contain faults that might lead to security vulnerabilities. Therefore, one goal of this thesis is to create a means for compliance-checking real-world systems in an automated manner. The compliance checking is targeted to be based on state machine models (as derived in RG2), as checking an accurate state machine uncovers consistent and inconsistent behavior both more comprehensively and efficiently, and, thus, more solid than using traditional conformance checking. The specification is modeled into a state machine and its behavior compared to that of learned state machine. Thereby, the behavioral aspect of the equivalence is crucial: it is not necessary that a system's state machine is *identical* to a specification state machine, only that both state machines *behave* exactly the same. These checks also provide confirmation or rebuttal of assumptions made in threat modeling (based on specifications tailored to these assumptions) creating a link to RG1. We also use model checking to assure the security of specifications (RG4).
Contributing papers: Paper II, Paper III, Paper V, Paper VI, Paper VII.

RG3: Demonstrate the applicability of behavioral equivalence as a method for compliance checking of a learned implementation against a given specification (e.g., based on a communication protocol standard)

3.1.4 RG4: Model Checking

Deriving test cases from a black box system in a generic manner is a difficult task. Since we can gain knowledge about a system under test through derived models (RG2 - see 3.1.2), we can use this information to analyze the system for security properties using model checking (see Section 2.9). In order to use a model checker, two things are needed: a) propositions that describe the attributes of each state in the model and b) LTL formulas that describe

[1] This depends on the RFQ. An RFQ may or may not contain a detailed specification that can be used for checking an implementation's compliance with it.

the properties to check. Since the derived models are Mealy Machines, they do not intrinsically possess propositions. One sub-goal is therefore to find a formalized methodology to augment mealy machine states with propositions, that should be generally applicable for models of different settings. The second sub-goal is to find LTL formulas to perform the model checking. These could be derived from protocol standards, ideally generalizing these rules. It is therefore a goal to determine to which degree such rules can be generalized (i.e., is it possible write general rules that are able to assess the security of, e.g., the authentication mechanism of multiple protocols). Another source could be attributes of elements in threat models. This way, the intrinsic assumptions of threat models (e.g., a secure authentication mechanism) could be checked for their validity in the following implementation, creating a link to RG1. To further align the verification process with threats from the design phase, we also derive LTL properties from threat models using LLMs.
Contributing papers: Paper VI, Paper VIII, Paper IX.

RG4: Create a method to generate checkable models from learned automata and derive checkable properties reflecting system security attributes.

3.2 Research Method

This thesis follows the Design Science Research Method.(DSRM) [43]. First thoughts on Design Science were made by Simon in 1969 [44], where he asked how to scientifically scrutinize artificial artifacts of a certain complexity. Artificial in that sense means everything not being deducted strictly by (apodictic) natural laws, i.e., everything human-made, including engineering. Based on this fundament, Nunamaker et al. provided a framework for DSRM for Information Systems (IS) [45]. The framework provides a multi-methodological approach to IS research considering theory building, systems development, experimentation, observation, and their relations to each other. Their work also provides a process to research IS consisting of the following activities and underlying research issues. [45]:

1. Construct a conceptual framework
    - State a meaningful research question
    - Investigate the system functionalities and requirements
    - Understand the system building processes/procedures

- Study relevant disciplines for new approaches and ideas

2. Develop a system architecture
   - Develop a unique architecture design for extensibility, modularity, etc.
   - Define functionalities or system components and interrelationships among them
3. Analyze and design the system
   - Design the database/knowledge base schema and processes to carry out system functions
   - Develop alternative solutions and choose one solution

4 Build the system

- Learn about the concepts, framework, and design through the system building process
- Gain insight about the problems and the complexity or the system

4 Experiment, observe, and evaluate the system

- Observe the use or the system by case studies and field studies
- Evaluate the system by laboratory experiments or field experiments
- Develop new theories/models based on the observation and experimentation of the system's usage
- Consolidate experiences learned

The conceptual framework (1) was done prior to the actual research by defining the overall objective and the research goals (Section 3.1) out of the motivational identified practical problems (Section 1.1), along with studying related work (Section 5). The system architecture (2) was defined in four ways: a) as a structured overall architecture for implementing a testing process (in paper I), b) a system architecture for model learning and model-based compliance testing (papers II, III, V, and VII), c) a conceptual architecture (paper IV) and an alternative LLM-based architecture (paper VIII) for test case generation from threat models, d) techniques to utilize model checking on Mealy Machine models (papers VI and IX) and e) an meta-architecture concept for integrating the components a-c (in this thesis). System for RG1 was designed (papers IV and VIII). Various approaches were considered and examined for RGs 2 (papers I, II, III and VII), 3 (papers III, V, VI, and VII), and 4 (Paper IX). Based on these designs, system implementations were built for threat model-based test generation (RG1) in paper VIII, learning (RG2) and compliance checking (RG3) in papers III and VII, and model checking (RG4) in paper IX. These prototypes were also used for extensive system evaluation in the same papers. Figure 3.2 contains an overview of the distribution of the research process steps among the published papers, also denoting the relation to the research goals. As a summary, the concept was taken early on (prior to and in Paper I), most papers (except V and VI) contributed to a (potentially different, depending on the research goal) system architecture, design and prototyping are covered in the papers from III onward, and the evalaution is done in the last three works (except for III). The main works with regards to the research goals are Papers IV and VIII for (two different approaches of) RG1, Paper III for RGs 2 and 3

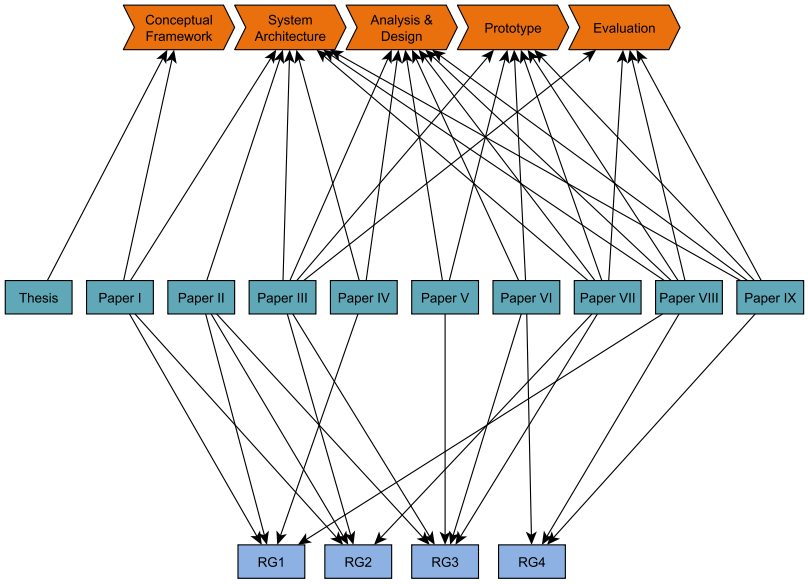

Figure 3.2: Overview of the research contributions in relation to the research process. Amber denotes the process steps, cyan the papers contributing to each step, and the (fulfilled) research goals in light blue
.

(VII is an extended version thereof) and IX for RG4. The other papers outline fundamental preliminary works or more specific applications.

Chapter 4

Research Contributions

This chapter contains the research contributions that have been made towards reaching the research goals stated in Section 3.1 and outlines the solutions, their novelty and their distribution among the publications. Table 4.1 gives an overview of the contributions of individual papers (outlined in Chapter 4.5) towards reaching the research goals.

Table 4.1: Publication contributions to the research goals.

Paper	RG1	RG2	RG3	RG4
I	X	X		
II	X	X	X	
III		X	X	
IV	X			
V			X	
VI			X	X
VII		X	X	
VIII	X			X
IX				X

For each research a different solution is presented, namely:

1 Threat Model-Based Test Generation (achieving RG1)

2 Automated Model Derivation and Algorithm Evaluation (achieving RG2)

3 Compliance Checking (achieving RG3)

4 Mealy Machine Model Checking (achieving RG4)

The threat model-based test generation (1) goes a little bit beyond RG1, as it also partially provides a V&V method selection, although the latter also is meant to contain model checking for test generation, which is reserved for future work (see Section 6.2). The automated behavior model (state machine) derivation (2) and the compliance checking (3) achieve to RG2 and RG3, respectively. The compliance checking yields a verdict that not only highlights deviations from a specification but also a counterexample that (manually) feeds back into threat model-based test generation by providing input for altering the threat modeling rule set (see Section 3.1). Analogously, model checking (4) provides a methods to achieve RG4. It provides a more targeted method to uncover cybersecurity flaws, as opposed to the compliance checking. Figure 4.1 provides an overview of the contributions in relation to the research goals and to the thesis papers. Blue boxes mark the contributions, surrounded by the research goals in red dashed lines. The cyan boxes mark previous work the contributions build on, while the dashed black boxes denote the respective papers including the contributions and previous work. The arrows indicate dependencies; solid ones indicate sub-parts and dashed ones indicate inputs.

4.1 Threat-Model-Based Test Generation

To fulfill RG1, Paper IV presents a (static, rule-based) method for transferring threat models, via attack trees and labeled transition systems (LTS), into attack descriptions conceived in a domain-specific language (DSL). The method bases on an existing threat modeling tool (actually, the graph-based representation of the attacks has the form of a Directed Acyclic Graph – DAG) [46] and an existing DSL called Agnostic domain-specific Language for the Implementation of Attacks (ALIA) [30] and concentrates on the transition between those two. ALIA is a procedural text-based language consisting of sequences of single actions (called test patterns) as a pseudo code that stand for specific steps of a composed attack in a technology-agnostic manner (not bound to a specific system-under-test). These steps will be translated into concrete executable instructions for specific systems-under-test based on Xtext and Xtend [47]. ALIA also supports pre and post conditions and flow controls like conditionals and loops. Since, the edges of an attack tree (or DAG, respectively) correspond with actions to be taken to get from one threat to another, we can attribute them to corresponding actions in ALIA (see Figure 4.2). We

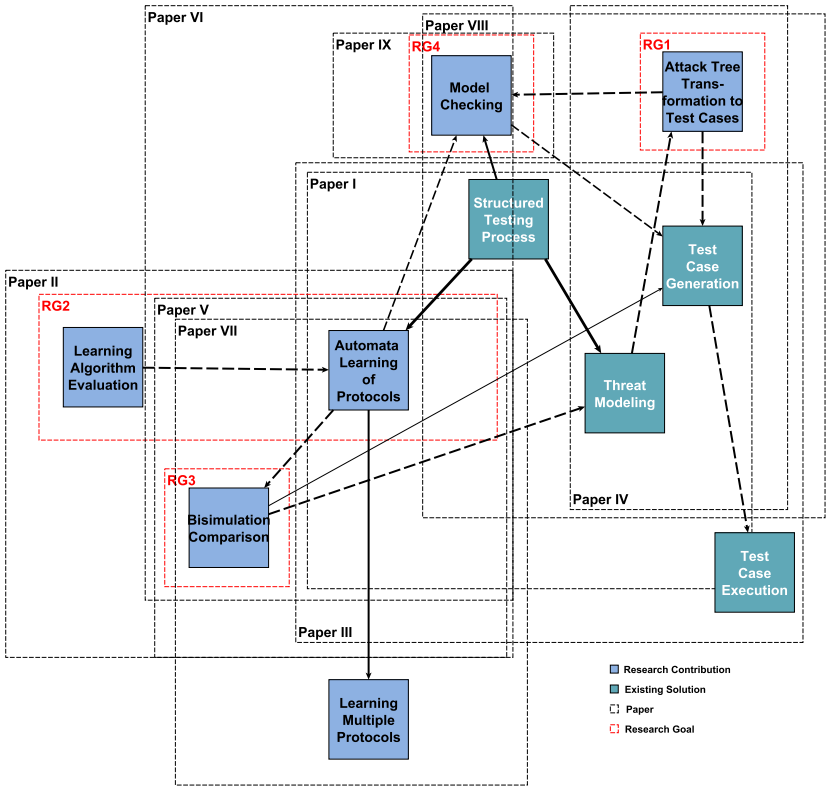

Figure 4.1: Relations of the main research contributions. Blue boxes mark the contributions, surrounded by the research goals in red dashed lines. The cyan boxes mark previous work the contributions build on, while the dashed black boxes denote the respective papers including the contributions and previous work. The arrows indicate dependencies; solid ones indicate sub-parts and dashed ones indicate inputs.

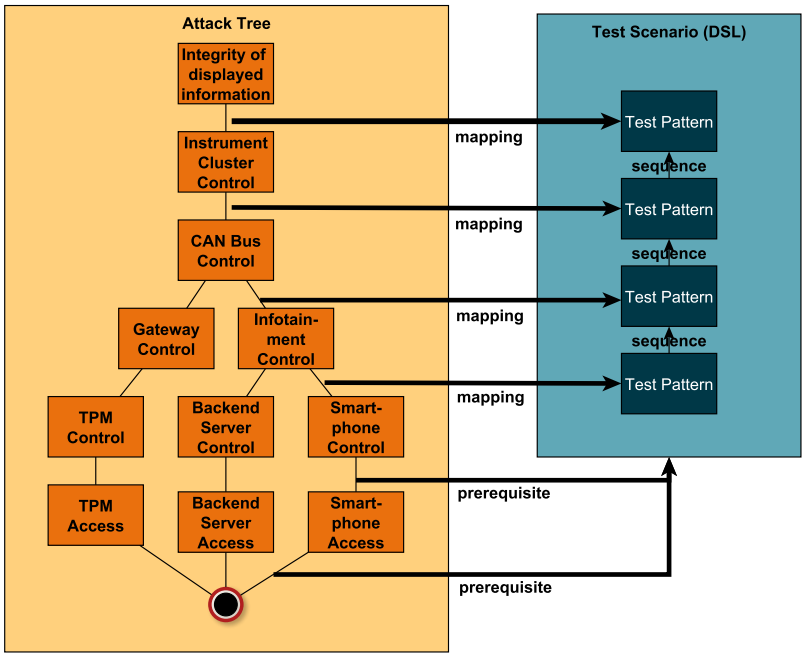

Figure 4.2: Attack tree to Test Scenario transformation example from Paper IV (modified to fit the example in Section 2.2).

Source		Target		Conn.	DSL Key
Name	SubType	Name	SubType	SubType	
Smart-phone	External Inter-actor	Info-tainment	ICU	WiFi	BBScan BBExploit OpenAndroidHotspot OpenADB
Info-tainment	ICU	CAN Bus	Wired Bus	CAN	InstallPythonEnv InstallPythonLib
CAN Bus	Wired Bus	Instru-ment Cluster	Control Unit	CAN	InstallAndroid- CANDosScript ADBPythonScript
Instru-ment Cluster	Control Unit	Dis-played Infor-mation	-	isAsset	-

Table 4.2: Example rules for translating attack trees into DSL scripts. Only the rules directly used for the mapping in Figure 4.2 are displayed. ICU means Infotainment Control Unit. That the names are only present for the reader's convenience, for the ruling the subtypes are sufficient. To obtain this information and correctly label an attack tree, we need correlate information from the attack tree (Figure 2.2) and the architectural model (Figure 2.1).

manually create a rule set that determines the action to be taken. Therefore, we categorize the threats that form the nodes of the DAG. We then define an action to be taken for each combination (i.e., an edge from a source node of type x to a target node of type y). The rule set for this actions forms the labeling function that turns the DAG into a Labeled Transitions System (LTS): $\forall q_1, q_2 \in Q, a \in ACT \cdot s_1 \xrightarrow{a} s_2 := a \times s_1 \rightarrow s_2$. With the labeling function of an LTS, the alphabet of ALIA will be attributed to paths within DAGs generated out of the threat model. Subsequently traversing the resulting LTS along a DAG's path will automatically sequence that input and generate an attack description in ALIA. From the ALIA, it is possible to generate concrete test cases, once an implementation of the architecture is ready.The rules (i.e. the partial labeling function) we apply is noted in Table 4.2. When fusing this with a basic structure template, we eventually gain an ALIA script as shown in Listing 4.1.

Paper VIII presents a method utilizing Large Language Models (LLMs) to

Listing 4.1: ALIA example of an automotive attack

```
PreConditions:
Actions:
    bbtarget: scan(type:BlueBorne, interface:BT_IF)
    bbshell: exploit(type:BlueBorne, target:bbtarget)
    wifitarget: exploit(type:OpenAndroidHotspot, target:
        bbtarget, shell:bbshell) default "XXXXX"
    adbshell: exploit(type:OpenADB, target:wifitarget)
    install_python_env: exploit(type:InstallPythonEnv,
        target:adbshell)
    install_python_lib: exploit(type:InstallPythonLib,
        shell:adbshell)
    attackScript: exploit(type:InstallAndroidCANDosScript
        , target:adbshell)
    can_attack: exploit(type:ADBPythonScript, target:
        mytarget, shell:adbshell, file:"CanAttackScript",
        interval:5)
PostConditions:
```

generate test cases from the same attack trees mentioned above. Concentrating on the automotive Unified Diagnostic Services (UDS), we therefore utilize a Retrieval-Augmented Generation (RAG) approach. This RAG consists of multiple stages of queries, particularly (1) an attack tree analyzer, (2) a document retriever that fetches relevant code parts from a vectorized database, (3) a grader that evaluates the retrieved documents and has a feedback loop to the retriever, (4) a generator that produces test cases in Python and (5) an evaluator that assesses the quality of the code an has a feedback loop to the generator (regenerating if the quality is not satisfactory). The vectorized database consists mainly of existing test cases from our AVL TestGuard test catalog and external sources that connect threats and test code [1]. Listing 4.2 shows a truncated example test case. As an alternative, we generate also generate properties in Linear Temporal Logic (LTL) with an adapted RAG. An example for a generated property from the same tree as the listing is *AUTHENTICATION ENFORCEMENT:*$\square(DIAG\_SESSION\_INIT \wedge UNAUTH \rightarrow false$[2]. This means that a UDS interactor has to first authenticate before entering a diagnostic ses-

[1] E.g., the Auto-ISAC Threat Matrix (ATM): https://atm.automotiveisac.com/

[2] This corresponds to $\square\neg(EXT \wedge \neg AUTH)$ for our annotated Mealy machines (see Section 4.4).

sion on the system. These properties can be used for model checking (RG4). However, in this case the rule is semantically wrong, since authentication in UDS is not necessary when entering such a session, but when accessing protected resources. This means that there is further research to perform to adjust the RAG in a way that it can cope with such protocol details. To improve the overall quality of both test cases and LTL properties, we use learned protocol state machines as additional input for the generator, drawing a link to RG2. Paper I describes a structured approach to derive a testing strategy from threat modeling RG1 and its connection to learning-based component testing (RG2), and Paper II describes the general embedding of threat modeling into a security testing process.

Listing 4.2: Example of an LLM-generated test case from an attack tree.

```
"test_cases": [
   {
      "id": 1,
      "name": "Bypass Physical and Logical Protections -
          Unauthorized Diagnostic Port Access",
      "description": "Tests whether an attacker with
          physical access can connect to the diagnostic
          port and initiate UDS communication without
          authentication.",
      "vulnerability_addressed": "Lack of Authentication
          on Diagnostic Interfaces",
      "setup": "# Setup instructions: ...",
      "test_code": "import can\\nimport os\\n ...",
      "teardown": "# Teardown: ...",
      "expected_result": "If the system is vulnerable,
          the ECU will  ..."
   },
   ...
]
```

This contribution contains several novelties, namely: (a) the approach of transferring a threat model into test descriptions using attack trees and LTS, (b) using attack trees in conjunction with protocol Mealy Machines as input to an LLM RAG for automated improved test case generation, and (c) utilizing an LLM RAG to derive properties for model checking.

4.2 Automated State Machine Derivation

A solution to achieve RG2 was developed by using active automata learning to infer behavioral communication protocol models (concretely state machines, also called automata) on the example of the handshake protocol (ISO 14443-3 [25]) of NFC systems. A similar system for a platooning protocol is in development (see Section 6.2). The developed solution consists of an NFC adapter interface library for the LearnLib library [48], containing the necessary adjustments (including compiling a new firmware) to an NFC hardware adapter along with an abstraction layer that transforms symbols from the learning algorithm to NFC hardware signals and vice versa. The solution also delivers insights on learning the ISO 14443-3 protocol, as it has some very characteristic features in the handshake protocol (particularly, two intertwined combination lock structures with almost identical states and the property that it does not send a response in case of a non-expected signal[3]). In this setting also, different propagated learning algorithms were evaluated for their suitability and (surprisingly) an older algorithm was found best performing to learn correct implementations (namely the L* algorithm with the closing strategy by Rivest/Schapire), while (less surprisingly) the modern TTT algorithm was best performing when it comes to detecting flaws (see next chapter). The theory for the solution was worked out in Paper I, while Paper II describes the solution concept. Paper III contains the full solution implementation with a description of the complete details for deriving a state machine of NFC handshake protocol implementations (example see Figure 4.3), including performance evaluations (see Table 4.3 for an example.

Paper VII, eventually, contains an extension to learn compound automata of multiple protocols running on the same device and gives a practical example with a device providing both an NFC and a Bluetooth Low-Engergy (BLE) interface (see Figure 4.4).

Specifically, there is quite some previous work on automata learning-based approaches for learning communications systems (even one for NFC banking cards - see Section 5), but for the ISO 14443-3 protocol, so far no automated black box-learning solution has been presented. Furthermore, the approach of inferring a compound Mealy Machine representing multiple protocols on a system in a single learning process is novel.

[3] On the other hand, it is not time sensitive. Therefore, a lot of engineering effort was put into error correction to make the learning robust. Since, we cannot learn non-deterministic automata with our approaches, timing-induced (and other) non-determinism has to be abstracted. We have, however, the opportunity to artificially re-introduce it for model checking (see Section 4.4).

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

Table 4.3: Runtime (minutes) per algorithm and maximum word length for the ISO/IEC 14443-3 protocol from Paper III.

4.3 Compliance Checking

RG3 was reached by comparing two automata using bisimulation and trace equivalence: one inferred using automata learning (see 4.2) and a second one based on a specification. While the basic concept was mentioned in Paper II, this was implemented for the NFC handshake protocol in Paper III. This way of checking the conformance is more comprehensive than traditional conformance checking trough testing, because it compares the complete behavior of a system with the complete behavior of a specification instead of merely testing a small subset in form of specific traces. The reason for using bisimulation and trace equivalence ($\sim$) instead of a much simpler full (graph) identity is that standards compliance does not require the state machines to be identical, but merely to behave the same way. A system with a deviating automaton could still behave equivalent to and therefore be compliant to a specification (or standard). Figure 4.3 shows a representative example of learned automata with an ISO/IEC 14443-3 compliant automaton and one deviating from the standard (the Tesla key fob). The deviation could easily automatically detected using this approach. We also use a similar methods to evaluate compound automata (see Section 4.2) in Paper VII. Using simulation ($\preceq$) and trace ($\subseteq$) preorder, we can evaluate whether two learned single-protocol automata (in our example NFC and BLE) are represented in a compound automaton that represents both. We can also determine if the compound automaton displays additional behavior. This is the case when a protocol (e.g., for a pairing process) utilizes both communication interfaces. This leads to additional behavior (i.e., states) in the compound automaton, that is not visible in both single-protocol counterparts. Figure 4.5 for an example, shows an additionally found τ state in a

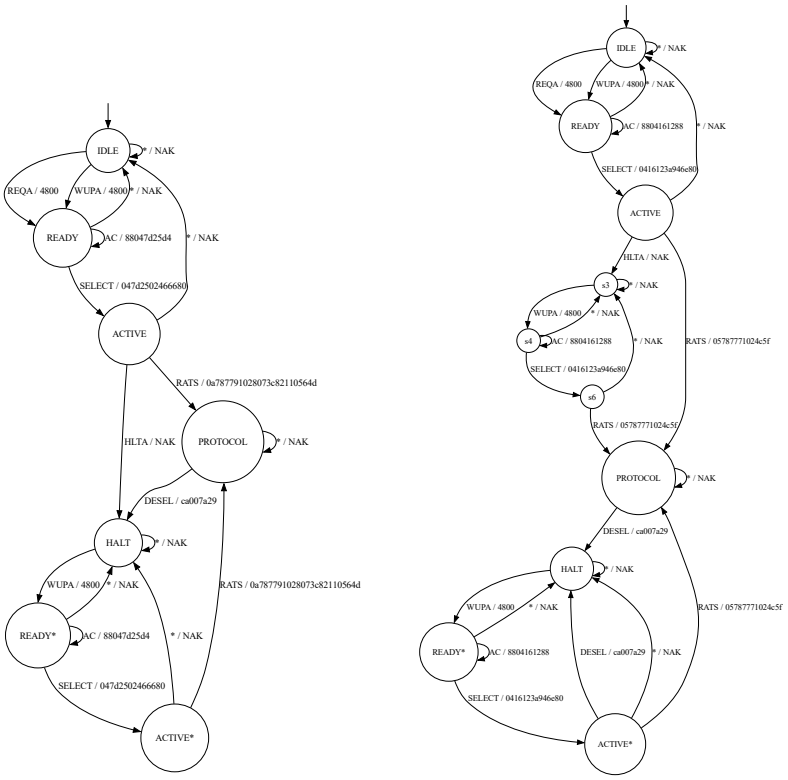

Figure 4.3: ISO/IEC 14443-3 automaton of an NXP test card (left, standard-compliant) and a Tesla car key fob (right, deviating from the standard with the transition *DESEL* in the *ACTIVE** state) learned with TTT from Paper III.

Tesla key fob's BLE behavior when learned in conjunction with NFC. We also present an approach to assure the security of the used specification automaton using model checking in a primitive form in Paper V and more elaborated in Paper VI, in which we also introduce the modeling of formal specifications in the Rebeca modeling language and converting them into a specification automaton. Since many standards are underspecified (leaving ambiguities in the desired behavior), in Papers V and VI we do not formally use Mealy machines for specifications, but non-deterministic Mealy-styled LTS. The difference is,

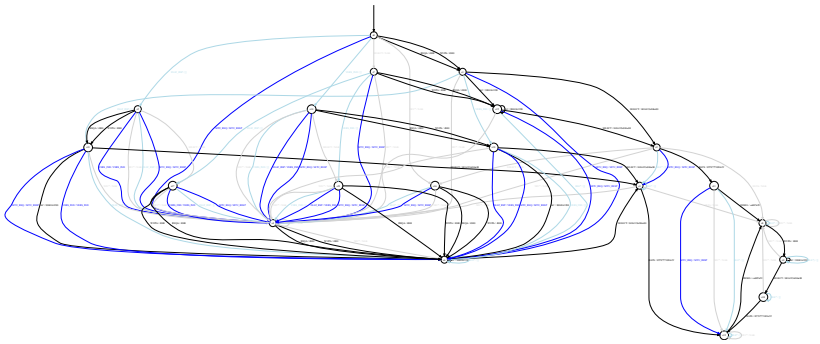

Figure 4.4: Compound NFC/BLE Tesla key fob model learned with TTT from Paper VII. Transitions in black are NFC and blue BLE, pale lines are without output (i.e., timeouts). We also used NFC* and BLE* for *all other* BLE and NFC inputs for better readability.

that we allow for non-deterministic behavior, i.e., an input symbol in a given state can have two transitions (with different goals and/or outputs). This allows us to model multiple possibilities for each input-state combination, if the respective specification leaves multiple options for behavior. Then simulation or trace preorder determines if the learned implementation's behavior is part of the specification's (i.e., if each transition of the implementation automaton is also present in the specification). Figure 4.6 shows an example for a learned model (a passport) and an automaton of an underspecified standard (the ICAO electronic machine-readable travel document standard).

There is (though few) previous work using the concept of automata learning paired with bisimulation for behavior comparison (see Section 5), however, no solution for practically working protocol conformance checking against a formal specification.

4.4 Model Checking

For RG4, we transformed the learned models (state machines) in a format that is suitable for for being checked for security properties. Since model checking is a technique that checks every state for a system and evaluates the given properties to hold (see Section 2.9), a model must fulfill two conditions: a) it must contain checkable properties and b) it must be in a format that a model

Figure 4.5: Single-protocol automaton of a Tesla key fob's BLE interface from Paper VII, learned with TTT (in blue), added (in light gray) is the additional state when learning a compound model.

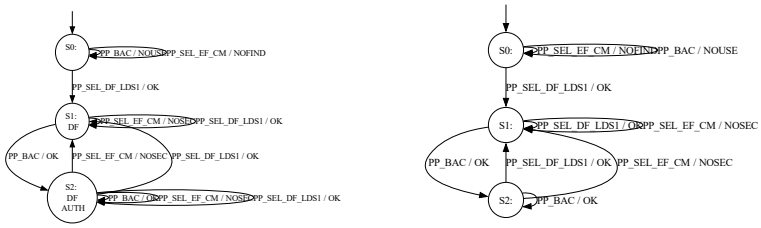

Figure 4.6: Simplified example of a Rebeca specification model converted in a non-deterministic Mealy-style LTS compared to a learned MRTD Mealy model from Paper VI. Note that the difference lies in additional transitions in the model, modeling optional behavior.

checker can interpret. With the fulfillment of RG2, we possess system models in Mealy Machine format. These models do not possess any state information (they are only labeled with a sequential numeric identifier). To fulfill condition a), we use protocol specific *Context-based Proposition Maps (CPMs)* C that define conditions for gaining (C_g) and losing (C_l) meaningful propositions, depending on the Mealy Machine's transition labels (i.e. input and output). A condition ($c \in C$) is a triplet of a set of propositions $S_P \in 2^P$, an input set $S_\Sigma \in 2^\Sigma$, and an output set $S_\Omega \in 2^\Omega$ ($\langle S_P, S_\Sigma, S_\Omega \rangle$) and P as set of all propositions in the annotated Mealy machine. For instance, passports use (among others) *basic authentication (BAC)* to authenticate a user who wants to read a protected resource. If a transition contains a symbol for *BAC* as an input and a success code (e.g., *9000* in NFC) as an output, the target sate gets the proposition authenticated (AUTH). We also define losing conditions when the system reaches a de-authenticated state (e.g., after a new session, etc.). In an automated process, we then iteratively propagate the propositions over all

transitions that do not contain a lose condition. That means that if a transition's origin state has the AUTH proposition, also the target state gets it, unless its transition label (i.e., input and output) are defined as a lose condition. This way, we annotate the complete Mealy Machine with propositions, effectively transforming it into a Kripke-like structure [49] $\mathcal{MK} = (Q, q_0, \Sigma, \Omega, , \delta, \lambda, \mathcal{L})$, which corresponds to the original Mealy machine with the function $\mathcal{L}$ attributing any combination of propositions ($S_P \in 2^P$) to each state. To fulfill condition b, we parse the graph of the resulting Kripke structure and transform it into a modeling language. We chose Rebeca for this purpose, since it is an actor-based language is designed to model reactive systems [50]. In our automated translation process, we formally define a Rebeca template with two ($\mathcal{M} := r_{sys}||r_{env}$): the system ($r_{sys}$), that displays the behavior of the Kripke structure and an environment actor (r_{env}), that provides inputs. Propositions are modeled by setting and unsetting state variables of the system actor. Certain outputs (e.g. for a successful read or write operation) are not system state propositions, but still vital for model checking. To be able to check these transition labels that contain vital information, we define *temporary propositions*, which basically represent an output rather than an actual state proposition, and are automatically cleared for following states. Formally, we split the transition into two parts, with an internal (τ) state in between. This state inherits all propositions from the origin state and additionally gets the temporary proposition. The model checker can then check for this temporary proposition when examining the τ state. We denote these temporary propositions in the CPM using a dedicated condition set (C_τ). In the Rebeca model, inputs are functions (message servers) of the system and outputs are functions (messages servers) called back based on the system state and calling input function. In the input functions the set the system variables, which are the (boolean) propositions and an integer representing the system state. Therefore, formally the system actor is $r_{sys} := \langle V_{sys}, M_{sys}, K_{sys} \rangle$, with the set of state variables $V_{sys} := \{q\} \cup P$, the method identifiers $M_{sys} := \Sigma$ and the set of known actors $K_{sys} := \{r_{env}\}$. The environment actor is $r_{env} := \langle V_{env}, M_{env}, K_{env} \rangle$, with $V_{env} := C_\tau$), $M_{env} := \Omega \cup \{req\}$ (the output symbols and an initial request function), and $K_{env} := \{r_{sys}\}$. What is left for checking the model using *RMC* (the Rebeca model checker) are properties to check. We therefore use a) general properties, b) protocol-specific properties, which represent behavior contradicting the respective standard and c) threat-model-dervied properties that check an implementation for the threats in the modeling phase being mitigated. The latter class we derive utilizing an LLM RAG system, described in Section 4.1. While the protocol-specific properties are derived case-by-case, the general

properties are derived from basic security requirements (both defined in Paper IX):

- **R1** (Authentication): GIVEN the system is in a *non-authenticated* state, WHEN a *access* operation on a *protected* resource occurs, THEN the operation *must not* return a positive response.
- **R2** (Confidentiality): GIVEN the system is in an arbitrary state, WHEN an *non-secured access* operation on a *protected* resource occurs, THEN it *should not* be successful.
- **R3** (Privilege Levels): GIVEN an *authenticated* state, WHEN the level of privilege is not sufficient, THEN a performed operation *must not* be successful.
- **R4** (Key Validity): GIVEN an *authentication* OR *secure access* operation, WHEN an actor is asked for a key, THEN providing an *invalid key must not* be successful.

From these requirements, we derive generic properties (that use propositions which are defined by the CPMs for each protocol[4]):

- **P1** (Authentication): $\Box(\neg AUTH \wedge PROT \rightarrow \neg ACCESSOK)$[5]
  A read operation on an protected resource without authentication must not be successful.
- **P2** (Confidentiality): $\Box(PROT \rightarrow \neg$ UREADOK$)$
  An unsecured read operation on a protected resource must not be successful.
- **P3** (Privilege Levels): $\Box((PRIV \rightarrow AUTH) \wedge (\neg PRIV \wedge CRIT \rightarrow \neg ACCESSOK))$
  A read operation on critical resources without having a privileged authentication must not be successful. The privileged status must be higher ($\geq$) than a normal authenticated status (implies relation).
- **P4** (Key Validity): $\Box(\neg INVKEYOK)$[6]
  An invalid key must never be accepted.

[4] E.g., a CPM for UDS defines when is a state *authenticated* in UDS (which inputs and outputs have to occur). For other protocols, it works analogously with their own CPMs.

[5] Afra does not support implications in the property file, therefore we use $\Box(\neg(\neg AUTH \wedge PROT) \vee \neg ACCESSOK)$, instead.

[6] This includes deprecated keys ($OLDKEY$) or wrong keys ($WRONGKEY$), like an all-zero key.

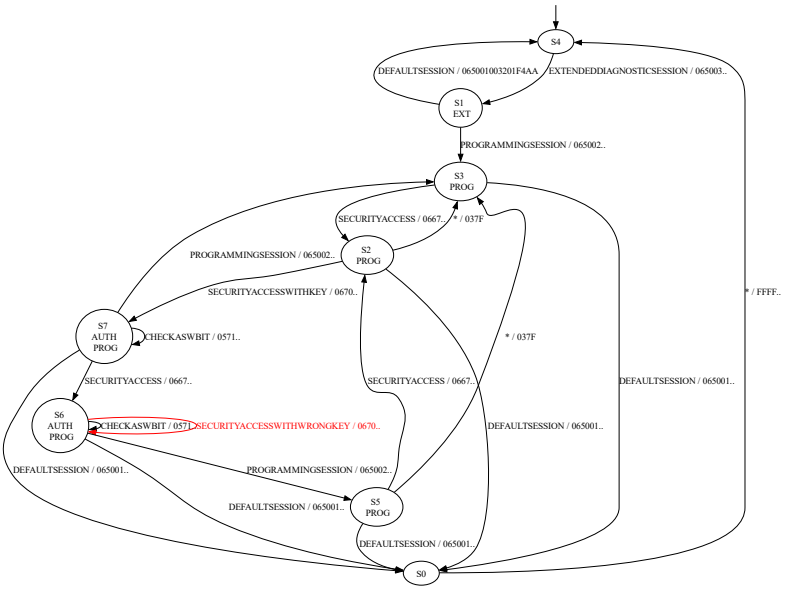

Figure 4.7: Simplified annotated Mealy machine of the learned UDS model from an automotive ECU. Self-loops that do not add to the understanding have been removed and the output truncated for readability. The star (*) denotes any other input not explicitly stated. The red transition violates property P4.

We check these properties with very different protocols, namely ICAO electronically Machine-Readable Travel Document (eMRTD) running via NFC on passports and Unified Diagnostic Services (UDS) running on an automotive control unit. In the former case, we could successfully verify these properties on different passport documents. In the latter case, we found R2/P2 being violated. This is expectable, since UDS on the can bus does not support encryption – a long known circumstance. But our specific device (from a German supplier used in car models from a Chinese manufacturer) also violates R4/P4, since it accepts wrong keys, once in an already authenticated session[7] (see Figure 4.7 for a model displaying this violation. Furthermore, we also define a slightly al-

[7]This flaw was discovered by colleagues from AVL in Paper III. Paper IX shows that the model checking approach can successfully detect this kind of flaws.

tered template that introduces non-deterministic transitions to a timeout state, which allows for checking this kind of behavior without manual modeling, despite the learning algorithms (Section 4.2) only allow inferring deterministic models.

4.5 Publications

This section contains an outline of each publication in this thesis consisting of an abstract, the work in this thesis' context, its contribution to the research goals and the author's contribution to the respective paper.

4.5.1 Paper I

Title: A Systematic Approach to Automotive Security
Authors: Masoud Ebrahimi, Stefan Marksteiner, Dejan Ničković, Roderick Bloem, David Schögler, Philipp Eisner, Samuel Sprung, Thomas Schober, Sebastian Chlup, Christoph Schmittner, and Sandra König
Abstract: We propose a holistic methodology for designing automotive systems that consider security a central concern at every design stage. During the concept design, we model the system architecture and define the security attributes of its components. We perform threat analysis on the system model to identify structural security issues. From that analysis, we derive attack trees that define recipes describing steps to successfully attack the system's assets and propose threat prevention measures. The attack tree allows us to derive a verification and validation (V&V) plan, which prioritizes the testing effort. In particular, we advocate using learning for testing approaches for the black-box components. It consists of inferring a finite state model of the black-box component from its execution traces. This model can then be used to generate new relevant tests, model check it against requirements, and compare two different implementations of the same protocol. We illustrate the methodology with an automotive infotainment system example. Using the advocated approach, we could also document unexpected and potentially critical behavior in our example systems.
Work in the thesis context: The paper outlines a structured process to verification by testing, containing threat modeling an black box-inferring behavioral models of systems using automata learning.
Contributes to research goals: RG1, RG2.
Thesis author's contribution: One equally contributing main author. Main

responsible for section 4 (security testing), contributed parts of sections 1 and 2, complete section 4.1 and parts of 4.2. This corresponds to co-developing the overall testing concept based on learning methods, describing the automata learning theory and general parts of the use cases.

4.5.2 Paper II

Title: Approaches for Automating Cybersecurity Testing of Connected Vehicles
Authors: Stefan Marksteiner, Peter Priller, and Markus Wolf
Abstract: Vehicles are on the verge building highly networked and interconnected systems with each other. This requires open architectures with standardized interfaces. These interfaces provide huge surfaces for potential threats from cyber attacks. Regulators therefore demand to mitigate these risks using structured security engineering processes. Testing the effectiveness of this measures, on the other hand, is less standardized. To fill this gap, this book chapter contains an approach for structured and comprehensive cybersecurity testing of contemporary vehicular systems. It gives an overview of how to define secure systems and contains specific approaches for (semi-)automated cybersecurity testing of vehicular systems, including model-based testing and the description of an automated platform for executing tests.
Work in the thesis context: The paper outlines a concept to automate automotive cybersecurity testing using automata learning, incorporating the results of a threat model, an a platform for automated execution based on a domain-specific language.
Contributes to research goals: RG1, RG2, RG3.
Thesis author's contribution: Main driver and main author of this paper. Contributed all content except the introductory sections 1 and 2, 3.1, 4.4 and 4.5 (delivered review for these sections). This corresponds with the main concept, an automotive life cycle testing description, a testing process and a model-based testing concept based on automata learning.

4.5.3 Paper III

Title: Using Automata Learning for Compliance Evaluation of Communication Protocols on an NFC Handshake Example
Authors: Stefan Marksteiner, Marjan Sirjani, and Mikael Sjödin

Abstract: Near-Field Communication (NFC) is a widely proliferated standard for embedded low-power devices in very close proximity. In order to ensure a correct system, it has to comply to the ISO/IEC 14443 standard. This paper concentrates on the low-level part of the protocol (ISO/IEC 14443-3) and 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 modelled after the standard, which provides a strong verdict of conformance or non-conformance. 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 automata of system that a) consist of two very similar structures and b) very frequently give no answer (i.e. a timeout as an output).
Work in the thesis context: This paper contains an examination how to efficiently infer automata of black box NFC systems using automata learning and automatically comparing the behavior (using bisimulation) to a specification automaton, therefore comprehensively assessing the standards compliance of the system-under-test.
Contributes to research goals: RG2, RG3.
Thesis author's contribution: Main driver and main author of this paper. Contributed most of the content.

4.5.4 Paper IV

Title: From TARA to Test: Automated Automotive Cybersecurity Test Generation Out of Threat Modeling
Authors: Stefan Marksteiner, Christoph Schmittner, Korbinian Christl, Dejan Ničković, Mikael Sjödin, and Marjan Sirjani
Abstract: The UNECE demands the management of cyber security risks in vehicle design and that the effectiveness of these measures is verified by testing. This mandates the introduction of industrial-grade cybersecurity testing in automotive development processes. The regulation demands also to keep the risk management current, which again creates the need of stretching the testing over the full life cycle of an automotive system. Currently, the automotive cybersecurity testing procedures are not specified or automated enough to be able to deliver tests in the amount and thoroughness needed to keep up with that regulation, let alone doing so in a cost-efficient manner. This paper introduces

an automotive security life cycle governance approach, that takes the currently being developed concepts of Cybersecurity Assurance Levels and Targeted Attack Feasibility into account and provides a means to automatically generate test cases at early development stages. These tests can also be used in later phases to verify and validate the implementations of developed systems. These formalized concepts increase the both the completeness and efficiency of automotive cybersecurity testing over vehicles' complete life cycles.
Work in the thesis context: The paper contains an approach to derive an attack tree from a threat model and a concept to transform into an agnostic attack script in a domain-specific language, with a labeled transition system (LTS) as an intermediate step, allowing for automatically creating a test case once the modeled system is implemented.
Contributes to research goals: RG1.
Thesis author's contribution: Main driver and main author of this paper. Contributed sections 1,2, 3.2, and 5. This corresponds to the paper's motivation and a concept for transforming a threat model-based attack tree into attack descriptions written in an (formal) domain-specific language.

4.5.5 Paper V

Title: Automated Passport Control: Mining and Checking Models of Machine Readable Travel Documents
Authors: Stefan Marksteiner, Marjan Sirjani, and Mikael Sjödin
Abstract: Passports are part of critical infrastructure for a very long time. They also have been pieces of automatically processable information devices, more recently through the ISO/IEC 14443 (Near-Field Communication – NFC) protocol. For obvious reasons, it is crucial that the information stored on devices are sufficiently protected. The International Civil Aviation Organization (ICAO) specifies exactly what information should be stored on electronic passports (also Machine Readable Travel Documents – MRTDs) and how and under which conditions they can be accessed. We propose a model-based approach for checking the conformance with this specification in an automated and very comprehensive manner: we use automata learning to learn a full model of passport documents and use trace equivalence and primitive model checking techniques to check the conformance with an automaton modeled after the ICAO standard. Since the full behavior is underspecified in the standard, we compare a part of the learned model and apply a primitive checking ruleset to assure proper authentication. The result is an automated (non-interactive), yet very

thorough test for compliance, despite the underspecification. This approach can also be used with other applications for which a specification automaton can be modeled and is therefore broadly applicable.
Work in the thesis context: The paper extends the conformance checking part to include underspecified standards. This allows a more hands-on usage of the approach and opens it up for application at a broader spectrum of systems to check.
Contributes to research goals: RG3.
Thesis author's contribution: Main driver and main author of this paper. Contributed most of the content.

4.5.6 Paper VI

Title: Black-box protocol testing using Rebeca and Automata Learning
Authors: Stefan Marksteiner and Mikael Sjödin
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.
Work in the thesis context: Introduces model checking of specification models and automatically generating specification automata from them. This assures specifications to be correct and bridges compliance checking (RG3) with model checking (RG4).
Contributes to research goals: RG3, RG4.
Thesis author's contribution: Main driver and main author of this paper. Contributed most of the content.

4.5.7 Paper VII

Title: Learning Single and Compound-protocol Automata and Checking Behavioral Equivalences
Authors: Stefan Marksteiner, David Schögler, Marjan Sirjani, and Mikael Sjödin
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.
Work in the thesis context: Extends the learning-based compliance checking to allow for examining multiple-protocol systems. It therefore extends the learning capabilities and utilizes the equivalence checking method to determine the differences between single-protocol and compound-protocol behavior.
Contributes to research goals: RG2, RG3.
Thesis author's contribution: Main driver and main author of this paper. Contributed most of the content, except for the BLE learner.

4.5.8 Paper VIII

Title:
Authors: Tanmay Kuhle, Stefan Marksteiner, Jose Alguindigue, Hannes Fuchs, Sebastian Fischmeister and Apurva Narayan
Abstract: In modern automotive development, security testing is critical for

safeguarding systems against increasingly advanced threats. Attack trees are widely used to systematically represent potential attack vectors, but generating comprehensive test cases from these trees remains a labor-intensive, error-prone task that has seen limited automation in the context of testing vehicular systems. This paper introduces STAF (Security Test Automation Framework), a novel approach to automating security test case generation. Leveraging Large Language Models (LLMs) and a four-step self-corrective Retrieval-Augmented Generation (RAG) framework, STAF automates the generation of executable security test cases from attack trees, providing an end-to-end solution that encompasses the entire attack surface. We particularly show the elements and processes needed to provide an LLM to actually produce sensible and executable automotive security test suites, along with the integration with an automated testing framework. We further compare our tailored approach with general purpose (vanilla) LLMs and the performance of different LLMs (namely GPT-4.1 and DeepSeek) using our approach. We also demonstrate the method of our operation step-by-step in a concrete case study. Our results show significant improvements in efficiency, accuracy, scalability, and easy integration in any workflow, marking a substantial advancement in automating automotive security testing methodologies. Using TARAs as an input for verification tests, we create synergies by connecting two vital elements of a secure automotive development process.
Work in the thesis context: Contains a more dynamic approach to generate test cases from attack trees. Furthermore, it utilizes behavior models to improve the quality of generated test and, finally, it generates properties for model checking.
Contributes to research goals: RG1.
Thesis author's contribution: Co-contributor to adapting an existing RAG to the automotive domain, contributed to prompt engineering, introduced UDS models to boost the LLM's result quality, engineered the LTL generation part.

4.5.9 Paper IX

Title: Learn, Check, Test - Security Test Generation Utilizing Automata Learning and Model Checking
Authors: Stefan Marksteiner, Marjan Sirjani, and Mikael Sjödin
Abstract: Cyber-physical systems are part of industrial systems and critical infrastructure. Therefore, they should be plumbed in a comprehensive manner to verify their correctness and security. At the same time, the complexity of

such systems demands such examinations to be systematic and, to a certain degree, automated for efficiency and accuracy. A method that provides these features is model checking. However, it requires a model that faithfully displays the behavior of the scrutinized system. This task is not trivial, as many of these systems can be examined only in black box settings, due to long supply chains or secrecy. We therefore utilize active black box learning techniques to mine behavioral models in the form of Mealy machines of such systems at runtime and translate it into a form that can be scrutinized using a model checker. On this behalf, we first annotate the model with propositions by mapping context information from the respective protocol to the model using *Context-based Proposition Maps*. We gain annotated Mealy machines that resemble Kripke structures. Creating a formally defined template, we then translate this structure into code in the Rebeca modeling language. Based on general security requirements (authentication, confidentiality, privilege levels, and key validity), we further define generic security properties that are composed of the propositions from the annotated Mealy machines. By defining the meaning of a propositions in the context of a specific protocol, CPMs assure the same propositions are present in each annotated Mealy machine, even when they represent different protocols. Using the Rebeca model checker, we can therefore subsequently check basic security requirements using the same generic properties for different protocols. Furthermore, the gained model can be easily altered to introduce non-deterministic behavior (like timeouts) or faults and examined if the properties still hold under these different conditions. Lastly, we demonstrate the versatility of the approach by providing case studies of very different systems (a passport and an automotive control unit), speaking different communication protocols (NFC and UDS), checked with the same tool chain and the same security properties.

Contributes to research goals: RG4.

Thesis author's contribution: Main driver and main author of this paper. Contributed most of the content.

Chapter 5

Related Work

This section considers other work in this field and adjacent fields. Given the research goals and contributions it is divided into contributions made in different fields namely: model-based test generation, automated state machine derivation and protocol learning, conformance checking, and applications of automata learning to cybersecurity.

5.1 Model-Based Test Case Generation

Model-based testing (MBT) uses a model representation (normally behavioral, but also structural or other kinds) of a system-under-test. Model-based test case generation is an automated test case generation based on model-based testing. It is used in very diverse application domains like Information and Communications Technology (ICT), Automotive, Consumer electronic, Railway, Aerospace, Avionic, Tourism, Agriculture, Finance, Management, Construction, Sport, Automation. The used models include broad variety of different types (like state machines, activity and sequence diagrams, Simulink models, pre/post models, Simulink models, etc.) and the approaches to generate tests include structural coverage (based on control-flow, data-flow, transitions or UML), data coverage (boundary values, statistical or pairwise testing), fault and requirements-based criteria, and explicit and statistical test generation, state, search; model checking, requirements, event, random-based and others [51, 52].

5.2 Attack-Trees-Based Security Testing

Attack trees are a formal (graphical) representation of the set of possibilities to attack a certain system and have been described in the late 1990ies [19, 20]. They connect specific small attacks (i.e., exploiting threats) to a system in order to attack a complete system or a specific target inside a system with a combined or concatenated attack. The single attacks can be underlaid with different information like necessary skills or features or a success probability, allowing for also calculating this information (i.e., a complete set of skills or features or the combined attack success probability) for the complete attack. It further allows for selecting different paths through that tree that are most efficient regarding defined criteria (e.g., maximized success probability). There is also an approach that combines security-related attack trees with safety-related fault trees and also provides a translation mechanism to transfer them into stochastic timed automata. These can be analyzed using model checking [53]. This can be used to generate test cases. More directly, attack trees have been used to build fault injection-based attacks that can be used directly onto a system-under-test [54]. There is also work to adopt attack trees for automotive systems [55]. There is also an automotive-related method to create attack trees from threat models [42]. The thesis builds upon the later work by providing a translation mechanism from attack trees into a formal attack description language that provides blueprints for cyberattacks in RG1 (Section 3.1.1).

5.3 Formalized Test Descriptions

There is quite extensive work on languages for describing attacks to computer systems [56, 57, 58, 59, 60, 61]. However, this thesis builds upon a domain-specific language (DSL) tailored for automating attacks on automotive systems called *Agnostic domain-specific Language for the Implementation of Attacks (ALIA)* co-authored by the thesis author [30]. The language concept stems from the principle to abstract attacks on specific automotive systems from their (proprietary) technology-specific traits, leaving a blueprint structure for an attack that needs to be concretized again to be executed against a different system. The ALIA DSL is therefore designed for describing attacks on automotive systems in a technology-agnostic way. Apart from the original intention of porting attacks from one (proprietary) system to another, this allows for specifying attacks at design time and concretizing them once an implementation is available. This thesis integrates this language as a formal description for attacks to

achieve RG1 (Section 3.1.1).

5.4 Automated State Machine Derivation and Protocol Learning

One of the key elements for fully automating model-based test case generation is automatically obtaining a suitable model to analyze. Finite state machines have been frequently used for correctness analyses [62, 63, 64, 65] possibilities to analyze them for their correctness and security properties. There are various approaches to automatically inferring (i.e., learning) state machines. Recurrent networks have been used to learn state machines already in the early 1990ies [66]. Some algorithms work on steering learning from traces by using a two-stage approach. They first analyze traces and mine a rule set and secondly using the rule set for learning automata from traces [67]. Others impose constraints on learning using linear temporal logic [68]. Many of the trace-based inferring methods base on the *KTail* algorithm [67]. This algorithm has been defined already 1972 by Biermann and Feldman [69]. Trace-based mechanisms are also used to generate other models like sequence diagrams [70]. Since the aim of this thesis is to black box-learning behavioral models of real-world systems, it concentrates on approaches actively querying a system. A method for this that has made many advances in the recent years is automata learning (for the basics see Section 2.6). There is quite some work of using automata learning for security analysis and testing, specifically for learning communication protocols [71, 72, 73, 74, 75, 76]. This also includes NFC but concentrates on the upper layer (ISO/IEC 14443-4) protocol, dodging the specific challenges of the handshake protocol [77]. Aichernig et al. provided a benchmark for different automata learning setups using existing benchmark data [78]. This thesis also provides and automata learning performance evaluation, which is however very specially tailored for the ISO/IEC 14443-3 protocol, with accordingly different results. Recent works also concentrated on making use of these techniques for practical use, e.g., for security analyses [79, 76] or model-based fuzz testing [75]. The thesis differentiates from these works by combining learning with compliance checking and also using this to checking assumptions in threat modeling.

5.5 Conformance Checking Using Equivalence of State Machines

There are, partly theoretic, approaches of learning a state machine and comparing it with other ones, targeting target DFAs [80] or probabilistic transition systems (PTS) [81]. For Mealy type machines, which (through their input and output behavior modeling) are better suited for describing reactive systems, Neider et al. provided some fundamental work, using automata learning and bisimulation [82]. Similar things were put into practice by viewing different machines as labeled transition systems (LTS) for model comparison [83, 84] and to verify inferred embedded control software models [85]. The closest work to ours is from Schuts et al., which uses Mealy machine learning and (trace and bisimulation) equivalence to compare the behavior of legacy industry system before and after refactoring [86]. However, there is no known comprehensive approach for using bisimulation for protocol compliance checking against a formal specification, which is the differentiation mark of this thesis compared to the described approaches (RG3 - Section 3.1.3)

5.6 Model Checking

Peled et al. [41] have provided a very influential paper regarding black box testing that combines automata learning with model checking. They propose a variant with a pre-learned model and a combined learning and testing approach. The latter approach is improved by Groce et al. [87] and Shijubo et al. [88] with adaptions in integrating model checking into the learning process to improve its performance. Our work is based on a similar idea, and we extended the approach by annotating Mealy machines with atomic propositions creating Kripke-like structures that we turned into Rebeca code for model checking. That way, we implemented all the phases for going through learning, checking, and testing real-world examples with an automated tool chain. Neider and Jansen used a symbolic approach to learn and model-check DFAs (but not Mealy or More machines) [89]. Similarily, Fiterau-Brostean et al. [90, 91] also used a symbolic approach (using the NuSMV model checker). Translating the learned model to Rebeca, our approach provides more advanced possibilities to manipulate the model. Since in Rebeca a model's behavior is defined similar to a programming language, altering it is trivial. Furthermore, Rebeca possess an operator for non-determinism (the ? operator). This allows for reintroducing some non-deterministic behavior that is usually abstracted away

during the learning process (since the used learning algorithms can only handle deterministic state machines).

Chapter 6

Conclusion and Future Work

This chapter summarizes the work included in the Licentiate thesis and outlines further directions to go from the current status of the research done in the thesis and generally in the research field.

6.1 Conclusions

The research described in this thesis aims for facilitating the usage of formal methods for generating tests to assure correctness and security with a focus on the automotive domain and on communication protocols. Following the need of the domain, we concentrated on generating test cases from the security analysis during the design phase, namely Threat Analysis and Risk Assessment (TARA) process and from the implementation, namely by checking the implementation's compliance with a specification using automata learning, as well as annotating the models with propositions and deriving properties for model checking. The latter part provides feedback for the design phase: since TARA models systems components with given properties that are based on assumptions about a later implementation (e.g., conforming to international communication protocol standards), the actual compliance to a specification can prove these assumptions to be correct or incorrect. The first part leads to RG1, which is creating attack descriptions out of threat models, which can be used to generate concrete test cases for automotive systems once systems are implemented (Section 3.1.1). The second part is twofold, first we aim for mining a suitable model for security and correctness analysis (RG2 - Section 3.1.2) and sec-

ond, we aim for a suitable methodology to use a behavioral equivalence with a specification as means for compliance checking (RG3 - 3.1.3). Furthermore, we aim for annotating the model with propositions and deriving properties in order to bring the model in a format that allows for verifying its security with a model checker (RG4 - 3.1.4.

Each of these research goals is met with a respective contribution namely, a method for test generation based on threat models (Section 4.1). This occurs by generating technology-agnostic test specifications written in the Agnostic domain-specific Language for the Implementation of Attacks (ALIA) out of attack trees derived from an existing tool for TARA (ThreatGuard) using labeled transition systems (LTS) as a means for the transformation. This approach is, to the best of our knowledge, novel. Alternatively, we utilized a RAG-based LLM system to create a) test scripts (in python) from attack trees (or DAGs) and b) derive LTL properties for model checking. Including learned protocol models into the RAG improved the quality of the generated tests. The second goal is met by automated state machine derivation based on active automata learning (Section 4.2). We showed the practical use of this technique by deriving state machines of Near-Field Communication (NFC) system for correctness and security analyses. We also provide insights on setups, abstraction and performance evaluations of different algorithms in special settings. The third goal was matched by a compliance checking method (Section 4.3). This method compares the behavior two state machines; one learned from an implementation and one modeled after a specification (e.g., the ISO 14443-3 standard). We therefore use bisimulation and trace equivalence, which in combination with automata learning is novel for protocol conformance checking. The fourth goal was achieved by a method that converts learned Mealy Machines into a checkable format (Section 4.4). Using *context maps* (rules that determine when a certain proposition or property occurs) that determine the semantics of a given protocol, we can annotate the model with atomic propositions (if necessary) and create appropriate model checking rules. We showed that these rules can be generalized to a certain extent, so that a set of rules can be used for checking multiple different communication protocols.

6.2 Future Directions

Despite the efforts taken in this thesis so far, quite some closely related problems have been left open to fulfill the overall objective in its entirety. Some of these directions are:

- Further implementations of adapters for specific protocols using the same, general learning framework. E.g., dealing with the specifics of V2X protocols or the reader parts of NFC systems in order to create a comprehensive multi-protocol learning framework. The research goal is to create a generally applicable method for protocol model inference.
- Utilize the learned models for fuzz test generation using different strategies based on node and transition properties of the learned models. The research goal is to create highly efficient approaches for fuzz testing in order to create effective zero-input testing methods. We propose a variety of strategies:
  - Use Galois lattices [92] to build equivalence groups for fuzzing
  - Averaging fuzz tests in different states (i.e., provide an equal distribution of fuzz tests over all system states).
  - Fuzzing inputs with high probability of inducing state changes
  - Fuzz inputs inside states that are frequent target states
  - Fuzz inputs inside states that have the highest number of successor states
- Utilize the learning and model checking framework to more specifically look for attack pattern (like downgrade attacks, old key usage, etc.). Therefore, we define needed input symbols for the learner, context maps for transferring in a checkable model, and LTL rules to check for the attacks.
- Implement a timed version of the tool chain to deal with time-sensitive protocols (e.g., use learning concepts like MM1Ts [93] and model checking with timed Rebeca).

Bibliography

[1] M. Ebrahimi, S. Marksteiner, D. Ničković, R. Bloem, D. Schögler, P. Eisner, S. Sprung, T. Schober, S. Chlup, C. Schmittner, and S. König, "A Systematic Approach to Automotive Security," in *Formal Methods* (M. Chechik, J.-P. Katoen, and M. Leucker, eds.), vol. 14000 of *Lecture Notes in Computer Science*, (Cham), pp. 598–609, Springer International Publishing, 2023.

[2] S. Marksteiner, P. Priller, and M. Wolf, "Approaches For Automating Cybersecurity Testing Of Connected Vehicles," in *Intelligent Secure Trustable Things* (M. Karner, J. Peltola, M. Jerne, L. Kulas, and P. Priller, eds.), Studies in Computational Intelligence, Springer Nature, 2023.

[3] S. Marksteiner, M. Sirjani, and M. Sjödin, "Using Automata Learning for Compliance Evaluation of Communication Protocols on an NFC Handshake Example," in *Engineering of Computer-Based Systems* (J. Kofroň, T. Margaria, and C. Seceleanu, eds.), vol. 14390 of *Lecture Notes in Computer Science*, (Cham), pp. 170–190, Springer Nature Switzerland, 2024.

[4] S. Marksteiner, C. Schmittner, K. Christl, D. Nickovic, M. Sjödin, and M. Sirjani, "From TARA to Test: Automated Automotive Cybersecurity Test Generation Out of Threat Modeling," in *Proceedings of the 7th ACM Computer Science in Cars Symposium*, CSCS '23, (New York, NY, USA), pp. 1–10, Association for Computing Machinery, Dec. 2023.

[5] S. Marksteiner, M. Sirjani, and M. Sjödin, "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, (New York, NY, USA), pp. 1–8, Association for Computing Machinery, July 2024.

[6] S. Marksteiner and M. Sjödin, "Black-box protocol testing using Rebeca and Automata Learning," in *Rebeca for Actor Analysis in Action: Essays Dedicated to Marjan Sirjani on the Occasion of Her 60th Birthday* (E. A. Lee, M. R. Mousavi, and C. Talcott, eds.), pp. 212–235, Cham: Springer Nature Switzerland, 2025.

[7] S. Marksteiner, D. Schögler, M. Sirjani, and M. Sjödin, "Learning single and compound-protocol automata and checking behavioral equivalences," *International Journal on Software Tools for Technology Transfer*, vol. 27, pp. 35–52, Feb. 2025.

[8] T. Kuhle, S. Marksteiner, J. Alguindigue, H. Fuchs, S. Fischmeister, and A. Narayan, "STAF: Leveraging LLMs for Automated Attack Tree-Based Security Test Generation," in *23er escar Europe : The World's Leading Automotive Cyber Security Conference (Frankfurt, 05. - 06.11.2025)*, 2025. Accepted, not yet published, arXiv preprint under DOI 10.48550/arXiv.2509.20190.

[9] S. Marksteiner, M. Sjödin, and M. Sirjani, "Learn, Check, Test - Security Test Generation Utilizing Automata Learning and Model Checking," *Computers & Security*, 2025. Submitted, not yet accepted, arXiv preprint under DOI 10.48550/arXiv.2509.22215.

[10] T. Blazek, F. Ademaj, S. Marksteiner, P. Priller, and H.-P. Bernhard, "Wireless Security in Vehicular Ad Hoc Networks: A Survey," *SAE International Journal of Connected and Automated Vehicles*, vol. 6, Aug. 2022.

[11] A. Roberts, S. Marksteiner, M. Soyturk, B. Yaman, and Y. Yang, "A Global Survey of Standardization and Industry Practices of Automotive Cybersecurity Validation and Verification Testing Processes and Tools," *SAE International Journal of Connected and Automated Vehicles*, vol. 7, Nov. 2023.

[12] D. Zhang., N. Azad., S. Fischmeister., and S. Marksteiner., "Zeroth-order optimization attacks on deep reinforcement learning-based lane changing algorithms for autonomous vehicles," in *Proceedings of the 20th International Conference on Informatics in Control, Automation and Robotics - Volume 1: ICINCO*, pp. 665–673, SciTePress / INSTICC, 2023.

[13] A. Sharma, A. Narayan, N. L. Azad, S. Fischmeister, and S. Marksteiner, "AVATAR: Autonomous Vehicle Assessment through Testing of Adversarial Patches in Real-time," *IEEE Transactions on Intelligent Vehicles*, pp. 1–14, 2024.

[14] M. Sirjani, E. Lee, Z. Moezkarimi, B. Pourvatan, B. Johansson, S. Marksteiner, and A. Papadopoulos, "Actors for Timing Analysis of Distributed Redundant Controllers," in *Concurrent Programming, Open Systems and Formal Methods: Essays Dedicated to Gul Agha to Celebrate His Scientific Career* (J. Meseguer, C. Varela, and N. Venkatasubramanian, eds.), no. 16120 in Lecture Notes in Computer Science, pp. 189–214, Springer Nature Switzerland, 2026.

[15] United Nations Economic and Social Council - Economic Commission for Europe, "Uniform provisions concerning the approval of vehicles with regards to cyber security and cyber security management system," Regulation "155", United Nations Economic and Social Council - Economic Commission for Europe, Brussels, 2021.

[16] M. H. ter Beek, R. Chapman, R. Cleaveland, H. Garavel, R. Gu, I. ter Horst, J. J. A. Keiren, T. Lecomte, M. Leuschel, K. Y. Rozier, A. Sampaio, C. Seceleanu, M. Thomas, T. A. C. Willemse, and L. Zhang, "Formal Methods in Industry," *Form. Asp. Comput.*, vol. 37, pp. 7:1–7:38, Dec. 2024.

[17] A. Shostack, *Threat Modeling: Designing for Security*. Indianaplois, IN: John Wiley & Sons, 2014.

[18] D. Ward, I. Ibarra, and A. Ruddle, "Threat Analysis and Risk Assessment in Automotive Cyber Security," *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, vol. 6, no. 2013-01-1415, pp. 507–513, 2013.

[19] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," in *Proceedings of the 1998 Workshop on New Security Paradigms*, (New York, NY, USA), pp. 71–79, ACM, 1998.

[20] B. Schneier, "Attack trees," *Dr. Dobb's journal*, vol. 24, no. 12, pp. 21–29, 1999.

[21] R. M. Keller, "Formal verification of parallel programs," *Communications of the ACM*, vol. 19, pp. 371–384, July 1976.

[22] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, Nov. 1987.

[23] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," in *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, (New York, NY, USA), pp. 411–420, Association for Computing Machinery, Feb. 1989.

[24] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), Lecture Notes in Computer Science, (Cham), pp. 307–322, Springer International Publishing, 2014.

[25] 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.

[26] Bluetooth SIG, "Bluetooth Specification," Core Specification v5.4, Bluetooth SIG, 2023.

[27] International Organization for Standardization and Society of Automotive Engineers, "Road Vehicles – Cybersecurity Engineering," ISO/SAE Standard "21434", International Organization for Standardization, 2021.

[28] VDA QMC Project Group 13, "Automotive SPICE - Process Reference and Assessment Model for Cybersecurity Engineering," Core Specification 1.0, Quality Management Center of the German Association of the Automotive Industry, 2021.

[29] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. USA: Prentice Hall PTR, 2nd ed., Feb. 2002.

[30] C. Wolschke, S. Marksteiner, T. Braun, and M. Wolf, "An Agnostic Domain Specific Language for Implementing Attacks in an Automotive Use Case," in *The 16th International Conference on Availability, Reliability and Security*, ARES 2021, (New York, NY, USA), pp. 1–9, Association for Computing Machinery, Aug. 2021.

[31] S. Marksteiner, "Verfahren und Vorrichtung zur Konformitätsprüfung eines Kommunikationsgerätes," Austrian Patent AT527044, AVL List GmbH, Austria, 2023.

[32] T. R. Ingoldsby, "Attack tree-based threat risk analysis," tech. rep., Amenaza Technologies Limited, 2021.

[33] M. Fowler and R. Parsons, *Domain-Specific Languages*. Pearson Education, Sept. 2010.

[34] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, pp. 1045–1079, Sept. 1955.

[35] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, Nov. 1987.

[36] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), Lecture Notes in Computer Science, (Cham), pp. 307–322, Springer International Publishing, 2014.

[37] F. Vaandrager, "Model learning," *Communications of the ACM*, vol. 60, pp. 86–95, Jan. 2017.

[38] B. Russell, "Mathematical Logic as Based on the Theory of Types," *American Journal of Mathematics*, vol. 30, no. 3, pp. 222–262, 1908.

[39] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57, Oct. 1977. ISSN: 0272-5428.

[40] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, Apr. 2008.

[41] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black Box Checking," in *Formal Methods for Protocol Engineering and Distributed Systems: FORTE XII / PSTV XIX'99 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX) October 5– 8, 1999, Beijing, China* (J. Wu, S. T. Chanson, and Q. Gao, eds.), IFIP Advances in Information and Communication Technology, pp. 225–240, Boston, MA: Springer US, 1999.

[42] S. Chlup, K. Christl, C. Schmittner, A. M. Shaaban, S. Schauer, and M. Latzenhofer, "THREATGET: towards automated attack tree analysis for automotive cybersecurity," *Inf.*, vol. 14, no. 1, p. 14, 2023.

[43] A. Hevner and S. Chatterjee, "Design Science Research in Information Systems," in *Design Research in Information Systems: Theory and Practice* (A. Hevner and S. Chatterjee, eds.), Integrated Series in Information Systems, pp. 9–22, Boston, MA: Springer US, 2010.

[44] H. A. Simon, *The Sciences of the Artificial*. Cambridge, MA, US: MIT press, 1969.

[45] J. F. Nunamaker, M. Chen, and T. D. Purdin, "Systems Development in Information Systems Research," *Journal of Management Information Systems*, vol. 7, pp. 89–106, Dec. 1990.

[46] C. Schmittner, B. Schrammel, and S. König, "Asset Driven ISO/SAE 21434 Compliant Automotive Cybersecurity Analysis with ThreatGet," in *Systems, Software and Services Process Improvement* (M. Yilmaz, P. Clarke, R. Messnarz, and M. Reiner, eds.), Communications in Computer and Information Science, (Cham), pp. 548–563, Springer International Publishing, 2021.

[47] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, Aug. 2016.

[48] M. Isberner, F. Howar, and B. Steffen, "The Open-Source LearnLib," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), Lecture Notes in Computer Science, (Cham), pp. 487–495, Springer International Publishing, 2015.

[49] S. A. Kripke, "A completeness theorem in modal logic," *The Journal of Symbolic Logic*, vol. 24, pp. 1–14, Mar. 1959.

[50] M. Sirjani, "Rebeca: Theory, Applications, and Tools," in *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures* (F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, eds.), vol. 4709 of *Lecture Notes in Computer Science*, pp. 102–126, Springer, 2006.

[51] M. L. Mohd-Shafie, W. M. N. W. Kadir, H. Lichter, M. Khatibsyarbini, and M. A. Isa, "Model-based test case generation and prioritization: A systematic literature review," *Software and Systems Modeling*, vol. 21, pp. 717–753, Apr. 2022.

[52] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Elsevier, July 2010.

[53] R. Kumar and M. Stoelinga, "Quantitative security and safety analysis with attack-fault trees," in *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, (Singapore), pp. 25–32, IEEE, 2017.

[54] A. Morais, E. Martins, A. Cavalli, and W. Jimenez, "Security protocol testing using attack trees," in *2009 International Conference on Computational Science and Engineering*, vol. 2, pp. 690–697, 2009.

[55] K. Karray, J.-L. Danger, S. Guilley, and M. Abdelaziz Elaabid, "Attack tree construction and its application to the connected vehicle," in *Cyber-Physical Systems Security*, pp. 175–190, Cham: Springer International Publishing, 2018.

[56] C. Michel and L. Mé, "Adele: An attack description language for knowledge-based intrusion detection," in *Trusted Information* (M. Dupuy and P. Paradinas, eds.), (Boston, MA), pp. 353–368, Springer US, 2001.

[57] M. Yampolskiy, P. Horváth, X. D. Koutsoukos, Y. Xue, and J. Sztipanovits, "A language for describing attacks on cyber-physical systems," *International Journal of Critical Infrastructure Protection*, vol. 8, pp. 40 – 52, 2015.

[58] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Chapter one - security testing: A survey," in *Advances in Computers* (A. Memon, ed.), vol. 101, pp. 1 – 51, Elsevier, 2016.

[59] P. X. Mai, F. Pastore, A. Goknil, and L. C. Briand, "A natural language programming approach for requirements-based security testing," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 58–69, 2018.

[60] P. Johnson, R. Lagerström, and M. Ekstedt, "A meta language for threat modeling and attack simulations," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, (New York, NY, USA), Association for Computing Machinery, 2018.

[61] S. Katsikeas., P. Johnson., S. Hacks., and R. Lagerström., "Probabilistic modeling and simulation of vehicular cyber attacks: An application of

the meta attack language," in *Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*, pp. 175–182, INSTICC, SciTePress, 2019.

[62] R. Alur and M. Yannakakis, "Model checking of hierarchical state machines," *ACM SIGSOFT Software Engineering Notes*, vol. 23, pp. 175–188, Nov. 1998.

[63] K. Winter, *Model checking abstract state machines*. PhD thesis, Technische Universität Berlin, 2001.

[64] C. K. F. Tang and E. Ternovska, "Model Checking Abstract State Machines with Answer Set Programming," in *Logic for Programming, Artificial Intelligence, and Reasoning* (G. Sutcliffe and A. Voronkov, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 443–458, Springer, 2005.

[65] F. Tsarev and K. Egorov, "Finite state machine induction using genetic algorithm based on testing and model checking," in *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '11, (New York, NY, USA), pp. 759–762, Association for Computing Machinery, July 2011.

[66] Z. Zeng, R. M. Goodman, and P. Smyth, "Learning Finite State Machines With Self-Clustering Recurrent Networks," *Neural Computation*, vol. 5, pp. 976–990, Nov. 1993.

[67] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, (New York, NY, USA), pp. 345–354, Association for Computing Machinery, Aug. 2009.

[68] N. Walkinshaw and K. Bogdanov, "Inferring Finite-State Models with Temporal Constraints," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 248–257, Sept. 2008.

[69] A. W. Biermann and J. A. Feldman, "On the Synthesis of Finite-State Machines from Samples of Their Behavior," *IEEE Transactions on Computers*, vol. C-21, pp. 592–597, June 1972.

[70] M. McGavin, T. Wright, and S. Marshall, "Visualisations of execution traces (vet) an interactive plugin-based visualisation tool," in *Proceedings of the 7th Australasian User interface conference-Volume 50*, pp. 153–160, 2006.

[71] C. Y. Cho, D. Babi ć, E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, (New York, NY, USA), pp. 426–439, Association for Computing Machinery, Oct. 2010.

[72] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias, "SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 1690–1701, Association for Computing Machinery, Oct. 2016.

[73] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of SSH implementations," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, (New York, NY, USA), pp. 142–151, Association for Computing Machinery, July 2017.

[74] P. Fiterău-Broştean and F. Howar, "Learning-Based Testing the Sliding Window Behavior of TCP Implementations," in *Critical Systems: Formal Methods and Automated Verification* (L. Petrucci, C. Seceleanu, and A. Cavalcanti, eds.), Lecture Notes in Computer Science, (Cham), pp. 185–200, Springer International Publishing, 2017.

[75] B. K. Aichernig, E. Muškardin, and A. Pferscher, "Learning-Based Fuzzing of IoT Message Brokers," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 47–58, Apr. 2021.

[76] I. Karim, A. A. Ishtiaq, S. R. Hussain, and E. Bertino, "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, May 2023.

[77] F. Aarts, J. De Ruiter, and E. Poll, "Formal Models of Bank Cards for Free," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pp. 461–468, Mar. 2013.

[78] B. K. Aichernig, M. Tappler, and F. Wallner, "Benchmarking Combinations of Learning and Testing Algorithms for Automata Learning," *Formal Aspects of Computing*, June 2023.

[79] A. Pferscher and B. K. Aichernig, "Fingerprinting and analysis of Bluetooth devices with automata learning," *Formal Methods in System Design*, May 2023.

[80] Y.-F. Chen, C.-D. Hong, A. W. Lin, and P. Rümmer, "Learning to prove safety over parameterised concurrent systems," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, pp. 76–83, Oct. 2017.

[81] C.-D. Hong, A. W. Lin, R. Majumdar, and P. Rümmer, "Probabilistic Bisimulation for Parameterized Systems," in *Computer Aided Verification* (I. Dillig and S. Tasiran, eds.), Lecture Notes in Computer Science, (Cham), pp. 455–474, Springer International Publishing, 2019.

[82] D. Neider, R. Smetsers, F. Vaandrager, and H. Kuppens, "Benchmarks for Automata Learning and Conformance Testing," in *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday* (T. Margaria, S. Graf, and K. G. Larsen, eds.), Lecture Notes in Computer Science, pp. 390–416, Cham: Springer International Publishing, 2019.

[83] F. Aarts, H. Kuppens, J. Tretmans, F. Vaandrager, and S. Verwer, "Improving active Mealy machine learning for protocol conformance testing," *Machine Learning*, vol. 96, pp. 189–224, July 2014.

[84] M. Tappler, B. K. Aichernig, and R. Bloem, "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.

[85] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software," in *Formal Methods and Software Engineering* (M. Butler, S. Conchon, and F. Zaïdi, eds.), (Cham), pp. 67–83, Springer International Publishing, 2015.

[86] M. Schuts, J. Hooman, and F. Vaandrager, "Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report," in *Integrated Formal Methods* (E. Ábrahám and M. Huisman, eds.), (Cham), pp. 311–325, Springer International Publishing, 2016.

[87] A. Groce, D. Peled, and M. Yannakakis, "Adaptive Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems* (J.-P. Katoen and P. Stevens, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 357–370, Springer, 2002.

[88] J. Shijubo, M. Waga, and K. Suenaga, "Efficient Black-Box Checking via Model Checking with Strengthened Specifications," in *Runtime Verification* (L. Feng and D. Fisman, eds.), (Cham), pp. 100–120, Springer International Publishing, 2021.

[89] D. Neider and N. Jansen, "Regular Model Checking Using Solver Technologies and Automata Learning," in *NASA Formal Methods* (G. Brat, N. Rungta, and A. Venet, eds.), (Berlin, Heidelberg), pp. 16–31, Springer, 2013.

[90] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining Model Learning and Model Checking to Analyze TCP Implementations," in *Computer Aided Verification* (S. Chaudhuri and A. Farzan, eds.), (Cham), pp. 454–471, Springer International Publishing, 2016.

[91] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of SSH implementations," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, (New York, NY, USA), pp. 142–151, Association for Computing Machinery, July 2017.

[92] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Structural Analysis in the Social Sciences, Cambridge: Cambridge University Press, 1994.

[93] F. Vaandrager, M. Ebrahimi, and R. Bloem, "Learning Mealy machines with one timer," *Information and Computation*, vol. 295, p. 105013, Dec. 2023.

II

Included Papers

Chapter 7

Paper I: A Systematic Approach to Automotive Security

Masoud Ebrahimi, Stefan Marksteiner, Dejan Ničković, Roderick Bloem, David Schögler, Philipp Eisner, Samuel Sprung, Thomas Schober, Sebastian Chlup, Christoph Schmittner, Sandra König.

Proceedings of the Formal Methods 2023 conference, In Lecture Notes in Computer Science, vol. 14000. Cham: Springer International Publishing, 2023. Reproduced with permission from Springer Nature. DOI: 10.1007/978-3-031-27481-7_34.

Abstract

We propose a holistic methodology for designing automotive systems that consider security a central concern at every design stage. During the concept design, we model the system architecture and define the security attributes of its components. We perform threat analysis on the system model to identify structural security issues. From that analysis, we derive attack trees that define recipes describing steps to successfully attack the system's assets and propose threat prevention measures. The attack tree allows us to derive a verification and validation (V&V) plan, which prioritizes the testing effort. In particular, we advocate using learning for testing approaches for the black-box components. It consists of inferring a finite state model of the black-box component from its execution traces. This model can then be used to generate new relevant tests, model check it against requirements, and compare two different implementations of the same protocol. We illustrate the methodology with an automotive infotainment system example. Using the advocated approach, we could also document unexpected and potentially critical behavior in our example systems.

7.1 Introduction

The advent of *connected, cooperative automated mobility* provides a huge opportunity to increase mobility efficiency and road safety. However, the resulting connectivity creates new attack surfaces that affect the vehicle's safety, security, and integrity. With an estimated 100 million lines of embedded code, modern vehicles are highly complex systems that need to provide consistent cyber-security assurances. Indeed, there are an alarming spike in cyber-attacks targeting connected cars, their electronic control units (ECUs), and the original equipment manufacturer (OEM) back-end servers.

Therefore, making the right security decisions from the early design stages is crucial. The ad-hoc security measures done by domain experts are insufficient to meet the requirements in the automotive domain. The standard ISO/SAE 21434 and the mandatory regulation UN R155 advocate for more systematic reasoning about system security. The United Nations Economic Commission for Europe (UNECE) has adopted new security regulations, such as UNECE R155 and R156, for the homologation of future vehicles that address the identified cyber-attack risks, for example, during software updates. Similarly, the cyber security standard ISO/SAE 21434, introduced in 2021, defines precise security requirements for vehicles during the entire product life cycle, from its development to its operation and maintenance. Hence, there is an urgent need for methods and tools that address multiple security-related aspects, from early vehicle design to deployment and operation phases.

This paper proposes a top-down methodology for systematically assessing automotive security at different stages of vehicle development. The proposed methodology follows the product cycle in several steps. During the early design phase, we use threat modeling, analysis, and repair to provide more systematic support for the concept design of secure (automotive) systems. These methods allow us to identify the system's weaknesses in security threats and develop structural measures to prevent and mitigate them. We then use the threat analysis results to capture the system's critical components concerning security properties and derive a verification and validation (V&V) plan. We apply established processes (fuzz testing, penetration testing, etc.) for testing the implemented system components. However, the source code of the component implementation is often unavailable to the V&V team, and they cannot efficiently use the classical testing methods and tools. In that case, we advocate using automata learning for testing that builds an explainable model of a black-box implementation of a component from a set of executed test cases that facilitates testing and other V&V activities. This methodology is a re-

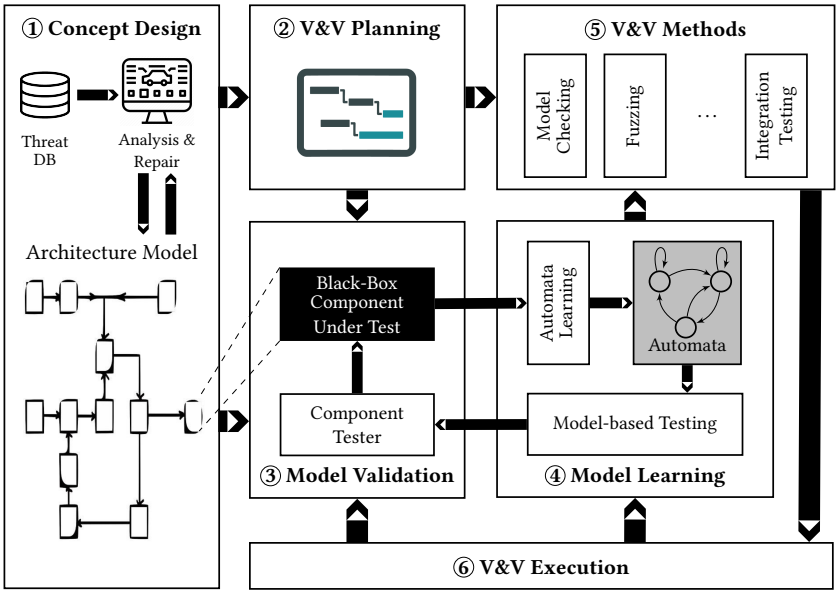

Figure 7.1: Overview of the TRUSTED methodology

sult of a joint research effort amongst the industrial and academic partners in TRUSTED[1], a project focusing on trust and security in autonomous vehicles. In implementing our proposed methodology, we were also supported by partners from the related LearnTwins[2] project, which focuses on learning-based testing methods for digital twins.

7.2 TRUSTED Methodology

The TRUSTED methodology starts with the concept design with a *threat model* of the vehicle; see Stage ① in Figure 7.1. The threat model consists of two components: (i) a system model architecture and (ii) a threat database. The system model architecture provides a structural view of the vehicle. This view includes vehicle components and subsystems (e.g., sensors, actuators, ECUs) and describes their (wireless or wired) interconnections. We can assign security

[1] https://TRUSTED.iaik.tugraz.at/
[2] https://learntwins.ist.tugraz.at/

attributes (e.g., authentication, encryption) to system components and communication links. A system model can define security boundaries that enclose trusted subsystems and assets we need to protect from potential attacks. The threat database contains a set of known threats—these threats from public domain sources, relevant standards, and previous experience. The threat model is an input to a threat analysis method allowing the detection of structural weaknesses in the system's architecture. We then combine the threat analysis with the repair activities to identify prevention and mitigation actions required to protect the system from identified threats.

The high-level threat analysis performed in the early stages of the design provides essential insights into the security-related weaknesses in the system architecture. We can take structural defense actions to improve the system's security based on threat repair outcomes (e.g., implementing authentication in a specific component). Yet, there is no guarantee that an attacker cannot break the resulting measures. Hence, it is imperative to have a solid verification and validation (V&V) plan. In the TRUSTED methodology, we use the insights gained by threat analysis and repair to identify risks and prepare an effective V&V plan corresponding to ② in Figure 7.1.

We use the system architecture model developed during the concept design phase to implement and integrate the components of the system. The implementation step is outside the scope of the TRUSTED methodology, but we assume the components are available as black boxes (see ③ in Figure 7.1). That is, we assume that we can execute components, but we cannot access their implementations.

During the development and integration of different components from the system architecture, verifying and testing safety and security functionalities becomes another critical aspect that we must address. Model validation (③ in Figure 7.1) tests the model for conformance against the component under test. This step provides either affirmation for the correctness (or completeness, respectively) of the model or counterexamples to refine the latter in a loop until the model is considered good enough to be used for test case generation.

We propose a learning-for-testing approach using automata learning (④ in Figure 7.1) as the core method for generating tests during V&V. In automata learning (see Section 7.4.1), we construct a Finite State Machine (FSM) of the System Under Test (SUT). We use the inferred FSM to: (1) obtain potential attack data, and (2) identify critical inputs that might show differences between the FSM and the SUT. We must automatically perform the necessary tests during the development and especially the maintenance phase to guarantee a quick response in the event of a threat.

We chose the learning-based testing approach due to its versatility and numerous V&V activities that we can undertake with the inferred FSM (⑤ in Figure 7.1). We can use the inferred FSM to: (1) visualize and understand the implementation, (2) model check it against its formalized requirements (possibly generating test cases on specification violations), (3) generate additional test cases by fuzz testing, and (4) Test for equivalence between implementation and a reference model or another implementation.

In the last phase (⑥ in Figure 7.1), we use various V&V strategies to verify the specified properties against the actual component under test. The test results are final verification outcomes; meanwhile, we can use them as counterexamples for the learning algorithms in ④ in Figure 7.1. This policy provides a feedback loop for refining the model in the learning-based testing approach. We execute and store tests using an automated test execution platform that augments generic test cases with additional information. This additional information comes from a test database or is provided in a grey box testing [1].

The threat model and the tests created during various design phases must be continuously maintained and updated throughout the vehicle lifecycle. We must incorporate new unknown threats and vulnerabilities into the model and re-evaluate the model to find new security issues. We must also integrate the changes to functions resulting from software updates into the system model and their impact on the vehicle's security analyzed and re-tested. This closely corresponds with the notions on testing in ISO 21434 and UNECE R155.

7.3 Automotive Security by Design

In this section, we demonstrate the use of THREATGET [2], a tool for threat modeling and analysis to improve the security of automotive applications during their early stages of design (step ① in Figure 7.1) and generate an appropriate V&V plan (step ② in Figure 7.1). We illustrate the approach with an automotive infotainment system developed by the industrial partner.

We first model the system using THREATGET (Section 7.3.1) and apply analysis to identify potential structural weaknesses in the system architecture (Section 7.3). We then use this analysis to derive a V&V plan (Section 7.3.3). Finally, we can augment it with threat repair to propose additional security measures [3].

7.3.1 System Architecture Model

We first create an accurate model of the automotive infotainment system (IS), shown in Figure 7.2. The IS is part of a larger ADAS reference model. It has several external interfaces that expose an attack surface of the vehicle. The external interfaces in Figure 7.2 are Bluetooth, WiFi, Interior Camera, and On-Board Diagnostics (OBD). The Multimedia Interface Hub (MIH) is an essential component of the infotainment system that (co-)implements core functionalities, including navigation, phone calls, and music playback. MIH also bridges external and internal interfaces. The Telematics Communication Unit (TCU) is the primary interface to the Internet. Many components in a modern vehicle depend on the TCU. For example, navigation systems use TCUs to access and update maps, and ECUs use them for over-the-air updates. Finally, all components except for TCU and Head Unit communicate through a CAN interface. We add two assets to the model – the confidentiality asset associated with the Head Unit and the availability asset associated with the TCU. The assets need to be protected, and their associated components are potential targets for attackers.

The IS is a weak security link in modern vehicles because it is more prone to successful cheap attacks than other components (e.g., *Body Control Unit* or the *Engine Control Unit*). This is due to versatile attack scenarios provided by the use of mainstream Unix-like operating systems, e.g., *Uconnect* and *Automotive Grade Linux*, the user requirements demanding functionalities like a built-in internet browser and installing third-party apps enabling remote code execution attacks, and the use of CAN bus that cannot guarantee communication integrity between the vehicle's external and internal interfaces.

7.3.2 Threat Analysis

We analyze the system model with THREATGET against its *threat database*, defining a set of possible threats formulated as *rules*. The threat descriptions are collected from multiple sources: automotive security standards and regulations (e.g., ISO/SAE 21434, ETSI, UNECE WP29 R155, and UNECE R156), publicly documented threats identified in past incidents, and expert knowledge.

We illustrate threat rules with two examples used during the analysis of the infotainment system model: the rule named "Gain Control of Wireless Interface (e.g., WiFi, Bluetooth, or BLE)" and the rule named "Flood CAN Communication with Messages". Both threat rules originate from automotive security analyses performed by domain experts. The first threat's formaliza-

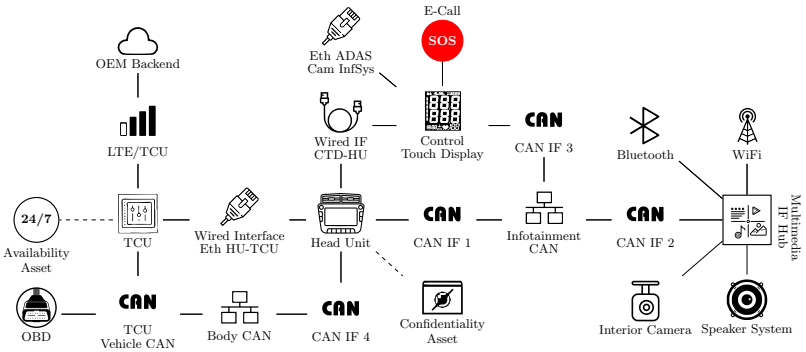

Figure 7.2: Automotive infotainment system model.

tion is

```
ELEMENT  : "Wireless Interface"{
    "Authorization" NOT IN ["Yes", "Strong"] &
    "Input Sanitization" != "Yes" &
    "Authentication" NOT IN ["Yes", "Strong"] &
    "Input Validation" != "Yes" &
    PROVIDES CAPABILITY "Control" := "true". }
```

This rule specifies that a wireless interface (e.g., WiFi or Bluetooth) that neither implements authorization and authentication nor sanitizes or validates its inputs is susceptible to threats. The last line in the rule explicitly states that if this threat is exploited, the malicious user can control the wireless interface. The "Threat Flood CAN Communication with Messages" threat is formalized as

```
FLOW {
    SOURCE ELEMENT  : "ECU"
        { REQUIRES CAPABILITY "Control" >= "true" } &
    TARGET ELEMENT  : "ECU" {
        HOLDS ASSET {
            "Cybersecurity Attribute" = "Confidentiality" &
            PROVIDES CAPABILITY "Read" := "true" } } &
    INCLUDES ELEMENT  : "BUS Communication" &
    INCLUDES NO ELEMENT  : "ECU" {"Anomaly Detection" = "Yes".} }
```

This rule states that the threat is present if there is a path starting from an ECU that is under the control of a malicious user to another ECU that holds the confidentiality asset and that there is a bus between them and no ECU on the path has implemented anomaly detection.

When applied to the infotainment system model, THREATGET identifies multiple threats. One threat is "Spoof messages in the vehicle network because of the missing components". It describes a pattern that starts at an Interface with no Authentication and ends at an ECU with no Input Validation and holds an asset. It includes a wired Shared Medium representing a vehicle's CAN BUS. Moreover, no element (of type Firewall, Server, ECU, or Gateway) on the flow from the Interface to the ECU takes care of Anomaly Detection.

We can address the identified threats with appropriate security measures. Threat repair [3] consists of preventing concrete threats by proposing security measures that can be implemented during the system's design. THREATGET implements *attribute repair*, a method that proposes changes in the components' security attributes as locally deployed measures with a simple cost model.

In the case of the automotive infotainment system model, e.g., the proposed threat repair measures include enabling authorization and implementing authentication in the WiFi and Bluetooth components. We note that threat repair does not remove the need for the planned V&V activities. The fact that authentication is integrated into the WiFi device, following the outcomes of threat repair, does not guarantee that the authentication algorithm's implementation is weakness free. On the contrary, systematic testing of the WiFi's authentication protocol is even more necessary to gain confidence that the WiFi device is not a possible entry point for malicious users.

7.3.3 V&V Planning

In addition to threat analysis, there is support for identifying and modeling more sophisticated threats using attack trees; c.f. [4]. This results in more knowledge about potential attackers' steps when intruding into a system. Simple rules can be assigned attributes called capabilities that are either required for an intrusion or can be gained through the intrusion of a system component. Moreover, we can define the different access levels to a component (e.g., $Access < Read < Modify < Control$). Depending on previously acquired capabilities, different attack tree rules trigger, yielding distinct attack trees. An example of such a generated attack tree is illustrated in Figure 7.3.

The attack tree depicted in Figure 7.3 shows how a malicious user can access the *confidentiality* asset associated with the Head Unit via external interfaces such as WiFi and Bluetooth. For instance, control of the Bluetooth interface can be gained if its security attributes (input validation and sanitization, authorization and authentication) are not implemented or have weak-

nesses. From there, the user can gain control of the Multimedia Interface Hub, which is not sufficiently secure, and then get control of the Head Unit and hence the access to the asset. The attack tree exposes the most critical components that need to be protected. We note that the attack tree from Figure 7.3 is not maximal nor unique – while THREATGET generates multiple trees for each asset in the model, including the maximal attack trees, we use a simpler tree for illustration purposes.

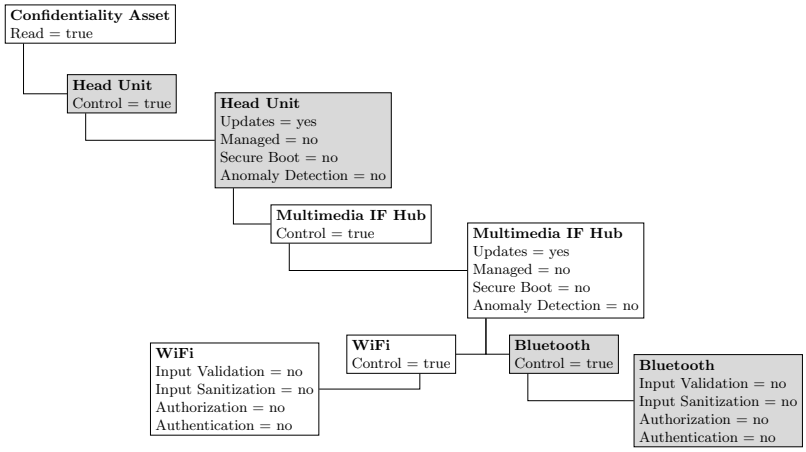

Figure 7.3: Attack tree derived from THREATGET. Multiple children from the same node are implicitly interpreted with an OR operation.

7.4 Automotive Security Testing

In this section, we advocate an approach based on learning to test critical components identified by the threat analysis methods during concept design, when these components are assumed to be black-box to the tester.

7.4.1 Automata Learning for Correctness

Many cyber-physical components in the automotive domain implement one or multiple finite state machines (FSMs).

Implementing larger automotive FSMs becomes cumbersome mainly because: (1) ensuring FSM's correctness w.r.t. its specification is expensive, (2)

correctly coding the structure of a large FSM is difficult, and (3) correct integration of FSMs in complex software is hard.

Unfortunately, many software-driven components in the automotive industry are black boxes from different manufacturers, hence are hard to verify and thus do not provide functional or non-functional guarantees.

Given an FSM of a black-box automotive component, we can test and verify it to increase our confidence in its correctness. Automata learning has proven to be a successful method for learning-based testing of communication protocols that are also used in the automotive domain, e.g., MQTT [5] or Bluetooth Low Energy [6]. We use automata learning [7] to infer an FSM model (concretely a Mealy machine) of the the SUT. In the learning context we refer to the SUT by system-under-learning (SUL). In automata learning, a *learner* asks an *oracle* two types of queries. First, *membership queries* to determine the SUL's output for a given input word. Second, *equivalence queries* check whether a learned model conforms to the SUL, to which the oracle returns positive answer or a counterexample. A counterexample is an input-output word distinguishing SUL from hypothesis. In practice, oracles for black box systems work with conformance testing.

Ordinarily, real-world systems' alphabets are not manageable for learning algorithms. Abstraction helps to both cope with this fact and to make inferred models more human-readable. Too much abstraction, however, might induce non-deterministic behavior and hide problems we intend to find. There are also automatic abstraction refinement approaches for an optimum of abstraction in a mapper [8, 9]. An abstraction mapper consists of a mapping function that converts a concrete input into an abstract symbol. It also observes the SUL's concrete outputs and sends an abstraction to the learner. To send a concrete input to the SUL, the mapper inverses the abstraction. There are multiple methods to assess the behavioral correctness of the learned FSMs, including (1) black-box checking [10], adaptive model checking [11], a combination of learning-based testing and machine learning [12] and symbolic execution [13].

7.4.2 Use-Case Scenarios

The attack tree (see Figure 7.3) poses the critical components that need to be tested for security. In this section, we illustrate our learning-based testing approach on the two components highlighted in gray color in Figure 7.3 - the Bluetooth interface (as an entry vector) and the Head Unit ECU.

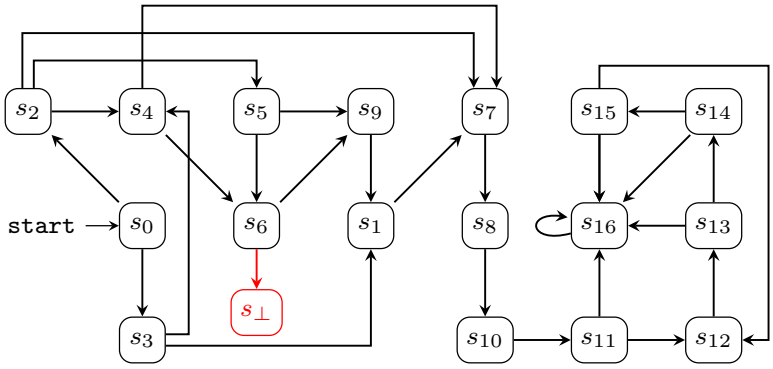

Figure 7.4: Inferred FSM structure for Bluetooth pairing.

Bluetooth and Bluetooth Low Energy

Bluetooth is a well-established standard for wireless audio used in most infotainment systems. Bluetooth Low Energy (BLE) grows in popularity for car access and sensor data transmission. The protocols have a variety of known vulnerabilities [14, 15, 16, 17, 18], some also specifically for automotive systems[3].

Learning Setup we use Intel Wireless Controllers (AC 8265 and AX200) implementing Bluetooth and BLE. The learning setups are similar, the difference is in the radio hardware and the physical layer, requiring three entities: (1) Radio Device, (2) Learner, and (3) Interface between the two with a mapper. The learner was implemented using the LearnLib framework [19].

Learned Model and Findings We inferred the pairing process models, which are used for encryption and therefore security-critical in the SULs. As a tangible result, we discovered a BLE deadlock state (red state in Figure 7.4) in the Linux BLE host software. With repeated out-of-order transmission of pairing requests of different types, we force the respective BLE stack into a state that limits the device to respond to basic link-layer control packets. After the state is reached, each following connection will start in this state until the controller is reset.

[3] https://research.nccgroup.com/2022/05/15/technical-advisory-tesla-ble-phone-as-a-key-passive-entry-vulnerable-to-relay-attacks/

Unified Diagnostic Services

Each ECU has a secure access mode reachable through its UDS implementation, available via vehicle's OBD connector. An attacker able to exploit UDS security features would be also able to manipulate data or even flash the ECU with a malicious firmware.

Learning Setup To communicate with the ECU we used a CAN interface. To learn a different ECU we only need to adapt the interface. We started by implementing a reduced UDS interface, consisting of instructions to put an ECU into secure access mode. Communications occures via a CAN bus interface. The learner was implemented using the AALpy framework [20].

Learned Model and Findings The learning experiment resulted in a reduced FSM of the UDS shown in Figure 7.5. An analysis of the results shows that once being successfully authenticated (state s_4), an incorrect authentication key will still result in the same state. This is unexpected and allows for prolonging a session without authentication. When requesting a new seed for re-authentication (s_5) this behavior persists. Moreover, on re-entering a secure session afterwards (from s_6), the ECU accepts an old key as well; an unexpected behavior after re-initiating the key authentication. Figure 7.5 marks all unexpected behaviors in red.

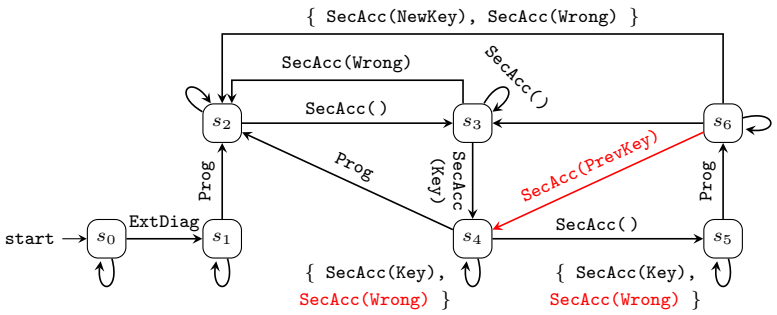

Figure 7.5: Inferred UDS FSM.

7.5 Conclusion

We introduced the TRUSTED methodology for designing and assessing trusted and secure automotive systems. The main novelty of the proposed methodology is its holistic and systematic approach to security, which starts at concept design and is carried down to the implementation and assessment of individual components. We instantiated the different parts of the methodology using the state-of-the-art methods and tools for threat modelling and analysis, automata learning and testing. We illustrated the use of the methodology by applying it step-by-step an automotive infotainment system. Using the learning-based testing approach we could document previously unpublished denial-of-service conditions in the examined BLE setups, as well as unexpected behavior allowing for extending secure UDS programming sessions on the scrutinized ECU.

Future Work We plan to further automate the transition from the concept design and V&V planning on one side, to the actual testing activities done on the level of components by devising a domain-specific test description language that can define abstract V&V plans derived from the attack trees, and be refined in a way so that eventually it can be executed on a platform (e.g., as in [21]). Second, the TRUSTED methodology mainly focuses on the transition from concept design to testing the implementation. We plan to also study the opposite direction – how to use the component testing results to update the system model and have a more refined threat analysis and a more realistic threat assessment.

Acknowledgements

This research received funding 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. 867558 (project TRUSTED) and within the ECSEL Joint Undertaking (JU) under grant agreement No. 876038 (project InSecTT). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Sweden, Spain, Italy, France, Portugal, Ireland, Finland, Slovenia, Poland, Netherlands, Turkey. The document reflects only the author's view and the Commission is not responsible for any use that may be made of the information it contains.

Bibliography

[1] S. Marksteiner, N. Marko, A. Smulders, S. Karagiannis, F. Stahl, H. Hamazaryan, R. Schlick, S. Kraxberger, and A. Vasenev, "A Process to Facilitate Automated Automotive Cybersecurity Testing," in *2021 IEEE 93rd Vehicular Technology Conference (VTC Spring)*, (New York, NY, USA), pp. 1–7, IEEE, 2021.

[2] C. Schmittner, S. Chlup, A. Fellner, G. Macher, and E. Brenner, "Threatget: Threat modeling based approach for automated and connected vehicle systems," in *AmE 2020 - Automotive meets Electronics; 11th GMM-Symposium*, (Berlin), pp. 1–3, VDE Verlag, 2020.

[3] T. Tarrach, M. Ebrahimi, S. König, C. Schmittner, R. Bloem, and D. Nickovic, "Threat repair with optimization modulo theories," *CoRR*, 2022.

[4] M. Ebrahimi, C. Striessnig, J. C. Triginer, and C. Schmittner, "Identification and verification of attack-tree threat models in connected vehicles," in *SAE Intelligent and Connected Vehicles Symposium*, (Shanghai, China), pp. 1–12, SAE International, 2022.

[5] M. Tappler, B. K. Aichernig, and R. Bloem, "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.

[6] A. Pferscher and B. K. Aichernig, "Fingerprinting Bluetooth Low Energy Devices via Active Automata Learning," in *Formal Methods* (M. Huisman, C. Pˇ asăreanu, and N. Zhan, eds.), Lecture Notes in Computer Science, pp. 524–542, Springer International Publishing, 2021.

[7] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, Nov. 1987.

[8] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. W. Vaandrager, "Automata learning through counterexample guided abstraction refinement," in *FM 2012*, (Berlin), pp. 10–27, Springer, 2012.

[9] F. Howar, B. Steffen, and M. Merten, "Automata Learning with Automated Alphabet Abstraction Refinement," in *Verification, Model Checking, and Abstract Interpretation* (R. Jhala and D. Schmidt, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 263–277, Springer, 2011.

[10] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black Box Checking," in *Formal Methods for Protocol Engineering and Distributed Systems: FORTE XII / PSTV XIX'99 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX) October 5–8, 1999, Beijing, China* (J. Wu, S. T. Chanson, and Q. Gao, eds.), IFIP Advances in Information and Communication Technology, pp. 225–240, Boston, MA: Springer US, 1999.

[11] A. Groce, D. Peled, and M. Yannakakis, "Adaptive Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems* (J.-P. Katoen and P. Stevens, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 357–370, Springer, 2002.

[12] K. Meinke, "Learning-Based Testing of Cyber-Physical Systems-of-Systems: A Platooning Study," in *Computer Performance Engineering* (P. Reinecke and A. Di Marco, eds.), Lecture Notes in Computer Science, (Cham), pp. 135–151, Springer International Publishing, 2017.

[13] B. K. Aichernig, R. Bloem, M. Ebrahimi, M. Tappler, and J. Winter, "Automata Learning for Symbolic Execution," in *2018 Formal Methods in Computer Aided Design (FMCAD)*, (Austin, Texas, USA), pp. 1–9, IEEE, 2018.

[14] D. Antonioli, N. O. Tippenhauer, and K. B. Rasmussen, "The KNOB is broken: Exploiting low entropy in the encryption key negotiation of bluetooth BR/EDR," in *28th USENIX Security Symposium, USENIX Security 2019* (N. Heninger and P. Traynor, eds.), (Santa Clara, CA, USA), pp. 1047–1061, USENIX Association, 2019.

[15] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, "BIAS: Bluetooth Impersonation AttackS," in *2020 IEEE Symposium on Security and Privacy (SP)*, (San Francisco, CA, USA), pp. 549–562, IEEE, May 2020.

[16] D. Antonioli, N. O. Tippenhauer, K. Rasmussen, and M. Payer, "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, (New York, NY, USA), pp. 196–207, Association for Computing Machinery, May 2022.

[17] B. Seri and G. Vishnepolsky, "The dangers of Bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern Bluetooth stacks.," tech. rep., Armis Inc., 2017.

[18] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, "Key Negotiation Downgrade Attacks on Bluetooth and Bluetooth Low Energy," *ACM Trans. Priv. Secur.*, vol. 23, pp. 14:1–14:28, June 2020.

[19] M. Isberner, F. Howar, and B. Steffen, "The Open-Source LearnLib," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), Lecture Notes in Computer Science, (Cham), pp. 487–495, Springer International Publishing, 2015.

[20] E. Muˇ skardin, B. K. Aichernig, I. Pill, A. Pferscher, and M. Tappler, "AALpy: An active automata learning library," *Innovations in Systems and Software Engineering*, vol. 18, no. 3, pp. 417–426, 2022-09-01.

[21] C. Wolschke, S. Marksteiner, T. Braun, and M. Wolf, "An Agnostic Domain Specific Language for Implementing Attacks in an Automotive Use Case," in *The 16th International Conference on Availability, Reliability and Security*, ARES 2021, (New York, NY, USA), pp. 1–9, Association for Computing Machinery, Aug. 2021.

Chapter 8

Paper II: Approaches For Automating Cybersecurity Testing Of Connected Vehicles

Stefan Marksteiner, Peter Priller and Markus Wolf.

In Intelligent Secure Trustable Things, M. Karner, J. Peltola, M. Jerne, L. Kulas, and P. Priller, Eds., in Studies in Computational Intelligence. Springer Nature, 2023. Reproduced with permission from Springer Nature. DOI: 10.1007/978-3-031-54049-3_13.

Abstract

Vehicles are on the verge building highly networked and interconnected systems with each other. This requires open architectures with standardized interfaces. These interfaces provide huge surfaces for potential threats from cyber attacks. Regulators therefore demand to mitigate these risks using structured security engineering processes. Testing the effectiveness of this measures, on the other hand, is less standardized. To fill this gap, this book chapter contains an approach for structured and comprehensive cybersecurity testing of contemporary vehicular systems. It gives an overview of how to define secure systems and contains specific approaches for (semi-)automated cybersecurity testing of vehicular systems, including model-based testing and the description of an automated platform for executing tests.

8.1 Introduction

Mobility is a high priority in our society. Statistics report global annual car sales between 60 and 75 million[1] during recent years. According to the European Automobile Manufacturers' Association (ACEA), just in Europe approximately 350 million cars are currently in use [1], and the number grows to beyond 1 billion for a worldwide estimation. Cars are ubiquitous, for many families and businesses around the world, since decades. What has changed, however, is the fact that today's vehicles have become complex IT systems, often also called "computers on wheels". Modern cars run 100+ million lines of source code (MLOSC), and host complex computer networks both internally (in-vehicle networks) and externally. Many modern cars are now connected via the Internet to (maybe even multiple) cloud services, as well as to cellular networks (3G, LTE, 5G), and to specific vehicular networks (also known as car-to-car (C2C) or vehicle-to-everything (V2X), like ITS-G5). And that's not all: most vehicles also provide local networking capabilities (also called personal area networking, PAN). Typically based on WIFI and Bluetooth, it is used to connect to users' personal devices like smart phones and tablets, or to their home WLAN. To complement that already impressive array of wireless communication interfaces, some car manufacturers (or Original Equipment Manufacturers - OEMs) might add ultra-wide band (UWB) radios to communicate with car access systems like owner's keys or keycards. In addition, advanced driver assistance systems (ADAS) and future fully automated driving (AD) capabilities add GNSS receivers (Global Navigation Satellite System), TMC receivers (Traffic Message Channel), and active radar systems. Modern vehicles combine deeply complex software with exposure to a wide range of wireless networking technologies to both public and closed networks). In cybersecurity, this is called opening a large attack surface. This is worsened by the fact that vehicles are exposed for a much longer time than, e.g., personal computers (PC) or mobile devices like smart phones. Cars are in operation for some 15 years and more, which increases the threat that a vulnerability is found, shared, and at some point in time exploited by an attack. With such significant high exposure, let's consider potential threats which could evolve from malign attacks. Vehicles are highly dynamic (by nature), provide high levels of energy (storing 100kWh and more), are valuable (sometimes beyond 100k€) and exist as worldwide accessible objects in public, thus unrestricted places. When exploited by an attack taking over remote control, vehicles could become dangerous weapons, for both passengers inside, and for other road participants.

[1] https://www.statista.com/statistics/200002/international-car-sales-since-1990/

Worse, if groups of vehicles would come under attacker's control, they could be used to stage threats on city or even national level. State-sponsored attackers could stage war or terror attacks of not-yet-seen scale. Other scenarios might be less about harming humans, but could include denial-of service on single vehicles (e.g., owners cannot use their vehicles unless a ransom is paid) or on fleet level (e.g., blocking important road infrastructure, threatening whole communities, and some serious damage of a brand's reputation). And of course, there is simple car theft. Data privacy is also an important aspect. Modern cars might "know" quite a lot about their users, including their past and present locations, driving habits, additional passengers, anything spoken in the vehicle, attention level while driving, contact information like phone numbers from connected personal devices, etc. An attack could therefore retrieve quite a lot of personal information and thus become considerable value to attackers. While not all of these threats have been discussed widely in public, the automotive industry is very much aware of it, and has stepped up efforts in designing more secure systems in cars, and establishing secure life cycle processes to provide necessary updates to fix vulnerabilities. An important part of securing these vehicular systems is the verification and validation of the effectiveness of taken security measures through testing. This testing needs to be done continuously through the life cycle (as new exploits might come up over time), and also as updating a system (or just a part of it) might alter its behavior an a way relevant to its security. In essence, (cyber)security testing must assure a system to display a small attack surface, be resilient and (possibly) to fix vulnerabilities before they are exploited in the wild.

The remainder of the chapter is structured the following way: Section 8.2 contains the current state of the art and related work. Section 8.3 contains measures for securing automotive systems. Section 8.4 contains specific approaches for automated cybersecurity testing of vehicular systems, including model-based testing and the description of an automated platform for executing tests. Section 8.5, eventually concludes this chapter.

8.2 State of the Art and Related Work

The automotive industry can draw from experience in other domains regarding security testing. General IT (managing e.g., corporate networks and IT systems) has established a history of penetration testing (abbreviated: pen testing), as simulated, authorized cyber-attacks. Typically executed by cybersecurity experts (acting as "white-hat hackers"), the goal is to identify weaknesses

by letting these experts try to hack into the system under test (SUT) under predefined constraints (e.g., no physical access, no permanent harm), typically within a defined time window. If successful, these tests can thereby discover and document weaknesses. Translated to automotive industry, several companies offer similar pen-testing as a service on different levels (component, system, vehicle). While pen-testing might provide highly valuable insights into what level of security has been achieved for the vehicle, and might even uncover previously unknown vulnerabilities, it suffers from limited scalability and repeatability, as it is driven by and dependent on human experts. Security experts have toolboxes with highly effective tools (like the open source Metasploit framework[2]), but often need to supervise and configure these tools, and to adapt existing or write new scripts for complete attack chains to match a specific SUT. This requires skills and labor, and often involves considerable costs, which clearly limits scalability. Due to the sheer complexity of automotive software code (100+ MLOSC), it is also quite challenging for the experts to correctly hypothesize vulnerabilities, and to select (and execute) the most effective attacks, given the limited time available. This might heavily depend on expertise of the human testers, further limiting repeatability and comparability between pen tests campaigns. The threat of cyber attacks by adversaries has, however, also been recognized by standards and regulatory bodies. The United Nations Economic Council for Europe (UNECE) has issued a regulation (R 155 [2]) that prescribes the installation of a cybersecurity management system (CSMS). A CSMS is a process framework that accompanies the automotive development process over the complete life cycle and assures cybersecurity in every phase. Consequently, the International Organization for Standardization (ISO) and the Society of Automotive Engineers (SAE) have issued a joint standard (ISO/SAE 21434 [3]) that defines such a CSMS. As testing guidelines in these standards are somewhat underrepresented in contrast to security engineering, a structured approach is needed, e.g., as defined in [4, 5]. It further became clear that in order to establish dependable security covering all variants of vehicle lines in their full life cycle, supporting the upcoming accelerated software development cycles (automotive DevOps), an advanced process based on smart automation was required, as suggested in [6].

[2] https://github.com/rapid7/metasploit-framework

8.3 Automotive Cybersecurtiy Lifecycle Management

In order to maintain secure (and through, security-related impacts, also safe) vehicular systems, the respective system needs a security concept. The cybersecurity testing (see Section 8.4) will eventually validate and verify the effectivness of that concept. To establish a security concept for the complete life cycle of a vehicle for testing, we mainly rely on five pillars:

1. Threat Modeling (see Section 8.3.1)
2. Variant Management
3. Vulnerability Assessment
4. Automated Test Generation (see Section 8.4.2)
5. Process Governance

Threat modeling (see Section 8.3.1) is a widely proliferated technique in the automotive industry, mainly as part of a threat analysis and risk assessment (TARA) process [7].

As an OEM's fleet contains various vehicle model configurations, all of which contain tens of ECUs all of which again may display different hardware and software versions, keeping track of this potentially vast number of variants is crucial to determine the security posture of each member of the fleet. Our approach to tackle this problem is to use calibration data management that links technical attributes with software calibrations, to keep track of all ECU variations over the system's life cycle [8, 9]. This system, CRETA, contains exhaustive information about the variants, including their ECU firmware binaries.

This allows for the stored firmwares to be subsequently analyzed, generating a digital model of the software. To do so, firstly the firmware is extracted by iterating through the file tree, using an extraction algorithm and validating the extraction's correctness. The extracted software undergoes a composition analysis that pre-processes executables and normalizes the software in order to compare to a large database of mapped components, identified e.g. by file paths, file names, and characteristic strings in the software or configuration data, yielding a Software Bill-of-Materials (SBOM). Subsequently, the model is analyzed for security properties using pattern recognition. Patterns of known attacks from Common Vulnerabilities and Exposures (CVEs) are

compared with each identified software library in the SBOM. Furthermore, the model undergoes a binary code analysis to find vulnerabilities not found in public databases: the binary is mapped in data and code sections, the code is then disassembled and later mapped into an intermediate language (for normalizing purposes) that allows for reconstructing the functions, analyzing the parameters and stack behavior and building control and data flows [10]. This matching, for instance, is able to identify common flaws like buffer overflows and, hence, is able to uncover zero-day vulnerabilities in software in a black box setting. Thirdly, patterns for proliferated code guidelines and relevant security standards are implemented, allowing for compliance checking against a given set of standards. This analysis, paired with full life cycle-coverage of the variants, allows for dealing with the parts lists and vulnerability management requirements mentioned above, as well as for verifying security requirements.

Vulnerabilities found in the code through pattern matching, however, are not necessarily exploitable for a variety of reasons. For instance, the location in the code could not be reachable, the impact of the vulnerability could be nullified through write protection of the memory or file system, or the interface might be protected by access controls. Therefore, the generated model also allows for model-based cybersecurity test case generation by using either the generated behavior model for model checking or by directly using the found patterns as basis for vulnerability exploitation [11]. We also aim for deriving test cases from threat modeling with a certain degree of automation (see Section 8.4.2).

To govern the process we developed our tool, FUSE, that guides activities of a given standard and provides standards-compliant documentation given the necessary input. We implemented ISO/SAE 21434 [3] and UNECE R155 [2] (as well as ISO 26262 [12], ISO 25119 [13]). The modeled objectives from the standards allow for providing all necessary artifacts for performing a review or audit, as well as keeping track of the conformance to relevant standards inside the development project.

8.3.1 Threat Modeling

One key element of cybersecurity analysis in all life cycle phases is threat modeling. This technique for security analysis is around for many years and well proliferated. It basically consists of modeling the information flows in an SUT and consequently examining them in a comprehensive way, e.g., via STRIDE or a similarly structured method [14].
Numerous software capable of performing a thread modeling process exists,

but prior to ThreatGet none was specifically developed for embedded or IoT systems. ThreatGet is a software tool developed by Austrian Institute of Technology (AIT) and based on Microsoft Enterprise Architect, a commonly used platform for systems model engineering [15].
It is used to examine models, objects, connections and charts in a system to enable iterative threat and risk analysis, covering the following categories:

- Actor,
- Sensor,
- Vehicle Unit,
- Data Store,
- Communication Interface,
- Communication Flow

Objects and connections in ThreatGet have so called tagged values at creation time. These describe analysis or security relevant properties of elements. It is recommended for users to extend the properties in addition to already proposed tagged values. Additionally, a database is used in the background that contains objects, which can also be extended by a user [15].

As an application example, Figure 8.1 shows the threat diagram of a communication flow inside ThreatGet. In this case, the environment data from the camera is directed to the ”Sensor Data Fusion and Decision Making” unit. After all diagrams are completed, a threat-overview is derived. An automatic risk evaluation consists of suggested values and can be adapted in a manual risk evaluation. In this step it is possible to rate the impact and occurrence of a threat at different levels and afterwards results can be exported in a report [15].

8.4 Cybersecurity Testing

In order to assure the cybersecurity of automotive systems and provide evidence for the appropriateness and effectiveness of security measures (according to a cybersecurity management system) , rigorous, structured and comprehensible testing is necessary [2]. Therefore a structured process, aligned with ISO/SAE 21434 [3] is recommendable. Such a process for testing could contain the following activities [5]:

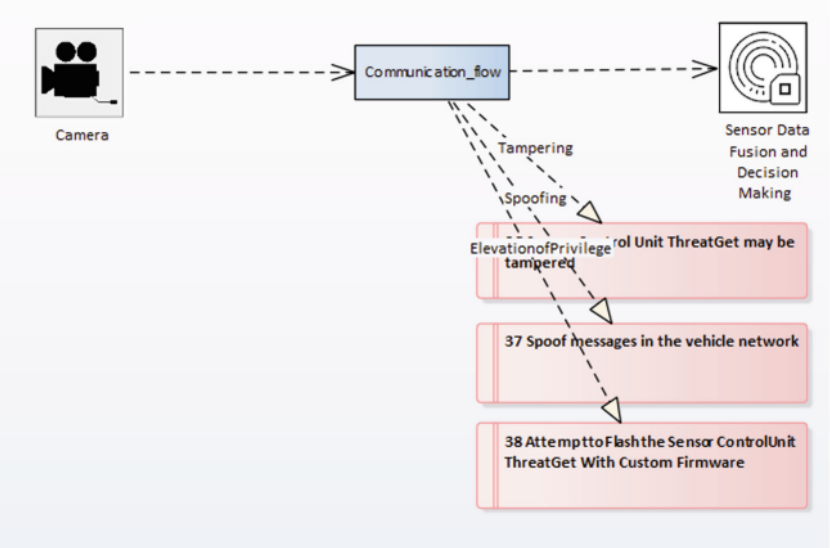

Figure 8.1: A list of found threats between the camera and the sensor data fusion and decision making [15].

1. Item Definition
2. Threat Analysis and Risk Assessment
3. Security Concept Definition (mainly including the test targets)
4. Test Planning and Scenario Development
   (a) Penetration Test Scenario Development
   (b) Functional and Interface Test Development
   (c) Fuzz Testing Scenario Development
   (d) Vulnerability Scanning Scenario Development
5. Test Script Development
   (a) Test Script Validation
6. Test Case Generation

   (a) Test Environment Preparation

7. Test Case Execution
8. Test Reporting

While items 1-3 correspond to a threat modeling process (see Section 8.3.1), the rest of them are the core testing process. To increase testing efficiency, these steps could be partially automated using model and learning-based approaches that can execute test planning and execution steps [6]. Here, the steps can be summarized into *concept design*. Item 4 *forms V &V planning*, while items 5 and 6 can be subsumed under *V & V Methods*. Finally, items 7 and 8 forms *V & V execution*. In between the planning and the methods, steps for automation can take effect: models from the concept design can be validated in an automated way and single components can be modeled using automated learning techniques and verified using methods from the V & V methods. An example of this used in the InSecTT project is described in Section 8.4.1. The full approach as described above consists of the following steps [6]:

1. Concept Design
2. V&V Planning
3. Model Validation
4. Model Learning
5. V&V Methods
6. V&V Execution

8.4.1 Learning-based Testing

Following the approach described above, we use learning, more concretely active automata learning to derive a model of a system [16]. The methodology uses a learner-teacher system where an all-knowing teacher answers the learning system queries about the SUT, in the context of cyber-physical systems ordinarily by providing the output to a series of inputs. The learner tries to infer a state machine from the given information. Once it has a hypothesis of a state machine that describes the observed behavior, it presents it to the teacher who then acknowledges the hypothesis as correct or gives a counterexample. This again, in real-world situations of black-box learning will mostly be simulated

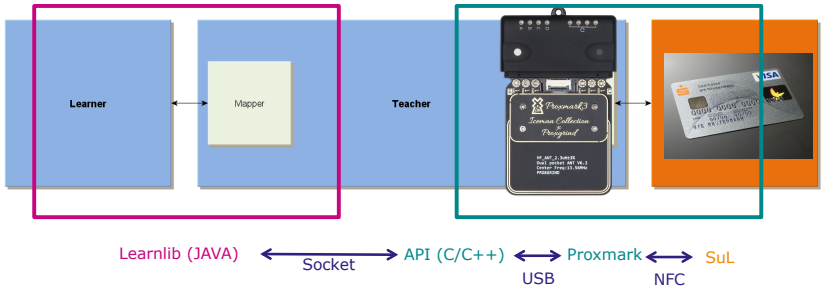

Figure 8.2: NFC Automata Learning Setup [23]

by conformance testing algorithms: if conformance is shown, the hypothesis is assumed as correct, otherwise a failing test sequence serves as a counterexample. The counterexample is taken as new input to refine the hypothesis and the learning continues until no more counterexamples are found. The this algorithm has been first formulated by Angluin [17] and has experienced significant improvements since (e.g., [18, 19]).

In accordance with the process outlined in Section 8.4, we use this technique to infer a model of a component. As a proof-of-concept we test a car access system based on Near-Field Communications (NFC). The testing setup consists on a learner (as described above) based on the *Learnlib* Java library [20] and a Proxmark NFC device [21] with an respective API that enables us to learn a model of the ISO 14443-3 NFC handshake protocol [22]. Figure 8.2 shows an overview of this setup. The used learning setup allows for inferring a state machine of the protocol and compare it to the specification in the standard to check its conformance. Figure 8.3 shows the learned model of the actual SUT (and NXP test card of a car access system prototype). Further use of the model is to do actual model checking or to use the model as an input for guided fuzz testing.

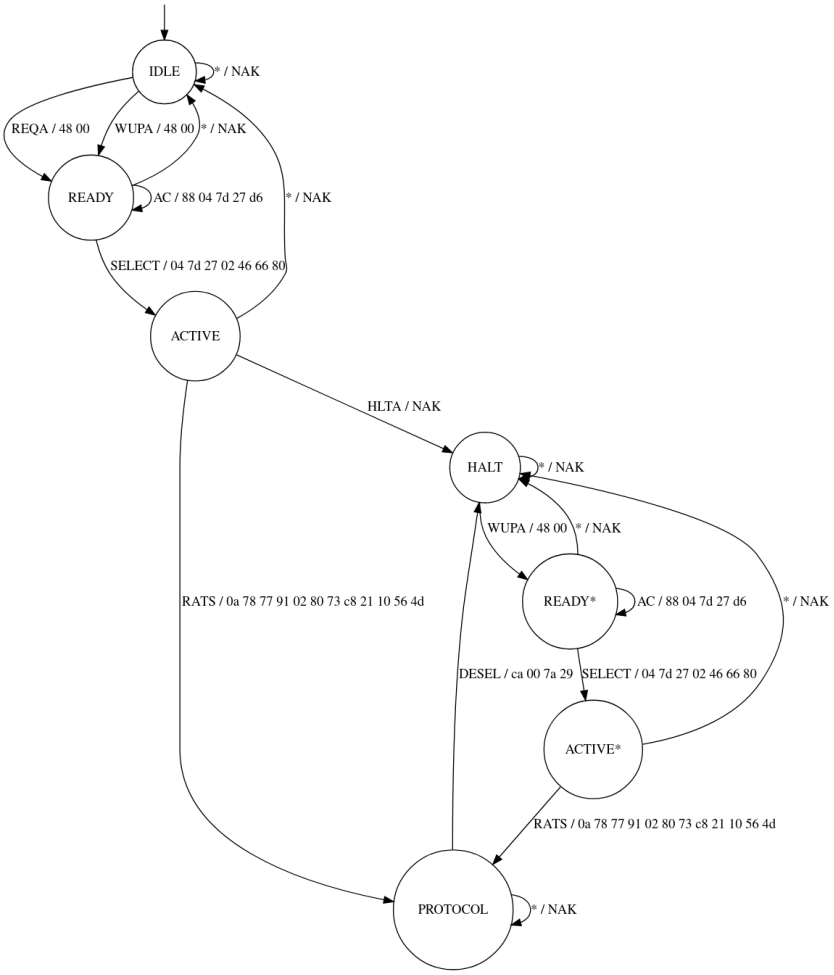

Figure 8.3: Learned Model of an NXP NFC Test Card

8.4.2 Model-based Test Case Generation

On a macroscopic level, a model of a complete vehicle as defined in the threat model (see Section 8.3.1) has to be explored in order to identify single com-

ponents and generate test cases based on an attack tree [24, 25], a petri net [26, 27], or similar. If the SUT is modeled manually and, therefore, the components are known, this is trivial. If the setting is a black or grey box situation, we follow the approach to assume a generic model as starting point and test various components of the model by, e.g., send certain CAN messages for enumeration or try out an exploit that is known to affect a very broad variety of systems. Based on a comparison of the expected and actual output of the test, one can narrow down the set of likely components and system architectures (as described in [28]), e.g., based on SAT solving [29].

In order to generate test cases on a component level, a model must be transformed into a form that can be examined using a model checker (e.g. the Rebeca model checker [30] or SLAM [31]). Violations of the specification found by a model checker point towards an interesting position for a test case that could be extrapolated out of the traces leading to the respective states. There is also work regarding a toolchain using the UPPAAL framework [32].

Subsequently properties defining the security of a system shall be defined and used in the model checking. For c) where the model checking fails, a security problem might be present. The trace of the counter example can help in building a test case. Moreover, the input sequences used for the automata learning of the model shall be used to make test cases for the actual system-under-test. Using the traces as test vectors eliminate false positives from the model checking, as the exploitability of specification violations is test on the actual system. To concrete the abstract input, fuzzing techniques may be used [33].

8.4.3 Testing Platform

To realize the testing in the faction outlined in Section 8.4, a testing framework was developed and implemented. The high-level architecture was derived from the approach outlined in [4]. It has been adapted to suit the need of performing test in any phase of the product life cycle by adding co-simulation techniques into the testing framework architecture (see Figure 8.4 for an overview). The core component is a Security Testing Framework (see Section 8.4.4). It gains test cases from a generation engine that is fed by two sources: security functional tests from security requirements and penetration test attack vectors that have been tried out before (see description in Section 8.4.4) from a library. The core framework executes the attacks directly onto the SUT or into a co-simulation platform (indicated as *framework interfaces* in the figure) that interconnects various simulation parts: environment (i.e. other vehicles' and infras-

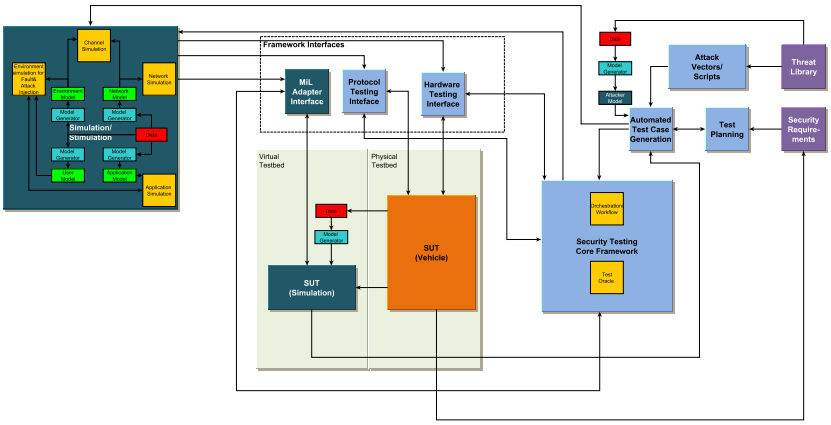

Figure 8.4: Overview of the Automotive Cybersecurity Testing Framework's high-level architecture

tructure's interference), network (generating mainly ITS-G5 traffic), channel (capable of simulating various physical layer signals as well as emitting them physically) and application (Section 8.4.4 contains an example with a platooning application). This way, each component can be stimulated the same way regardless if it is a physical or simulated component.

8.4.4 Automated Test Execution

For test execution, the test cases that were derived as described in previous chapters are fed into the automated test execution environment. Test cases are either manually written or generated in the ALIA DSL [34] format, which aims to provide an abstract and system agnostic representation of logical steps in a test case. Out of the main test-script and its included sub-scripts (containing frequently occurring blocks that handle a specific task such as opening a listener) a JSON Object is generated. These test case descriptions in DSL and JSON format are stored into a Database and can be accessed through the Orchestration Application; a platform independent web application that allows a user to manage information about the current SUT, schedule test execution and review results. This Orchestration Application then sends the test cases that the user wants to execute to the Execution Engine (AXE) and afterwards generates a report out of the received output from the AXE and the Test Oracle.

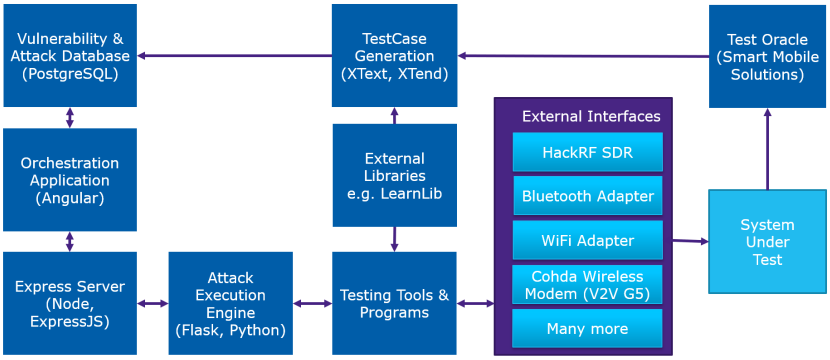

Figure 8.5: AACT Test Execution Framework

The AXE is a Python based software that runs on an instance of Kali Linux and utilizes a variety of different interfaces, libraries and other software tools to perform a test case execution. It takes either a single test case or a structured collection of tests as input in JSON format and starts to subsequently execute contained steps. Figure 8.5 shows an overview of this architecture. This modular approach allows not only to target a specific SUT but also to control and parameterize whole (semi-) virtual SUT environments to manage SUT-behavior during a test scenario. Furthermore, it is possible to define and address different processes for tool execution which enables for example to host a malicious server, start a netcat listener and execute exploit code sequentially in a single test and afterwards perform code execution in an obtained reverse shell in the listener process.

One proof-of-concept use case implemented in the framework was security testing of the Ensemble platooning protocol [35] in a simulated environment. The concrete setup consisted of two truck simulations running on low-cost hardware connected via physical ITS-G5 [36] connection via Cohda modems. Another modem is used as an adversary to eavesdrop and interfere with the connection. The testing framework is able to start the simulation, so that the simulated trucks form a platoon. The actual test consists of a) listening to the communications b) distilling the session key out of a package c) cracking the key (for testing purposes, the key was reduced to eight bits) d) injecting a malicious message to disband the platoon. Figure 8.6 shows an overview of this setup. The result was that the injection failed for timing reasons, because the platoon keep-alive messages were sent in such a high frequency that they in-

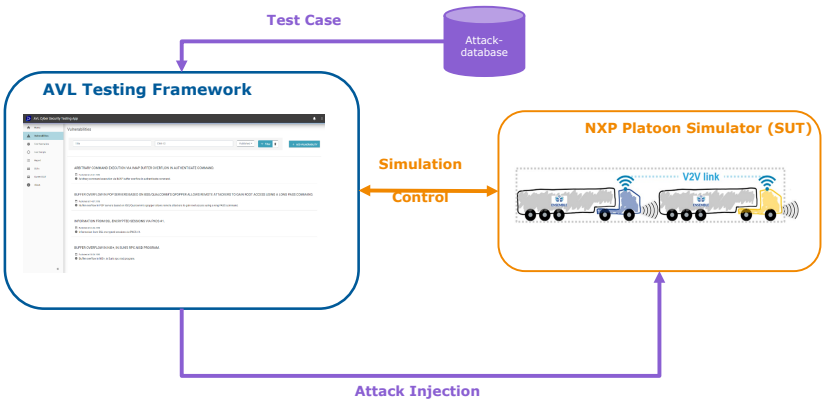

Figure 8.6: Platooning Use Case Overview

terfered with the break-up sequence. Even with reduced key (from AES-256 down to 8 bits) the protocol was secure against the tested attack. Furthermore, ITS-G5 built-in signatures, that were disabled for the test, would have prevented a successful injection. The test could therefore show the security of the protocol in an automated way as described above.

8.4.5 Fuzzing

The goal of fuzzing is to reach a non-intended state of a SUT by using completely or partially random input. The latter technique may use a structured frame structure that is compliant with communication standards used by the SUT and randomized payload data. [37] In case of a CAN-Bus, a fuzzing tool can create packets that consist of the standard ID Range (0 to 2047) and a previously learned or sniffed payload [38]. A fuzzer should include the following components [39]:

- A fuzz generator that assembles input from non-random components and random components with a sufficient amount of randomness
- A deliver mechanism that sends the generated inputs to the SUT
- A monitoring system (test oracle), which interprets the results such as

SUT responses, monitored network communication, debug interface output, system signals or other physical responses and performs decisions based on it e.g. if a test passes or fails.

By using this approach, no in-depth knowledge about the SUT is needed and every component that provides external interfaces can be targeted for testing, including ECU software, ECU hardware, protocols and busses (e.g. CAN). Fuzzing may be utilized in the automotive environment to [40]:

- Reverse engineer messages on busses
- Disrupt an in-vehicle communication network
- perform a cyber-attack
- lead to vehicle component damage.

Depending on the used interface and protocol it may not be possible to fuzz-test every possible combination of input in its entirety in a feasible time frame. Therefore, it makes sense to pre-select meaningful value and position ranges for randomized content. Because of this potentially large test case space, fuzzing may be applied in parallel to other test methods as long as the complete run-time is still in a defined range and produces positive results.

In case of the AVL AXE, fuzzing CAN bus signals is a very common use case. A fuzzing software e.g. booFuzz, American Fuzzy Lop or caring caribou is armed with valid CAN Messages or a template with a specification which parts of messages should be randomized and then handles the tasks of subsequently sending the (generated) data to the SUT as well as receiving and interpreting the feedback (such as Vector Tools CANoe).

8.5 Conclusion

This chapter showed a holistic approach of cybersecurity testing of modern vehicles over the complete life cycle. It showed how, proceeding from threat modeling and variant management, test cases can (semi-)automatically be derived using structured processes and learning techniques. The generated tests are subsequently executed on an automated platform that is capable of controlling the test and/or simulation setup and applying the respective attack vector. The described methodology provides an end-to-end means to test vehicular systems over the complete life cycle.

Acknowledgements

This research received funding within the ECSEL Joint Undertaking (JU) under grant agreement No. 876038 (project InSecTT). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Sweden, Spain, Italy, France, Portugal, Ireland, Finland, Slovenia, Poland, Netherlands, Turkey. The document reflects only the author's view and the Commission is not responsible for any use that may be made of the information it contains.

Bibliography

[1] European Automobile Manufacturers' Association (ACEA), "Vehicles in use europe 2022," tech. rep., European Automobile Manufacturers' Association (ACEA), 2021.

[2] United Nations Economic and Social Council - Economic Commission for Europe, "Uniform provisions concerning the approval of vehicles with regards to cyber security and cyber security management system," Regulation "155", United Nations Economic and Social Council - Economic Commission for Europe, Brussels, 2021.

[3] International Organization for Standardization and Society of Automotive Engineers, "Road Vehicles – Cybersecurity Engineering," ISO/SAE Standard "21434", International Organization for Standardization, 2022.

[4] S. Marksteiner and Z. Ma, "Approaching the Automation of Cyber Security Testing of Connected Vehicles," in *Proceedings of the Central European Cybersecurity Conference 2019*, CECC 2019, (New York, NY, USA), ACM, 2019.

[5] S. Marksteiner, N. Marko, A. Smulders, S. Karagiannis, F. Stahl, H. Hamazaryan, R. Schlick, S. Kraxberger, and A. Vasenev, "A Process to Facilitate Automated Automotive Cybersecurity Testing," in *2021 IEEE 93rd Vehicular Technology Conference (VTC Spring)*, (New York, NY, USA), IEEE, 2021.

[6] M. Ebrahimi, S. Marksteiner, D. Ničković, R. Bloem, D. Schögler, P. Eisner, S. Sprung, T. Schober, S. Chlup, C. Schmittner, and S. König, "A systematic approach to automotive security," in *Formal Methods* (M. Chechik, J.-P. Katoen, and M. Leucker, eds.), (Cham), pp. 598–609, Springer International Publishing, 2023.

[7] D. Ward, I. Ibarra, and A. Ruddle, "Threat Analysis and Risk Assessment in Automotive Cyber Security," *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, vol. 6, no. 2013-01-1415, pp. 507–513, 2013.

[8] T. Dobes, T. Kaserer, N. Schuch, and G. Storfer, "Smart Variant Calibration with Data Analytics," *ATZ - Automobiltechnische Zeitschrift*, no. Extra August 2018, 2018.

[9] M. Rathfelder, H. Hsu, T. Brandau, and G. Storfer, "Calibration Data Management for Porsche Chassis Systems," *ATZ worldwide*, vol. 118, pp. 16–21, June 2016.

[10] A. C. Franco da Silva, S. Wagner, E. Lazebnik, and E. Traitel, "Using a Cyber Digital Twin for Continuous Automotive Security Requirements Verification," *IEEE Software*, pp. 0–0, 2022.

[11] S. Marksteiner, S. Bronfman, M. Wolf, and E. Lazebnik, "Using Cyber Digital Twins for Automated Automotive Cybersecurity Testing," in *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pp. 123–128, Sept. 2021.

[12] International Organization for Standardization and Society of Automotive Engineers, "Road vehicles – Functional safety," ISOStandard 26262, International Organization for Standardization, 2018.

[13] International Organization for Standardization, "Tractors and machinery for agriculture and forestry – Safety-related parts of control systems," ISOStandard 25119, International Organization for Standardization, 2018.

[14] A. Shostack, *Threat Modeling: Designing for Security*. John Wiley & Sons, 2014.

[15] M. El Sadany, C. Schmittner, and W. Kastner, "Assuring compliance with protection profiles with ThreatGet," in *Computer Safety, Reliability, and Security* (A. Romanovsky, E. Troubitsyna, I. Gashi, E. Schoitsch, and F. Bitsch, eds.), (Cham), pp. 62–73, Springer International Publishing, 2019.

[16] F. Vaandrager, "Model learning," *Communications of the ACM*, vol. 60, pp. 86–95, Jan. 2017.

[17] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, Nov. 1987.

[18] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), Lecture Notes in Computer Science, (Cham), pp. 307–322, Springer International Publishing, 2014.

[19] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," in *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, (New York, NY, USA), pp. 411–420, Association for Computing Machinery, Feb. 1989.

[20] M. Isberner, F. Howar, and B. Steffen, "The Open-Source LearnLib," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), Lecture Notes in Computer Science, (Cham), pp. 487–495, Springer International Publishing, 2015.

[21] F. D. Garcia, "Tutorial: Proxmark, the Swiss Army Knife for RFID Security Research," tutorial at 8th Workshop on RFID Security and Privacy (RFIDSec 2012), Radboud University, Nijmegen, 2012.

[22] 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.

[23] S. Marksteiner, M. Sirjani, and M. Sjödin, "Using Automata Learning for Compliance Evaluation of Communication Protocols on an NFC Handshake Example," in *Engineering of Computer-Based Systems* (J. Kofroň, T. Margaria, and C. Seceleanu, eds.), vol. 14390 of *Lecture Notes in Computer Science*, (Cham), pp. 170–190, Springer Nature Switzerland, 2024.

[24] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," in *Proceedings of the 1998 Workshop on New Security Paradigms*, pp. 71–79, ACM, 1998.

[25] B. Schneier, "Attack trees," *Dr. Dobb's journal*, vol. 24, no. 12, pp. 21–29, 1999.

[26] C. A. Petri, *Kommunikation mit Automaten*. PhD thesis, Technische Universität Darmstadt, 1962.

[27] V. Varadharajan, "Petri net based modelling of information flow security requirements," in *[1990] Proceedings. The Computer Security Foundations Workshop III*, pp. 51–61, 1990.

[28] S. Marksteiner and P. Priller, "A Model-Driven Methodology for Automotive Cybersecurity Test Case Generation," in *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pp. 129–135, Sept. 2021.

[29] S. Otten, T. Glock, C. P. Hohl, and E. Sax, "Model-based Variant Management in Automotive Systems Engineering," in *2019 International Symposium on Systems Engineering (ISSE)*, pp. 1–7, 2019.

[30] M. Sirjani, "Rebeca: Theory, applications, and tools," in *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures* (F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, eds.), vol. 4709 of *Lecture Notes in Computer Science*, pp. 102–126, Springer, 2006.

[31] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "Slam and static driver verifier: Technology transfer of formal methods inside microsoft," in *International Conference on Integrated Formal Methods*, (Berlin, Heidelberg), pp. 1–20, Springer, 2004.

[32] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. W. Vaandrager, "Automata learning through counterexample guided abstraction refinement," in *FM 2012*, (Berlin), pp. 10–27, Springer, 2012.

[33] B. K. Aichernig, E. Muškardin, and A. Pferscher, "Learning-Based Fuzzing of IoT Message Brokers," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 47–58, Apr. 2021.

[34] C. Wolschke, S. Marksteiner, T. Braun, and M. Wolf, "An Agnostic Domain Specific Language for Implementing Attacks in an Automotive Use Case," in *The 16th International Conference on Availability, Reliability and Security*, ARES 2021, (New York, NY, USA), pp. 1–9, Association for Computing Machinery, Aug. 2021.

[35] A. Ladino, L. Xiao, K. Adjenugwhure, N. Deschle, and G. Klunder, "Cross-platform simulation architecture with application to truck platooning impact assessment," in *ITS World Congress*, 2021.

[36] European Telecommunications Standards Institute, "Intelligent transport systems (its); vehicular communications; basic set of applications; definitions," ETSI "TS 102 638", European Telecommunications Standards Institute, 2009.

[37] R. McNally, K. K.-H. Yiu, D. A. Grove, and D. Gerhardy, "Fuzzing: The state of the art," Tech. Rep. ADA558209, Defense Technical Information Center, 2012.

[38] P. Lapczynski, H. Heinemann, T. Schöneberger, and E. Metzker, "Automatically generating fuzz tests from automotive communication databases," in *5th escar USA, Detroit, isits AG*, June 2017.

[39] H. Lee, K. Choi, K. Chung, J. Kim, and K. Yim, "Fuzzing can packets into automobiles," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pp. 817–821, 2015.

[40] D. S. Fowler, J. Bryans, S. A. Shaikh, and P. Wooderson, "Fuzz testing for automotive cyber-security," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 239–246, 2018.

Chapter 9

Paper III: Using Automata Learning for Compliance Evaluation of Communication Protocols on an NFC Handshake Example

Stefan Marksteiner, Marjan Sirjani, and Mikael Sjödin.

Proceedings of the Engineering of Computer-Based Systems Conference (ECBS) 2023, J. Kofroň, T. Margaria, and C. Seceleanu, Eds., in Lecture Notes in Computer Science, vol. 14390. Cham: Springer Nature Switzerland, 2023. Reproduced with permission from Springer Nature. DOI: 10.1007/978-3-031-49252-5_13.

Abstract

Near-Field Communication (NFC) is a widely adopted standard for embedded low-power devices in very close proximity. In order to ensure a correct system, it has to comply to the ISO/IEC 14443 standard. This paper concentrates on the low-level part of the protocol (ISO/IEC 14443-3) and 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 modelled after the standard, which provides a strong verdict of conformance or non-conformance. 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 system that a) consist of two very similar structures and b) very frequently give no answer (i.e. a timeout as an output).

9.1 Introduction

In this paper we describe an approach of very thoroughly evaluating the compliance of Near-Field Communications (NFC)-based chip systems with the ISO/IEC 14443-3 NFC handshake protocol [1] using formal methods, concretely automata learning and equivalence checking. We present a tool chain that is easy to use - both the learning and the equivalence checking can run fully automatic. A complete automaton of the system-under-test (SUT) compared with a specification automaton modeled after the standard, provides a strong complement to conformance testing. The remainder of this paper structures as follows. First we provide its motivation and contribution. Section 9.2 gives an overview of basic concepts in this paper, including a formal definition of bisimulation for Mealy Machines as used in this paper. Section 9.3 describes the developed interface for automata learning of NFC systems, while Section 9.4 describes the learning setup including a comparison of different algorithms and calibrations to be most suitable for the specifics of the NFC handshake protocol. Section 9.5 shows real-world results, while Section 9.6 compare them to the works of others. Section 9.7, eventually, concludes the paper and gives and outlook on future work.

9.1.1 Motivation

As the NFC protocol is widely adopted in a broad variety of different, often security-critical, chip systems like banking cards, passports, access systems, etc., that use relatively weak hardware, a correct implementation is utterly important. While there are many works about security weaknesses in NFC (e.g., [2, 3]), also specifically regarding the ISO/IEC 14443-3 handshake (e.g., [4, 5]), there is few works on comprehensive testing (see Section 9.6). Assuring the correctness of the system is a principal step in the quest to trustworthy systems. As there is, to the best of our knowledge, no comprehensive works regarding assessment of the handshake protocols, as the fundament of secure protocols build atop, we aim for a strong verdict of ISO compliance for NFC systems. 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 Section 9.2.2).

9.1.2 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 strong compliance checking:

- Insights regarding the specifics of learning NFC using active automata learning
- An evaluation on the performance of different learning algorithms in systems with very similar structures
- Developing an NFC interface for a learning system
- An approach for automated compliance checking using bisimulation and trace equivalence

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 system-under-learning 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 [6] with Rivest/Schapire (LSR) closure [7] 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 [8] 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. Lastly, we 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.

9.2 Preliminaries

This section outlines the theoretical fundamentals of state machines and automata learning including a definition of equivalence and bisimilarity in the context of this paper. It further briefly describes the used framework and the basics and characteristics of the scrutinized protocol.

9.2.1 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 [9]. 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, Ω the output alphabet (that may or may not 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 ($Q \times \Sigma \rightarrow Q \times \Omega$). An even simpler type of automaton is a deterministic finite acceptor (DFA) [10]. 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 of DFAs 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 [11].

9.2.2 Transitions and Equivalence

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 [1] describe 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). Figure 9.1 shows a very simple example of a three-state automaton and its behavior-equivalent (minimal) two-state counterpart.

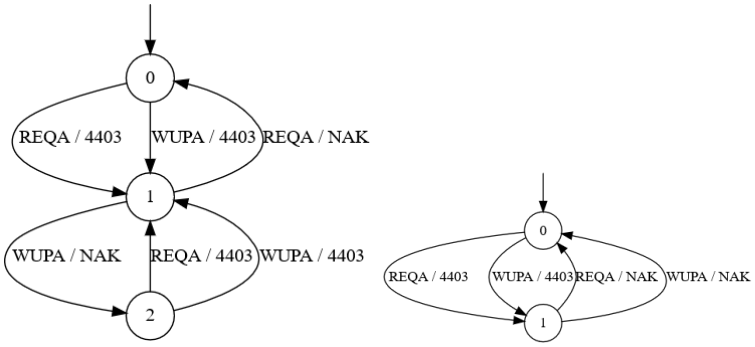

Figure 9.1: Example for a partial automaton and its minimal counterpart.

To compare this type of equivalence between two LTS LTS_1 and LTS_2, commonly used are (various degrees of) simulation, bisimulation (noted as $LTS_1 \sim LTS_2$) and trace equivalence. Simulation means that one automaton can completely reproduce the behavior of the other, for the bisimulation, this relation becomes bidirectional (i.e. functional). Trace equivalence compares the respective output of automata. Just (uni-directional) simulation alone is not sufficient as this would only 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 $LTS = (S, Act, \rightarrow, I, AP, L)$, with S 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.

Definition 1 (Bisimilarity)**.** *Bisimlarity of two LTS LTS_1 LTS_2 is defined as exhibiting a binary relation $R \subseteq QxQ$, such that [12]:*

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

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 $Post(s, \alpha) = \{s\prime \in S | s \xrightarrow{\alpha} s\prime\}$ and $Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$, meaning the union of all action successions, which again are again the result the transition function with a defined action and state as input. 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 [13]. Based on this, we define Mealy bisimilarity ($M_1 \; M_2$) for our purpose follows:

Definition 2 (Mealy Bisimilarity)**.** *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 $q_1 \in Q_1, q_2 \in Q_2 \cdot (q_1, q_2) \in R$ *must hold*

1) $\sigma \in \Sigma \cdot \lambda_1(q_1, \sigma) = \lambda_2(q_2, \sigma)$
2) *if* $q_1\prime \in Post(q_1)$ *then there exists* $q_2\prime \in Post(q_2)$ *with* $(q_1\prime, q_2\prime) \in R$
3) *if* $q_2\prime \in Post(q_2)$ *then there exists* $q_1\prime \in Post(q_1)$ *with* $(q_1\prime, q_2\prime) \in R$

As the transition function is dependent on the input, we define $Post(q, \sigma) = \delta(q, \sigma)$ and $Post(\sigma) = \bigcup_{\sigma \in \Sigma} Post(q, \sigma)$, which is essentially the same as for LTS brought into the notation of Section 9.2.1. There are a couple of different bisimulation types that differentiate by the handling of non-observable (internal) transitions (ordinarily labeled as τ transitions), e.g. strong and weak bisimulation, and branching bisimulation to give a few examples. 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 SUTs are black boxes. This means that internal state changes (commonly denoted as τ) are not visible, which determines the kind of bisimulation. 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 $Q = Q_{M_l} = Q_{M_s}; n = |Q|; \forall n \in Q | q_{nM_l} = q_{n_{M_s}}$).

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

Definition 3 (Trace equivalence)**.** $Traces(LTS_1) = Traces(LTS_2)$

Although both bisimulation and trace equivalence might be principally capable of comparing a specification with an implementation automaton for determining the standard compliance, determining bisimulation is a problem to be solved in efficiently, whereas trace equivalence is PSPACE complete [14]. However, this might be negligible with a relatively low number of states and transitions. 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 [12]. 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 Section 9.4.4). Therefore, trace equivalence is preferred as it is sufficient for standard compliance, but bisimilarity might be used in cases where more efficient checking algorithms are necessary.

9.2.3 Automata Learning

The classical method of actively learning automata of systems, was outlined in Angluin's pivotal work known as the L* algorithm [6]. This work uses a *minimally adequate Teacher* that has (theoretically) perfect knowledge of the SUT (in this case called System-under-learning – 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) [15]. 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[8], is deemed currently the most efficient [16]. Other widely used algorithms include a modified version of the original L* with a counterexample handling strategy by Rivest and Schapire [7], or the tree-based Direct Hypothesis Construction (DHC) [17] and Kearns-Vazirani (KV) [18] 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 [19]. Originally for Deterministic Finite Automata, this learning method could be used to learn Mealy Machines [20]. This preferred for learning black box reactive systems (e.g. cyber-physical systems), as modeling these as Mealy is comparatively simple.

9.2.4 LearnLib

To utilize automata learning we use a widely adopted Java library called LearnLib [21]. 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 systems under test. The interface classes further allow for defining the input alphabets that the algorithm routines uses to factor queries used to fill an observation table or tree. Depending on the used algorithms, the li-

brary is capable of inferring DFAs, NFAs (Non-deterministic finite acceptors), Mealy machines or VPDAs (Visibly Pushdown Automata).

9.2.5 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 device PCD) that also serves as field for data transmission. There are a couple of defined procedures that allow for operating proximity cards in presence of other wireless objects in order to exchange data [22]. One standard particularly defines two handshake procedures based on cascade-based anti-collision and card selection (called type A and type B), one of which NFC proximity cards must be compliant with [1]. This handshake is the particular target system-under-learning (SUL) of this paper, with the purpose of providing very strong evidence for compliance. Due to the proliferation and the nature of the given system-under-learning, this paper concentrates on type A devices. Therefore, all statements on NFC and its handshake apply for type A only.

9.2.6 The NFC Handshake Automaton

ISO 14443-3 contains a state diagram that outlines the Type A handshake procedure for an NFC connection (see Figure 9.2). This diagram is not a state machine of the types described in Section 9.2.1, for it lacks both output and final states. As we learn Mealy machines, we augmented it with abstract outputs (see Sections 9.4.2 and 9.4.4) 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 [22]) 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) occur. In the latter state, the card waits for a request to answer-to-select (*RATS*), which brings it into

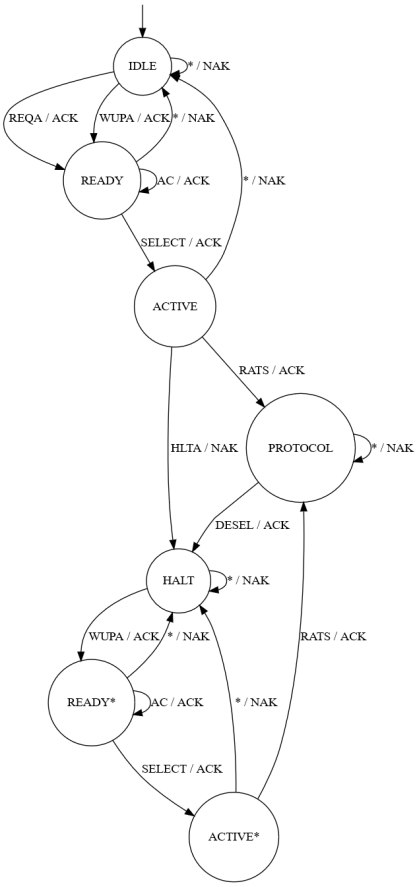

Figure 9.2: NFC handshake automaton after ISO 14443-3 [1] augmented with abstract outputs. Note: star (*) as input means any symbol that is not explicitly stated in another outbound transition of the respective state.

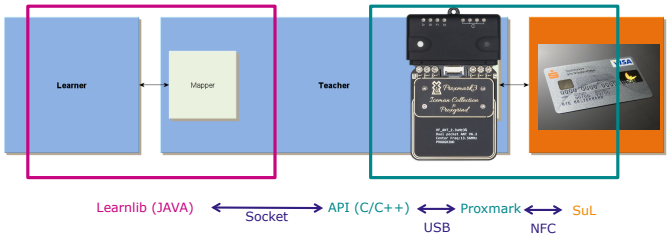

Figure 9.3: NFC interface setup.

said *PROTOCOL* state. In all of these states, an unexpected input would return the system to the *IDLE* state, no giving an 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 a *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** state 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 Section 9.4.2)

There is also one specific feature to the NFC handshake protocol: unlike most communication protocols, an unexpected or wrong input yield to no output. This has an implication to learning, as a timeout will be interpreted as a general error message.

9.3 NFC Interface

As Learner, we use the algorithm implementations in the Learnlib Java library (see Section 9.2.4), configured as outlined in Section 9.4. To interact with the NFC SUL, a Proxmark RFID/NFC device (see Section 9.3.1) is used that works with an adapter written in C++ (see Section 9.3.2). Figure 9.3 provides an overview of the setup.

9.3.1 Learner Interface Device

The interface with an NFC SUL is established via Proxmark3. Proxmark3 is a pocket-size NFC device capable of acting as an NFC reader (PCD) or tag (PICC), as well as sniffing device [23]. 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 Section 9.3.2).

9.3.2 Adapter Class

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 Section 9.4.2) 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 non-deterministic 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 re-implemented to send bulk inputs (i.e. send complete input words instead of single symbols), which improved the throughput significantly and solved non-determinism.

Firmware Modifications

In order to be able to transfer traces word-wise instead of symbol-wise, 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 SUT. 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 time outs these modifications, eliminating a great portion of the latency times of frequently used USB connections, boost the performance of the learning using different learning algorithms significantly (for a performance evaluation see Section 9.4.1).

9.4 Learning Setup

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. A positive output on the other hand, ordinarily consists of a standardized status code or information that is needed

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

Table 9.1: Runtime (minutes) per algorithm and maximum word length.

for the next phase of the handshake, e.g., parts of a card's unique identifier (UID). The non-answer to undefined is a characteristic feature of the NFC standard. This directly affects the learning because it yields many identical answers and efficient time-out handling is essential. It is therefore necessary to evaluate different state-of-the-art learning algorithms for their specific fitness (see Section 9.4.1) well as determining the optimal parameter set (Section 9.4.1). 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.

9.4.1 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*), is determined by the method and number of conformance tests. Generally speaking, a too short (maximum) word length results in an incompletely learned (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 9.1 shows a comparison of the runtime of different algorithms with different maximum word lengths (in red the respective algorithm's shortest runtime that learned the correct 7-state model). Some of the non-steadiness in the results can be explained by the fact that some calibrations with shorter word lengths required more equivalence queries and, thus, refinement proce-

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	10192	2588	5124	7932	7607	6628	6213
EQs	2	3	2	5	5	4	4

Table 9.2: Performance evaluation of different algorithms for a compliant system with their respective fastest calibration in the given setting.

dures. Table 9.2 shows the results with the best performing (correct) run of the respective algorithm. This, however, only covers the performance of learning a correct implementation. The opposite side, discovering a bug, shows a different picture. We therefore used a SUT with a slightly deviating behavior (see Section 9.5.3). 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 by their 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 non-compliant 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 SUT's actual behavior model, KV was performing worst. The runtimes roughly correspond with the amount of sent symbols, in this case the a very long timeout has to be set to avoid non-determinism. The classical L* is not in the list, as the algorithm crashed after more than 24 hours of runtime. Table 9.3 provides an overview of minimum word lengths, run time, words, symbols and equivalence queries. Lower minimum word lengths yielded false negatives (i.e. the result showed a correct model with the deviation not uncovered).

9.4.2 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 (∇), 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 rationales 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 section 9.3.2).

Input Alphabet

For the input alphabet we use the one needed for successfully establishing a handshake (cf. Figure 9.2), according to the state diagram for Type-A cards in the ISO 14443-3 standard [1]:

- 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 [22]. 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.

Output Alphabets

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

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	14637	15262	23867	19241	13353	11769
Eqs	5	3	6	6	5	5

Table 9.3: Performance evaluation of different algorithms for a non-compliant system with their respective fastest calibration in the given setting.

if a system behaved non-deterministicly, 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 anti-collision (*AC*) and *SELECT* yields a different output, which introduces non-deterministic 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 Section 9.2.5). 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 Section 9.2.6). This idea was therefore forfeit in favor of raw output for the learning. We still maintained this higher abstraction for the equivalence checking (see Section 9.4.4 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 anti-collision 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.

9.4.3 Labeling and Simplification

An implementation that conforms to the standard will automatically labeled correctly, as the labelling 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 acknowledgement 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 non-compliance with the ISO 14443-3 standard (given that only the messages defined in that standard are used as an input alphabet - see Section 9.4.2).

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.

9.4.4 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 Section 9.2.2) [24]. As the Learnlib toolset provides to possibility to store the learned automata in a couple of formats, including Aldebaran, setting up the tool chain is easy, even though some re-engineering was necessary. Learnlib's standard function for exporting in the Aldebaran format does not include outputs. This accepts transitions as equal that are in fact not (as they distinguish only through the output). 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, amongst others, the algorithm of Jansen et al. [25] for bisimilarity checking. We therefore simply model the specification in form of the handshake diagram (see Figure 9.2) 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 learnt implementations. The models of SUTs, although, could differ greatly event if the behavior is similar . 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. E.g., 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 non-compliance to the ISO 14443-3 standard of the scrutinized implementation. If a non-compliance (i.e. a missing or additional state or transition actually countering the bisimulation relation) is found, all we need is to do a simple conformance test. A trace of the non-compliant state/transition is trivial to extract from the automaton (see the example in Section 9.5.3). If that trace is executed on the system-under-test and actually behaves like predicted in the model, we have found the actual specification violation in the real system, disproving the compliance.

Alternatively, an actual positive verdict of compliance of a learned model is simple. A full compliance proof can be made when doing identity equivalence, that is comparing the learned model state by state and transition by transition with the model manually derived from the ISO 14443-3 standard. If every state and transition is equal, we consider the system as compliant. More formally, the learned machine M_l must be fully equal the specification machine M_s, i.e. $M_l = M_s \wedge (M_l = M_s \models Q_{M_l} = Q_{M_s} \wedge \Sigma_{M_l} = \Sigma_{M_s} \wedge \Omega_{M_l} = \Omega_{M_s} \wedge \delta_{M_l} = \delta_{M_s} \wedge \lambda_{M_l} = \lambda_{M_s} \wedge q_{0_{M_l}} = q_{0_{M_s}})$. This, obviously, is a simpler but stronger relation that is not coersive for ISO protocol compliance. The probability of learning (with a sufficient amount of conformance testing) an incorrect model that is still compliant with the standard is negligible.

9.5 Evaluation

In this section we briefly outline the achieved results with the described tool chain. We used serveral 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.

9.5.1 Test Cards and Credit Cards

We used five different NFC test 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 are conform to the standard, with only minor differences. One of these deffenrences is replying with diffent ATQA to REQA/WUPA messages with 4400 and 4800 respectively. Overall, the results with these cards are very similar. Figure 9.4 shows an example of a learnt automaton (left side).

9.5.2 Passports

We also examined two different passports from European Union countries: one German and one Austrian. The main noticeable difference (at ISO 14443-3 levlel) between the 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.

9.5.3 Tesla Key Fob

Apart from significantly slower answers than the other devices, which required to adapt the timeouts to avoid nondeterministic behavior, the learned automaton slightly differs when learnt with the TTT algorithm. Figure 9.4 (right side) shows a model of a Tesla car key fob learnt with TTT. The (unnamed) states 3,4 and 6 are very similiar 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

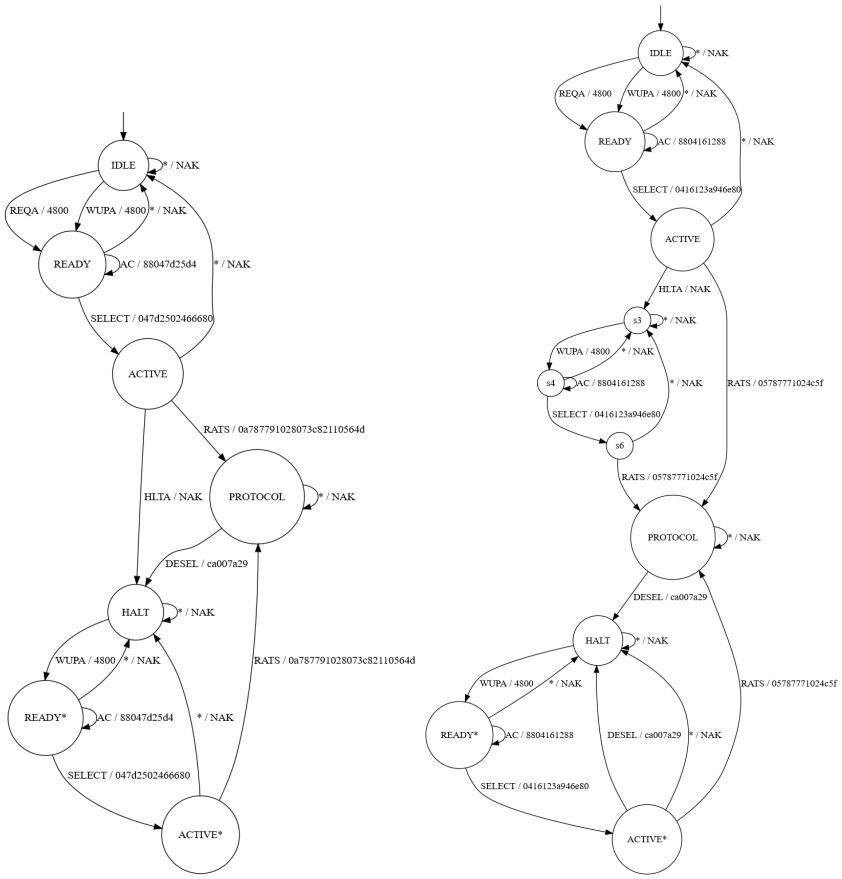

Figure 9.4: Automaton of an NXP test card (left) and a Tesla car key fob (right) learnt with TTT.

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 ($\langle$*REQA/ACK*$\rangle$, $\langle$*SELECT/ACK*$\rangle$, $\langle$*RATS/ACK*$\rangle$, $\langle$*DESEL/ACK*$\rangle$, $\langle$*WUPA/ACK*$\rangle$, $\langle$*SELECT/ACK*$\rangle$, $\langle$*DESEL/ACK*$\rangle$). According to the specification, the last label should

be ⟨DESEL/NAK⟩.

9.6 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 [26] or probabilistic transition systems (PTS) [27]. Neider et al. [28] contains some significant theoretic fundamentals of using automata learning and bisimulation for different types of state machines, including Mealys. It also contains the important observation that (generalized) Mealy Machines are bisimilar if their underlying LTS are bisimilar. Tappler et al. [13] 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 [29]. There is also previous work on using automata learning for inferring models of NFC cards [30], 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 protocol.

9.7 Conclusion

In this paper, we demonstrated the usage of automata learning to infer models of systems under test 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' performances.

9.7.1 Discussion

Using our learning setup on real-world devices, we found little differences between the SUTs – all examined systems were compliant to ISO/IEC 14443-3. Observed differences were mainly in the privacy-related random UIDs sent by passports and the slow answers and a slightly different automaton of the Tesla key fob. 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 (IDLE/READY/ACTIVE and HALT/READY*/ACTIVE*) 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. However, TTT performed best in finding a non-compliant system, which is the actual purpose of the testing and that the minimum word length has an impact on the ability to find incompliances. This might give some hints for optimization of learning strategies for similar structures.

9.7.2 Outlook

The compliance checking is but a first step towards assuring correctness and, subsequently, cybersecurity for NFC systems. Concretely, further research directions include test case generation using model checking and using the model to guide an intelligent fuzzer to leverage cybersecurity validation and verification (V&V). The target of these V&V activities are on the one hand upper layer protocols and on the other hand NFC reader devices to search for faults that might lead to exploitable security vulnerabilities. To talk to readers, because of the low latency of NFC communications, it is crucial to already know what to send before a conversation, which is satisfied by the predefined input words in the automata learning process.

Acknowledgements

This research received funding within the ECSEL Joint Undertaking (JU) under grant agreement No. 876038 (project InSecTT) 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 programme and Austria, Sweden, Spain, Italy, France, Portugal, Ireland, Finland, Slovenia, Poland, Netherlands, Turkey. The document reflects only the author’s view and the Commission is not responsible for any use that may be made of the information it contains.

Bibliography

[1] 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.

[2] W. Issovits and M. Hutter, "Weaknesses of the ISO/IEC 14443 protocol regarding relay attacks," in *2011 IEEE International Conference on RFID-Technologies and Applications*, pp. 335–342, Sept. 2011.

[3] J. Vila and R. J. Rodríguez, "Practical Experiences on NFC Relay Attacks with Android," in *Radio Frequency Identification* (S. Mangard and P. Schaumont, eds.), Lecture Notes in Computer Science, (Cham), pp. 87–103, Springer International Publishing, 2015.

[4] G. Hancke, "Practical attacks on proximity identification systems," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 6 pp.–333, May 2006.

[5] M. Maass, U. Müller, T. Schons, D. Wegemer, and M. Schulz, "NFCGate: An NFC relay application for Android," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, (New York, NY, USA), pp. 1–2, Association for Computing Machinery, June 2015.

[6] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, Nov. 1987.

[7] R. L. Rivest and R. E. Schapire, "Inference of Finite Automata Using Homing Sequences," *Information and Computation*, vol. 103, pp. 299–347, Apr. 1993.

[8] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), Lecture Notes in Computer Science, (Cham), pp. 307–322, Springer International Publishing, 2014.

[9] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, pp. 1045–1079, Sept. 1955.

[10] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, Dec. 1943.

[11] E. F. Moore, "Gedanken-Experiments on Sequential Machines," in *Automata Studies*, vol. 34 of *AM-34*, pp. 129–154, Princeton University Press, 1956.

[12] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, Apr. 2008.

[13] M. Tappler, B. K. Aichernig, and R. Bloem, "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.

[14] L. Aceto, A. Ingolfsdottir, and J. Srba, "The algorithmics of bisimilarity," in *Advanced Topics in Bisimulation and Coinduction*, pp. 100–172, Cambridge University Press, 2011.

[15] B. Jacobs and A. Silva, "Automata Learning: A Categorical Perspective," in *Horizons of the Mind. A Tribute to Prakash Panangaden: Essays Dedicated to Prakash Panangaden on the Occasion of His 60th Birthday* (F. van Breugel, E. Kashefi, C. Palamidessi, and J. Rutten, eds.), Lecture Notes in Computer Science, pp. 384–406, Cham: Springer International Publishing, 2014.

[16] F. Vaandrager, "Model learning," *Communications of the ACM*, vol. 60, pp. 86–95, Jan. 2017.

[17] M. Merten, F. Howar, B. Steffen, and T. Margaria, "Automata Learning with On-the-Fly Direct Hypothesis Construction," in *Leveraging Applications of Formal Methods, Verification, and Validation* (R. Hähnle,

J. Knoop, T. Margaria, D. Schreiner, and B. Steffen, eds.), vol. 336, pp. 248–260, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[18] M. J. Kearns and U. Vazirani, *An Introduction to Computational Learning Theory*. MIT Press, Aug. 1994.

[19] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black Box Checking," in *Formal Methods for Protocol Engineering and Distributed Systems: FORTE XII / PSTV XIX'99 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX) October 5–8, 1999, Beijing, China* (J. Wu, S. T. Chanson, and Q. Gao, eds.), IFIP Advances in Information and Communication Technology, pp. 225–240, Boston, MA: Springer US, 1999.

[20] M. Shahbaz and R. Groz, "Inferring Mealy Machines," in *FM 2009: Formal Methods* (A. Cavalcanti and D. R. Dams, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 207–222, Springer, 2009.

[21] M. Isberner, F. Howar, and B. Steffen, "The Open-Source LearnLib," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), Lecture Notes in Computer Science, (Cham), pp. 487–495, Springer International Publishing, 2015.

[22] 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.

[23] F. D. Garcia, G. de Koning Gans, and R. Verdult, "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.

[24] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse, "The mCRL2 Toolset for Analysing Concurrent Systems," in *Tools and Algorithms for the Construction and Analysis of Systems* (T. Vojnar and L. Zhang, eds.), Lecture Notes in Computer Science, (Cham), pp. 21–39, Springer International Publishing, 2019.

[25] D. N. Jansen, J. F. Groote, J. J. A. Keiren, and A. Wijs, "An O(m log n) algorithm for branching bisimilarity on labelled transition systems," in *Tools and Algorithms for the Construction and Analysis of Systems* (A. Biere and D. Parker, eds.), Lecture Notes in Computer Science, (Cham), pp. 3–20, Springer International Publishing, 2020.

[26] Y.-F. Chen, C.-D. Hong, A. W. Lin, and P. Rümmer, "Learning to prove safety over parameterised concurrent systems," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, pp. 76–83, Oct. 2017.

[27] C.-D. Hong, A. W. Lin, R. Majumdar, and P. Rümmer, "Probabilistic Bisimulation for Parameterized Systems," in *Computer Aided Verification* (I. Dillig and S. Tasiran, eds.), Lecture Notes in Computer Science, (Cham), pp. 455–474, Springer International Publishing, 2019.

[28] D. Neider, R. Smetsers, F. Vaandrager, and H. Kuppens, "Benchmarks for Automata Learning and Conformance Testing," in *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday* (T. Margaria, S. Graf, and K. G. Larsen, eds.), Lecture Notes in Computer Science, pp. 390–416, Cham: Springer International Publishing, 2019.

[29] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software," in *Formal Methods and Software Engineering* (M. Butler, S. Conchon, and F. Zaïdi, eds.), (Cham), pp. 67–83, Springer International Publishing, 2015.

[30] F. Aarts, J. De Ruiter, and E. Poll, "Formal Models of Bank Cards for Free," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pp. 461–468, Mar. 2013.

Chapter 10

Paper IV: From TARA to Test: Automated Automotive Cybersecurity Test Generation Out of Threat Modeling

Stefan Marksteiner, Christoph Schmittner, Korbinian Christl, Dejan Ničković, Mikael Sjödin, and Marjan Sirjani. Proceedings of the 7th ACM Computer Science in Cars Symposium, CSCS'23. New York: Association for Computing Machinery, 2023. DOI: 10.1145/3631204.3631864.

Abstract

The United Nations Economic Commission for Europe (UNECE) demands the management of cyber security risks in vehicle design and that the effectiveness of these measures is verified by testing. Generally, with rising complexity and openness of systems via software-defined vehicles, verification through testing becomes a very important for security assurance. This mandates the introduction of industrial-grade cybersecurity testing in automotive development processes. Currently, the automotive cybersecurity testing procedures are not specified or automated enough to be able to deliver tests in the amount and thoroughness needed to keep up with that regulation, let alone doing so in a cost-efficient manner. This paper presents a methodology to automatically generate technology-agnostic test scenarios from the results of threat analysis and risk assessment (TARA) process. Our approach is to transfer the resulting threat models into attack trees and label their edges using actions from a domain-specific language (DSL) for attack descriptions. This results in a labelled transitions system (LTS), in which every labelled path intrinsically forms a test scenario. In addition, we include the concept of Cybersecurity Assurance Levels (CALs) and Targeted Attack Feasibility (TAF) into testing by assigning them as costs to the attack path. This abstract test scenario can be compiled into a concrete test case by augmenting it with implementation details. Therefore, the efficacy of the measures taken because of the TARA can be verified and documented. As TARA is a de-facto mandatory step in the UNECE regulation and the relevant ISO standard, automatic test generation (also mandatory) out of it could mean a significant improvement in efficiency, as two steps could be done at once.

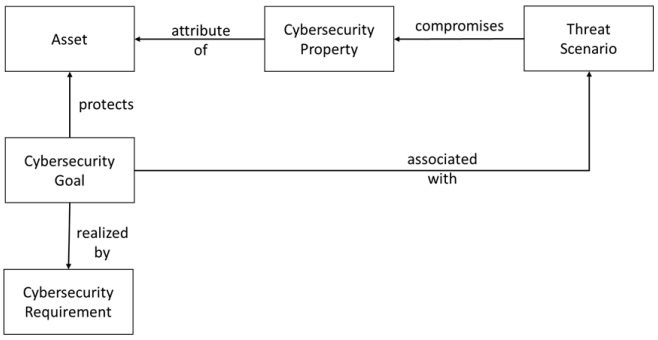

Figure 10.1: Relationship for risk mitigation.

10.1 Introduction

The market introduction of vehicle-to-x (V2X) functions and advanced driving assistance systems (ADAS) to automotive systems make them increasingly complex. At the same time, cybersecurity incidents (increasingly induced by criminals) display an exponential growth [1]. This is being recognized by standards and regulation bodies. For example, the United Nations Economic Commission for Europe (UNECE) issued a regulation (R155) that demands cybersecurity concerns to be addressed over the complete life cycle and verify the measures through testing [2]. Therefore, a holistic approach for cybersecurity engineering and testing over the complete life cycle is needed. This paper presents the confluence of a life cycle governance and a structured semi-automated testing approach to provide fast, comprehensive and cost-efficient cybersecurity testing over the complete automotive life cycle in conjunction with the concepts of Cybersecurity Assurance Levels (CALs) and Targeted Attack Feasibility (TAF). Section 10.2 describes the latter concepts and their integration in a security testing process. Section 10.3 elaborates automating the process of generating suitable threat models and attack trees. Section 10.4 describes the transfer mechanism from attack trees to agnostic test cases and their application to an actual implementation. Section 10.5 describes the application of the process in a small case study. Section 10.6 gives an overview of different work in this direction and Section 10.7, eventually, concludes the paper.

10.1.1 Motivation

As current standards (most prominently ISO/SAE 21434 and UNECE R155) lack the details of how to test, there are two initiatives ongoing in ISO's standardization: ISO/SAE PAS 8475 (WIP)[1] [3] that copes with Cybersecurity Assurance Levels (CALs) and Targeted Attack Feasibility (TAF) and ISO/SAE PAS 8477 (WIP) [4] that deals with verification and validation (V&V) methods. In order to include these concepts-in-development into security processes, giving clarity to Original Equipment Manufacturers (OEMs) and suppliers, this paper aims for giving suggestions how to align security testing on CALs and TAFs originating from the earliest stages of the (security) engineering process. Furthermore, the aim is to turn the overhead necessary for formalizing the combined engineering and testing process into an advantage by automatizing them. More specifically, these formalized processes can be used to automate test case generation from threat models. As a result, test case blueprints can be generated during the modeling process, that can be later on (semi-)automatically compiled into executable test cases. This allows for structured and efficient testing of the fulfillment of the requirements stemming from the threat analysis.

10.1.2 Contribution

This paper contributes mainly four things to the body of knowledge:

1. A structural concept how to incorporate CALs and TAFs into the cybersecurity engineering process.
2. A process to align testing on CALs and TAFs.
3. A method to generate attack trees from TARA.
4. A concept to transform attack trees into technology-agnostic test scenarios automatically as a blueprint to verify and validate security claims and requirements.

Item 1 explains the upcoming developments of ISO/SAE 8475 and describes the usage of CALs and TAFs (Section 10.2.1). Item 2 discusses the merit of the CAL/TAF usage in security testing (Section 10.2.1). Item 3 shows an approach how the formalization necessary to include the first two items can be used to increase the efficiency of testing by generating attack trees from a Threat

[1] Work-in-Progress

Analysis and Risk Assessment (TARA) (Section 10.3.1). Item 4 provides a method to transform attack trees into abstract test scenarios by labelling the edges with actions from the alphabet of a domain-specific language (DSL) for attack descriptions (see Section 10.4.2).

10.2 Automotive Security Communication

Effective communication plays a pivotal role in the automotive industry, particularly within the complex network of Original Equipment Manufacturers (OEMs) and Tier 1 and 2 suppliers. Especially in the cybersecurity domain, with interlocking layers of defense [5] the criticality of clearly communicating expected requirements, is required for achieving optimal outcomes. By fostering a shared understanding of risk mitigation strategies, OEMs and suppliers can collaboratively address cybersecurity challenges, enhance product security, and streamline operations. ISO/SAE 21434 defines here a framework in which during the Threat Assessment and Risk Analysis cybersecurity goals are defined. A cybersecurity goal is aimed at reducing the risk of threat scenarios and realized by cybersecurity requirements (see Figure 10.1). This process can be applied during all phases of the development, at item (see Section 10.2.3), system, or component level. Cybersecurity goals can be defined by the OEM and by the supplier. Cybersecurity requirements are assigned to components and implemented.

10.2.1 Cybersecurity Assurance Level (CAL)

An important aspect is here on the interplay between customer requirements and regulatory needs. As mentioned in the introduction, UNECE requires in the new UN R155 [2] that cybersecurity in a vehicle has to be tested and demonstrated during the type approval. With the complexity of modern vehicles, this testing effort needs to be distributed through the supply chain. ISO/SAE 21434 already establishes as an informative part the concept of the Cybersecurity Assurance Level (CAL). Inspired from assurance level schemes like the Common Critiera Evaluation Assurance Levels (EALs) [6], the goal of CAL is to describe the expected level of assurance and rigor for a defined cybersecurity goal. ISO/SAE 21434 defines an informative framework regarding the mapping of CAL to the impact and the attack vector. In addition, for concept and product development potential aspects that can be adjusted by CAL like testing effort or independence are given. CAL is assigned per cybersecurity goal

and derived requirements inherit the CAL. If a requirement addresses multiple cybersecurity goals, the highest CAL is inherited.

10.2.2 Target Attack Feasibility (TAF)

In practical applications of CAL and ISO/SAE 21434, there has been a noticeable lack of clarity regarding the expected strength of security controls. This ambiguity becomes particularly evident when suppliers attempt to translate high-level security goals and requirements into technical specifications and implementations. While CAL provides insights into the engineering rigor, it falls short in communicating their actual strength. To address this gap, the concept of Target Attack Feasibility (TAF) has been introduced. TAF is designed to be associated with specific security controls, offering a measure of their expected strength. For instance, a security goal such as ”protect the integrity of the message” could be interpreted through various security controls based on their TAF levels:

- TAF1: cryptographic hash
- TAF2: symmetric encryption
- TAF3: asymmetric encryption

However, the temporal relevance of TAF is still a topic of debate. As more TAF levels are designated to specific security control technologies, there's an increasing risk that these assignments might become obsolete over time. One potential solution is to map TAF levels to Attack Feasibility, where, for example, TAF1 would necessitate a specific level of expertise, equipment, and time to breach. This approach, in contrast to a fixed technological assignment, offers a more flexible interpretation, though it also introduces a degree of subjectivity.

10.2.3 Integrating CAL and TAF in security testing

Due to the impact of CAL and TAF on the overall process and especially on the cybersecurity testing, a well-structured process is necessary. We adapt here a testing process, presented in [7] and adapted to include CAL and TAF. The process is aligned with ISO/SAE 21434 [8]. The activities are basically sequential, although some activities provide input for more than one subsequent activity. Figure 10.2 provides an overview.

I Item Definition

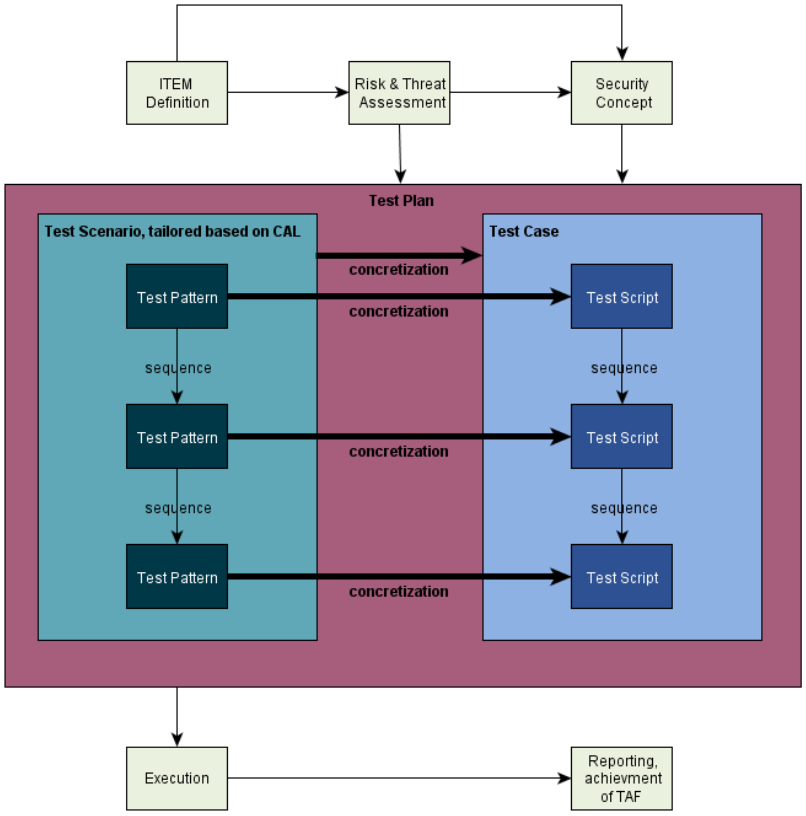

Figure 10.2: Layout of the security testing process from [7].

II Risk and Threat Assessment

III Security Concept Definition (including the test targets)

IV Test Planning and Scenario Development

(a) Penetration Test Scenario Development

(b) Functional and Interface Test Development

(c) Fuzz Testing Scenario Development

(d) Vulnerability Scanning Scenario Development

V Test Script Development

VI Test Script Validation

VII Test Case Generation

(a) Test Environment Preparation

VIII Test Case Execution

IX Test Reporting

In the item definition (i), the scope of the development is defined. This can range from a complete car model to specific systems or combination of systems. Risk and threat assessment (ii) (e.g., TARA [9, 10]) identifies potential vulnerabilities to be addressed and prioritizes them, focusing on certain threats that are deemed graver, while neglecting others. Here CAL and TAF are assigned for each Cybersecurity goal. The security concept definition (iii) mainly aims at anticipating measures to counter the threats from the previous activity. Measures that should be present and effective to counter specific threats that should be validated in the course of this process. TAF plays a major role in the selection of suitable security measures, that achieve a sufficient level of risk reduction. The test planning and scenario development (iv) derives an abstract test plan, consisting of scenarios, based on the security targets from the previous activity. The test plan should contain an overall test strategy. Tests are based on threats and focus on risky areas, denoted by an increased CAL. Test data inputs are selected based on threats from the risk analysis [11] and match test patterns which represent abstract (symbolic) actions in a distinct sequence. The scenarios are categorized into four classes [8]: penetration testing, functional and interface test, fuzz testing and vulnerability scanning. Although derived from the analysis of a test item, the scenario description is used to be generic: no specific information of an item on a lower technical level should be incorporated for portability reasons. Sensibly, descriptions could be composed in a domain specific language (DSL) for attack descriptions [12, 13, 14, 15]. Selection of scenarios and also independence of persons who test the SUT are based on CAL. The test script development (v) turns the test patterns from the scenarios into executable scripts. It should develop a script to match a test pattern by either using an existing exploit from an available database or develop an own attack on the system. This means that the pattern must be equipped

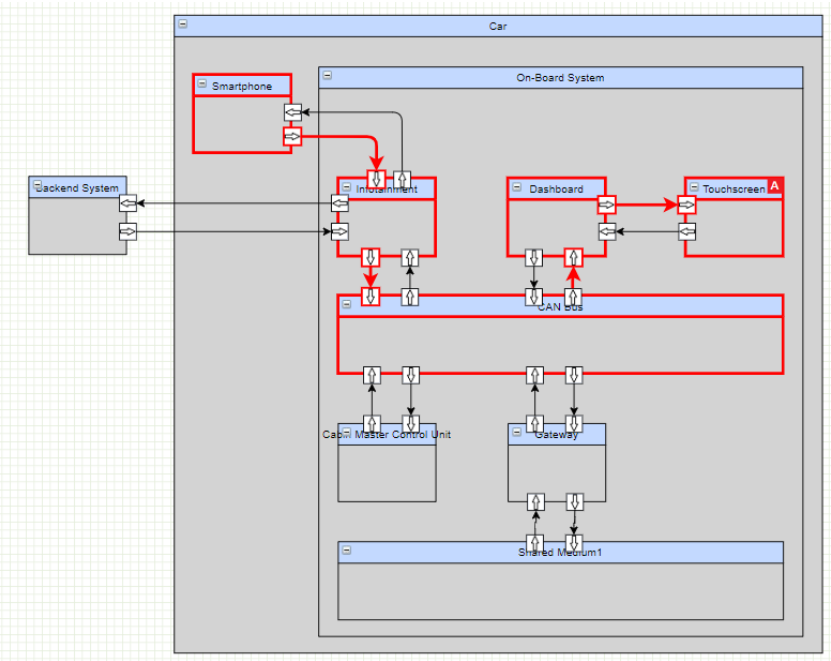

Figure 10.3: Example Threat Model.

with specific information and brought in a form that it is executable on a testing system, e.g., on a Linux shell. The test case generation (vi) assembles the test scripts to a consistent test case (a full attack on an SUT) by processing a DSL-based description (the generic test scenario) and using additional information from an SUT database, as well as using combinatorial methods to economically increase the test coverage [16]. Lastly, the tests have to be executed (vii) and their result reported (viii). These activities also include proper feedback from the test. If the process is to be automated, proper information for an autonomous test oracle has to be provided in the form of pre and post conditions that have to be fulfilled in order to assess a positive or negative (or even inconclusive) test result. Here the achievement of the intended TAF has to be included.

10.3 Threat Modeling

In this Section, we present an approach that generates test scenarios in a technology-agnostic manner out of a threat model. In the context of this paper, we conceptualize threat modeling as an iterative process used to identify and analyze potential threats in information technology (IT) systems. This iterative process basically requires two major components as inputs [17]. The first component is a threat model that summarizes the accumulated knowledge of known and documented threats, vulnerabilities, and weaknesses for the domain under study, such as automotive and IoT . It serves as a comprehensive repository of potential threats that could compromise the system. The second component is a systematic and abstract representation of the system under consideration. This representation contains all the key information required for a thorough threat analysis. Our approach uses an adapted version of the internal SysML block diagram that facilitates the representation of the relationships and properties of the system components, providing the basis for a comprehensive analysis.

The modeling process itself is the comparative analysis between the threat model and the system model. This critical comparison helps derive a list of existing threats, which is the completion of one cycle of the process. This list is expanded by recognizing the intrinsic interdependencies of the identified threats, which overcomes the limitation of looking at threats in isolation [18]. By leveraging the data revealed by the identified threats, we can explore the intricacies of their interdependencies. It is worth noting that threats rarely occur in a vacuum; they primarily build on previous steps and can trigger subsequent events.

To map these interdependencies, we use the concept of pre- and post-conditions. With this strategy, we can not only detect these dependencies, but also visually represent this additional information using attack graphs and attack trees to improve the understanding and analysis of potential threat interactions. In Section 3, we elaborate on the intricacies of this enhanced process, detailing the concept of threat interdependencies and the resulting strengthened approach to threat modeling.

Figure 10.3 shows an example of a threat model based on [19]. In this example, the electrical/electronic (E/E) architecture of an autonomous low-speed shuttle is presented. This architecture was modelled in ThreatGet, a tool for threat modelling and analysis, to facilitate automated security analysis and demonstrate the process from TARA to CAL and TAF.

We denote here one of many potential assets, with is the integrity of the Master Controller. If an attacker would be able to modify the firmware, he

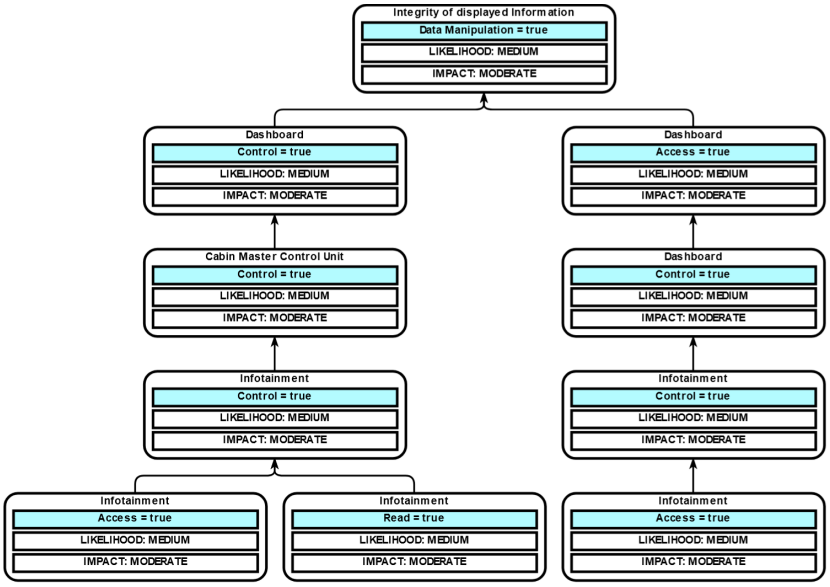

Figure 10.4: Example attack tree.

could send any command and cause potential safety and operational issues (due to the low speed of the vehicle)

- Asset: Firmware of the Master Controller (Integrity)
- Damage Scenario: Unintended steering causing collision with an obstacle ASIL C

An analysis shows a potential attack starting from an unencrypted wireless connection between external services and the AI & Drive Algorithm (see Figure 10.4). This allows an attacker to reach the dashboard and manipulate data on this element (=¿ violating the integrity of the displayed information).

In order to address this a security goal is defined, which states that the master controller has to be protected and this security goal gets a CAL assigned, based on the potential impact (CAL 3). This security goal is then mapped to a security requirement, that encryption with at least TAF 3 is added to external connections. TAF 3 could be mapped to asymmetric encryption.

```
PreConditions:
    bbshell: (bttarget)

Actions:
    bttarget: scan(type:BlueBorne, interface:BT_IF)
    bbshell: exploit(type:BlueBorne, target:bttarget)
    wifitarget: exploit(type:OpenAndroidHotspot, target:bttarget, shell:bbshell) default "
    adbshell: exploit(type:OpenADB, target:wifitarget)
    install_python_env: exploit(type:InstallPythonEnv, shell:adbshell)
    install_python_lib: exploit(type:InstallPythonLib, shell:adbshell)
    attackScript: exploit(type:InstallAndroidCANDosScript, shell:adbshell)
    can_attack: exploit(type:ADBPythonScript, shell:adbshell, file:"CanAttackScript", interval:5)


PostConditions:
    //can_attack: Oracle.CAN_MESSAGE("
```

Figure 10.5: Example for a test scenario in the used attack description language.

10.3.1 Threat-Interdependencies and Attack Trees

The Threat Analysis and Risk Assessment (TARA) process aims to identify potential threats and assess the associated risks to ensure effective risk mitigation [9, 19]. It involves systematically investigating threats, assessing their likelihood and impact, and developing strategies to address the identified risks [20]. The first step of the TARA process is to analyse for potential threats. This step is essential because only what has been identified can be assessed later. It involves identifying vulnerabilities, weaknesses or potential attack vectors [19, 21]. It is not advisable to look at threats solely in isolation as part of the TARA process, as this approach ignores the interactions between different threats. Threats often interact with or reinforce each other, resulting in attack chains or paths. Failure to consider these interactions can result in missing relevant risks and inadequate prioritization of resources for effective risk mitigation [22]. The concept of pre- and post-conditions for threats can be used to represent the interdependencies of threats within the TARA process. Preconditions represent the necessary circumstances or events that must be met for a threat to occur, while postconditions represent the possible consequences or outcomes that result from the occurrence of a particular threat. It should also be emphasized that the postconditions of some threats may be the preconditions of others. By identifying and analysing these preconditions and postconditions, we can better understand how threats are connected and how they propagate or influence each other [18]. In an attack tree [23, 24], the hierarchical structure illustrates the connections between threats, their relationships, and the different attack scenarios. The root of an attack tree is usually

connected to the attack target, which is the overall goal of an attacker. From this attack target, a security objective can be derived, which represents the desired outcome of the attack defence. By visualizing threats in an attack tree, we can analyse the preconditions and postconditions associated with each threat. Considering the interdependencies of threats within the attack tree not only simplifies, but also improves, the assessment of target attack feasibility [25]. By visualizing the connections and dependencies between different threats, it becomes easier to analyse the feasibility of attacking a particular target. Understanding how multiple threats contribute to a given postcondition provides a more comprehensive view of the potential attack surface and the likelihood of a successful attack [21]. Considering the inter-dependencies within the attack tree improves understanding of the overall risk landscape and facilitates more informed decision-making regarding resource allocation, security control implementation, and mitigation prioritization. This approach improves the accuracy and effectiveness of target attack feasibility assessments and results in more robust and proactive security measures. In addition, consideration of dependencies enables organizations to effectively prioritize remediation efforts. By identifying critical paths and dependencies within the attack tree, resources can be strategically allocated to protect the most vulnerable areas. While the CAL can be easily derived based on the impact, the TAF can focus on elements in the tree which have the highest contribution to the Attack Feasibility. In summary, considering interdependencies in the TARA process and attack tree not only simplifies but also improves threat assessment and increases overall cybersecurity. By understanding the interrelationships and dependencies, organizations can effectively identify, prioritize, and mitigate risks, resulting in higher CAL and greater confidence in the security of their systems.

10.4 Automated Testing

This section is concerned with the automated generation of security test cases stemming from a TARA using attack trees (see Section 10.3.1). The principal idea is to use the resulting attack tree and create blueprints for testing in the form of implementation-agnostic test scenarios, through mapping rule sets. These agnostic test scenarios can later be concretized and executed on a specific system implementation.

10.4.1 Security Tests and their relationship with the Security Analysis

Following the method in Section 10.2.3, we store blueprints for test cases in a system-agnostic manner in the ALIA DSL [12] as test scenarios (see Figure 10.5 for an example). These test scenarios are an abstract representation of actions to be taken to execute a test case. The actions are accompanied by preconditions that determine if an action is to be carried out (i.e., is the step sensible in the current situation). Postconditions determine the expected result and contain therefore information for a test oracle. The respective steps in the scenarios (test patterns) use symbolic instructions. Concrete test cases are compiled by augmenting the scenarios with concrete information about the system-under-test (e.g., exploit code, or specific messages on the CAN bus that would yield an expected result). This scenario can be seen as a recipe for an attack with the concrete information as ingredients. The result is a concretely executable set of instructions (in JSON format) to be ran on a Linux-based attack system. To generate tests that would subsequently provide evidence for the successful satisfaction of the requirements derived from the TARA, taking CAL and TAF into account, we propose a flow that uses the attack tree analysis' results and transforms it into attack scenarios that can be augmented with concrete implementation details in later phases of the development.

10.4.2 Security Test Generation

Using an attack graph (such as a tree, but also other structures like petri nets [26] are thinkable) allows for closing the loop from TARA to testing through an automated process. The missing link to achieve this pervasive chain is a transform mechanism from paths in the generated attack graph structure to test scenarios in the DSL. We therefore propose a mechanism that transforms a specific path in an attack tree (see Section 10.3.1) into a test scenario. This is achieved by mapping the edges of that path with actions in a DSL-based test scenario. The basic idea is that an action is required to realize a threat. Therefore, traversing trough a path in an attack tree requires a set of actions, each action responsible getting from one node in the tree to another. As the test patterns in the DSL principally consist of such (abstract) actions (accompanied by optional sets of pre- and postconditions), the resolution is a rule-based translation function to simply map the tree edges to test patterns. Figure 10.6 gives an overview of this process. Formally, the attack tree can be seen as a directed graph with rules (sequencing and parallelization). This resembles a *Transition*

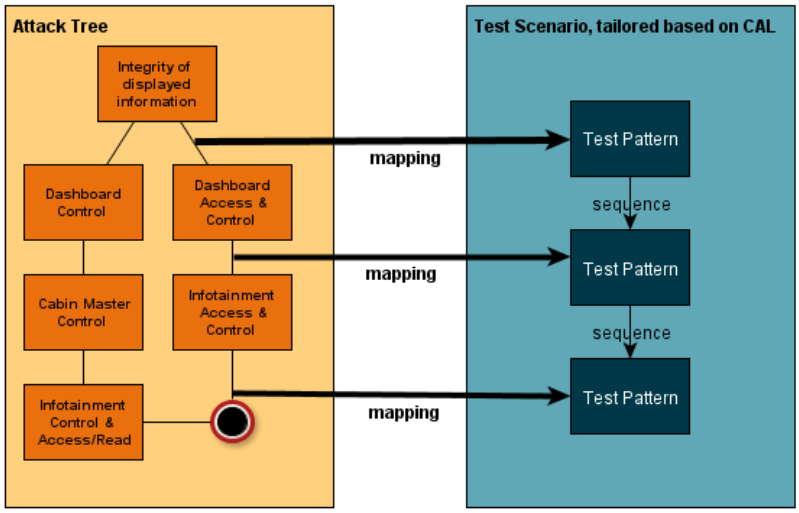

Figure 10.6: Attack tree to Test Scenario transformation example.

System (TS), defined as a set of states (Q) and a transition relation ($\rightarrow \in Q \times Q$, with $q, q\prime \in Q; q \rightarrow q\prime$). In this case, Q is the set of nodes in the attack graph, while $\rightarrow$ is determined by the edges and rules in the tree. A *Labelled Transition System (LTS)* additionally possesses a set of labels (Σ), such that each transition is named with a label σ in Σ ($q, q\prime \in Q, \sigma \in \Sigma; q \xrightarrow{\sigma} q\prime$) [27]. The set of labels is taken from the set of test patterns (i.e., possible actions) in the DSL. A labeling function attributes a label σ to a transition using an associative array. Once this LTS has been established, generating the abstract test case is trivially conducted by traversing along the respective path in the LTS an collecting the labels. The sequential set of collected labels (i.e., test patterns) automatically constitutes a test scenario. In simple words, we use an attack tree to select actions needed for an test scenario out of the set of all available test patterns and

brings them into sequence. The way the DSL is currently structured, an action can be identified by the tuple keyword (currently one of *scan*, *exploit*, and *execute* - the first two are to detect and attack devices, while the latter is a generic keyword for auxiliary tasks) and type (which defines the action closer). There are other attributes like *interface*, *target*, and *shell*, that depend on the action type. More than one action can be necessary to change the state in an attack tree (i.e. to traverse from one node to another). In this case the label attributed to the transition contains both actions. As an abstract example, the transition from *access to a system* to *control of a system* could require *execute, escalate privilege* as an action from the DSL. Therefore the resulting transition in the LTS would be $A_s \xrightarrow{xc_{ep}} C_s$ with A_s is the system access, C_s system control and xc_pe the execute (xc) privilege escalation (pe). A more concrete example follows in the case study in Section 10.5.

The course of action to use the TARA results for test cases also allows for prioritizing test cases, as attack paths can have calculated path costs (based on CALs and TAFs). As perfect security is infeasible, a *sufficiently* secure system can be defined as a system that does not exhibit an attack path with a cost below a certain threshold. Through the test case generation, it can be verified that relevant attack paths discovered through the threat modeling are mitigated through the measures in the security concept and effectively blocked in the implementation later.

The reason for using an LTS as a transition model is that it can be regarded as a more powerful structure than a tree (a tree can be viewed as a subset of an LTS in this regard) and can be easily converted into other structures like Directed Acyclic Graphs or even a general directed graph (in case of allowed loops needed), which makes it suitable as an internal structure. It can also be practically used in a three-layered process in this application. First, the attribution between tree edges and DSL actions (i.e., the labelling function) must be established only once initially and if the base set of node types in the TARA process or the possible actions in the DSL change (this happens rarely). Second, the LTS generation (low effort if the labelling function is present) must be done once, when an attack tree is generated or updated. Thirdly, the test cases have to be generated based on selection of paths, defining an origin (i.e. an entry point into the system) and a target is trivial, just collecting labels.

10.5 Case Study

To practically demonstrate the approach, we give an example of a realistic use case scenario. This use case has been practically tested using our test system. It consists of a standard car model that possesses a single can bus with an aftermarket infotainment system, running on Android, built in. The conducted test was to manipulate the speed gauge using a wireless access an entry point. We first created an architecture model using ThreatGet. The critical components for the attack are the infotainment system, the CAN bus the attacked dash board (including a screen) and a smart phone that is under the attacker's control (i.e., it is the attacker's smartphone) – see Figure 10.3. The threat analysis using the tool yielded a list of 103 threats (using the STRIDE methodology [28]). Using the methodology in Section 10.4.2, we generate attack trees and respective paths using different origin and destination points in the architecture diagrams and the threat attributions along the way. One specific result of this process is the attack tree in Figure 10.4. In this sequence, access to infotainment is followed by control of the infotainment, which is succeed by control (implying access) to the dashboard. This enables to corrupt the integrity of displayed information. This in practice means e.g., fake readings on the speed and RPM gauges or similar things - including potential safety implications – Table 10.1 provides an overview of threats applicable to the display. Please note that those apply directly to the display, while the attack tree allows for applying threats indirectly not requiring direct access to the system. The key element is the CAN bus, any device (also the cabin master control unit) connected to the (right) CAN bus (cf. Figure 10.3) that is taken control of could be used to gain access to the dashboard and manipulate the display under certain circumstances modeled in the threat model and attack tree. The transitions between these items have been matched with fitting action items from the DSL. To reach access to the Infotainment from an initial state in the LTS, a wireless scan and already an exploit (labels $s_{BlueBorne}$ and $xp_{BlueBorne}$ for scanning and executing a BlueBorne attack, with s for a scan and xp for an exploit) has to be take as actions. Please note that this is one of more possibilities to gain access, there could be others. To gain control, we use the actions of opening a connection to a remote hotspot using the access ($xp_{OpenAndroidHotspot}$) and opening an Android Debug (ADB) shell ($xp_{OpenADB}$). The rest of the tree is a special case, as the access to the Dashboard, its control and the data manipulation can occur in one step by sending fake CAN messages. These messages are represented by the different step *can_attack* in the DSL ($xp_{CanAttack}$). Figure 10.5 shows the resulting attack description in the DSL. The steps immediately preceding

the CAN attack (*install_python_env*, *install_python_lib*, and *attackScript*) are intrinsic, as these are just necessary steps to fulfill the last one. In that sense, they can also be seen as part of gaining control over the infotainment, as it is only after these three steps capable of carrying out the rest of the attack. The concretization for a specific system eventually works by generating a JSON code that contains executed environments and exploit code, as well as information as CAN packet structures from a database or directly given information from the tester as form of a grey-box test. This is out-of-scope of this paper and already published elsewhere [12] in detail, but for the sake of the functioning of the approach it should be briefly mentioned that the DSL items (i.e. *Test Patterns*) are augmented with information from a systems database containing information about the systems-under-test (partially pre-filled and completed by a client in a grey box setting or penetration testers in a black box setting) with the necessary information (e.g., pieces of code to exploit a certain software or version, specific data of CAN messages to send, etc.). This is translated into a JSON format containing an environment (e.g., BASH, Python, a framework like Metasploit, etc.) and sent to an execution engine that is instrumented with the SUT and calls the respective software tools tools to execute the concrete attack.

10.6 Related Work

Threat modeling is an approach that responds to the increasing need to address security concerns from the early phases of product development. The popularity of threat modelling is reflected by a variety of available methods and tools, ranging from open-source academic prototypes to full-fledged commercial solutions. There are roughly speaking three categories of threat modeling approaches. The first class of tools only allow manual modeling based on Excel sheets and questionnaires [29]. Threat identification and mitigation is identified without and automated reasoning support. The second class of tools improves the modelling experience by providing a graphical modelling environment but without a rigorous formal model [30, 31, 32, 33]. Finally, the third class of tools are model-based system engineering solutions with an underlying formal threat model and provide full support for automated threat analysis [30, 31, 32, 33].

Attack trees [34] describe sophisticated attack patterns that capture sequences of basic attack steps and describe how these can be combined to reach a target. Graphical modelling and analysis of attack trees is supported by sev-

Table 10.1: Threats related to the display in the case study example (MED=medium; MOD=moderate; SEV=severe)

Target	Affected Asset	Damage Scenario	Threat Title	Category	Impact Cat.	Likel.	Impact	Risk
Touchscreen	n/a	n/a	Tamper through external ports	TAMPERING		MED	MOD	3
Touchscreen	n/a	n/a	Physical Tampering	TAMPERING		MED	MOD	3
Touchscreen	Information Availability	Operational impact	Physical Tampering	TAMPERING	Operational	MED	SEV	1
Touchscreen	Information Integrity	Operational impact,	Physical Tampering	TAMPERING	Operational	MED	SEV	1
Touchscreen	Information Integrity	Safety impact	Physical Tampering	TAMPERING	Safety	MED	SEV	1

eral tools [35, 36]. Attack trees can be extended with additional attributes such as possibility, cost, resources [34] or time [37]. Attack trees can be combined with fault trees for a more integrated safety and security analysis or with defender's mitigation measures resulting in the attack-defence tree model [38]. Attack trees are complementary to the more static threat model and the relation between the two has been only seldomly investigated. Isograph AttackTree [35] supports threat analysis and risk assessment from the attack tree, following the relevant ISO standards. On the other hand, THREATGET allows automatic generation of attack trees from threat analysis results [39].

The integration of the threat and attack tree modeling and analysis and testing has not been sufficiently investigated in the literature. The only work that

we are aware of on this topic is about test generation from attack trees has been studied in the context of the vehicle security in the automotive domain [40]. In this paper, we propose a methodology that goes from threat modelling to the generation of test cases, where attack trees are used as an intermediate step in this process.

10.7 Conclusion

We described a method to automatically generate abstract test scenarios out of a TARA using attack trees and LTSs. The main improvement of this method is that these test scenarios can be derived from a process that is mandated by a CSMS in a simple, automated, and resource-efficient way, which surpasses manual test case generation while still maintaining targeted tests as a result. The resulting scenarios can be further compiled into executable test cases with very low effort once the details of the implemenation are known. We also showed incorporation of CALs and TAF into a security analysis and testing pipeline. These concepts define the level of thoroughness of testing as well as providing a metric for the effectiveness of included safeguards. The required formalization in this manifestation of a testing process is used to increase completeness and efficiency in security testing by using the products of formalized steps in an automated process. Overall, this paper demonstrates a workflow originating from CALs and a TARA, which results are used to generate test cases in an automated manner (via attack tree generation). These tests can be used at various stages of the life cycle and also determine TAFs in the practical implementation stages. Future work includes to utilize machine learning to attribute the test patterns to attack tree edges (instead of a fixed function). This allows for more flexible and experience-based test case generation.

Acknowledgements

This research has received funding within the ECSEL Joint Undertaking (JU) under grant agreements No.101007326 (project AI4CSM) and No. 101007350 (project AIDOaRt). 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. The document reflects only the author's view, and the Commission is not responsible for any use that may be made of the information it contains.

Bibliography

[1] Upstream Security, "Upstream Security Global Automotive Cybersecurity Report," tech. rep., Upstream Security, 2020.

[2] United Nations Economic and Social Council - Economic Commission for Europe, "UN Regulation on Uniform Provisions Concerning the Approval of Vehicles with Regard to Cyber Security and of Their Cybersecurity Management Systems," Tech. Rep. ECE/TRANS/WP.29/2020/79, United Nations Economic and Social Council - Economic Commission for Europe / United Nations Economic and Social Council - Economic Commission for Europe, Brussels, 2020.

[3] International Organization for Standardization and Society of Automotive Engineers, "ISO/SAE PAS8475 (WIP) Road Vehicles – Cybersecurity Assurance Levels and Targeted Attack Feasibility - SAE International." https://www.sae.org/standards/content/iso/sae%20pas8475/, 2022.

[4] International Organization for Standardization and Society of Automotive Engineers, "ISO/SAE PAS8477 (WIP) Road vehicles - cybersecurity verification and validation - SAE International." https://www.sae.org/standards/content/iso/sae%20pas8477/, 2023.

[5] G. Macher, H. Sporer, R. Berlach, E. Armengaud, and C. Kreiner, "SAHARA: A security-aware hazard and risk analysis method," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, (Grenoble, France), pp. 621–624, IEEE, Mar. 2015.

[6] International Organization for Standardization, "Information security, cybersecurity and privacy protection – Evaluation criteria for IT security – Part 2: Security functional components," ISO/IEC Standard 15408-2:2022, International Organization for Standardization, 2022.

[7] S. Marksteiner, N. Marko, A. Smulders, S. Karagiannis, F. Stahl, H. Hamazaryan, R. Schlick, S. Kraxberger, and A. Vasenev, "A Process to Facilitate Automated Automotive Cybersecurity Testing," in *2021 IEEE 93rd Vehicular Technology Conference (VTC Spring)*, (New York, NY, USA), pp. 1–7, IEEE, 2021.

[8] International Organization for Standardization and Society of Automotive Engineers, "Road Vehicles – Cybersecurity Engineering," ISO/SAE Standard "21434", International Organization for Standardization, 2021.

[9] D. Ward, I. Ibarra, and A. Ruddle, "Threat Analysis and Risk Assessment in Automotive Cyber Security," *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, vol. 6, no. 2013-01-1415, pp. 507–513, 2013.

[10] C. Schmittner, B. Schrammel, and S. König, "Asset Driven ISO/SAE 21434 Compliant Automotive Cybersecurity Analysis with ThreatGet," in *Systems, Software and Services Process Improvement* (M. Yilmaz, P. Clarke, R. Messnarz, and M. Reiner, eds.), Communications in Computer and Information Science, (Cham), pp. 548–563, Springer International Publishing, 2021.

[11] C. C. Michael, K. van Wyk, and W. Radosevich, "Risk-Based and Functional Security Testing," tech. rep., U.S. Deparmtent of Homeland Security, 2005.

[12] C. Wolschke, S. Marksteiner, T. Braun, and M. Wolf, "An Agnostic Domain Specific Language for Implementing Attacks in an Automotive Use Case," in *The 16th International Conference on Availability, Reliability and Security*, ARES 2021, (New York, NY, USA), pp. 1–9, Association for Computing Machinery, Aug. 2021.

[13] F. Cuppens and R. Ortalo, "Lambda: A language to model a database for detection of attacks," in *International Workshop on Recent Advances in Intrusion Detection*, (Berlin, Heidelberg), pp. 197–216, Springer, 2000.

[14] C. Michel and L. Mé, "ADeLe: An Attack Description Language for Knowledge-Based Intrusion Detection," in *Trusted Information* (M. Dupuy and P. Paradinas, eds.), IFIP International Federation for Information Processing, (Boston, MA), pp. 353–368, Springer US, 2001.

[15] M. Yampolskiy, P. Horváth, X. D. Koutsoukos, Y. Xue, and J. Sztipanovits, "A language for describing attacks on cyber-physical systems," *International Journal of Critical Infrastructure Protection*, vol. 8, pp. 40–52, Jan. 2015.

[16] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical combinatorial testing," SP 800-142, National Institute of Standards and Technology, 2010.

[17] A. Shostack, *Threat Modeling: Designing for Security*. Indianaplois, IN: John Wiley & Sons, 2014.

[18] H. S. Lallie, K. Debattista, and J. Bal, "A review of attack graph and attack tree visual syntax in cyber security," *Computer Science Review*, vol. 35, p. 100219, Feb. 2020.

[19] R. Sell, M. Leier, A. Rassõlkin, and J.-P. Ernits, "Autonomous Last Mile Shuttle ISEAUTO for Education and Research," *International Journal of Artificial Intelligence and Machine Learning*, vol. 10, pp. 18–30, Jan. 2020.

[20] D. Eng, "Integrated Threat Modelling," Master's thesis, University of Olso, 2017.

[21] T. R. Ingoldsby, "Attack tree-based threat risk analysis," tech. rep., Amenaza Technologies Limited, 2021.

[22] M. S. Haque and T. Atkison, "An Evolutionary Approach of Attack Graph to Attack Tree Conversion," *International Journal of Computer Network and Information Security*, vol. 9, pp. 1–16, Nov. 2017.

[23] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," in *Proceedings of the 1998 Workshop on New Security Paradigms*, (New York, NY, USA), pp. 71–79, ACM, 1998.

[24] B. Schneier, "Attack trees," *Dr. Dobb's journal*, vol. 24, no. 12, pp. 21–29, 1999.

[25] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, (New York, NY, USA), pp. 217–224, ACM, 2002.

[26] C. A. Petri, *Kommunikation mit Automaten*. PhD thesis, Technische Universität Darmstadt, 1962.

[27] R. M. Keller, "Formal verification of parallel programs," *Communications of the ACM*, vol. 19, pp. 371–384, July 1976.

[28] R. Khan, K. McLaughlin, D. Laverty, and S. Sezer, "Stride-based threat modeling for cyber-physical systems," in *2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, (New York, NY), pp. 1–6, IEEE, 2017.

[29] Tutamantic Ltd., "Tutamen threat model automator." Online, 2020. Accessed: 2020-11-29.

[30] Foreseeti AB, "Foreseeti." Online, 2020. Accessed: 2020-11-29.

[31] J. Was, P. Avhad, M. Coles, N. Ozmore, R. Shambhuni, and I. Tarandach, "Owasp pytm." Online, 2020. Accessed: 2020-11-29.

[32] M. E. Sadany, C. Schmittner, and W. Kastner, "Assuring compliance with protection profiles with threatget," in *SAFECOMP 2019 Workshops*, Lecture Notes in Computer Science, (Berlin), pp. 62–73, Springer, 2019.

[33] K. Christl and T. Tarrach, "The analysis approach of threatget," *CoRR*, vol. abs/2107.09986, 2021.

[34] S. Mauw and M. Oostdijk, "Foundations of attack trees," in *Information Security and Cryptology - ICISC 2005* (D. H. Won and S. Kim, eds.), vol. 3935, pp. 186–198, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[35] Isograph, "Isograph attacktree." Online, 2023. Accessed: 2023-10-03.

[36] Amenaza Technologies Limited, "Securitree." Online, 2023. Accessed: 2023-10-03.

[37] J. Bryans, H. N. Nguyen, and S. A. Shaikh, "Attack defense trees with sequential conjunction," in *2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)*, (Hangzhou, China), pp. 247–252, IEEE, 2019-01.

[38] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Foundations of attack–defense trees," in *Formal Aspects of Security and Trust* (P. Degano,

S. Etalle, and J. Guttman, eds.), vol. 6561, pp. 80–95, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

[39] S. Chlup, K. Christl, C. Schmittner, A. M. Shaaban, S. Schauer, and M. Latzenhofer, "THREATGET: towards automated attack tree analysis for automotive cybersecurity," *Inf.*, vol. 14, no. 1, p. 14, 2023.

[40] M. Cheah, H. N. Nguyen, J. Bryans, and S. A. Shaikh, "Formalising systematic security evaluations using attack trees for automotive applications," in *Information Security Theory and Practice* (G. P. Hancke and E. Damiani, eds.), vol. 10741, pp. 113–129, Cham: Springer International Publishing, 2018. Series Title: Lecture Notes in Computer Science.

Chapter 11

Paper V: Automated Passport Control: Mining and Checking Models of Machine Readable Travel Documents

Stefan Marksteiner, Marjan Sirjani, Mikael Sjödin.

Proceedings of the 19th International Conference on Availability, Reliability and Security, New York: Association for Computing Machinery, 2023. DOI: 10.1145/3664476.3670454

Abstract

Passports are part of critical infrastructure for a very long time. They also have been pieces of automatically processable information devices, more recently through the ISO/IEC 14443 (Near-Field Communication – NFC) protocol. For obvious reasons, it is crucial that the information stored on devices are sufficiently protected. The International Civil Aviation Organization (ICAO) specifies exactly what information should be stored on electronic passports (also Machine Readable Travel Documents – MRTDs) and how and under which conditions they can be accessed. We propose a model-based approach for checking the conformance with this specification in an automated and very comprehensive manner: we use automata learning to learn a full model of passport documents and use trace equivalence and primitive model checking techniques to check the conformance with an automaton modeled after the ICAO standard. Since the full behavior is underspecified in the standard, we compare a part of the learned model and apply a primitive checking ruleset to assure proper authentication. The result is an automated (non-interactive), yet very thorough test for compliance, despite the underspecification. This approach can also be used with other applications for which a specification automaton can be modeled and is therefore broadly applicable.

11.1 Introduction

Passports are among critical infrastructure and subject to forgery for a very long time. This has been aggravated by the fact – it is even mandatory for member states of the International Civil Aviation Organization (ICAO) that all documents that are not machine-readable travel documents (MRTDs) are expired since 2015[1]. More recently, passports have opened up for wireless reading via Near-Field Communication (NFC). The NFC communication protocol specified in the ISO/IEC 14443 standard series (most prominently in ISO/IEC 14443-4 [1] for data communications), while the respective commands for interacting with integrated-circuit identification cards are defined in ISO/IEC 7816-4 [2]. The ICAO Doc 9303 series specifies a logical data structure and further details regarding commands, as well as rule sets for accessing the data inside the defined structure for MRTDs [3]. To sum it up, we use automata learning with SELECT, READ, UPDATE and AUTHENTICATE (implementing Basic Access Control - BAC [4]) symbols from the ISO/IEC 14443-4 protocol to infer an automaton and compare it using trace and bisimilarity equivalence to an automaton modeled after the ICAO MRTD specification. The remainder of this paper is structured as follows: Section 11.1.1 contains this paper's additions to the body of knowledge, Section 11.2 outlines the necessary prerequisite knowledge, Section 11.3 describes the learning and conformance checking setup, Section 11.4 gives evaluation results, Section 11.5 gives an overview of important related work, and Section 11.6 concludes the paper with a discussion and an outlook on further research directions.

11.1.1 Contribution

This paper outlines a process how to very thoroughly analyze passports and automatically check their conformance with the ICAO MRTD specifications using formal methods. The contribution to the body of knowledge is threefold. The paper provides:

- A concise summary of the ICAO MRTD specification – this information can be used by researchers to build compliance checking systems.
- A(n incomplete) state machine model of the specification.

[1] https://www.icao.int/Newsroom/Pages/Last-Week-for-States-to-Ensure-Expiration-of-Non-Machine-Readable-Passports.aspx

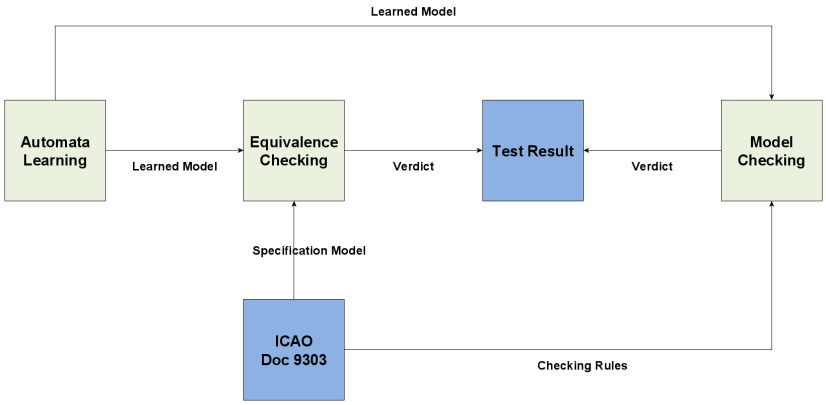

Figure 11.1: Overview of the approach. Green are processes and blue are artifacts.

- An automata learning setup for ISO/IEC 14443-4, ISO/IEC 7816-4, and ICAO Doc 9303 including an input alphabet definition and a practical implementation.
- A practically implemented method for compliance checking based on equivalence and primitive model checking for an underspecified standard.

The approach described in this paper can also be adapted, e.g., by using the specification parts model (Sections 11.2.2, 11.3.3, and 11.3.4) to create model-based tests from it (removing the learning part) or by using the learning part only (Sections 11.3.1, 11.3.2, and 11.4) and creating rules for a model checker to check the model for desired properties (removing the equivalence checking part). Figure 11.1 provides an overview of this approach and its parts.

11.1.2 Limitations

Since the ICAO specification is not strict enough to build a feasible specification automaton (through many optionalities, there could be many automata that represent a valid model of the specification), the behavioral equivalence only covers a part of the conformance checking. For other parts (namely, files should not be read and/or writable if certain conditions are not met), we impose model

checks. These are, however, pretty primitive and might be improved. Also, with the lack of a Password Authenticated Connection Establishment (PACE) and Terminal Authentication implementation, we do not check for these types of authentication [2]. Lastly, the case study is limited, as only Austrian passports were available as examination objects.

11.2 Preliminaries

This section outlines the fundamentals of the NFC protocol family, the ICAO MRTD specifications and the concepts of automata learning.

11.2.1 Near Field Communication

Near Field Communication (NFC), standardized in the ISO/IEC 14443 series, is a wireless communications protocol that allows for reader devices (proximity coupling devices - PCDs) to communicate completely passive, powerless devices (proximity integrated circuit cards - PICCs) at low data rates (up to 424 kbits/s). The PICCs are therby powered via an inductive field that also transports the data. After a handshake [5], which is out of scope of this paper, communications is standardized in ISO/IEC 14443-4 [1], which defines basic types of messages 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. The actual data interchange format is defined in ISO/IEC 7814-4 [2], including commands for data selection, data manipulation and security functions. In general, data on NFC cards is segmented into different applications (*dedicated files - DFs*), which are comparable to directories on file systems, that contain data files (*elementary files - EFs*) as actual data storages. Both types are selected using the *SELECT* command with different parameters. The standard also defines manipulation operators that can be applied on EFs. One flavor are the *READ/WRITE/UPDATE/APPEND/SEARCH/ERASE/COMPARE BINARY* commands. These are for manipulating *Data Units*, which can inside the EF be controlled by using offsets. Another set is the *READ/WRITE/UPDATE/APPEND/SEARCH/ERASE/ACTIVATE/DEACTIVATE RECORD* commands for manipulating *Records*, that can be addressed by record identifiers instead of raw binary offsets. Under records reside *Data objects (DOs)*, which can be

[2] A German passport was available, but supported PACE only. We could not include it in the evaluation for the reason stated above.

addressed with the *GET/PUT/UPDATE/COMPARE DATA* commands. Apart from that, the standard defines security functions like the *GETCHALLENGE* and different forms of *AUTHENTICATE* and *verification* commands. These commands build the base of the input alphabet for learning NFC models, the rest of the commands inside an abstraction layer will be defined by respective identifiers defining the content.

The respective answers to these commands could consist of data (which can be encrypted as well), but in any case contains two status bytes (as defined in ISO/IEC 7816-4). These bytes are set to have a scheme distinguishing between a completed process with normal processing (*9000* and *61XX* - the latter means that are data bytes left to transmit) or warning processing (*62XX* and *63XX*), as well as aborted processing with execution error (*64XX* and *66XX*) or checking eroor (*67XX* and *6FXX*). The most common codes encountered when working with passports are (empirical :

- *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 satisified (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)

11.2.2 Machine Readable Travel Document Specification

The International Civil Aviation Organization's (ICAO) Doc 9303 series specifies the appearance and behavior of passports and machine readable travel documents (MRTDs). Particularly Part 10 [3] specifies the logical data structure (LDS) of MRTDs and defines access rights (i.e. what authentication is necessary to read or manipulate) for data.

The standard defines four applications, referenced by dedicated files:

- eMRTD (ID A0 00 00 02 47 10 01)
- Travel records (ID A0 00 00 02 47 20 01)
- Visa records (A0 00 00 02 47 20 02)
- Additional biometrics (A0 00 00 02 47 20 03)

The first (following the LDS1) is mandatory, while the latter three (follwing the LDS2) are optional. The eMRTD application contains all of the data that is normally on the main page of a passport (like number, name, birth date, expiration date, etc.) plus additional data including electronic photos, finger and iris scans. This application contains data that should be immutable in the document and readable with authentication, namely with the older Basic Access Control (BAC) or the newer Password Authenticated Connection Establishment (PACE), with sensitive biometrics (fingerprints and iris scan) additionally needs a terminal authentication to determine the reader is authorized. Due to this is mandatory and the other applications are not implemented in many (including EU) passports[3], we concentrate on this part. The other applications contain potentially mutable records, and certificates stored within the application to display authenticity against a reader – assuring that visa and electronic travel stamps are genuine. The applications require different levels of authentication (see Section 11.2.2). Figure 11.2 gives an overview of the layout and the different applications and Table 11.1 gives an overview of the defined EFs with their IDs, DFs, requisiteness, and access requirements.

Electronic Machine Readable Travel Document Application

Concentrating on the Electronic Machine Readable Travel Document (eMRTD) application, ICAO Doc 9303 Part 10 defines various EFs that contain

[3] We also concretely tested an expired Austrian and a valid Austrian and German passport for these applications. The answer was unanimously the status code *6A82* for *File or application not found*.

personal and document data, along with access requirements. In particular it defines

- *Common (EF.COM)*: containing metadata (version, encoding, etc.) of the application
- *Data Group 1 (EF.DG 1)*: containing the machine readable zone.
- *Data Group 2 (EF.DG 2)*: containing the holder's face image.
- *Data Group 3 (EF.DG 3)*: containing the holder's fingerprints image.
- *Data Group 4 (EF.DG 4)*: containing the holder's iris image.
- *Data Group 5 (EF.DG 5)*: containing holders displayed portrait(s).
- *Data Group 6 (EF.DG 6)*: is reserved for future use.
- *Data Group 7 (EF.DG 7)*: containing the holder's displayed signature.
- *Data Group 8 (EF.DG 8)*: containing data features.
- *Data Group 9 (EF.DG 9)*: containing structure features.
- *Data Group 10 (EF.DG10)*: containing substance features.
- *Data Group 11 (EF.DG11)*: containing additional personal details (e.g., localized name, place-of-birth).
- *Data Group 12 (EF.DG12)*: containing additional document details (e.g., issuing authority, date-of-issue).
- *Data Group 13 (EF.DG13)*: containing optional details.
- *Data Group 14 (EF.DG14)*: containing data elements.
- *Data Group 15 (EF.DG15)*: containing the public key info for active authentication.
- *Data Group 16 (EF.DG16)*: containing persons to notify.
- *Document Security Object (EF.SOD)*: containing hash values of the data group for integrity checking.
- *Country Verifying Certification Authorities (EF.CVCA)*: containing public keys of CVCA for teminal authentication (see Section 11.2.2).

Name	ID	DF	Mandatory	SELECT	READ
EF.ATR/INFO	2F01	Master	No[1]	ALWAYS	ALWAYS
EF.DIR	2F00	Master	No[1]	ALWAYS	ALWAYS
EF.CardAccess	011C	Master	No[2]	ALWAYS	ALWAYS
EF.CardSecurity	011D	Master	No[2]	PACE	PACE
EF.DG1,2	0101,02	LDS1.eMRTD	Yes	BAC/PACE	BAC/PACE
EF.DG3,4	0103,04	LDS1.eMRTD	No	BAC/PACE+TA	BAC/PACE+T
EF.DG5,7-13,16	0105,07-D,10	LDS1.eMRTD	No	BAC/PACE	BAC/PACE
EF.DG6 (RfFU)	0106	LDS1.eMRTD	No	-	-
EF.DG14	010E	LDS1.eMRTD	No[2]	BAC/PACE	BAC/PACE
EF.DG15	010F	LDS1.eMRTD	No[3]	BAC/PACE	BAC/PACE
EF.COMMON	011E	LDS1.eMRTD	Yes	BAC/PACE	BAC/PACE
EF.SOD	011D	LDS1.eMRTD	Yes	BAC/PACE	BAC/PACE
EF.CVCA	011C	LDS1.eMRTD	Yes	BAC/PACE	BAC/PACE
EF.Certificates	011A	All LDS2	No	PACE+TA	PACE+TA
EF.ExitRecords	0102	LDS2.Travel Records	No	PACE+TA	PACE+TA
EF.EntryRecords	0101	LDS2.Travel Records	No	PACE+TA	PACE+TA
EF.VisaRecords	0103	LDS2.Visa Records	No	PACE+TA	PACE+TA
EF.Biometrics1-64	0201-0240	LDS2.Add. Biometrics	No	PACE+TA	PACE+TA

Table 11.1: Files from ICAO Doc 9303-10 with their names, IDs, application, requisiteness and access requirements.
[1]Conditional - required if LDS2 files are present.
[2]Conditional - required if PACE is implemented.
[3]Conditional - required if active authentication is implemented.

- Key files for authentication (see Section 11.2.2).

All of these files can be mandatory (DG1, DG2), optional (DGs 3-5, 7-13, and 16), or conditional (DG14 - if PACE is implemented, DG15 - if AA is implemented) and can be read if authenticated (via BAC or PACE), except for DGs 3 and 4, which require additional terminal authentication (see Section 11.2.2) - none of these files should be manipulated (no write/append access). Table 11.1 gives an overview of these files, along with those from the LDS2 applications.

Authentication

Since passports can considered critical infrastructure devices, authentication is crucial. The ICAO defines two mechanisms for access to the MRTD chips in

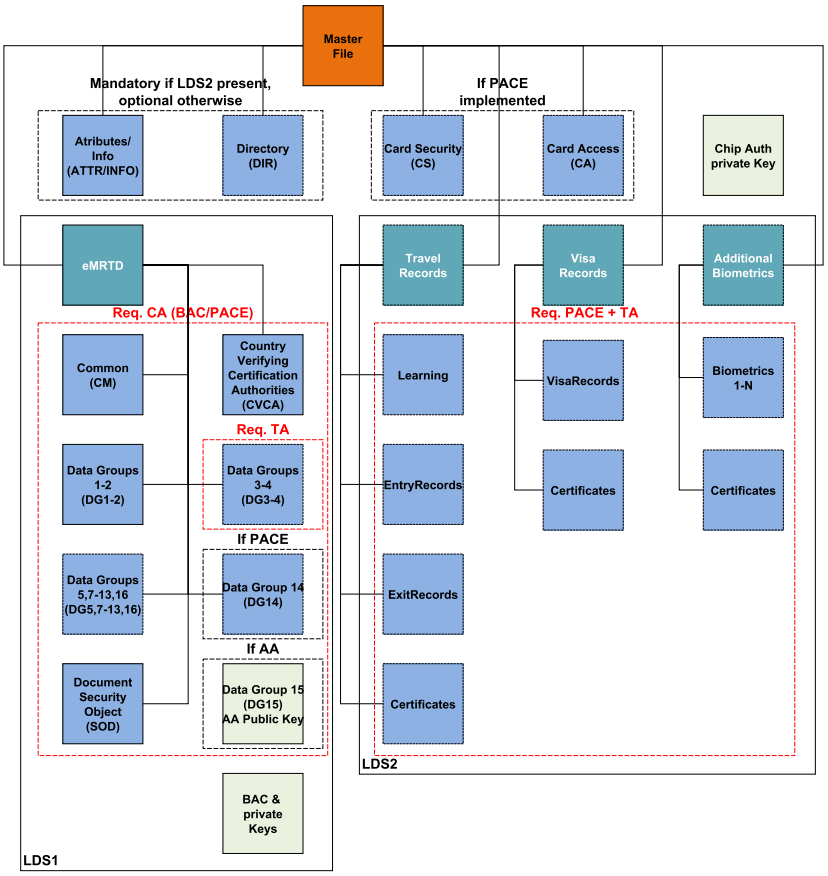

Figure 11.2: Locical Data Structure of Machine Readable Travel Documents. Amber is the master file (MF), Cyan are dedicated files (DF), Blue are Elementary Files (EF), and Green are key files. Solid frames means mandatory files, dashed ones optional files. Solid boxes donate the LDS contexts, dashed black boxes requirements, and dashed red boxes necessary authentication.

its Doc 9303-11 standard [4]:

- Basic Access Control (BAC) and
- Password Authenticated Connection Establishment (PACE).

BAC is the older one, it had known privacy issues [6, 7] and may become deprecated in future. Currently an MRTD must implement one or both mechanisms. Additionally LDS2 applications *must* and additional biometrics, LDS1 data groups 3 (fingerprints) and 4 (iris), *may* be secured by a terminal authentication procedure.

Since we have a BAC, but not a PACE implementation available and our available devices-under-test (see Section 11.4) all support BAC but only partially PACE, we use BAC only for modeling and evaluation. BAC uses is challenge-response by encrypting a received nonce (via a *GETCHALLENGE* command) with a key derived from three components[4]: the passport number, the expiration date and the holder's birth date. These can be obtained from the machine readable zone or the main page of the passport. The encrypted nonce is subsequently sent through an *EXTERNAL AUTHENTICATE* command to finish the authentication. The authentication answer contains additional key material for establishing session keys. All instructions (and respective responses) operating on data protected by this particular BAC are encrypted using these session keys. The rationale is to prevent unnoticed wireless data extraction from an MRTD.

11.2.3 Automata Learning

Automata learning is a method to infer state machine models (originally deterministic finite acceptors - DFAs) from a system using a learner-teacher framework [8]. The learner may ask two kinds of questions: *Membership queries* and *Equivalence queries*. The former is used to determine if an input (specifically an input word, which is a combination of input symbols) is *well-formed* i.e., if it is a valid word inside this language. The answer to this query is a yes or a no from the teacher. The latter type of queries is to determine the correctness of a learned automaton. The teacher answers with yes if a hypothesis (inferred after a sufficient amount of membership queries) is correct or gives a counter example the learner could use to improve the hypothesis until it is correct. To apply this to real-world reactive systems (mostly software or cyber-physical systems), we use Mealy machines (formally $M = (Q, \Sigma, \Omega, \delta, \lambda, q_0)$, with Q being a set of states, Σ an input alphabet, Ω an output alphabet, δ a transition

function ($\delta : Q \times \Sigma \rightarrow Q$), λ an output function ($\lambda : Q \times \Sigma \rightarrow \Omega$) and q_0 an initial state. This alters the framework in the sense that the membership queries yield a Mealy output instead of a binary answer. In practice, the teacher is implemented in a way that the membership queries deliver the (abstracted - see Section 11.3.1) result directly obtained from the reactive system and the equivalence queries are realized as set of conformance tests that deliver a positive answer or a counterexample in the form of a failed test's input. The respective systems can also be viewed as labeled transition systems (LTS) [9]. We can therefore use trace equivalence ($Traces(LTS_1) = Traces(LTS_2)$) to check the behavioral equivalence with a specification automaton [10].

11.3 Learning Setup

We use the widely used Java library LearnLib [11] to mine (Mealy type) state machine models of passports. This library provides classes for developing adapters to a system-under-learning (SUL), as well as various learning algorithms (L* [8] and variants thereof [12], KV [13], DHC [14] and TTT [15]).

The SUL classes interact with a C++ program that serves as an interface for a Proxmark3 NFC adapter device [16], which allows for sending arbitrary NFC commands to a device and process the respective responses. This program also contains an abstraction layer (see Section 11.3.1).

11.3.1 Abstraction

Since, in principle, any combination of bytes can be sent to a SUL, the input space is very large (only bounded by the maximum transmission units for NFC). To keep the learning within a feasible time frame, the input must be limited to sensible set of discrete instructions i.e., the input alphabet for a Mealy machine. The C++ adapter translates input symbols to data to be send. Similarly, we abstract the output of the operations. This is a necessity, since some of the commands yield a different output every time (e.g., through random cryptographic nonces, session keys, etc.). However, conviently all of the answer messages contain a status code (see Section 11.2.1), which is even in clear text for encrypted messages. Also, the status code already contains the relevant information for checking ICAO conformance, since it determines whether a file could be successfully read or manipulated. We therefore use the answer status codes as abstracted outputs.

11.3.2 Input Alphabet

The input alphabet in our case consists of a combination of instructions from ISO/IEC 7816-4 (select DF, select EF, GETCHALLENGE, EXTERNAL AUTHENTICATION, READ BINARY, and UPDATE BINARY) and the file structure with the DF and EFs outlined in Section 11.2.2. Except for the BAC, all instructions are used in two forms: unencrypted and encrypted. The rationale is to check if after a successful authentication insecure access might become possible. We use READ BINARY as representative for all reading operations and UPDATE BINARY as representative for writing operations. As stated above we concentrate on the LDS1 application, making this the only select DF instruction. The codes for card access (CA) and (CVCA), as well as for card security (CS) and the document security object (SOD) are identical (only executed in different context) yielding to only one encrypted and unecrypted input symbol for each. The BAC is abstracted in to one input symbol (combining the GETCHALLENGE and EXTERNAL AUTHENTICATE instructions along with all necessary key calculations). The compelete input alphabet is therfore ¡SEL_EF.CA¿, ¡SEL_DF.LDS1¿, ¡SEL_EF.CM¿, ¡SEL_EF.DG1¿, ¡SEL_EF.DG2¿, ¡SEL_EF.DG3¿, ¡SEL_EF.DG4¿, ¡SEL_EF.DG5¿, ¡SEL_EF.DG6¿, ¡SEL_EF.DG7¿, ¡SEL_EF.DG8¿, ¡SEL_EF.DG9¿, ¡SEL_EF.DG10¿, ¡SEL_EF.DG11¿, ¡SEL_EF.DG12¿, ¡SEL_EF.DG13¿, ¡SEL_EF.DG14¿, ¡SEL_EF.DG15¿, ¡SEL_EF.DG16¿, ¡SEL_EF.SOD¿, ¡SEL_EF.ATR¿, ¡SEL_EF.DIR¿, ¡RD_BIN¿, ¡BAC¿, ¡SSEL_EF.CA¿, ¡SSEL_DF.LDS1¿, ¡SSEL_EF.CM¿, ¡SSEL_EF.DG1¿, ¡SSEL_EF.DG2¿, ¡SSEL_EF.DG3¿, ¡SSEL_EF.DG4¿, ¡SSEL_EF.DG5¿, ¡SSEL_EF.DG6¿, ¡SSEL_EF.DG7¿, ¡SSEL_EF.DG8¿, ¡SSEL_EF.DG9¿, ¡SSEL_EF.DG10¿, ¡SSEL_EF.DG11¿, ¡SSEL_EF.DG12¿, ¡SSEL_EF.DG13¿, ¡SSEL_EF.DG14¿, ¡SSEL_EF.DG15¿, ¡SSEL_EF.DG16¿, ¡SSEL_EF.SOD¿, ¡SSEL_EF.ATR¿, ¡SSEL_EF.DIR¿, ¡SRD_BIN¿. Using this alphabet in the abstracted learner yields models of passports (see Section 11.4). This model can be used to compare with a specification automaton or to use with model checking.

11.3.3 Specification Automaton

Following the specification, we modeled a minimal automaton that is able to behave a required by ICAO specification Doc 9303-10. This is not straightforward, since, compared with the possible NFC command alphabets, the ICAO document is under-specified. Additional behavior is specified in Doc 9303-11 (particularly that the LDS1.eMRTD application *must* be selected before performing BAC authentication). Since much of the behavior is not defined by

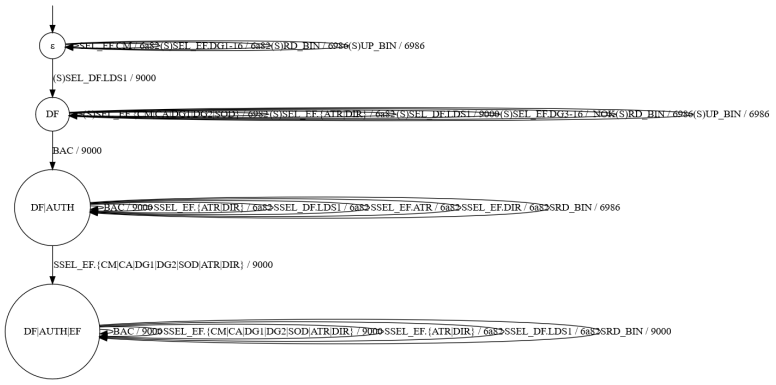

Figure 11.3: Graph of the specification automaton. The diagram was simplified for better readability.

ICAO (e.g., behavior on multiple authentications and selections), we modeled the bare functional minimum. The automaton contains 5 states, which, for better comprehensiveness, we label according to already happened events: the initial state (ε), a state with a valid EF from the Master DF (EF), the LDS1 application selected (DF), the LDS1 application selected and authenticated with BAC ($DFAUTH$), and the LDS1 application selected, authenticated with BAC and a valid EF from the LDS1 application selected $DFAUTHEF$. Since some files are optional we also leave out the selection transitions in the state where they could be successfully accessed, since the respective transitions could have a positive (*9000*) or negative (*6a82*) output. Figure 11.3 show a graph of the specification automaton. Note that the automaton is not a complete one, as not every input has a defined output and target state in the output and transition relations, respectively. This yields a specification automaton that can be used for either a learning and behavior comparison approach as described in this paper, or to create model-based tests.

11.3.4 Simplification and Labeling

To allow for a sensible comparison of the learned model with the specification, we want to identify the distinct states with a hard definition in the standard in order to separate it from not defined parts, as not the full behavior but only the access rules and requiredness of files are prescribed (see Section 11.2.2). Also, semantically labeled states are more convenient for both the hu-

man reader and electronic processing. We therefore use a simple algorithm that accesses mandatory files in a specific order, namely:

- 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*

This names the states after attributed properties: *EF* for a selected elementary file, *DF* for a selected decicated file (i.e. the LDS1.eMRTD application), AUTH for a successful authentication (i.e., BAC), FAILAUTH for a failed authentication (mainly happens because no file that needs authentication was selected beforehand), *DEAUTH* for a revoked authentication (which occurs when insecure – i.e., non-encrypted – commands are executed). This label states where taken from different Austrian passports.

11.3.5 Specification Conformance

As stated above, through underspecification there is room for diverse behavior patterns. As only the access level and optionality of files is defined, we created a minimal automaton that modeled the access rules for present files. This automaton only contains the ε, *DF*, *DF—AUTH*, and *DF—AUTH—EF* states. File operations (i.e. *READ BINARY* should only be possible in the *DF—AUTH—EF* state. We therefore abstracted the output into successful operations (*9000*) and unsuccessful operations (*NOK*) for optional files in the other states. It is, however, not significant for an optional EF if access to it has been denied because of lack of authentication or because the file is not present. Since we identified 22 different non-mandatory EFs (all inside the Master and the eMRTD DFs), that would otherwise have led to 2^{22} possibilities equalling just as many specification automata. Inside the *DF—AUTH—EF* we left the transitions out for optional files (since, according to the specification, an operation may or may not successful) and modeled successful read operations for mandatory ones. For conformance checking, two steps were necessary: a) positive checking and b) negative checking.

For a) we used a trace equivalence check with the specification automaton, removing any states and transitions from the learned one that are not in the specification. The pracical implementation is realized by removing all transitions from a learned automaton that lead from or to a state that is not covered within the specification automaton (which, again is not complete and is missing ambiguous transitions, i.e. such for optional files). For the remainder we perform a trace equivalence check between the learned and the specification automata. We realize this by converting the LearnLib output in the Graphviz (.dot) format into the Aldebaran (.aut) format and feed it into the mCRL2 tool [17] for trace equivalence checking. However, all non-covered transitions will be removed in an examined learned automaton as well, so the trace equivalence shall hold if the SUL conforms.

For b) we performed a primitive form of model checking using a simple rule set:
(i) Since for all mandatory files reside in the LDS1.eMRTD application and authentication is required for access, a READ BINARY must be secured (*SRD_BIN*) and executed for from the *DF—AUTH—EF* state) to yield a positive result (*9000*).
(ii) For *DF—AUTH—EF* to be in a *guaranteed* authenticated state, every transition targeting that state must come from *DF—AUTH* or come through a suc-

cessful authentication (*BAC / 9000*) transition[4]. This enables for efficient, automatic conformance checking that is as comprehensive as the specification allows.

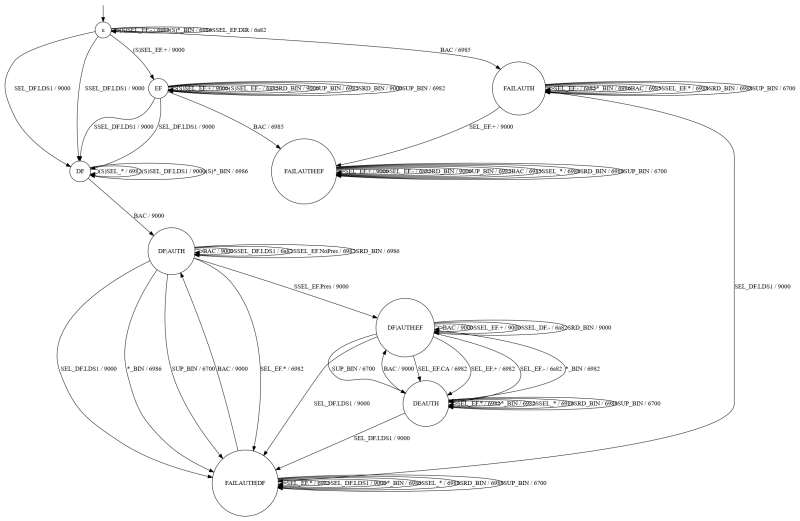

Figure 11.4: Learned and labeled model of an Austrian passport. The diagram was simplified for better readability.

11.4 Evaluation

We put the methodology to test at two different passports from the Republic of Austria: one current and one expired 5 years ago. We were able to infer models of these passports and observed subtle differences, particularly that the elder one obviously does not support PACE (missing the respective CardAccess and CardSecurity files in the master record). Figure 11.4 shows a diagram of the automaton of the current passport, which is simplified for readability but a full model in the sense that it is not the reduced version used for conformance checking as outlined in Section 11.3.5. The model shows additional states

[4]While other possibilities are possible in principle, they cannot be guaranteed to be conform with the standard; as a de-authentication might have occurred, the authentication can not be take for granted.

as compared to the (partial) specification, namely a de-authenticated state, as well as failed authentications and combinations of this new and the other states. Concretely, the additional states are *EF* (a selected Elementary File in the Master file), *FAILAUTH*, *FAILAUTH—EF*, *FAILAUTH—DF*, and *DEAUTH* (a de-authentication after a wrongly selected EF in the *DF—AUTH—EF* state. The main task for a checker is to make sure that no illegal operation (i.e., reading or manipulation operation) occurs in these states. Both examined objects passed the conformation tests (equivalence and simple model checking) as outlined in Section 11.3.5.

11.5 Related Work

We 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 [9]). Apart from that, usage of similar approaches for compliance checking is sparse. For Mealy Machines, Tappler et al. [18] used bisimulation for comparison and there is work for a similar approach checking an embedded control software for its correctness [19].

11.6 Conclusion

In this paper, we demonstrated the usage of automata learning to infer models of passports and check whether they comply to international machine readable travel document specification. We therefore distilled the relevant information to create a specification automaton out of the relevant documents and modeled a labeled transition system out of it, which contains the standard-compliant behavior. Since many of the files are not mandatory, we abstracted the output for the comparison of optional files outside the state where a file was selected and authentication successful. We therefore coped with an underspecified standard using incomplete automata. This system was compared with a learned model of an actual passport, using the input alphabet described in Section 11.3.2. Since the models displayed equivalent behavior, evidence for conformance with the ICAO standard was provided.

11.6.1 Discussion

The current implementation is not complete and has therefore some limitations. Most prominently, no other authentication mechanisms than BAC was implemented, therefore parts of the specification (particularly PACE-related) could not be tested – this also ruled out some newer passports, e.g., current German ones, as systems-under-test, as BAC can be completely abandoned as authentication mechanism in favor of PACE. Also, no systems containing LDS2 applications were available, so these could also not be tested.

11.6.2 Outlook

The methods in this paper provide principally a very thorough method of NFC-based data systems (particularly passports). This method can easily adapted to be used with other systems and protocols, once provided with an adequate specification, learner adapter and input alphabet. Another direction to move forward is to test more specifically: instead of checking equivalent behavior with a specification automaton, specific rules can be applied for a model checker to check the system for certain properties (particularly, security properties). Looking in another direction, the specification models are created manually so far. To further automate the process techniques like Natural Language Processing (NLP) using small or large language models can be used to create specifications automata from standards or specification documents. This also facilitates the use case of Original Equipment Manufacturers (OEMs) being able to very thoroughly examine the

Acknowledgements

This research received funding within the CHIPS Joint Undertaking (JU) under grant agreement No. 101007350 (project AIDOaRt). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Sweden, Spain, Italy, France, Portugal, Ireland, Finland, Slovenia, Poland, Netherlands, Turkey. The document reflects only the author's view and the Commission is not responsible for any use that may be made of the information it contains.

Bibliography

[1] 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.

[2] International Organization for Standardization, "Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange," ISO/IEC Standard "7816-4", International Organization for Standardization, 2020.

[3] I. C. A. Organization, "Machine Readable Travel Documents – Part 9: Deployment of Biometric Identification and Electronic Storage of Data in MRTDs (Eigth Edition)," ISO/IEC Standard 9303-9, International Civil Aviation Organization, 2021.

[4] International Civil Aviation Organization, "Machine Readable Travel Documents – Part 11: Security Mechanisms for MRTDs (Eigth Edition)," ISO/IEC Standard 9303-11, International Civil Aviation Organization, 2021.

[5] 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.

[6] T. Chothia and V. Smirnov, "A Traceability Attack against e-Passports," in *Financial Cryptography and Data Security* (R. Sion, ed.), pp. 20–34, Springer, 2010.

[7] M. Arapinis, V. Cheval, and S. Delaune, "Composing Security Protocols: From Confidentiality to Privacy," in *Principles of Security and Trust* (R. Focardi and A. Myers, eds.), pp. 324–343, Springer, 2015.

[8] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, Nov. 1987.

[9] S. Marksteiner, M. Sirjani, and M. Sjödin, "Using Automata Learning for Compliance Evaluation of Communication Protocols on an NFC Handshake Example," in *Engineering of Computer-Based Systems* (J. Kofroˇ n, T. Margaria, and C. Seceleanu, eds.), vol. 14390 of *Lecture Notes in Computer Science*, pp. 170–190, Springer Nature Switzerland, 2023.

[10] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, Apr. 2008.

[11] M. Isberner, F. Howar, and B. Steffen, "The Open-Source LearnLib," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), Lecture Notes in Computer Science, (Cham), pp. 487–495, Springer International Publishing, 2015.

[12] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," in *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, (New York, NY, USA), pp. 411–420, Association for Computing Machinery, Feb. 1989.

[13] M. J. Kearns and U. Vazirani, *An Introduction to Computational Learning Theory*. MIT Press, Aug. 1994.

[14] M. Merten, F. Howar, B. Steffen, and T. Margaria, "Automata Learning with On-the-Fly Direct Hypothesis Construction," in *Leveraging Applications of Formal Methods, Verification, and Validation* (R. Hähnle, J. Knoop, T. Margaria, D. Schreiner, and B. Steffen, eds.), vol. 336, pp. 248–260, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[15] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), Lecture Notes in Computer Science, (Cham), pp. 307–322, Springer International Publishing, 2014.

[16] F. D. Garcia, G. de Koning Gans, and R. Verdult, "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.

[17] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse, "The mCRL2 Toolset for Analysing Concurrent Systems," in *Tools and Algorithms for the Construction and Analysis of Systems* (T. Vojnar and L. Zhang, eds.), Lecture Notes in Computer Science, (Cham), pp. 21–39, Springer International Publishing, 2019.

[18] M. Tappler, B. K. Aichernig, and R. Bloem, "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.

[19] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software," in *Formal Methods and Software Engineering* (M. Butler, S. Conchon, and F. Zaïdi, eds.), (Cham), pp. 67–83, Springer International Publishing, 2015.

Chapter 12

Paper VI: Black-box protocol testing using Rebeca and Automata Learning

Stefan Marksteiner, Mikael Sjödin.

In Rebeca for Actor Analysis in Action: Essays Dedicated to Marjan Sirjani on the Occasion of Her 60th Birthday, E. A. Lee, M. R. Mousavi, and C. Talcott, Eds., in Lecture Notes in Computer Science, vol. 15560. Springer Nature, 2025. Reproduced with permission from Springer Nature. DOI: 10.1016/0890-5401(87)90052-6.

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.

12.1 Introduction

12.1.1 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 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 [1] (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.

12.1.2 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.

12.1.3 Approach

Starting from existing work on learning a behavioral model using automata learning and comparing it with a (partial) specification [2, 3], we use Rebeca to create a partial model of the International Civil Aviation Organization's (ICAO) Doc 9303 part 9 standard [4], 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 [5]) and standardized integrated-circuit interfaces (ISO/IEC 7816-4 [6]). 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 [3]. Rebeca's integration environment (Afra) comes with a specific model checker (Modere) [1]. 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 [7] 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 12.4.3). We can then check whether the learned implementation model is included (using simulation or trace preorder – see Section 12.4) in the verified specification. We use the MCRL2 toolset's [8] *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 12.4.2) and implemented this into a tool-supported process (see Figure 12.1).

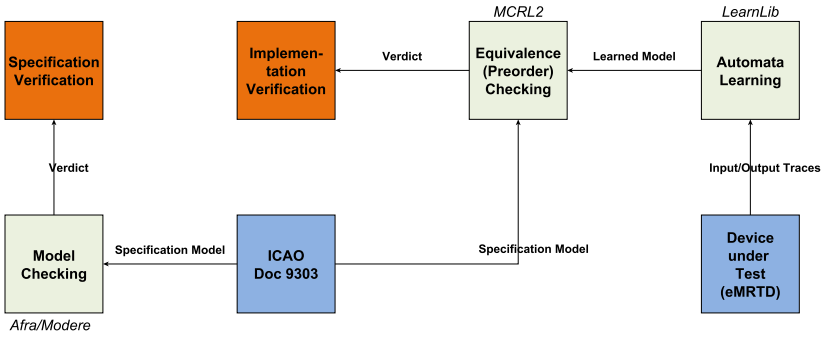

Figure 12.1: Overview of the approach. Green are processes, blue are external inputs, and amber are outputs (i.e., results), the arrow labels are artifacts (input artifacts for arrows from blue to green boxes, output artifacts from green to amber boxes. The italic labels next to the green boxes denote the used tool sets/frameworks.

12.1.4 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.

12.2 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.

12.2.1 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 labeling 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 labeling function [9]. 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 [10]. To model real-world, reactive systems, automata types that provide input and output are used. The two most common are Mealy [11] and Moore machines [12], 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, Σ the input alphabet, 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 ($\lambda : Q \times \Sigma \rightarrow \Omega$) – 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 [13].

12.2.2 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 12.2.3) model of the implemented system with a correct (see Section 12.2.5) 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 [9]. 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) [14]. Formally defined:

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 [9]:*

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 $Post(s, \alpha) = \{s\prime \in Q | s \rightarrow s\prime\}$ [9]. 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 5 (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 6 (Bisimilarity)**.** *of two LTS ($LTS_1 \sim LTS_2$) has a binary relation $R \subseteq Q \times Q$, such that [9]:*

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 7 (Trace equivalence)**.** $Traces(LTS_1) = Traces(LTS_2)$

12.2.3 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 [15], 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_σ 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_ω 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 λ (simultaneously traversing through the sates according to δ). 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 [16]. 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) [17], Direct Hypothesis Construction (DHC) [18], TTT [19], and L# [20]. 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*[1] amount of (potentially long) queries after a certain strategies, e.g., random walks [21].

12.2.4 LearnLib

Learnlib [7] is arguably the most widely used library for automata learning (however, there are others, e.g., AALpy [22] or Libalf [23]). 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 [24]. 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).

12.2.5 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 [9]. 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 [25] formulas with temporal modalities. Those modalities are

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

- *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.

12.2.6 Rebeca

Rebeca is a modeling language that can be used to model reactive systems with a Java-like syntax [1]. It possesses its own modeling IDE (Afra [26]), which has also a built-in model checker (Modere [27]) 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.

12.2.7 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 [5] 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.

12.2.8 Integrated Circuit Access

ISO/IEC 7816-4 [6] defines data structures for transmission (both wired and wireless) to and from integrated circuit cards, including PICCs in the NFC protocol. 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)

12.2.9 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 [4]. 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

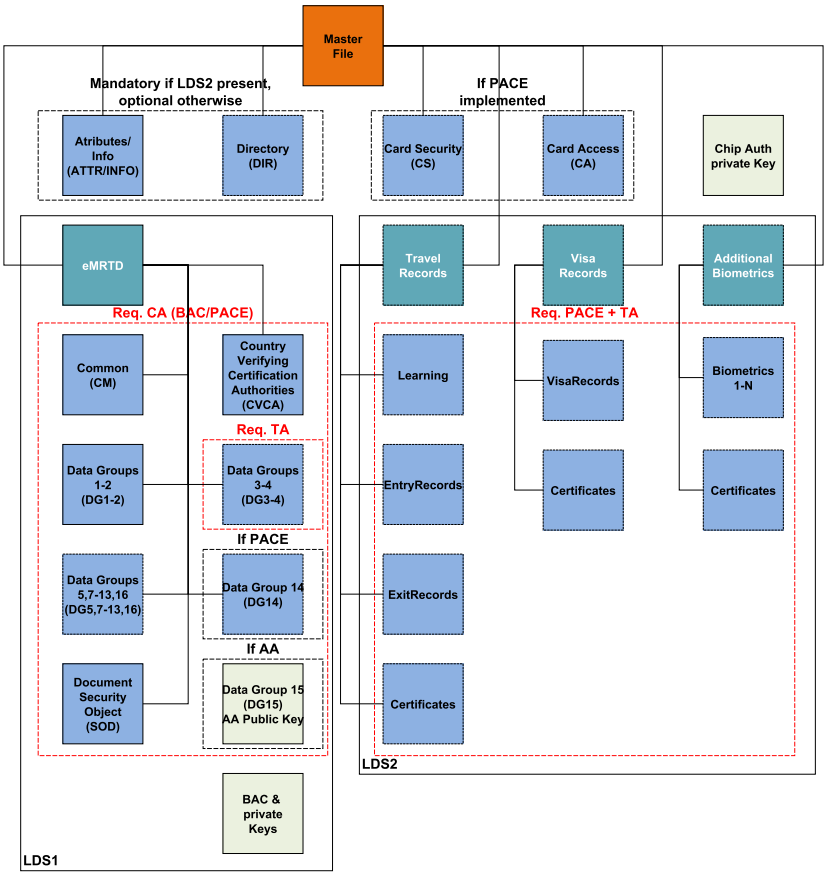

Figure 12.2: Logical Data Structure of Machine Readable Travel Documents from [3].
Amber is the master file (MF), Cyan are dedicated files (DF), Blue are Elementary Files (EF), and Green are key files. Solid frames means mandatory files, dashed ones optional files. Solid boxes donate the LDS contexts, dashed black boxes requirements, and dashed red boxes necessary authentication.

is underspecified. This means that more than one implementation is correct, which makes the standard non-deterministic from a modeling perspective.

12.3 Learning

We use Learnlib (see Section 12.2.4) 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 [3], which we extend with the aspect of checking the specification model.

12.3.1 NFC Interface

To interact with eMRTDs, we created an NFC interface for Learnlib [2]. We connect the developed SUL class via a Socket to an API (based on C++) that operates a Proxmark3 device [28], running a custom firmware enhanced to support automata learning.

12.3.2 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.

12.3.3 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 12.2.9, 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.

12.3.4 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 [3]:

- 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.

12.4 Compliance Evaluation

To determine an implementation's compliance with the standards, we compare a learned implementation model from previous works [3] with a specification model derived from the standards (see Section 12.4.1). 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.

12.4.1 Specification Model

We model the eMRTD specification as outlined in Section 12.2.9 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 12.1 gives an example of the secure SELECT EF CA and CVCA command[2]. 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 12.1: 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;
```

[2]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.

```
        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 12.2 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 12.2: Exemplary message server for the OK answer followed by a reset.

```
[...]
msgsrv reset(){
    OK=false;
    rdBinOK=false;
    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 12.3 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.

12.4.2 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: $\Box(\neg\ (\text{SSELEFOK} \wedge \neg(\text{DF} \wedge \text{AUTH})))$;

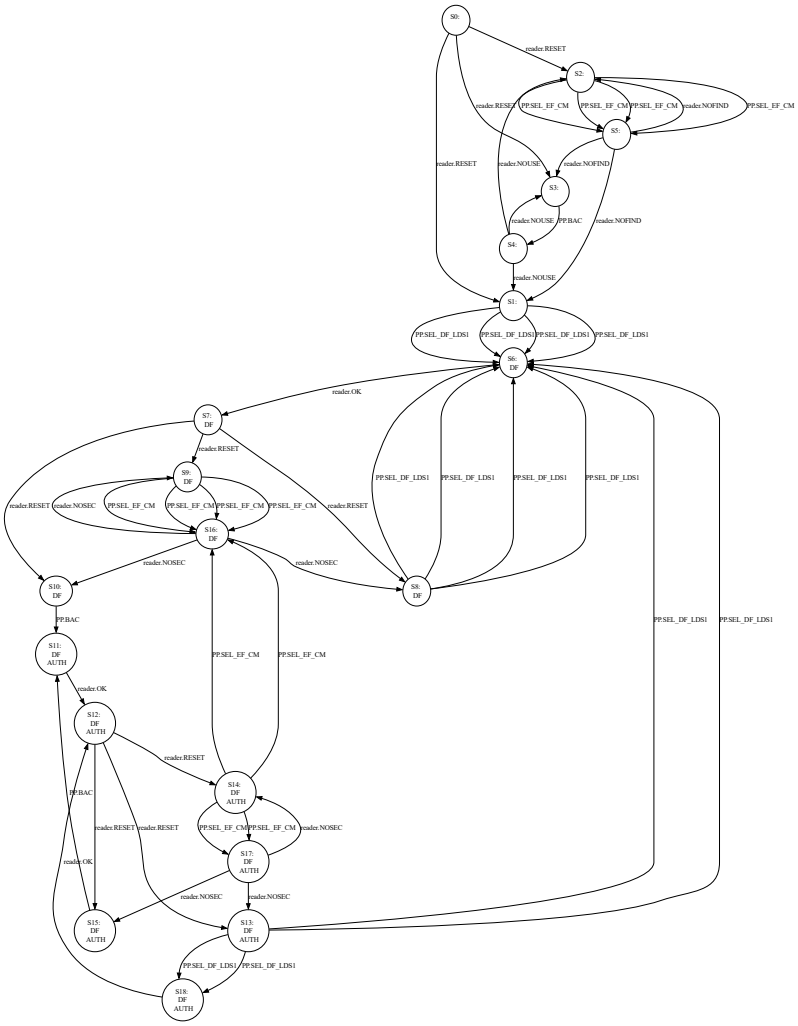

Figure 12.3: Example of a simple Rebeca model. It only covers selecting one EF (CM), one DF (LDS1) and a BAC authentication command.

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).

12.4.3 Converting the Specification Model

A full state space export (see Section 12.4.2) comes in an XML format, that can be converted into the Graphviz (DOT) format. However, displaying two communicating reactive systems (see Section 12.3), 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

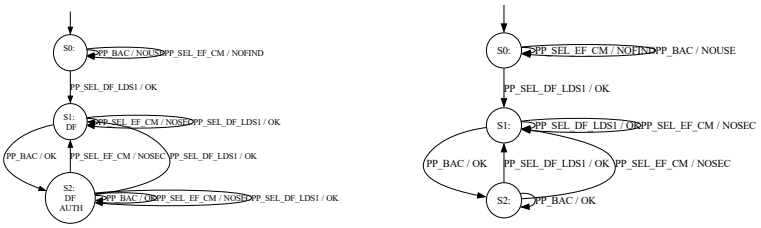

Figure 12.4: Simplified example of a Rebeca model converted in a non-deterministic Mealy-style LTS compared to a learned MRTD Mealy model. Note that the difference lies in additional transitions in the Rebeca version, modeling optional behavior. The Rebeca model shows the same behavior as the one in Figure 12.3.

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 12.4 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 12.5).

12.5 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 12.5), 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 12.6 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 [8] 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).

12.6 Related Work

There are approaches for combining automata learning and bisimulation algorithms [29, 30]. However, to the best of our knowledge, there are no approaches for combining model checking, learning and preorder checking as ours. We funded this approach on our method on partly specifying the ICAO eMRTD specification [3] 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 [2]).

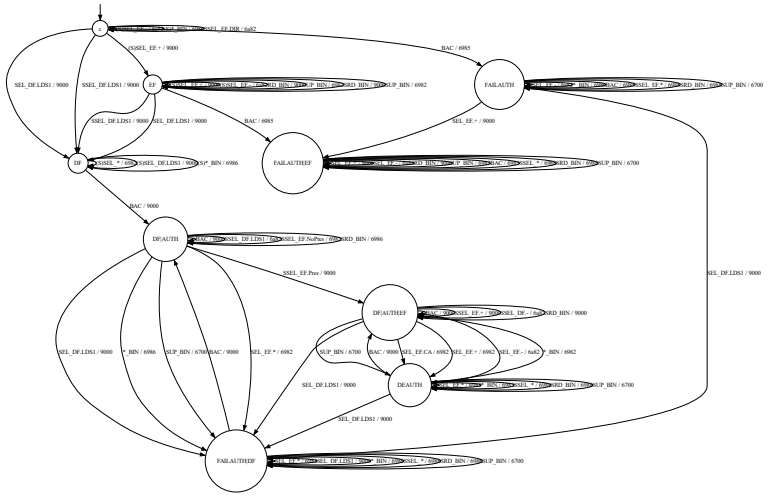

Figure 12.5: Labeled model of an Austrian passport learned with TTT from [3].

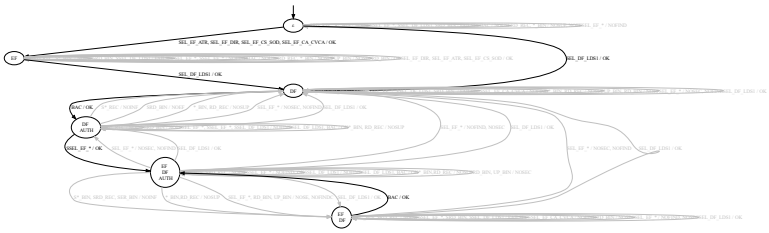

Figure 12.6: Example of the modeled ICAO specification. Please note that we simplified the transitions because of the complexity. Solid lines are transitions towards more elevated access rights, light lines are transitions leading towards the same or lower access privileges.

12.7 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.

12.7.1 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.

12.7.2 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.

Acknowledgements

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! We further acknowledge the support of the Swedish Knowledge Foundation via the industrial doctoral school RELIANT, grant nr: 20220130.

Bibliography

[1] M. Sirjani, "Rebeca: Theory, Applications, and Tools," in *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures* (F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, eds.), vol. 4709 of *Lecture Notes in Computer Science*, pp. 102–126, Springer, 2006.

[2] S. Marksteiner, M. Sirjani, and M. Sjödin, "Using Automata Learning for Compliance Evaluation of Communication Protocols on an NFC Handshake Example," in *Engineering of Computer-Based Systems* (J. Kofroň, T. Margaria, and C. Seceleanu, eds.), vol. 14390 of *Lecture Notes in Computer Science*, (Cham), pp. 170–190, Springer Nature Switzerland, 2023.

[3] S. Marksteiner, M. Sirjani, and M. Sjödin, "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, (New York, NY, USA), pp. 1–8, Association for Computing Machinery, July 2024.

[4] I. C. A. Organization, "Machine Readable Travel Documents – Part 9: Deployment of Biometric Identification and Electronic Storage of Data in MRTDs (Eigth Edition)," ISO/IEC Standard 9303-9, International Civil Aviation Organization, 2021.

[5] 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.

[6] International Organization for Standardization, "Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange," ISO/IEC Standard "7816-4", International Organization for Standardization, 2020.

[7] M. Isberner, F. Howar, and B. Steffen, "The Open-Source LearnLib," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), Lecture Notes in Computer Science, (Cham), pp. 487–495, Springer International Publishing, 2015.

[8] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse, "The mCRL2 Toolset for Analysing Concurrent Systems," in *Tools and Algorithms for the Construction and Analysis of Systems* (T. Vojnar and L. Zhang, eds.), Lecture Notes in Computer Science, (Cham), pp. 21–39, Springer International Publishing, 2019.

[9] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, Apr. 2008.

[10] J. F. Groote and F. Vaandrager, "Structured operational semantics and bisimulation as a congruence," *Information and Computation*, vol. 100, pp. 202–260, Oct. 1992.

[11] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, pp. 1045–1079, Sept. 1955.

[12] E. F. Moore, "Gedanken-Experiments on Sequential Machines," in *Automata Studies*, vol. 34 of *AM-34*, pp. 129–154, Princeton University Press, 1956.

[13] M. Tappler, B. K. Aichernig, and R. Bloem, "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.

[14] J. M. Champarnaud and F. Coulon, "NFA reduction algorithms by means of regular inequalities," *Theoretical Computer Science*, vol. 327, pp. 241–253, Nov. 2004.

[15] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, Nov. 1987.

[16] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," in *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, (New York, NY, USA), pp. 411–420, Association for Computing Machinery, Feb. 1989.

[17] M. J. Kearns and U. Vazirani, *An Introduction to Computational Learning Theory*. MIT Press, Aug. 1994.

[18] M. Merten, F. Howar, B. Steffen, and T. Margaria, "Automata Learning with On-the-Fly Direct Hypothesis Construction," in *Leveraging Applications of Formal Methods, Verification, and Validation* (R. Hähnle, J. Knoop, T. Margaria, D. Schreiner, and B. Steffen, eds.), vol. 336, pp. 248–260, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[19] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), Lecture Notes in Computer Science, (Cham), pp. 307–322, Springer International Publishing, 2014.

[20] F. Vaandrager, B. Garhewal, J. Rot, and T. Wißmann, "A New Approach for Active Automata Learning Based on Apartness," in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Fisman and G. Rosu, eds.), (Cham), pp. 223–243, Springer International Publishing, 2022.

[21] D. Lee, K. Sabnani, D. Kristol, and S. Paul, "Conformance testing of protocols specified as communicating finite state machines-a guided random walk based approach," *IEEE Transactions on Communications*, vol. 44, pp. 631–640, May 1996.

[22] E. Muškardin, B. K. Aichernig, I. Pill, A. Pferscher, and M. Tappler, "AALpy: An active automata learning library," *Innovations in Systems and Software Engineering*, vol. 18, pp. 417–426, Sept. 2022.

[23] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon, "Libalf: The Automata Learning Framework," in *Computer Aided Verification* (T. Touili, B. Cook, and P. Jackson, eds.), (Berlin, Heidelberg), pp. 360–364, Springer, 2010.

[24] L. Kruger, S. Junges, and J. Rot, "State Matching and Multiple References in Adaptive Active Automata Learning," in *Formal Methods*

(A. Platzer, K. Y. Rozier, M. Pradella, and M. Rossi, eds.), (Cham), pp. 267–284, Springer Nature Switzerland, 2025.

[25] B. Russell, "Mathematical Logic as Based on the Theory of Types," *American Journal of Mathematics*, vol. 30, no. 3, pp. 222–262, 1908.

[26] E. Khamespanah, M. Sirjani, and R. Khosravi, "Afra: An Eclipse-Based Tool with Extensible Architecture for Modeling and Model Checking of Rebeca Family Models," in *Fundamentals of Software Engineering* (H. Hojjat and E. Ábrahám, eds.), (Cham), pp. 72–87, Springer Nature Switzerland, 2023.

[27] M. M. Jaghoori, A. Movaghar, and M. Sirjani, "Modere: The model-checking engine of Rebeca," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, (New York, NY, USA), pp. 1810–1815, Association for Computing Machinery, Apr. 2006.

[28] F. D. Garcia, G. de Koning Gans, and R. Verdult, "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.

[29] Y.-F. Chen, C.-D. Hong, A. W. Lin, and P. Rümmer, "Learning to prove safety over parameterised concurrent systems," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, pp. 76–83, Oct. 2017.

[30] C.-D. Hong, A. W. Lin, R. Majumdar, and P. Rümmer, "Probabilistic Bisimulation for Parameterized Systems," in *Computer Aided Verification* (I. Dillig and S. Tasiran, eds.), Lecture Notes in Computer Science, (Cham), pp. 455–474, Springer International Publishing, 2019.

Chapter 13

Paper VII: Learning Single and Compound-protocol Automata and Checking Behavioral Equivalences

Stefan Marksteiner, David Schögler, Marjan Sirjani, Mikael Sjödin.

In International Journal on Software Tools for Technology Transfer, Volume 27, pages 35–52. Springer Nature, 2025. Reproduced with permission from Springer Nature. DOI: 10.1007/s10009-025-00797-y.

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 non-conformance. 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.

13.1 Introduction

This article is based on a paper [1], 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 [2] 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) [3] 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 structures as follows. First we provide its motivation and contribution. Section 13.2 gives an overview of basic concepts in this paper, including a formal definition of bisimulation for Mealy Machines as used in this paper. Sections 13.3.1, 13.3.2, and 13.3.3 describe the developed interfaces for automata learning of NFC systems, BLE systems, and compound-protocol learning, respectively. Sections 13.4.1, 13.4.2, and 13.4.3 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 Section 13.5 describes the methods for conformance checking and the respective protocols' specifics in this regard. Section 13.6 shows real-world results, while Section 13.7 compare them to the works of others. Section 13.8, eventually, concludes the paper and gives and outlook on future work.

13.1.1 Motivation

Both the NFC and BLE protocol are widely adopted protocol 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., [4, 5]), also specifically regarding the ISO/IEC 14443-3 handshake (e.g., [6, 7]), and BLE (e.g., [8]) there are few works on comprehensive testing (see Section 13.7). 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 Section 13.2.2).

13.1.2 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 [9] with Rivest/Schapire (LSR) closure [10] 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 [11] 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 Section 13.5.2).

13.2 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.

13.2.1 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 [12]. 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) [13]. 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 [14].

13.2.2 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 [2] describe 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 [15], and by Paige and Tarjan [16]. Fig. 13.1 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 is used. Simulation (as used in the simulation preorder, see below) means that one automaton can completely reproduce the behavior

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

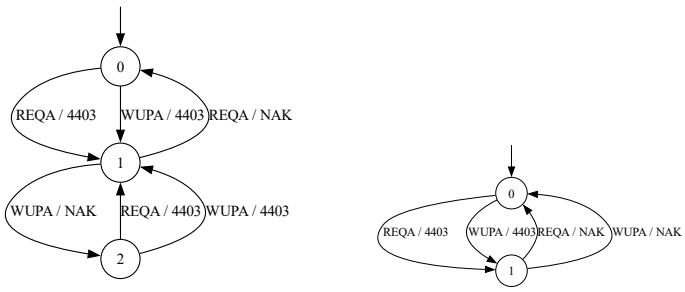

Figure 13.1: Example for a partial automaton and its minimal counterpart.

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 [17]. Trace equivalence compares the respective input/output sequences of automata (see below). Just uni-directional 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.

Definition 8 (Bisimilarity)**.** *Bisimlarity of two LTS ($LTS_1 \sim LTS_2$) is defined as exhibiting a binary relation $R \subseteq Q \times Q$, such that [18]:*

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

Condition A of Definition 8 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 [18]. 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 [19]. Based on this, we define Mealy bisimilarity ($M_1 \sim M_2$) for our purpose follows:

Definition 9 (Mealy Bisimilarity)**.** *Bisimliarity 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 $q_1 \in Q_1, q_2 \in Q_2 \cdot (q_1, q_2) \in R$ *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 Section 13.2.1, 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 non-observable (internal) transitions (ordinarily labeled as τ transitions), e.g., strong and weak bisimulation, and branching bisimulation to give a few examples [18]. 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 Section 13.2.3) 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 Section 13.5.2) with weak bisimulation.

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

Definition 10 (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 path (or path fragment) $\pi = q_0, q_1, ... | trace(\pi) = L(q_0), L(q_1), ..$ [18]. Since in a Mealy machine, we use the output function as a labeling function, traces are sequences of outputs $trace(\pi) = \lambda(q_0, \sigma_0), \lambda(q_1, \sigma_1), ...$ As only the input and output is directly observable (not the states themselves), we use an input/output as notion in the form $\langle \sigma_0/\omega_0 \rangle, \langle \sigma_1/\omega_1 \rangle, ..$ with $\sigma \in \Sigma$ and $\omega \in \Omega$.

Both bisimulation and trace equivalence might be principally 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 [20, 16, 21]. 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 [18]. 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 Section 13.5.1).

Simulation preorder means that only one system has to be able to simulate the other. Informally, that 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:

Definition 11 (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 [18]:*

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

.

This basically means a not (necessarily) symmetric bisimulation relation. Analogously, this means for trace preorder 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 12 (Trace preorder)**.** *Trace preorder of two LTS is defined as* $Traces(LTS_1) \subseteq Traces(LTS_2)$

13.2.3 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 [22] define a hiding operator (τ_I) that removes any actions in certain set of actions (I) and labels them as non-observable (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 [23].

13.2.4 Automata Learning

The classical method of actively learning automata of systems, was outlined in Angluin's pivotal work known as the L* algorithm [9]. 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) [24]. 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 [11], is deemed currently the most efficient [25]. Other widely used algorithms include a modified version of the original L* with a counterexample handling strategy by Rivest and Schapire [10], or the tree-based Direct Hypothesis Construction (DHC) [26] and Kearns-Vazirani (KV) [27] 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 [28]. Originally for Deterministic Finite Automata, this learning method could be used to learn Mealy Machines [29]. This preferred for learning black box reactive systems (e.g., cyber-physical systems), as modeling these as Mealy is comparatively simple.

13.2.5 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 (∇), 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 rationales 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 section 13.3.1).

13.2.6 LearnLib

To utilize automata learning we use a widely adopted Java library called LearnLib [30]. 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 uses to factor queries used to fill an observation table or tree. Depending on the used algorithms, the library is capable of inferring DFAs, NFAs (Non-deterministic finite acceptors), Mealy machines or VPDAs (Visibly Pushdown Automata).

13.2.7 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 device PCD) that also serves as field for data transmission. There are a couple of defined procedures that allow for operating proximity cards in presence of other wireless objects in order to exchange data [31]. One standard particularly defines two handshake procedures based on cascade-based anti-collision and card selection (called type A and type B), one of which NFC proximity cards must

be compliant with [2]. 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.

13.2.8 The NFC Handshake Automaton

ISO 14443-3 contains a state diagram that outlines the Type A handshake procedure for an NFC connection (see Fig. 13.2). This diagram is not a state machine of the types described in Section 13.2.1, for it lacks both output and final states. As we learn Mealy machines, we augmented it with abstract outputs (see Sections 13.4.1 and 13.5.1) 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 [31]) 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) occur. 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, no giving an 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 a *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** state 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

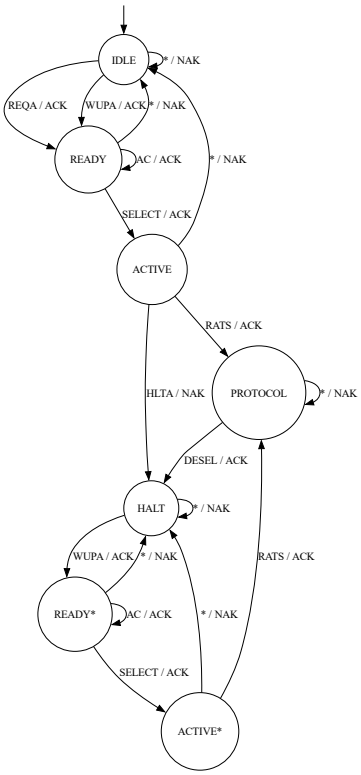

Figure 13.2: NFC handshake automaton after ISO 14443-3 [2] augmented with abstract outputs. Note: star (*) as input means any symbol that is not explicitly stated in another outbound transition of the respective state.

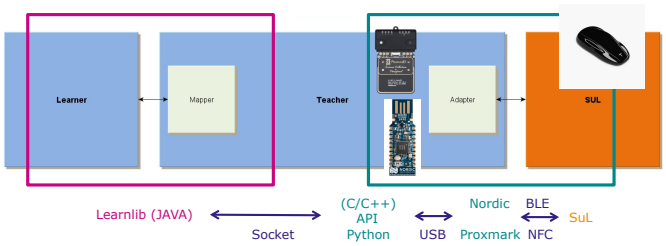

Figure 13.3: Learner interface setup.

*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 Section 13.2.5)

There is also one specific feature to the NFC handshake protocol: unlike most communication protocols, an unexpected or wrong input yield to no output. This has an implication to learning, as a timeout will be interpreted as a general error message.

13.2.9 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 [3].

13.3 Learning Interfaces

As a learner, we use the algorithm implementations in the Learnlib Java library (see Section 13.2.6). This section outlines the details for the setup for NFC, BLE and compound learning, and the respective teacher and SUL interfaces.

13.3.1 NFC Interface

The learner is configured as outlined in Section 13.4.1 using an adapter class (cf. 13.3.1). To interact with the NFC SUL, a Proxmark RFID/NFC device (see Section 13.3.1) is used that works with an adapter written in C++ (see Section 13.3.1). The l Fig. 13.3 provides an overview of the setup.

Learner Interface Device

The interface with an NFC SUL is established via Proxmark3. Proxmark3 is a pocket-size NFC device capable of acting as an NFC reader (PCD) or tag (PICC), as well as sniffing device [32]. 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 Section 13.3.1).

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 Section 13.2.5) 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 non-deterministic 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 re-implemented to send bulk inputs (i.e. send complete input words instead of single symbols), which improved the throughput significantly and solved non-determinism.

Firmware Modifications

In order to be able to transfer traces word-wise instead of symbol-wise, 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 de-serializes 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 time outs these modifications, eliminating a great portion of the latency times of frequently used USB connections, boost the performance of the learning using different learning algorithms significantly (for a performance evaluation see Section 13.4.1).

13.3.2 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 [33]. 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. 13.3).

Learner Interface Device

For learning BLE systems, we used a Nordic nRF52840, which is a multi-protocol SoC that supports most of the defined hardware functionalities in the Bluetooth Core Specifications 5.3 [34]. 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.

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 [3].

The communication with the firmware is split into two kind of messages:

- Packet transmitted or received
- Commands to change connection parameter

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 to prevent re-querying the system with duplicate traces for the equivalence oracle and allows to resume learning.

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 off 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 [35], we re-implemented

[2] https://scapy.net/

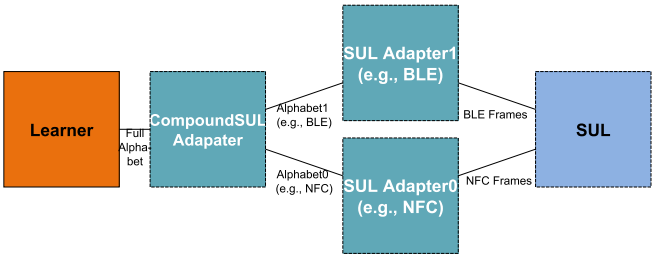

Figure 13.4: Compound protocol interface setup. Amber is the Learner class, turquoise the SUL adapter classes, and blue the actual SUL.

the BLE controller to have an open access to all link layer functions necessary for proper learning.

13.3.3 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. 13.4).

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 the one 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.

13.4 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.

13.4.1 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 is a characteristic feature of the NFC standard. This directly affects the learning because it yields many identical answers and efficient time-out handling is essential. It is therefore necessary to evaluate different state-of-the-art learning algorithms for their specific fitness (see Section 13.4.1), as well as determining the optimal parameter set (Section 13.4.1). 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.

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*), is determined by the method and number of conformance tests. Generally speaking, a too short (maximum) word length results in an incompletely learned (which, if the implementation is correct, should contain seven states). The

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

Table 13.1: Runtime (minutes) per algorithm and maximum word length.

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 13.1 shows a comparison of the runtime of different algorithms with different maximum word lengths (in red the respective algorithm's shortest runtime that learned the correct 7-state model). Some of the non-steadiness 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 13.2 shows the results with the best performing (correct) run of the respective algorithm. This, however, only covers the performance of learning a correct implementation. The opposite side, discovering a bug, shows a different picture. We therefore used a SUL with a slightly deviating behavior (see Section 13.6.2). 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 by their 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 non-compliant 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 worst. The runtimes

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	10192	2588	5124	7932	7607	6628	6213
EQs	2	3	2	5	5	4	4

Table 13.2: Performance evaluation of different algorithms for a compliant system with their respective fastest calibration in the given setting.

roughly correspond with the amount of sent symbols, in this case the a very long timeout has to be set to avoid non-determinism. The classical L* is not in the list, as the algorithm crashed after more than 24 hours of runtime. Table 13.3 provides an overview of minimum word lengths, run time, words, symbols and equivalence queries. Lower minimum word lengths yielded false negatives (i.e. the result showed a correct model with the deviation not uncovered).

Input and Output Alphabets

For the input alphabet we use the one needed for successfully establishing a handshake (cf. Fig. 13.2), according to the state diagram for Type-A cards in the ISO 14443-3 standard [2]:

- 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 [31]. 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.

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	14637	15262	23867	19241	13353	11769
Eqs	5	3	6	6	5	5

Table 13.3: Performance evaluation of different algorithms for a non-compliant system with their respective fastest calibration in the given setting.

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 non-deterministically, 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 anti-collision (*AC*) and *SELECT* yields a different output, which introduces non-deterministic 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 Section 13.2.7). 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 Section 13.2.8). This idea was therefore forfeit in favor of raw output for the learning. We still maintained this higher abstraction for the equivalence checking (see Section 13.5.1 for the reasoning). Raw output, however, retains this problematic non-determinism. We therefore introduce a caching strategy to cope with this issue. Whenever a valid (partial) UID is received as an answer to an anti-collision 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.

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 non-compliance with the ISO 14443-3 standard (given that only the messages defined in that standard are used as an input alphabet - see Section 13.4.1).

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.

13.4.2 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 connections times out and we end in a non-responsive 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. 13.6). This observation is specific to the used SUL, namely a Tesla key fob (see Section 13.6.2). The optimized learner, configured with more input symbols, yielded different results with other systems [36], which also applies to an updated version of the compound learner (see Section 13.6.2).

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.

13.4.3 Learning Compound-Protocol Automata

For compound protocols, we use a container SUL that de-serializes and dispatches SULs to child SULs that handle the interface to the protocols present in the compound SUL (cf. Section 13.3.3 and Fig. 13.4). The actual protocol SULs are invisible to the learner.

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 Section 13.6.2). On adding a protocol SUL, it's 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 de-serialized 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.

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.

13.5 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 multi-protocol 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.

13.5.1 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 Section 13.2.2) [37]. 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 re-engineering 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. [38] for bisimilarity checking. For NFC, we therefore simply model the specification in form of the handshake diagram (see Fig. 13.2) 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.

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. E.g., 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 non-compliance to the ISO 14443-3 standard of the scrutinized implementation. A trace of the non-compliant state/transition is trivial to extract from the automaton (see the example in Section 13.6.2). 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.

BLE Specifics

One specific of the BLE handshake is that the first request always has to be 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 a non-responsive sink state after two to four sent messages, if the timeout is not reset through a message.

13.5.2 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 Section 13.2.2), 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 Section 13.2.3) 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 generate 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.

13.6 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.

13.6.1 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 are 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. Fig. 13.5 shows an example of a learned automaton (left side). 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 [39].

13.6.2 Tesla Key Fob

Apart from significantly slower answers than the other devices, which required to adapt the timeouts to avoid nondeterministic behavior, the learned automaton slightly differs when learned with the TTT algorithm. Fig. 13.5 (right side) shows a model of a Tesla car key fob learned with TTT. The (unnamed) states 3,4 and 6 are very similiar 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

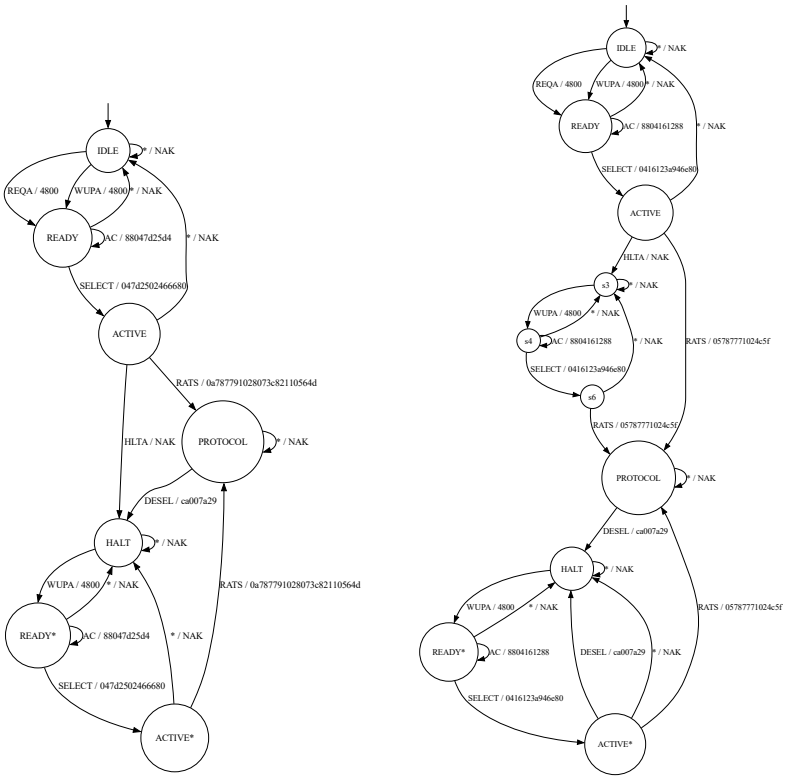

Figure 13.5: NFC automaton of an NXP test card (left) and a Tesla car key fob (right) learned with TTT.

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 ($\langle$*REQA/ACK*$\rangle$, $\langle$*SELECT/ACK*$\rangle$, $\langle$*RATS/ACK*$\rangle$, $\langle$*DESEL/ACK*$\rangle$, $\langle$*WUPA/ACK*$\rangle$, $\langle$*SELECT/ ACK*$\rangle$, $\langle$*DESEL/ACK*$\rangle$). According to the specification, the last label should be $\langle$DESEL/NAK$\rangle$. We also learned a BLE automaton with a reduced set used for comparison with the compound-

Figure 13.6: Automaton of a Tesla key fob's BLE interface, learned with TTT without BLE optimizations (in blue), added (in light gray) are the additional state class when learning a compound model (see Section 13.6.2).

protocol automata (see also Section 13.4.2). Fig. 13.6 shows this automaton, including the deviations from the compound automaton.

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 states automaton that combines both protocols (see Fig. 13.7). We subsequently used the equivalence checks described in Section 13.5.2. For NFC, we found some deviations with both checking methods, because in the compound setting, the learner failed to spot the non-compliance with the ISO/IEC standard (see above in 13.6.2). We subsequently compared it with the standard's specification automaton (cf. Section 13.5.1) 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 non-determinism in trace #1391), the learning took a different path, which resulted in the discovery of the non-compliant transition. For BLE, the preorder checks were positive, meaning that the compound automaton includes the behavior of the standalone 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.

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 standalone 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. 13.7) that differ, i.e., their successor states do not belong to the same label equivalence classes as in the standalone BLE model (Fig. 13.6). This means that these four states form their own class and are not mergeable with the other – 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. 13.7). 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 it's the 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. 13.6). 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.

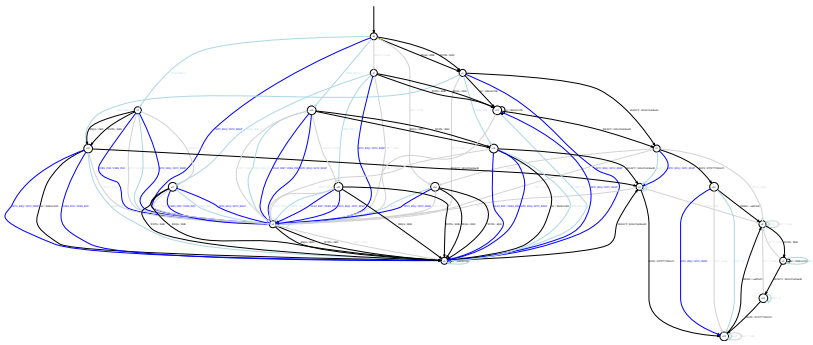

Figure 13.7: Compound NFC/BLE Tesla key fob model learned with TTT. Transitions in black are NFC and blue BLE, pale lines are without output (i.e., timeouts). We also used NFC* and BLE* for *all other* BLE and NFC inputs for better readability.

13.7 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 [40] or probabilistic transition systems (PTS) [41]. Neider et al. [42] contains some significant theoretic fundamentals of using automata learning and bisimulation for different types of state machines, including Mealys. It also contains the important observation that (generalized) Mealy Machines are bisimilar if their underlying LTS are bisimilar. Tappler et al. [19] 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 [43]. There is also previous work on using automata learning for inferring models of BLE systems, with the main target of fuzzing [44]. Another approach that targets compliance checking based on model checking that also sequences the sub-protocol of BLE is from Karim et al. [45]. There is also working on learning NFC card models [46], 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.

13.8 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' performances. 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 complex-

ity made it more difficult to learn. The same learner configuration failed to spot the NFC non-conformance 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.

13.8.1 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 non-compliant system and the minimum word length has an impact on the ability to find non-compliances. 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 non-determinism): one in-time and one timed-out. This issue might be overcome by using timed automata, but this is a complex solution.

13.8.2 Outlook

The compliance checking is but a first step towards assuring correctness and, subsequently, cybersecurity for NFC systems. Concretely, further research di-

rections include test case generation using model checking and targeting upper layer protocols (partly addressed in [39], 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 non-determinism, we would need the ability to recognize time properties. Timed automata, however, are hard to create [47]. To avoid this burden, while still considering timing, Mealy Machines with one [48] or multiple timers [49] 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. 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 nr: 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.

Bibliography

[1] S. Marksteiner, M. Sirjani, and M. Sjödin, "Using Automata Learning for Compliance Evaluation of Communication Protocols on an NFC Handshake Example," in *Engineering of Computer-Based Systems* (J. Kofroň, T. Margaria, and C. Seceleanu, eds.), vol. 14390 of *Lecture Notes in Computer Science*, (Cham), pp. 170–190, Springer Nature Switzerland, 2023.

[2] 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.

[3] Bluetooth SIG, "Bluetooth Specification," Core Specification v5.4, Bluetooth SIG, 2023.

[4] W. Issovits and M. Hutter, "Weaknesses of the ISO/IEC 14443 protocol regarding relay attacks," in *2011 IEEE International Conference on RFID-Technologies and Applications*, pp. 335–342, Sept. 2011.

[5] J. Vila and R. J. Rodríguez, "Practical Experiences on NFC Relay Attacks with Android," in *Radio Frequency Identification* (S. Mangard and P. Schaumont, eds.), Lecture Notes in Computer Science, (Cham), pp. 87–103, Springer International Publishing, 2015.

[6] G. Hancke, "Practical attacks on proximity identification systems," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 6 pp.–333, May 2006.

[7] M. Maass, U. Müller, T. Schons, D. Wegemer, and M. Schulz, "NFCGate: An NFC relay application for Android," in *Proceedings of the 8th ACM*

Conference on Security & Privacy in Wireless and Mobile Networks, WiSec '15, (New York, NY, USA), pp. 1–2, Association for Computing Machinery, June 2015.

[8] D. Antonioli, N. O. Tippenhauer, K. Rasmussen, and M. Payer, "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, (New York, NY, USA), pp. 196–207, Association for Computing Machinery, May 2022.

[9] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, Nov. 1987.

[10] R. L. Rivest and R. E. Schapire, "Inference of Finite Automata Using Homing Sequences," *Information and Computation*, vol. 103, pp. 299–347, Apr. 1993.

[11] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), Lecture Notes in Computer Science, (Cham), pp. 307–322, Springer International Publishing, 2014.

[12] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, pp. 1045–1079, Sept. 1955.

[13] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, Dec. 1943.

[14] E. F. Moore, "Gedanken-Experiments on Sequential Machines," in *Automata Studies*, vol. 34 of *AM-34*, pp. 129–154, Princeton University Press, 1956.

[15] J. Hopcroft, "An n log l Algorithm for Minimizing States in a Finite Automaton," in *Theory of Machines and Computations* (Z. Kohavi and A. Paz, eds.), pp. 189–196, Academic Press, Jan. 1971.

[16] R. Paige and R. E. Tarjan, "Three Partition Refinement Algorithms," *SIAM Journal on Computing*, vol. 16, pp. 973–989, Dec. 1987.

[17] D. Sangiorgi, "On the origins of bisimulation and coinduction," *ACM Trans. Program. Lang. Syst.*, vol. 31, pp. 15:1–15:41, May 2009.

[18] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, Apr. 2008.

[19] M. Tappler, B. K. Aichernig, and R. Bloem, "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.

[20] L. Aceto, A. Ingolfsdottir, and J. Srba, "The algorithmics of bisimilarity," in *Advanced Topics in Bisimulation and Coinduction*, pp. 100–172, Cambridge University Press, 2011.

[21] J. E. Hopcroft and R. M. Karp, "A Linear Algorithm for Testing Equivalence of Finite Automata," tech. rep., Cornell University, Dec. 1971.

[22] J. F. Groote and M. R. Mousavi, *Modelling and Analysis of Communicating Systems*. The MIT Press, 2023.

[23] C. A. R. Hoare, *Communicating Sequential Processes*, vol. 178. Englewood Cliffs: Prentice-Hall, 1985.

[24] B. Jacobs and A. Silva, "Automata Learning: A Categorical Perspective," in *Horizons of the Mind. A Tribute to Prakash Panangaden: Essays Dedicated to Prakash Panangaden on the Occasion of His 60th Birthday* (F. van Breugel, E. Kashefi, C. Palamidessi, and J. Rutten, eds.), Lecture Notes in Computer Science, pp. 384–406, Cham: Springer International Publishing, 2014.

[25] F. Vaandrager, "Model learning," *Communications of the ACM*, vol. 60, pp. 86–95, Jan. 2017.

[26] M. Merten, F. Howar, B. Steffen, and T. Margaria, "Automata Learning with On-the-Fly Direct Hypothesis Construction," in *Leveraging Applications of Formal Methods, Verification, and Validation* (R. Hähnle, J. Knoop, T. Margaria, D. Schreiner, and B. Steffen, eds.), vol. 336, pp. 248–260, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[27] M. J. Kearns and U. Vazirani, *An Introduction to Computational Learning Theory*. MIT Press, Aug. 1994.

[28] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black Box Checking," in *Formal Methods for Protocol Engineering and Distributed Systems* (J. Wu, S. T. Chanson, and Q. Gao, eds.), IFIP Advances in Information and Communication Technology, pp. 225–240, Boston, MA: Springer US, 1999.

[29] M. Shahbaz and R. Groz, "Inferring Mealy Machines," in *FM 2009: Formal Methods* (A. Cavalcanti and D. R. Dams, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 207–222, Springer, 2009.

[30] M. Isberner, F. Howar, and B. Steffen, "The Open-Source LearnLib," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), Lecture Notes in Computer Science, (Cham), pp. 487–495, Springer International Publishing, 2015.

[31] 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.

[32] F. D. Garcia, G. de Koning Gans, and R. Verdult, "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.

[33] D. Schögler, "An Automata Learning Framework for Bluetooth Low Energy," Master's thesis, Graz University of Technology, Graz, Austria, 2023.

[34] J. Ribas Sobreviela, *Bluetooth Low Energy Based on the nRF52840 USB Dongle*. Bachelor thesis, Universitat Politècnica de Catalunya, Oct. 2019.

[35] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sun, and E. Kurniawan, "SweynTooth: Unleashing mayhem over bluetooth low energy," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, (USA), pp. 911–925, USENIX Association, July 2020.

[36] M. Ebrahimi, S. Marksteiner, D. Ničković, R. Bloem, D. Schögler, P. Eisner, S. Sprung, T. Schober, S. Chlup, C. Schmittner, and S. König, "A Systematic Approach to Automotive Security," in *Formal Methods* (M. Chechik, J.-P. Katoen, and M. Leucker, eds.), vol. 14000 of *Lecture Notes in Computer Science*, (Cham), pp. 598–609, Springer International Publishing, 2023.

[37] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse, "The mCRL2 Toolset for Analysing Concurrent Systems," in *Tools and Algorithms for the Construction and Analysis of Systems* (T. Vojnar and L. Zhang, eds.), Lecture Notes in Computer Science, (Cham), pp. 21–39, Springer International Publishing, 2019.

[38] D. N. Jansen, J. F. Groote, J. J. A. Keiren, and A. Wijs, "An O(m log n) algorithm for branching bisimilarity on labelled transition systems," in *Tools and Algorithms for the Construction and Analysis of Systems* (A. Biere and D. Parker, eds.), Lecture Notes in Computer Science, (Cham), pp. 3–20, Springer International Publishing, 2020.

[39] S. Marksteiner, M. Sirjani, and M. Sjödin, "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, (New York, NY, USA), pp. 1–8, Association for Computing Machinery, July 2024.

[40] Y.-F. Chen, C.-D. Hong, A. W. Lin, and P. Rümmer, "Learning to prove safety over parameterised concurrent systems," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, pp. 76–83, Oct. 2017.

[41] C.-D. Hong, A. W. Lin, R. Majumdar, and P. Rümmer, "Probabilistic Bisimulation for Parameterized Systems," in *Computer Aided Verification* (I. Dillig and S. Tasiran, eds.), Lecture Notes in Computer Science, (Cham), pp. 455–474, Springer International Publishing, 2019.

[42] D. Neider, R. Smetsers, F. Vaandrager, and H. Kuppens, "Benchmarks for Automata Learning and Conformance Testing," in *Models, Mindsets, Meta: The What, the How, and the Why Not?* (T. Margaria, S. Graf, and K. G. Larsen, eds.), Lecture Notes in Computer Science, pp. 390–416, Cham: Springer International Publishing, 2019.

[43] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software," in *Formal Methods and Software Engineering* (M. Butler, S. Conchon, and F. Zaïdi, eds.), (Cham), pp. 67–83, Springer International Publishing, 2015.

[44] A. Pferscher and B. K. Aichernig, "Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning," in *NASA Formal Methods* (J. V.

Deshmukh, K. Havelund, and I. Perez, eds.), vol. 13260 of *Lecture Notes in Computer Science*, (Cham), pp. 373–392, Springer International Publishing, 2022.

[45] I. Karim, A. A. Ishtiaq, S. R. Hussain, and E. Bertino, "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, May 2023.

[46] F. Aarts, J. De Ruiter, and E. Poll, "Formal Models of Bank Cards for Free," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pp. 461–468, Mar. 2013.

[47] R. Brouwer, "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, Netherlands, Mar. 2020.

[48] F. Vaandrager, M. Ebrahimi, and R. Bloem, "Learning Mealy machines with one timer," *Information and Computation*, vol. 295, p. 105013, Dec. 2023.

[49] V. Bruyère, B. Garhewal, G. A. Pérez, G. Staquet, and F. W. Vaandrager, "Active Learning of Mealy Machines with Timers," Mar. 2024. Preprint: arXiv:2403.02019.

Chapter 14

Paper VIII: STAF: Leveraging LLMs for Automated Attack Tree-Based Security Test Generation

Tanmay Khule, Stefan Marksteiner, Jose Alguindigue, Hannes Fuchs, Sebastian Fischmeister, Apurva Narayan.

Proceedings of the 23rd escar Europe : The World's Leading Automotive Cyber Security Conference, ESCAR EUROPE'25. Bochum: Ruhr Universität Bochum, 2025. Paper accepted, not yet published. Preprint available on arXiv. DOI: 10.48550/arXiv.2509.20190.

Abstract

In modern automotive development, security testing is critical for safeguarding systems against increasingly advanced threats. Attack trees are widely used to systematically represent potential attack vectors, but generating comprehensive test cases from these trees remains a labor-intensive, error-prone task that has seen limited automation in the context of testing vehicular systems. This paper introduces STAF (Security Test Automation Framework), a novel approach to automating security test case generation. Leveraging Large Language Models (LLMs) and a four-step self-corrective Retrieval-Augmented Generation (RAG) framework, STAF automates the generation of executable security test cases from attack trees, providing an end-to-end solution that encompasses the entire attack surface. We particularly show the elements and processes needed to provide an LLM to actually produce sensible and executable automotive security test suites, along with the integration with an automated testing framework. We further compare our tailored approach with general purpose (vanilla) LLMs and the performance of different LLMs (namely GPT-4.1 and DeepSeek) using our approach. We also demonstrate the method of our operation step-by-step in a concrete case study. Our results show significant improvements in efficiency, accuracy, scalability, and easy integration in any workflow, marking a substantial advancement in automating automotive security testing methods. Using TARAs as an input for verification tests, we create synergies by connecting two vital elements of a secure automotive development process.

14.1 Introduction

Security testing is a crucial component of modern software and system development. With the increasing sophistication of attacks, ensuring the robustness of systems against such threats is critical. Attack trees form a convenient way to systematically categorize the different ways in which a system can be attacked [1]. An attack tree is a hierarchical diagram that represents the various ways an attacker might compromise a system. The root node represents the ultimate goal of the attacker, and the child nodes show the steps or sub-goals required to achieve it. These trees are instrumental in both the identification of system threats and the generation of test cases that simulate potential attack paths. Despite its importance, security test case generation often consumes significant time and resources, potentially overlooking critical details in complex systems. As a result, automation in this process has emerged as a critical area of research to ensure comprehensive coverage of potential attack vectors. This research addresses the challenge of automating security test case generation from attack trees using Large Language Models (LLMs) and a novel Retrieval-Augmented Generation (RAG) framework [2]. LLMs offer promising opportunities for automating security test generation, as demonstrated by recent research in generating test programs for compiler bug isolation [3]. However, they often face challenges in generating reliable and accurate test scripts, particularly in areas like security evaluation [4, 5]. To address these limitations, we propose Security Test Automation Framework (STAF), a novel solution using a custom multi-step self-corrective RAG framework specifically designed for automotive security test case generation from attack trees. By combining LLMs' code generation capabilities with this framework, we enable the automated generation of comprehensive and executable security test in Python as well as model checking properties in Linear Temporal Logic (LTL). Unlike traditional methods that focus on either attack tree analysis [6, 7] or test generation [8, 9, 10] in isolation, our approach integrates both aspects. This integration provides a complete workflow from threat identification to security testing, representing a significant advancement in the field. We also show the necessary adjustments that are needed to enable general-purpose LLMs to generate sensible, tailored automotive test cases. The key contributions of our work are:

- A method for generating security test cases from attack trees.
- An approach to create sensible executable tests for automotive systems.

- Integration with an existing system analysis tool for practical demonstration.
- Derivation of linear temporal logic (LTL) properties for model checking.

The remainder of this paper is organized as follows: Section 14.2 reviews related work in the field of automated security testing; Section 14.3 describes our proposed Security Test Automation Framework (STAF) in detail; Section 14.4 present the experimental setup and results; Section 14.5 provides a real-world scenario-based case study, and Section 14.6 concludes the paper.

14.2 Related Work

This section reviews and synthesizes recent research on attack tree analysis, security-focused test case generation, and the application of Large Language Models (LLMs) in software testing and evaluation. We identify key advances and limitations in these areas, highlighting the gaps that our work addresses.

14.2.1 Advancements in Automotive Attack Tree Analysis

Modern vehicles are increasingly characterized by their complexity as cyber-physical systems, incorporating numerous electronic control units (ECUs), diverse communication protocols (e.g., CAN, Ethernet [11, 12]), and increasing connectivity, all of which expand the attack surface [11, 13, 14]. To help manage these growing cybersecurity risks, standards like ISO/SAE 21434 and UN Regulation No. 155 emphasize the importance of structured threat modeling. As part of this process, Threat Analysis and Risk Assessment (TARA) plays a key role, with attack trees often used to map out potential paths an attacker might take [13, 14]. Despite their utility, the manual construction and maintenance of attack trees remain labor-intensive and susceptible to human error, particularly given the scale and dynamic nature of modern vehicle architectures [11, 14]. This has driven the development of tools that automate attack path identification using threat anti-patterns and system model analysis [14]. Such tools support ISO/SAE 21434-compliant TARA processes by generating attack graphs and trees that help identify threats early in the design phase. Although progress has been made, a critical gap remains: the transformation of attack trees into executable security test cases is still largely manual. Bridging this gap is essential for validating system resilience in practice. Large Language Models (LLMs) have shown promise in parsing structured data [15], but

their application to generating test cases from attack trees (especially in the automotive domain) remains underexplored. Our proposed framework, STAF, addresses this challenge by automating the generation of security test cases from attack trees, enabling scalable and context-aware security validation for automotive systems.

14.2.2 Works addressing the problem end-to-end

Recent research has explored the use of attack trees for security analysis in automotive systems. Umezawa et al. [16] applied attack trees to systems like the Tesla Model S, integrating threat databases to map attack vectors. Mishina et al. [17] combined fault tree and attack tree analysis to enhance security assessment. Cheah et al. [18] formalized systematic security evaluations using attack trees for automotive applications, while dos Santos et al. [19] proposed a formal model to facilitate security testing in modern automotive systems. While these studies advance attack tree modeling and threat analysis, they primarily focus on static analysis or threat documentation without automating test case generation; especially for general software systems. This highlights a critical gap; the lack of automated, adaptable security test case generation from attack trees for broader software applications.

14.2.3 LLMs in Test Case Generation

The application of large language models (LLMs) in software testing, particularly for test case generation, has garnered increasing attention in recent years. However, their use in security testing remains relatively underexplored. Plein et al. [20] investigated the use of LLMs, specifically ChatGPT, to generate test cases from bug reports. Their findings indicated that LLMs could produce executable test cases for approximately half of the bugs in their dataset. Nonetheless, the study also highlighted key limitations, including difficulties in achieving completeness and maintaining contextual accuracy, particularly for complex or domain-specific issues. Yu et al. [21] examined LLM-driven test script generation for mobile applications, identifying challenges related to cross-platform compatibility and the handling of platform-specific features. These findings underscore the need for more sophisticated approaches when generating test cases for security-critical systems, where interactions are often complex and context-dependent. Wang et al. [22] conducted a broader evaluation of LLMs across various software testing tasks. While their results demonstrated the potential of LLMs in generating unit tests and identifying bugs, they

also revealed persistent challenges. Specifically, LLMs struggled to produce comprehensive test suites and to manage the intricacies of security-sensitive scenarios. Although progress has been made in attack tree analysis and LLM-based test generation, several critical gaps remain. Existing approaches often lack the adaptability and depth required to address rapidly evolving security threats. The potential of LLMs to parse and analyze attack trees for security testing purposes has yet to be fully realized. Moreover, current methods frequently fall short in generating complete and context-aware test cases, particularly in domains with stringent safety and security requirements. These limitations are especially pronounced in the automotive sector, where embedded systems, real-time constraints, and regulatory compliance introduce additional complexity. To address these challenges, we propose the Security Test Automation Framework (STAF), which leverages LLMs within a self-corrective retrieval-augmented generation pipeline. STAF automates the generation of executable security test cases from attack trees, offering a scalable and domain-adapted solution for security validation in automotive systems.

14.3 Method

The goal of STAF is to create test cases from threat models. Our used threat modeling tool, AVL ThreatGuard[1], is capable of creating attack trees from TARAs. We use these attack trees as an input for STAF to create test cases in the form of executable Python scripts from that TARA. As an alternative, we use queries that create LTL properties to check implementation models in a model checker [23]. STAF then streamlines the process of generating security test cases directly from attack trees. For that, we build a RAG architecture. RAGs usually consist (apart from orchestration and user interface) of a source module, a retriever, a generator, and an evaluator/re-ranker [24]. For the source, we provide the LLM with a closed-loop knowledge base and, alternatively, a web search. The closed-loop information contains specific automotive cybersecurity knowledge and consists of a vectorized database containing a variety of specific automotive cybersecurity sources, particularly the Automotive ISAC Automotive Threat Matrix (ATM)[2], and the test libraries included in our automotive testing platform, AVL TestGuard[3]. To further improve the quality of the scripts, we include behavioral models (particularly Mealy machines)

[1] https://experience.avl.com/products/avl-threatguard
[2] https://atm.automotiveisac.com/
[3] https://experience.avl.com/products/avl-testguard

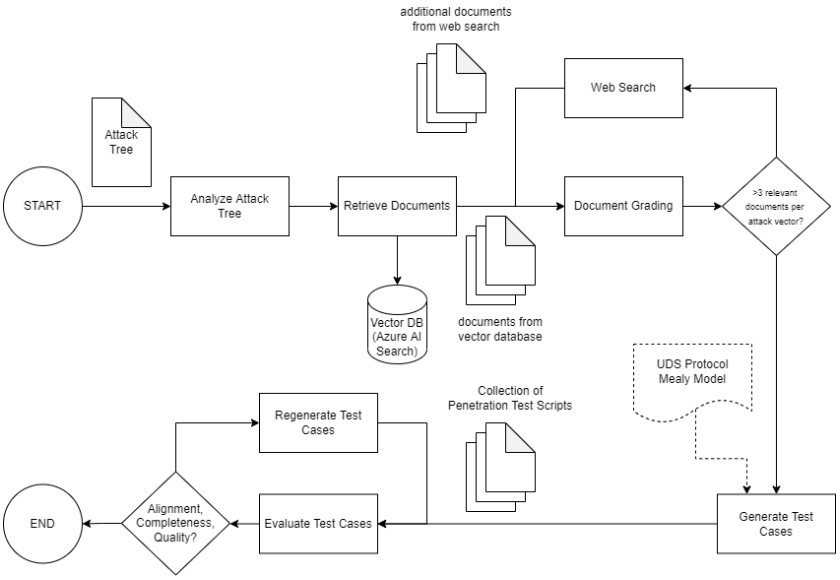

Figure 14.1: Workflow of STAF's self-corrective information retrieval mechanism. This process ensures the relevance and timely updates of the knowledge base by combining vector data store retrieval with web queries when necessary, enhancing the accuracy of generated security test cases. If applicable protocol Mealy models in DOT format are provided in the initial test generation prompt.

of tested protocols that improves the LLM's context knowledge of the protocol (see Section 14.5 for an example). We automatically inferred the models using automata learning [25]. We then implement a customized retriever that also contains an iterative grading loop. The grading consists of customized input prompts providing guidance for the LLM to evaluate the relevance of the retrieved documents. If the number of relevant documents exceeds a defined threshold, they serve as an input for the generator. The generator is another customized prompt that uses the retrieved inputs (and, if applicable, Mealy models) to generate actual test scripts. These test scripts will be evaluated and regenerated, by another prompt. The test generation process can be split in four interconnected stages: 1) Attack-tree Analysis, 2) Adaptive Information Retrieval, 3) Test-case Generation, and 4) Iterative Refinement (see Figure 14.1). Each stage contributes to the overall goal of translating complex attack tree

data into actionable security test cases. The process begins with LLM-guided attack tree analysis to extract threats. The framework then employs adaptive information retrieval to gather relevant knowledge, which informs the generation of structured test cases. Finally, an iterative refinement process ensures the quality and relevance of the generated test cases. We will discuss individual components in sections below.

14.3.1 Analyze Threats

The first stage of the framework involves analyzing attack trees encoded in a structured JSON format to interpret relationships among threats, attack vectors, and system weaknesses. STAF employs an LLM to extract insights from the attack tree through a carefully structured prompting strategy. This strategy guides the LLM in identifying individual threats, including the affected components and potential impacts, and extracting associated attributes such as preconditions, required access levels, and exploit complexity. For example, in the case of an attack tree node representing a threat in the Unified Diagnostic Services (UDS) protocol, the LLM would extract details about UDS services, sessions, seed-key, etc. This detailed analysis not only allows the framework to understand individual threats but also reveals their inter-dependencies and potential sequences of compromise.

14.3.2 Self-corrective Information Retrieval

The adaptive information retrieval component, illustrated in Figure 14.1, ensures that the system maintains a current and contextually relevant knowledge base for analyzing identified threats. Initially, the system formulates a query using keywords and threat attributes extracted during attack tree analysis. This query is used to perform a semantic search within a vectorized data store, which contains preprocessed documents represented as vectors. The semantic capabilities of the data store enable contextual matching, allowing the system to retrieve documents that are relevant beyond simple keyword overlap. To enhance the automotive relevance of the retrieval process, the vector store is enriched with domain-specific sources. These include the AUTO-ISAC Automotive Threat Matrix, which provides structured threat intelligence tailored to vehicle systems; a proprietary test case database from AVL's internal testing platform, which offers real-world examples of security validation procedures; and automata models of ECU Unified Diagnostic Services (UDS) stacks, which represent the behavioral logic of diagnostic protocols in modern vehicles. These

resources collectively improve the system's ability to retrieve and reason over automotive-specific threats and test scenarios. Once candidate documents are retrieved, an LLM-based grading mechanism evaluates their relevance based on contextual alignment, technical depth, and applicability to the identified threats. If the retrieved documents are insufficient or misaligned, the system initiates a self-corrective feedback loop. This involves refining the query and performing a web search using the Tavily API [4], aiming for targeted, use-case relevant sources such as vendor advisories, automotive security bulletins, and technical documentation. The results from both the vector store and web search are then integrated to form a comprehensive and up-to-date knowledge base for the subsequent test case generation phase.

14.3.3 Generate Test Cases

Building upon the accumulated knowledge from the previous stages, STAF generates structured test cases in JSON format. We employ domain adaptation through the strategic use of threat information and in-context learning capabilities of LLMs. This is done by structuring the prompt. This prompt includes threat analysis and the retrieved documents to ensure LLMs understand the underlying context. Additionally, the prompt has instructions to ensure LLMs include essential elements such as a descriptive title, an overview of the test scenario, setup instructions for preparing the environment, executable test scripts, tear-down procedures to restore the system to its original state, and expected outcomes indicating successful test results. For example, a test for authentication mechanism, would try to perform an action on usually protected resources without authentication. A success would indicate a vulnerability.

14.3.4 Chain of Improvement

For iterative refinement we use an LLM-as-a-judge approach [26]. This stage ensures that the generated test cases align with the original attack tree and conform to established security testing standards. The framework employs an LLM with specialized instructions to evaluate each test case across various metrics, including alignment with the attack tree, completeness of the test case components, runnability without additional modifications, and overall quality, clarity, and effectiveness. It also identifies any threats that are inadequately addressed and provides specific recommendations for refining the test cases. Lastly, it gives improvement suggestions to regarding error-free, runnable and

[4] https://tavily.com/

sensible code for regeneration. If a test case does not meet the quality benchmarks, the framework adjusts or regenerates the test case based on the suggested improvements and then re-evaluates it against the established criteria. This cycle continues until the test case achieves satisfactory scores across all metrics or a predefined number of iterations is reached. By incorporating this dynamic refinement process, the framework ensures high levels of precision and coverage in addressing potential vulnerabilities, thereby adding a layer of quality assurance to the testing process.

14.4 Evaluation

The evaluation of our proposed Security Test Automation Framework (STAF) is crucial to demonstrate its effectiveness in automating the generation of high-quality security test cases from attack trees. By conducting comprehensive tests, our aim is to show how STAF enhances the performance of the model in terms of alignment with identified threats, runnability of the generated code, and completeness of test cases.

14.4.1 Evaluation Criteria

Apart from the number of tests (with and without redundancies), we manually assessed each test case based on three key metrics, each evaluated on a scale of 0 to 10. Below is a detailed overview of the judgment criteria:

Alignment (0-10 points) : This metric measures the degree to which the generated test cases align with the identified threats present in the attack tree. A rating of 0 points is given if the test case does not address a threat present in the attack tree; we only want to test what is present in the attack tree. Up to 5 points are awarded for quality and specificity of the test case to address the target threat. Up to 3 points are awarded if the test case addresses advanced or subtle aspects of the threat, reflecting a higher level of depth and nuance. Showing deeper knowledge about the protocols/systems available. Up to 2 points are awarded for the variety of testing a threat, e.g., by trying different inputs or approaches to test the threat.

Runnability (0-10 points) : This criterion assesses the practical executability of the generated test code. A rating of 0 points is given if the test is not runnable for whatever reason. A deduction of 2 points for every placeholder

Model	#Tests	#Unique	Alignm.	Runnab.	Completen.	*Overall*
GPT (vanilla)	9	9	7.00	9.00	5.50	7.17
GPT (STAF)	60	22	**9.80**	9.00	8.50	**9.11**
GPT (STAF&MM)	65	27	9.00	**9.67**	7.33	8.67
DeepSeek (vanilla)	5	5	6.50	4.17	4.67	5.11
DeepSeek (STAF)	21	18	8.83	0.00	7.67	5.50
DeepSeek (STAF&MM)	30	14	9.33	0.00	**9.00**	6.11

Table 14.1: This table presents a detailed comparison of STAF's and STAF&MM's performance using GPT (4.1) and DeepSeek (V3) against their pure versions. For each category three scripts were analyzed.

or implicit assumption that will lead to a silent failing of tests (e.g., usage of unconfirmed CAN IDs). A deduction of 2 points for every case where the test might fail based on conditions (e.g. if the branch would execute successfully but the other branch would fail).

Completeness (0-10 points) : This metric evaluates how thoroughly the generated test cases cover all facets of the identified threats. Up to 4 points are awarded for including all the necessary external files, data, or resources required for the test cases. Up to 3 points can be earned by providing comprehensive setup and tear-down procedures, ensuring that the test environment is correctly initialized and cleaned up. An additional 3 points are given for well-documented test cases that clearly specify their purpose, procedures, and expected outcomes, thereby facilitating reproducibility and understanding.

An overall score is calculated as the arithmetic mean of these three metrics, providing a single comprehensive measure of the model's performance in generating security test cases. The LLM evaluator provides a detailed breakdown of how each score was calculated, along with a brief explanation of the evaluation. This approach ensures a transparent and consistent evaluation process across different models and methodologies.

14.4.2 Results

We selected two recent state-of-the-art models, GPT-4.1 and DeepSeek-V3-0324, as the backbone for STAF and STAF incl. Mealy Models (STAF&MM). As shown in Table 14.1, we compared their individual performance with their vanilla versions to highlight the enhancements achieved through STAF and STAF&MM. The results indicate that integrating STAF leads to significant improvements across all metrics for both models. The most obvious difference

is the rise in the number of generated tests (#Tests) and tests after removing redundant ones (#Unique). It gradually rose with STAF and MM introduction, with the exception that with deepseek, STAF alone generates more uniques than with MMs. The addition of learned protocol models as additional context for the models also increases the alignment and completeness of the generated test cases. This effect is more observable for DeepSeek-V3, while GPT-4.1 seems to inherently better understand certain protocols such as UDS out of the box. For GPT-4.1, the overall score increased from 7.17 to 9.11 upon integrating STAF, reflecting an improvement of **1.94 points**. The most notable enhancement is in the *Alignment* metric, which rose from 7.00 to 9.80, indicating that STAF effectively helps the model to generate more relevant test cases and supports addressing the threats in the attack trees more effectively. The *Completeness* score also improved from 5.50 to 8.50, demonstrating more comprehensive test case generation. The *Runnability* score stayed at 9.00 for both, the overall improvement suggests that there are considerable benefits in appyling the STAF approach. Similarly, DeepSeek-V3-0324 experienced an overall score increase from 5.11 to 5.50 (STAF), and 6.11 (STAF&MM), marking an enhancement of up to **1.0 points**. The *Completeness* metric showed a substantial rise of up to **3.00 points**, from 4.67 to 7.67 and respective 9.00 if the protocol model was applied. The *Alignment* score also saw an increase from 6.50 to 8.83 (STAF) and 9.33 (STAF&MM). These gains highlight STAF's ability to considerably supplement the models performance for this task. The *Runnability* was evaluated as 0.0 for half of the test cases generated by DeepSeek-V3 vanilla and all of the test cases generated with DeepSeek-V3 STAF and DeepSeek-V3 STAF&MM, since the model would always include natural language placeholders and comments within the Python scripts, which yielded them unusable without modification. In terms of individual metrics:

- **Alignment**: GPT-4.1 (STAF) achieved the highest score of 9.80, indicating excellent adherence to the identified threats. DeepSeek-V3 (STAF&MM) also performed well with a score of 9.33. GPT-4.1 (STAF&MM) placed third with a score of 9.00.
- **Runnability**: GPT-4.1 (STAF&MM) achieved the highest runnability score of 9.67, indicating that its generated test cases are highly executable without additional modifications. GPT-4.1 (STAF) and GPT-4.1 vanilla performed also well with a score of 9.00.
- **Completeness**: The STAF and STAF&MM models significantly out-

performed the vanilla models in completeness. This suggests that the STAF framework enables the models to generate more comprehensive test cases that cover all critical aspects of the threats.

These results demonstrate that STAF and STAF&MM outperform even GPT-4.1, in crafting effective test cases. This outcome demonstrates the importance of a structured approach in security testing. STAF's well-defined framework and targeted protocols ensure the retrieval and application of highly relevant data. This focus enables STAF to generate more precise, context-specific, and robust test cases. The framework's ability to filter, process, and apply domain-specific knowledge effectively translates into superior test case quality, even when compared to models with wider information access. The addition of the protocol Mealy models provides accurate knowledge of the communication rules, increasing the quality of interaction with a system under test. This finding emphasizes that in specialized tasks like security test case generation, a tailored approach with domain-specific protocols can yield better results than general-purpose models with broader but less focused information sources. Qualitatively, we saw not only improvements in the scoring itself, but also the appearance of certain traits – for instance, the introduction of a UDS mealy model induced the LLM to use session changes in test cases, which did not appear without the models. Besides, all versions considered a comparable amount of infeasible tests (e.g., physical bypass[5]), written in a Python pseudo code. We have seen similar results for LTL properties (see Section 14.5), but the refinement process in the RAG was more extensive.

As an example for the improvements possible by using GPT 4.1 STAF&MM versus GPT-4.1 (vanilla) consider the following code listings 14.1 and 14.2. While GPT-4.1 (vanilla) without a learned protocol model of the devices UDS was only able to create a test by sending a too short message, the 14.2 shows specific, targeted test content utilizing undocumented sub-functions.

```
1 import can
2 import time
3
4 can_interface = 'can0'
5 bus = can.interface.Bus(channel=can_interface, bustype='socketcan
      ')
6
7 # Malformed UDS message: too short (only 1 byte)
8 malformed_uds_data = bytes([0x10])
```

[5]This test involves physical tampering with the system. This is not feasible to test in a pure digital manner. We therefore exclude this from test sets, since the goal is to perform automated cybersecurity testing, which intrinsically does not involve physical attacks.

```

# test code
"""
Purpose: Send a malformed UDS message and observe ECU behavior for secure error handling.
Attack Steps:
 1. Send the malformed UDS message onto the CAN bus.
 2. Listen for a response or observe for any abnormal behavior (e.g., ECU crash, diagnostic session reset).
Expected Secure Behavior: ECU should respond with a negative response (e.g., 0x7F) or ignore the message, without crashing or leaking information.
"""
tx_msg = can.Message(arbitration_id=0x7DF, data=malformed_uds_data, is_extended_id=False)
bus.send(tx_msg)

response = None
start_time = time.time()
while time.time() - start_time < 5:
    msg = bus.recv(timeout=1)
    if msg and msg.arbitration_id == 0x7E8:
        response = msg
        break
if response:
    if response.data[0] == 0x7F:
        print('[PASS] ECU responded with negative response to malformed message.')
    else:
        print('[WARNING] ECU responded with unexpected data: {}'.format(response.data.hex()))
else:
    print('[PASS] No response to malformed message. ECU may be securely ignoring invalid input.')

# Teardown: Close the CAN bus interface
bus.shutdown()
```

Listing 14.1: GPT-4.1 (vanilla) - Test error handling of Malformed UDS Messages

```
# [...] definitions and imports skipped

if __name__ == "__main__":
    CAN_IFACE = 'can0'
    TX_ID = 0x7E0
    RX_ID = 0x7E8
    try:
```

```
        sock = ISOTPNativeSocket(CAN_IFACE, rx_id=RX_ID, tx_id=
    TX_ID, basecls=UDS, padding=True)
    except Exception as e:
        print(f"[ERROR] Failed to open ISOTP socket: {e}")
        assert False, "ISOTP socket unavailable."
    # Rare/undocumented subfunctions
    rare_subfunctions = [0x7F, 0x99, 0xFE]
    for sub in rare_subfunctions:
        send_multiframe(sock, 0x10, sub, payload_len=32)
    # Malformed multi-frame (fragmentation)
    send_multiframe(sock, 0x2E, 0x01, payload_len=64)
    # Check ECU is still alive
    try:
        resp = sock.sr1(UDS()/UDS_TP(), timeout=2, verbose=False)
        assert resp is not None, "ECU did not respond after
    protocol-specific attack (possible crash)!"
        print("[PASS] ECU is still operational after advanced
    protocol-specific attack.")
    except Exception as e:
        print(f"[ERROR] Scapy operation error during post-attack
    check: {e}")
        assert False, "ECU not responsive after protocol-specific
     attack!"
```

Listing 14.2: GPT-4.1 (STAF&MM) - Protocol Specific Attacks - UDS Subfunctions and Multi-Frame Fragmentation

14.5 Case Study

We conducted a case study using STAF with an attack tree Battery Management System (BMS). The purpose of this case study is to evaluate how well STAF can automatically generate security test cases derived from attack trees. By leveraging a realistic application, we provide a practical demonstration of the framework's potential and its current limitations when tested in a real-world scenario.

14.5.1 Setup

We used the TARA of a Battery Management System (BMS) as a basis. Figure 14.2 shows the architectural layout of the BMS' components as designed in a threat model. With the help of the threat modeling tool (see Section 14.3), we analyze the system and create several attack trees. For this case study, we particularly use an attack tree that targets *Man-in-the-Middle Attack via UDS Mes-*

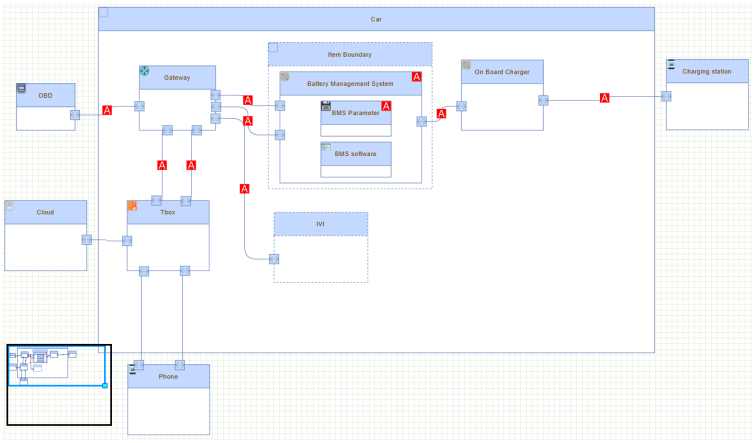

Figure 14.2: Architecture of the Battery Management System used as system under test.

sage Collection, which has the following attack vectors: *Intercept UDS Communication* (subvector *Exploit Unencrypted Communication Channel*), *Inject Malicious UDS Messages* (subvector *Replay or Modify Captured Messages*, and *Bypass Physical and Logical Protections* (subvector *Exploit Lack of Encryption and Trust*). These are described with attack steps for message replay: 1. *Use previously captured UDS messages as a template*, 2. *Modify message parameters to target specific ECUs or functions*, 3. *Inject modified messages into the communication channel*, and 4. *Observe vehicle response to determine success of injection*. This attack tree provides the main input to STAF.

14.5.2 Walk Through

STAF is based on LangChain and is running on a server behind a FastAPI application. The process of generation is triggered by sending an attack tree in JSON format to the server. The Attack-tree Analysis (*Analyze Attack Tree* step in Figure 14.1) is always conducted and results in keywords used for the Adaptive Information Retrieval (*Retrieve Documents* step), which will fetch the top ten relevant documents in the vector database. Here, during ingestion of documents in the vector database a verbal description of the document content was generated, for example existing Python test scripts for different attacks and automotive protocols. The Adaptive Information Retrieval will not only grade

the documents using a binary metric (*Document Grading* step), checking if the document is relevant for addressing the threat, but also conduct a web-search if less than three documents were rated positive (*Web Search* step). Afterwards, the attack tree is searched for protocol names such as UDS. If a protocol name appears in the attack tree, the Mealy model of the protocol standard is also provided in DOT language format. For STAF&MM, the protocol model could be added, for STAF standalone, the protocol model support was deactivated. Subsequently, the *Generate Test Cases* step is conducted and the LLM will receive a comprehensive prompts with instructions, the documents retrieved, if applicable the protocol model as well as formatting instructions for the result. As result, a list of test cases is provided (an example is shown in Listing 14.3), including metadata like a unique ID, the threat addressed, necessary implementations for setup and tear-down of the test environment as well as the test implementation itself.

```
"test_cases": [
   {
      "id": 1,
      "name": "Bypass Physical and Logical Protections - Unauthorized Diagnostic Port Access",
      "description": "Tests whether an attacker with physical access can connect to the diagnostic port and initiate UDS communication without authentication.",
      "threat_addressed": "Lack of Authentication on Diagnostic Interfaces",
      "setup": "# Setup instructions: ...",
      "test_code": "import can\\nimport os\\n ...",
      "teardown": "# Teardown: ...",
      "expected_result": "If the system is vulnerable, the ECU will  ..."
   },
   ...
]
```

Listing 14.3: Example of a Test Case.

The full collection of test scripts is then provided to an LLM for review (in the *Evaluate Test Cases*)step, where the tests are checked against the attack tree for alignment and completeness and the implementation for runnability, using a dedicated prompt. This prompt requests a scoring of 0-100 in each category and a list of missing threats, as well as a list of improvement suggestions where necessary. Listing 14.4 shows an example for improvement suggestions. If the

scoring is below a certain, configured threshold, the suggestions are improved by the generator using these suggestions (*Regenerate Test Cases* step) in a loop until the quality is satisfactory.

```
{
  "test_case_name": "Test Case 10: UDS Message Injection
    Test",
      "details": [
        "Add more detailed checks for message
    authentication mechanisms",
        "Include verification of physical access
    requirements"
      ]
},[...]
```

Listing 14.4: Example improvement suggestions.

Now up to four cycles of Iterative Refinements are conducted. The refinement is stopped if either four cycles were done, or the rating increases to above 90 for all categories. In each regeneration step, the original test scripts, the missing threats and the suggestions for improvement are provided to the LLM. This results in a continuous extension and quality improvement of the generated test cases. Listing 14.5 the prompt template for the Iterative Refinement. The same process for LTL properties yielded for example a property like *AUTHENTICATION ENFORCEMENT:*$\square(DIAG\_SESSION\_INIT \wedge UNAUTH \rightarrow false$.

```
prompt = """
    You are an elite security test engineer [..]. Your critical
    task is to **modify specific test cases** based on the
    provided improvement suggestions, and **add new test cases**
    for any missing vulnerabilites.
    ### Test Cases to Modify: {test_cases_to_modify}
    ### Improvement Suggestions: {improvements_map}
    ### Missing Vulnerabilites: {missing_vulnerabilites}
    ### Instructions:
    1. **Modify the test cases listed above** to incorporate [..]
    2. For each **missing vulnerability**, **create a new test
    case** that exactly addresses the vulnerability.
    3. Ensure that all test cases use appropriate and actual code
     relevant to the system under test, [..].
    4. Include all necessary **setup**, including required
    imports [..]
    5. The test code must be **complete, runnable Python code**.
     [..]
```

```
12      6. Follow **best practices** for the system or domain you are
         testing, and use appropriate methods and calls.
13      7. Each test case should demonstrate both the **vulnerable
        state and the secure state**.
14      8. Use **assert statements** to clearly indicate what
        constitutes a pass or fail condition.
15      {format_instructions}
16      """
```

Listing 14.5: Prompt template for iterative refinement.

14.6 Conclusion

In this paper, we introduced and evaluated STAF, a Security Test Automation Framework that automates the generation of security test cases from attack trees using LLM-guided analysis, adaptive retrieval, and iterative refinement. STAF reduces manual effort by transforming attack vectors into actionable, structured test cases. Through a combination of a robust knowledge base, adaptive retrieval, and self-corrective mechanisms, the framework generates comprehensive test cases tailored to specific threats in the attack trees for automotive systems. We could demonstrate that a specialized RAG architecture like STAF could help to overcome the hurdles of making LLM-based test generation usable and scalable for automotive systems and produce usable Python test code, as well as LTL properties for model checkers. It shows that attack trees can be used to structure test suites, while the RAG's context information helps to generate meaningful and executable test cases. The introduction of behavior models (Mealy machines) produced yet more and more aligned test cases.

14.7 Limitations & Future Work

While STAF shows promising results, a couple limitations surfaced during its evaluation. Particularly, one significant limitation is the lack of implementation details in an attack tree. Therefore, specific basic information (e.g., CAN baud rates or arbitration IDs) should be included. This results in test cases which are not immediately executable but require manual modification, hence STAF is not yet fully autonomous. Further, the framework requires multiple iterations to ensure that the test cases meet quality targets and align with the original attack tree. Especially for more complex applications this is resource-intensive.

This hinders the scalability of STAF when applied in dynamic or large-scale settings where speed is crucial. Overall, while STAF enhances the automation of security test generation, it still requires manual adjustments and human oversight to handle certain limitations. Additionally, improving the framework's ability to handle complex, large-scale applications efficiently, while maintaining up-to-date knowledge retrieval and minimizing manual interventions, will be critical areas for future development. Future work will include the output of test cases in Domain Specific Language (DSL), to allow injecting implementation details into generated tests as post-processing. Further, integration into a testing framework with a feedback loop to refine test cases will help to increase both the test quality and degree of automation – error messages and tool outputs will provide valuable feedback for the test case evaluator to generate more practial-oriented feedback and better test integration. The framework could also be improved with fine-tuning: test sets and script code can be separated, which could provide more accurate feedback and smaller context windows.

Acknowledgment

We want to acknowledge the help of our colleague Thomas Grandits for pointing out future directions. Parts of this work are funded by the European Union under the Horizon Europe program (grant agreement 101194245, project Shift2SDV) as well by the Natural Sciences and Engineering Research Council of Canada (NSERC). Other parts are funded in the frame of the Important Project of Common European Interest (IPCEI) on Microelectronics and Communication Technologies (ME/CT). We further acknowledge the support of the Swedish Knowledge Foundation via the industrial doctoral school RELIANT under grant No. 20220130.

Bibliography

[1] S. Mauw and M. Oostdijk, "Foundations of Attack Trees," in *Information Security and Cryptology - ICISC 2005* (D. H. Won and S. Kim, eds.), (Berlin, Heidelberg), pp. 186–198, Springer Berlin Heidelberg, 2006.

[2] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, (Red Hook, NY, USA), Curran Associates Inc., 2020.

[3] H. Tu, Z. Zhou, H. Jiang, I. N. B. Yusuf, Y. Li, and L. Jiang, "LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation," *ArXiv*, vol. abs/2307.00593, 2023.

[4] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 06, pp. 1340–1359, 2024.

[5] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software Vulnerability Detection using Large Language Models," in *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 112–119, Oct 2023.

[6] R. Kumar, S. Schivo, E. Ruijters, B. M. Yildiz, D. Huistra, J. Brandt, A. Rensink, and M. Stoelinga, "Effective analysis of attack trees: A model-driven approach," in *Fundamental Approaches to Software Engineering: 21st International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*

2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings 21, pp. 56–73, Springer International Publishing, 2018.

[7] G. Falco, A. Viswanathan, and A. Santangelo, "CubeSat Security Attack Tree Analysis," in *2021 IEEE 8th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pp. 68–76, 2021.

[8] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3Test: Assertion-Augmented Automated Test case generation," *Information and Software Technology*, vol. 176, p. 107565, 2024.

[9] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM," *Proc. ACM Softw. Eng.*, vol. 1, July 2024.

[10] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated Unit Test Improvement using Large Language Models at Meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, (New York, NY, USA), p. 185–196, Association for Computing Machinery, 2024.

[11] D. Ward and P. Wooderson, "Automotive Cybersecurity: An Introduction to ISO/SAE 21434," in *Automotive Cybersecurity: An Introduction to ISO/SAE 21434*, pp. i–xii, SAE International, 2021.

[12] L. Zhang, *Intrusion Detection Systems to Secure In-Vehicle Networks*. PhD thesis, University of Michigan-Dearborn, Dearborn, Michigan, USA, 2023.

[13] A. Lautenbach, M. Almgren, and T. Olovsson, "Proposing HEAVENS 2.0 – an automotive risk assessment model," in *CSCS '21, pages = 1–12*, ACM, 2021.

[14] S. Chlup, K. Christl, C. Schmittner, A. M. Shaaban, S. Schauer, and M. Latzenhofer, "THREATGET: Towards Automated Attack Tree Analysis for Automotive Cybersecurity," *Information*, vol. 14, no. 1, 2023.

[15] U. A. Khan, "LLM-powered parsing and analysis of semi-structured & Structured Documents," Aug 2024.

[16] K. Umezawa, Y. Mishina, and K. Takaragi, "Threat analyses using vulnerability databases —Possibility of utilizing past analysis results," *2019 IEEE International Symposium on Technologies for Homeland Security (HST)*, pp. 1–6, 2019.

[17] Y. Mishina, K. Takaragi, and K. Umezawa, "A Method of Threat Analysis for Cyber-Physical System using Vulnerability Databases," *2018 IEEE International Symposium on Technologies for Homeland Security (HST)*, pp. 1–7, 2018.

[18] M. Cheah, H. N. Nguyen, J. Bryans, and S. A. Shaikh, "Formalising Systematic Security Evaluations Using Attack Trees for Automotive Applications," in *Information Security Theory and Practice* (G. P. Hancke and E. Damiani, eds.), (Cham), pp. 113–129, Springer International Publishing, 2018.

[19] E. dos Santos, A. C. Simpson, and D. Schoop, "A Formal Model to Facilitate Security Testing in Modern Automotive Systems," in *Proceedings of the Joint Workshop on Handling IMPlicit and EXplicit knowledge in formal system development (IMPEX) and Formal and Model-Driven Techniques for Developing Trustworthy Systems (FM&MDD)* , 2018.

[20] L. Plein, W. C. Ou'edraogo, J. Klein, and T. F. Bissyand'e, "Automatic Generation of Test Cases based on Bug Reports: a Feasibility Study with Large Language Models," *ArXiv*, vol. abs/2310.06320, 2023.

[21] S. Yu, C. Fang, Y. Ling, C. Wu, and Z. Chen, "LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities," *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pp. 206–217, 2023.

[22] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing with Large Language Models: Survey, Landscape, and Vision," 2024.

[23] S. Marksteiner, M. Sirjani, and M. Sjödin, "Automated Passport Control: Mining and Checking Models of Machine Readable Travel Documents," in *ARES'24*, ARES '24, (New York, NY, USA), pp. 1–8, Association for Computing Machinery, July 2024.

[24] R. Shan and T. Shan, "Retrieval-Augmented Generation Architecture Framework: Harnessing the Power of RAG," in *Cognitive Computing* -

ICCC 2024 (R. Xu, H. Chen, Y. Wu, and L.-J. Zhang, eds.), (Cham), pp. 88–104, Springer Nature, 2025.

[25] e. Ebrahimi, "A Systematic Approach to Automotive Security," in *Formal Methods*, vol. 14000 of *Lecture Notes in Computer Science*, (Cham), pp. 598–609, Springer International Publishing, 2023.

[26] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging LLM-as-a-judge with MT-bench and Chatbot Arena," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, (Red Hook, NY, USA), pp. 46595–46623, Curran Associates Inc., 2023.

Chapter 15

Paper IX: Learn, Check, Test – Security Testing Using Automata Learning and Model Checking

Stefan Marksteiner, Mikael Sjödin, Marjan Sirjani.

Computers & Security. Amsterdam: Elsevier, 2026. Paper submitted, not yet accepted. Preprint available on arXiv. DOI: 10.48550/arXiv.2509.XXXXX.

Abstract

Cyber-physical systems are part of industrial systems and critical infrastructure. Therefore, they should be examined in a comprehensive manner to verify their correctness and security. At the same time, the complexity of such systems demands such examinations to be systematic and, if possible, automated for efficiency and accuracy. A method that can be useful in this context is model checking. However, this requires a model that faithfully represents the behavior of the examined system. Obtaining such a model is not trivial, as many of these systems can be examined only in black box settings due to, e.g., long supply chains or secrecy. We therefore utilize active black box learning techniques to infer behavioral models in the form of Mealy machines of such systems and translate them into a form that can be evaluated using a model checker. To this end, we will investigate a cyber-physical systems as a black box using its external communication interface. We first annotate the model with propositions by mapping context information from the respective protocol to the model using *Context-based Proposition Maps (CPMs)*. We gain annotated Mealy machines that resemble Kripke structures. We then formally define a template, which we use to transfer the structures in to a format to be processed by a model checker. We further define generic security properties based on basic security requirements (authentication, confidentiality, privilege levels, and key validity). Due to the used CPMs, we can instantiate these properties with a meaningful context to check a specific protocol, which makes the approach flexible and scalable. Furthermore, the gained model can be easily altered to introduce non-deterministic behavior (like timeouts) or faults and examined if the properties still hold under these different conditions. Lastly, we demonstrate the versatility of the approach by providing case studies of very different systems (a passport and an automotive control unit), speaking different communication protocols (NFC and UDS), checked with the same tool chain and the same security properties.

15.1 Introduction

Cyber-physical systems in all domains of everyday life become more interconnected and complex. Various domains like automotive, production, but also critical infrastructures like electronic government systems naturally have to provide communication interfaces to provide more smart and usable functionality. It is therefore crucial to create methodologies to verify the correctness and security of these interfaces. Since those systems are mostly proprietary, verifying their security has become an increasingly laborious task that is afflicted with a high degree of uncertainty. Many approaches rely on static, predefined test suites that do not take any architectural or behavioral information into account. Therefore, validation of industrial systems and critical infrastructure often include the use of formal methods that provide more rigor in verification [1]. Such methods, including model checking, often come with the disadvantage of a high effort (especially with modeling systems) in conjunction with being hard to apply at black box systems. To mend these disadvantages, we try to automate the model creation using active automata learning. This technique allows for an automated model generation to be used in a checking process, and is also a technique that is destined for black box systems [2]. We therefore combine the static model checking with dynamic active learning. Furthermore, we want to be able to not only verify the systems, but also correct errors and simulate different behavior. We can also use this technique to create corner-cases for verification by altering the system behavior to a Maximum Credible Accident (MCA) scenario [3]. We therefore follow the approach to translate the learned models into a modeling language (see Sections 15.2.4 and 15.5.2).

15.1.1 Contribution

This paper combines formal methods, namely active automata learning of existing systems and model checking, to create a tool chain for checking cyber-physical systems (particularly communication protocols) for their correctness and security. Our main contributions are:

- An approach for annotating learned models (Mealy machines [4]) with propositions using *context-based proposition maps* (*CPMs* - see Section 15.5.1), effectively creating Kripke structures [5] (combined into a single model with the Mealy machine as a hybrid annotated Mealy machine),
- A formally defined and implemented template that allows for creating

Rebeca models [6] from annotated Mealy machines,

- Generalized properties to check security requirements along with a easily usable, scalable method for concretization using CPMs,
- Two case studies of communication protocols (NFC [7] and UDS [8]) demonstrating the applicability and versatility of the approach.

15.1.2 Approach

We utilize active automata learning via the *LearnLib* Java library to infer Mealy machine models of a respective System-under-learning (SUL). We therefore created dedicated protocol adapters and defined input alphabets (Section 15.4). Since this model does not intrinsically contain any properties, we annotate it with sensible propositions. For that, we define using *context-based proposition maps (CPMs)* that define when propositions become true or false (Section 15.5.1). We created a tool that uses these maps to annotate Mealy machines represented in the GraphViz format, receiving an combined annotated Mealy machine. We then convert this structure into Rebeca code and verify it to be faithful to the original model (Sections 15.5.2 and 15.5.6). Once this step has been performed, we utilize the Rebeca Model Checking tool, RMC [9] to check the models for properties defined in linear temporal logic (LTL). These properties are partly specific to the scrutinized protocol, but can also partly be generalized to check very diverse protocols for the same properties. Eventually, if a property violation is found, the trace can be concretized via the learner's interface and be used as a test case for verification on the live system. If this confirms the violation, that indicates an issue. Otherwise, the different behavior serves as a counterexample for the learner in a new iteration of the process. That way, we implemented all the phases for going through learning, checking, and testing real-world examples. We automated the complete process by combining (a) the *learner(s)* using LearnLib [10] (Section 15.4), (b) a tool for generating Kripke structures and translating it into a Rebeca model to be checked (Section 15.5) and (c) the Rebeca *model checking* tool [9], used with generic security properties (Section 15.6).

The remainder of this paper outlines as follows: Section 15.2 contains basics, while Section 15.3 outlines general security requirements to be checked. Section 15.4 outlines the learner interfaces, Section 15.5 the Rebeca code generation including a formally defined template and a formal description of the CPM concept. Section 15.6 the model checking, including property definition. Section 15.7 briefly describes the relation to testing. Section 15.8 presents the

case studies, Section 15.9 the related work and Section 15.10 concludes the paper.

15.2 Background

In this section, we briefly describe the basics of automata, automata learning, model checking, as well as the frameworks we used and protocols used in the case studies.

15.2.1 Automata and Kripke Structures

Automata, or state machines, are a fundamental concept of computer science [11, 12]. To describe cyber-physical systems, we use a certain type of automata, namely Mealy machines [4]. A Mealy machine $\mathcal{M}$ is defined as $\mathcal{M} = (Q, \Sigma, \Omega, \delta, \lambda, q_0)$, with Q being the set of states, Σ the input alphabet, Ω the output alphabet (that may or may not be identical to the input alphabet), δ the transition function ($\delta : Q \times \Sigma \to Q$), λ the output function ($\lambda : Q \times \Sigma \to \Omega$), and q_0 the initial state. A state machine can describe a system in a way that for any defined set of inputs (input word), it delivers the correct set of outputs (output word) [13]. While Mealy machines are widely used to model system behavior, Kripke structures [5] are basic structures used for model checking [14]. A Kripke structure is similar to a Mealy machine, as both can be seen as types of Labeled Transition Systems (LTS) [15, 16]. However, they differ in that instead of labeling the transitions, Kripke structures have their states labeled with *propositions*. Coming from propositional logic, these are logical attributes that can be true or false. Any state in which this attribute is true gains it as a proposition. Each state can have an element of the power set of all propositions. The formal definition of a Kripke structure ($\mathcal{K}$) is $\mathcal{K} = (S, S_0, R, \mathcal{L})$ with S being the set of states, $S_0 \subseteq S$ the set of input states, $R \subseteq S \times S$ a transition relation and $\mathcal{L} = S \to 2^{AP}$ the labeling function that attributes an element of the power set of atomic propositions (AP) to the states.

15.2.2 Automata Learning

Automata learning was developed as a method to learn regular languages [17]. A learner (i.e., an algorithm) is allowed to ask two kinds of questions: *membership queries* and *equivalence queries*. We then assume a teacher that knows

all of the rules that determine well-defined words of that language. Membership queries give examples of input words($\sigma \in \Sigma | W_\sigma = \langle \sigma_1, \sigma_2..\sigma_n \rangle$), where the teacher answers wether the example is part of the language. After a sufficient amount of queries, the learner infers a hypothesis model of an automaton describing the language (in this case a Deterministic Finite Acceptor – DFA). It may then submit an equivalence query, where it presents the hypothesis to the teacher. The teacher either answers that the hypothesis is correct or presents a counterexample. To fit to real-world cyber-physical systems, approach has been adapted to learn Mealy machines as well, alongside with more efficient algorithms [18]. In this case, membership queries yield corresponding output words ($\omega \in \Omega | W_\omega = \langle \omega_1, \omega_2..\omega_n \rangle$). One popular field to use automata learning is to learn and test black box systems. In this case there is no teacher available that knows the interna of the examined system (SUL). Therefore, membership queries are just passed on to the live system[1]. Equivalence queries are implemented by a sufficient amount of conformance tests [19]. If all are passed, the hypothesis is accepted of correct. Otherwise, a failed conformance test automatically poses a counterexample. In this work, we use Learnlib [10], a well proliferated Java library for automata learning. It provides a classes for interfacing with SULs, implementations of well-known algorithms (L*, Rivest-Schapire, AAAR, ADT, KV, DHC and TTT, as well as an addon for L# [20]), and conformance testing strategies (complete depth-bounded exploration, random words, random walk, W-method, Wp-method). Other libaries are AALpy [21] and Libalf [22]).

Interface and Abstraction Layer

To interact with real-world systems, we not only need the input alphabet (Σ), but also a means to communicate with the SUL. To keep the as universal as possible, we therefore define an intermediate layer that serves two purposes: provide an interface to the SUL (including necessary error correction, timeout-handling, etc.) and abstracting the input layer. The latter, called mapper, is necessary to reduce the input space to a size feasible to be exhaustively explored. The mapper is formally defined as a state machine $A = (\Sigma, \Omega, Q, q_0, \delta, X, Y, \nabla)$, with Σ the abstract input alphabet, Ω the abstract output alphabet, Q being the set of states, q_0 the initial state, δ the transition function ($\delta : Q \times \Sigma \to Q$), λ the output function ($\lambda : Q \times \Sigma \to \Omega$), X the concrete input alphabet, Y the concrete output alphabet, and ∇ the abstraction function

[1] We are talking about *active* automata learning here. It is also possible to take the answers from recorded data in *passive* automata learning.

($\nabla : \Sigma \cup \Omega \rightarrow X \cup Y$. Inversely, the same mapper can convert the received concrete output from the SUL back to abstract output to present the learner by replacing the abstraction function with its inverse $\nabla^{-1} : X \cup Y \rightarrow \Sigma \cup \Omega$. The latter must not necessarily be fully defined in all cases. For many protocols, the learner is able to handle concrete outputs and to learn Ω on-the-fly. For these cases a partial mapper with $A = (\Sigma, Q, q_0, \delta, X, \nabla_{in})$ is sufficient where $\nabla_{in} : \Sigma \rightarrow X$. However, for some concrete outputs ($Y\prime \subseteq Y$) the inverse mapper must still be defined ($\nabla_{out}^{-1} : Y\prime \rightarrow \Omega$), since some inputs provoke non-deterministic values in the sense that they would not yield the same output even with the exact equal input sequence, which a non-deterministic Mealy machine learner cannot handle. Examples of such outputs are timestamps, counters, and random numbers (such as nonces).

15.2.3 Linear Temporal Logic and Model Checking

Linear Temporal Logic (LTL) is an extension of propositional logic [23] that allows for expressing logical temporal[2] modalities [24]. That is, LTL combines atomic propositions (attributes of a state that can be true or false) formulas using logical operators and extends them with the following modalities:

- *always* ($\Box$, or G for Globally): the proposition must hold in any subsequent state
- *eventually* ($\Diamond$, or F for Finally): the proposition must hold in some (i.e., any arbitrary) subsequent state (may or may not hold before)
- *next* ($\bigcirc$, or X for neXt): the proposition must hold in the immediately subsequent state and
- *until* ($\mathcal{U}$ or U): the proposition A_1 must hold until another defined proposition A_2 occurs ($A_1 \mathcal{U} A_2$). A_1 may or may not hold *after* A_2 has occurred. In any case, A_2 has to occur at some point.

A model checker runs through all states of a model and checks if their properties hold [24]. The properties are often expressed in LTL or similar logics like Computation tree logic (CTL) [25] or branching-time logic (CTL*) [26]. In automata-theoretic model checking, a model checker ordinarily creates Büchi automata of both the scrutinized model and the negation of the LTL property to check, and examines if their cross product's accepted language is empty [19].

[2] In this context, *temporal* is not to be confused with *timed*. LTL only allows statements about the modality and succession order of events to occur, not about a duration of any kind.

15.2.4 Rebeca

Rebeca is a modeling language with model checking support specifically designed to model and model check cyber-physical (i.e., reactive) systems [6]. Rebeca resembles the Java programing language to ease its usage for engineers. It conceptually uses *rebecs* to model reactive actors. Rebecs are similar to Java objects, but possess interfaces (*message servers*) to interact with each other and contain a message queue to process messages from other Rebecs in a FIFO manner. Its IDE, Afra [9], contains a model checker, Modere [27]. Each Rebec can have state variables that can constitute checkable properties. The properties are defined as LTL formula in a separate property file. Apart from the usability for software engineering and modeling and model checking cyber-physical systems, it is possible to convert code into Lingua Franca [28]. Lingua Franca is a coordination language, providing a deterministic concurrency model, that supports C, C++, Python, TypeScript, and Rust as targets [29]. Eventually, a Rebeca code can produce a verified program in one of these languages via Lingua Franca. Formally, a Rebeca model $\mathcal{M}$ is defined as $\mathcal{M} = ||_{i \in \mathcal{I}}\, r_i$, which is the set of concurrent actors (also called *rebec* for *re*active o*bjec*t) r_i, with $i \in \mathcal{I}$ being an identifier from the set of all identifiers in the model [30]. An actor is defined as $r_i := \langle V_i, M_i, K_i \rangle$, where i is a unique identifier, Vi the set of state variables, M_i the set of method identifiers (local methods and remotely callable methods called *message servers*), and K_i the list of other actors known to actor i. Remote method (i.e., message server) calls between actors occur by sending messages $msg = \langle sendid, i, mtdid \rangle$, with $sendid \in \mathcal{I}$ being the sender actor's identifier, $i \in \mathcal{I}$ the receiver actor's identifier, and $mtdid \in M_i$ the identifier of the method called on the sender side.

15.2.5 Near Field Communication and Integrated Circuit Access

Near Field Communication (NFC) is a communication standard [31, 7] that defines wireless communication for small, powerless embedded systems with low data rates. It defines messages for data transmission (information or *I* blocks), signaling (supervisory or *S* blocks), and acknowledgments (receive-ready or *R* blocks) and the modalities operate communications. The main applications are RFID-tags and wireless chip card access. For the latter, it uses the structures of ISO/IEC 7816-4 [32]. This standard logically segments circuits into applications (comparable to directories in a file system) that can be accessed via *Dedicated Files (DFs)* and contains *Elementary Files (EFs)* as data stores.

The standard defines commands to access (*SELECT*) data and applications, as well as manipulate data (*READ*, *WRITE*, *UPDATE*, etc.). It also defines *GETCHALLENGE* and *AUthenTICATE* commands to implement challenge-response-based authentication mechanisms that protect sensitive data. Usually, an authentication procedure yields a session key, that is subsequently used to encrypt the access to the protected parts of the system. Every command induces a response that contains an unencrypted two-bytes status code (even if the data itself is encrypted). This code is either *9000* for positive results or *6XXX* for various error codes.

15.2.6 Electronically Machine-Readable Travel Documents

Electronically Machine-Readable Travel Documents (eMRTDs) are a logical data structure to store travel document (e.g., passport) data on chips. The International Civil Aviation Organization (ICAO) standardizes this format in their Doc 9303 part 10 standard [33]. It defines two *DF* areas *LDS1* (containing the eMRTD application) and *LDS2* (containing travel records, visa records, and additional biometrics). The eMRTD contains the the Common (CM), the Country Verifying Certification Authorities (CVCA), and the Document Security Object (SOD) EFs, as well as EFs for 16 data groups that contain various types of data, like personal data, document data, biometrics (fingerprint, iris). The latter are more sensitive and therefore additionally protected, while the rest is protected by either the older Basic Access Control (BAC) or the newer PACE (Password Authenticated Connection Establishment). BAC generates a key based on some cryptographic operationes with the passport number, expiration date, and the owner's date of birth. PACE uses a password to encrypt a nonce, which then is the base for a Diffie-Hellman-Merkle key exchange that creates a session key. Besides there are some conditional files outside of both LDS, namely Attributes/Info (ATTR/INFO), Directory (DIR), Card Access (CA), and Card Security (CS). Figure 15.1 shows an overview of the ICAO eMRTD schema.

15.2.7 Unified Diagnostic Servcies

Unified Diagnostic Services (UDS) is an ISO standard that specifies services for diagnostic testers to control diagnostic (and further) functions on an on-vehicle Electronic Control Unit (ECU) [8]. Therefore, it defines a variety of services, whereby the concrete elaboration mostly lies in the hands of an implementer. The protocol operates on layer 7 of the OSI reference model.

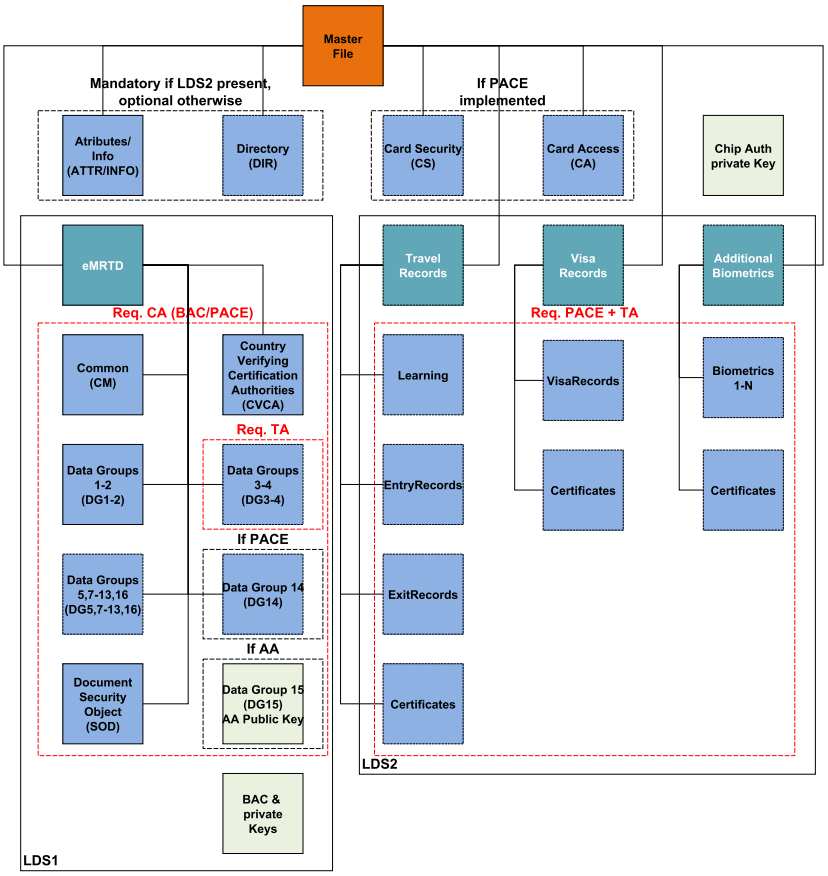

Figure 15.1: Logical Data Structure of Machine Readable Travel Documents from [34]. Amber is the master file (MF), Cyan are dedicated files (DF), Blue are Elementary Files (EF), and Green are key files. Solid frames means mandatory files, dashed ones optional files. Solid boxes donate the LDS contexts, dashed black boxes requirements, and dashed red boxes necessary authentication.

Therefore, it relies on an underlying communication protocol. Although it can use multiple protocols, the most proliferated is the Controller Area Network (CAN) protocol. As a client-server-based protocol it defines requests and responses. The requests contain a service ID, optionally followed by sub-service IDs. Particularly, it defines diagnostic and management services: Diagnostic Session Control (0x10), ECUReset (0x11), SecurityAccess (0x27), CommunicationControl (0x28), TesterPresent (0x3E): a keepalive mechanism,, AccessTimingParameter (0x83), SecuredDataTransmission(0x84), ControlDTCSetting (0x85), ResponseOnEvent (0x86), ResponseControl(0x87). It further defines data transmission services: ReadDataByIdentifier (0x22) , ReadMemoryByAddress (0x23) , ReadScalingDataByIdentifier (0x24) , ReadDataByPeriodicIdentifier (0x2A) , DynamicallyDefineDataIdentifier (0x2C) , WriteDataByIdentifier (0x2E) , WriteMemoryByAddress (0x3D), ClearDiagnosticInformation (0x14), ReadDTCInformation (0x19). Lastly also I/O, routine control, and file transfer functions: InputOutputControlByIdentifier (0x2F) , RoutineControl (0x31), RequestDownload (0x34), RequestUpload (0x35), TransferData (0x36), RequestTransferExit (0x37), RequestFileTransfer (0x38). We will use a set of these services as an input alphabet for learning the UDS state machine (see Section 15.4.2).

15.3 Generic Security Requirements

In this section, we take some long established security goals [35, 36, 37] that are also aligned with our case studies' standards documents (ICAO [38] and UDS [39]) and formulate them as checkable requirements in given-when-then[3] format [40]. We use the terms MUST, MUST NOT, SHOULD, SHOULD NOT and MAY as defined by the Internet Engineering Task Force (IETF) [41, 42][4]. These requirements use attributes, we later (in Section 15.6.1) use as propositions[5] for defining logic properties.

15.3.1 Authentication

To establish authenticity as a goal, we must require *authentication* of a user before accessing PROTECTED resources (not every resource must be protected,

[3] Using *given, when,* and *then* as keywords in italics.
[4] Using the same CAPITALIZED style.
[5] Using SMALL CAPITALS style.

some may be openly accessible – this depends on the system's security design). As a requirement , we define it as follows:

Requirement R1: *Given* the system is not in an AUTHENTICATED state, *when* an ACCESS operation on a PROTECTED resource occurs, *then* the operation MUST NOT return a positive response.

15.3.2 Confidentiality

To acquire confidentiality (also called secrecy in cryptographic contexts), most protocols allow for encryption (and may require it under certain conditions). We therefore recommend encryption for protected resource access:

Requirement R2: *Given* the system is in an arbitrary state, *when* an UNSECURED ACCESS operation on a PROTECTED resource occurs, *then* it SHOULD NOT be successful.

15.3.3 Privilege Levels

It is common practice for more complex systems and protocols to define more than one privilege level according to different roles and according privileges of an actor in order to provide access to more critical resources. This also applies for UDS, where this is implemented by different session and security levels [8]. Likewise, for eMRTDs, sensitve data (particularly biometric data like fingerprints and iris scans [43]) have to be additionally protected via terminal authentication according to European extended access control [33, 44].

Requirement R3: *Given* different PRIVILEGE levels, *when* the level of PRIVILEGE is not sufficient, *then* a performed operation MUST NOT be successful.

15.3.4 Key Validity

Another important attribute of secure connections is key validity and recency[6]. An *invalid key* in this sense is either a *wrong key* or an *old key* (not recent[7]). Obviously, this is a fundament for the prior requirements, since an authentication or encryption that accepts invalid keys is intrinsically flawed.

[6]To avoid misunderstandings, for this work we use the term to refer to using the most *recent* key, i.e., no newer key must exist. Since we do not consider a time component in the automata, we do not use recency in the sense of freshness.

[7]To avoid replay and similar attacks [45, 46]

Requirement R4: *Given* an AUTHENTICATION OR SECURE ACCESS operation, *when* an actor is asked for a key, *then* providing an *invalid key* MUST NOT be successful.

15.4 Building the Mealy Machines using Automata Learning

We use Learnlib (see Section 15.2.2) to create a setup to learn models of different protocol SULs, particularly NFC-based eMRTDs and UDS communications of automotive ECUs. Learnlib provides a variety of learning algorithms of which we use the TTT algorightm [47] for its comparable high performance [18]. We rely on previous work for the learning parts of eMRTDs [34] and UDS [48], respectively. In this work, we annotated the learned models' states with labels for propositions and translate them into code to allow for model checking. In this section we provide some details of the learning processes for the two protocols. While the learner is the same, the SUL adapters differ due to different input alphabets and different means to access the respective SUL.

15.4.1 NFC Interface

We use NFC to access and learn models of eMRTDs. This section explains the interface and inputs used to learn models of eMRTDs.

Input Alphabet

The used input symbols for learning eMRTDs contain effectively selecting the dedicated file for LDS1, selecting elementary files (particularly, CA/CVCA, CM, CS/SOD, ATR, DIR, and data groups DG1-16[8], read binary and update binary[9]. These symbols are used in a plain and a secure (encrypted with a key obtained in an authentication process) version. Furthermore, we defined additionally modified versions of the secured commands both using a wrong key (actually an all-zero key) and an old key from a previous authentication

[8] Selecting CA and CVACA, as well as CS and SOD, respectively, works with the same inputs. The difference is whether the DF for LDS1 is selected or not (see Figure 15.1).

[9] We also tried write binary, search binary, erase binary, and read record. These were not supported on used SULs (error code 0x6D00).

process[10]. For authentication, also a basic authentication (BAC) process is triggered via a dedicated input symbol. Since we lack an implementation of another authentication method (e.g., PACE or Terminal Authentication), BAC is the only authentication input we use.

Interface Device

We use a Proxmark3 NFC interface devices to access eMRTDs [49]. This devices is capable of handling communication with a broad variety of different NFC dialects, as well as reading from, writing on, flashing and emulating them. Besides, it allows for crafting arbitrary frames, altering any part of the data stream to specific needs.

SUL Adapter

The Java learner communicates with the SUL via a distinctive class that handles input, reset, etc. This SUL class then uses a socket to transmit the inputs to a C++ based adapter program that translates the input symbols into NFC frames and pushes it to the interface device. We altered both the standard proxmark interface program and firmware for efficiency and stability reasons. As a result, we send and receive complete input and output words instead of single symbols [50].

15.4.2 UDS Interface

Input Alphabet

We use the following inputs from the UDS standard (with the respective service IDs and sub IDs in parentheses): DefaultSession (10_01), ProgrammingSession (10_02), ExtendedDiagnosticSession (10_03), SafetySystemDiagnosticSession (10_04), Clear Diagnostic Information (PowerTrainDTC - 14), SecurityAccess (27), SecurityAccessWithKey (as response to a seed with a legit key - 27_0A), SecurityAccessWithWrongKey (same as SecurityAccessWithKey, but with a random key), Communication Control (28), Authentication (29), TesterPresent (3E), Secured Data (84), Routine Test (31), ReadDataByIdentifyer (22 with IDs 0xF100, 0xF150, and 0xF180), RequestDownload (34), RequestUpload (35), TransferData (36 - using dummy data), TransferExit (37).

[10]Secret keys for secured commands are obtained by successfully carrying out authentication. If no authentication happened in an input trace before using a secured command, an all-zero key is used. The same applies if an old key should be used but none or only a recent one is available.

Interface Device

In contrast to NFC, the hardware adapter does not need to be modified. To access the SUL, we use an off-the-shelf Peak Systems PCAN-FD device [51], that provides a standard CAN-FD interface for Linux (where it is registered under e.g., */dev/can0*) or Windows systems. We access this interface via a Python-based interface described in Section 15.4.2 to send and receive UDS messages over CAN.

SUL Adapter

Analogous to the NFC interface, the learner uses a distinctive SUL class. This class communicates with the device adapter program via a socket interface and passes the input symbols forward. It calls a respective input handler function in a python script. This handler function creates a raw frame (i.e., the raw UDS hex sequence encapsulated in a CAN message) corresponding to the input symbol. It then sends it over the system CAN interface and returns the respective response frame to the learner. This python interface builds on previous work [48] and was expanded with some error handling procedures, CAN-FD support and a variety of additional input symbols.

15.5 Generating the Rebeca Code

Once the learner provides a model of the SUL (stored in GraphViz format), it can be transformed to code in Rebeca for model checking. This happens in two phases: annotating the model with propositions and subsequently translating into code. The model checker also needs properties to check in the form of linear temporal logic (LTL). This section describes how to obtain these three components.

15.5.1 Annotating the Mealy Machine with Propositions using CPMs

Automata learning provides us with a SUL's Mealy machine (see Section 15.2.2). In a Mealy machine we only have inputs and outputs to distinguish different states and each state does not include any other information. Our goal is to determine the security of a system based on certain properties and it is more intuitive to specify the properties based on the information about the states.

We therefore annotate the states with relevant propositions. These propositions should provide the necessary context for sensible security checking.

We define intuitive and usable propositions for our context for each state (e.g., authenticated) and provide a method to annotate the model with these propositions in a correct way. Since the only way of determining the propositions is through the system's behavior (i.e., input and output), a straightforward approach is to define a *context-based proposition map (CPM)* that describes which input and output would make a propositions true or false. Therefore, we define *gain* (for becoming true) and *lose* (for becoming false) conditions. These condition sets (C) define when a state gains (C_g) and losses (C_l), respectively, a proposition and constitute the CPM. We define a condition ($c \in C$) as a triplet of a set of propositions $S_P \in 2^P$, an input set $S_\Sigma \in 2^\Sigma$, and an output set $S_\Omega \in 2^\Omega$ ($\langle S_P, S_\Sigma, S_\Omega \rangle$) and P as set of all propositions in the annotated Mealy machine. Which means that a condition could apply to multiple properties ($p \in P$) to be set by one multiple inputs ($\sigma \in \Sigma$) producing one of multiple outputs ($\omega \in \Omega$). The set C_g is applied to all transition labels of the model. The *target* state of each transition gains S_P if the transition's input and output equals $\sigma \in \S_\Sigma$ / $\omega \in S_\Omega$ of c according to the Mealy machine's output function λ. To automatically apply the propagation of propositions, we create an inheritance map for all propositions. That is, we determine for each transition in the system whether it would propagate the proposition in question. This is the case for any transition which input/output does not match σ/ω for a respective proposition p in the lose set C_l. With this inheritance map, we iteratively determine the cascading of propositions. For each proposition, we start at the states that possess the respective proposition according to Cg and annotate all target states of transitions in the propagation map for the proposition with p. With the updated model, we repeat that step until no new states are to annotate. Eventually, we have a state machine annotated with all propositions according to the defined conditions. As a result, we have labeled states like in a Kripke structure. Since we still maintain the Mealy machine labels of transitions, this resulting *annotated Mealy machine* can be seen as combined Mealy-Kripke structure. We therefore formally define the annotated Mealy Machine $\mathcal{MK}$ as $\mathcal{MK} = (Q, q_0, \Sigma, \Omega, , \delta, \lambda, \mathcal{L})$, with Q being the set of states, q_0 the initial state, Σ the input alphabet, Ω the output alphabet, δ the transition function and λ the output function of the Mealy machine. Finally, $\mathcal{L}$ denotes the labeling function that attributes propositions $p \in P$ to the states. $\mathcal{L}$ corresponds to the used CPM. For the model checking process, we also need to check for results of performed operations (e.g., a successful read operation) that are described in the transition labels of the Mealy machine. Since we use automata-based

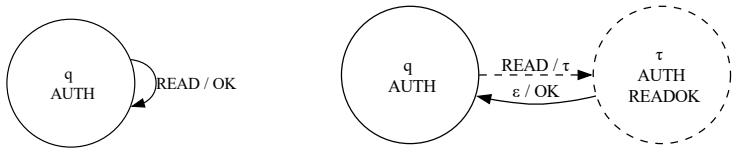

Figure 15.2: We need temporary propositions to specify *certain* properties where the properties are related to labels on the transitions and we do not have a corresponding proposition in the state for them. Here is a general example for a temporary proposition, stemming from a transition output. Left is the original state (q), already labeled with a proposition ($AUTH$). On the right side, we introduce an internal state (τ) to gain a checkable proposition for a successful read ($READOK$). It has this temporary proposition and inherits all other propositions from q.

model checking, we need to define a proposition for each result we want to check. Since the information is not contained within a state, we cannot directly set a proposition. For an example, a state x contains such an operation (we want to attribute with proposition p) as a self-loop transition. Before performing the operation p is false, afterwards it becomes true. In such a case it is undecidable if state x has proposition p. It might, however, still be needed to check if p does not occur in combination with some other proposition (e.g., a read must only occur if authentication is also set – see Property P1 in Section 15.6.1). To resolve this, define temporary or *internal (τ)* propositions (C_τ). That is, propositions that are true only in a single, internal state. We only use the internal properties in the translation to Rebeca code, they are not directly visible in the annotated Mealy machine (see Section 15.5.2). For any input and output matching $c \in C_\tau$, we split the respective transition (labeled *IN/OUT*) into two parts: an internal transition (labeled *IN/*τ) from the origin state (q) to a new internal (τ) state and an empty-input transition (labeled ε*/OUT*) from τ the target state ($q\prime$, which may or may not be equal to q). The generated τ state does not have any transitions than these two fragments of the original one. We then let state τ inherit all propositions from q and additionally set the proposition p that is defined in c, which is then (only) true in τ. Since we do not alter the target state, there is automatically no further inheritance. For the authentication example (see also Figure 15.2), if we check the property that

reading is only possible with prior authentication (with a proposition $AUTH$ set), the generated τ will receive a proposition $p = READOK$. Further, it also inherits the propositions of the origin state q. Therefore, if q is an authenticated state, the property holds (since both $AUTH$ and $READOK$ are true in τ). If q is not authenticated, the property is violated (since in τ, $READOK$ is true, but $AUTH$ is false). Therefore, CPMs enable to check different complex protocols with the same generic properties, as they instantiate the properties by populating it with the necessary (but protocol-specific) propositions.

15.5.2 Mapping the Annotated Mealy Machine into Rebeca Code

After annotating the Mealy machine with propositions, a fully automated procedure implemented in Java translates the resulting annotated Mealy machine to Rebeca code (see a formal definition in Section 15.5.3). In the annotated Mealy machine we see input and outputs of each transition. In the corresponding Rebeca code we have two actors: *environment* (an external entity giving input and collecting output) and a *system* (the entity whose behavior is defined by the Annotated Mealy machine). The *environment* actor contains a *request* message server and message servers corresponding to each output in the annotated Mealy machine. In the *request* message server we non-deterministically send messages corresponding to each possible input symbol to the *system*. For these messages related to each input symbol there is message server in the *system* actor. Each of these message servers calls back the message server of the *environment* actor that corresponds to the respective output in the respective state (by call back we mean sending a return message). We keep track of each state using a state variable in the *system*. Additionally, the *system* entertains state variables for the propositions. Every time a proposition changes from one state to another (gaining or losing) it is set to true or false, respectively, along with changing the state and ? calling the appropriate output message server on the *environment* actor. Additionally, the algorithm considers the temporary propositions. If C_τ defines a temporary proposition, a corresponding state variable is set on the *environment* actor. However, all temporary propositions are reset (i.e., set to false) in the *environment* actor's request message server, which is subsequently called by the output message server. This assures that the temporary proposition is present in a state for the model checker, but not maintained further, as it is not part of any state of the Mealy machine. As a summary, the main part of the Mealy machine's logic is modeled in the *system* actor, while the *environment* actor can be seen as an external input giver to the

system.

15.5.3 Formal Definition of a Rebeca Template

We formalize the translation of the annotated Mealy machine by defining a formal Rebeca template by substituting the annotated Mealy components into a Rebeca model definition (see Section 15.2.4). We define this Rebeca model with two actors: $\mathcal{M} := r_{sys}||r_{env}.r_{sys}$ is for a *system* actor that obtains the annotated Mealy machine's behavior and r_{env} for an *environment* actor that serves as an input giver. We define the *system* as $r_{sys} := \langle V_{sys}, M_{sys}, K_{sys}\rangle$. We further define $V_{sys} := \{q\} \cup AP$. So r_{sys} holds a state variable with an identifier of the current state ($q \in Q$ from the annotated Mealy machine, initially set to q_0) and P, which is the annotated Mealy machine's set of atomic propositions, for each of which $V_{:}sys$ contains a boolean state variable. The only known actor is the *environment* ($K_{sys} := \{r_{env}\}$) and the method identifiers are equal the set of input symbols ($M_{sys} := \Sigma$). The *environment* ($r_{env} := \langle V_{env}, M_{env}, K_{env}\rangle$) contains the *temporary propositions* from the CPM as state variables ($V_{env} := C_\tau$)) and *system* as other known actor ($K_{env} := \{r_{sys}\}$). The set of method identifiers consists the output alphabet and a *request* method that handles the calls of the system and resets the temporary propositions ($M_{env} := \Omega \cup \{req\}$). In each of the methods of r_{sys} ($i \in M_{sys}$) two things happen: a) we set the curent state identifier (q) along with all properties ($p \in AP$) according to the respective transition in the annotated Mealy machine's transition function that corresponds to the current state and input symbol $\delta(q, \sigma)$[11]. b) we call an output method in r_{env} ($oinM_{env}$) according to the current sate and the annotated Mealy machine's output function $\lambda(q, \sigma)$, which means sending the message $\langle r_{sys}, r_{env}, M_{env}(\omega)\rangle$. The *environment* actor's (r_{env}) *request* method (req), resets all state variables (which represent the temporary propositions from C_τ) to false and randomly picks one input by calling a method from M_{sys} by sending the message $\langle r_{env}, r_{sys}, M_{sys}(\sigma)\rangle$, with a randomly picked $\sigma \in \Sigma$. Each of the other methods (mapping to a ω from the annotated Mealy machine) sets a state variable corresponding to a *temporary proposition* to true, if there is a respective entry in the CPM (C_τ)[12] and then calls the *request* method (with the message $\langle r_{env}, r_{env}, req\rangle$) to start another interaction cycle. Lastly, we note that the *request* method is also called

[11] Note: the Method $m \in M_{sys}$ has a direct, bijective representation in the set of Input Symbols ($\sigma \in \Sigma$): $M_{sys} \leftrightarrow \Sigma$. Additionally having the state variable, we can therefore directly use the transition (δ) and output (λ) functions of the annotated Mealy machine.

[12] Depening on the calling methods, which is submitted as a parameter in the message.

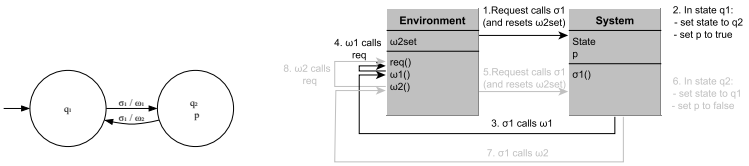

Figure 15.3: A simple annotated Mealy machine example (left) with an UML diagram of its Rebeca representation (right), as outlined in Section 15.5.2. The black arrows in the UML diagram show an example flow an input of σ_1 in state q_1, the gray ones a flow of an input of σ_1 in state q_2.

by r_{env}'constructor, which starts the execution of the Rebeca model ($\mathcal{M}$.

15.5.4 Illustrative Example

To give an illustrative example, we define a Mealy machine with two sates ($Q = \{q_1, q_2\}$), one input ($\Sigma = \{\sigma_1\}$), and two output ($\Omega = \{\omega 1, \omega 2\}$) symbols. Further, we have the initial state $q_0 = q_1$, the transition function $\delta(q, \sigma) := \{\ q_1$ if $q = q_2 \wedge \sigma = \sigma_1;\ \ q_2$ if $q = q_1 \wedge \sigma = \sigma_1;\ \}$, and the output function $\lambda(q, \sigma) := \{\ \omega_1$if $q = q_1 \wedge \sigma = \sigma_1;\ \ \omega_2$if $q = q_2 \wedge \sigma = \sigma_1;\ \}$ We further define the CPM as $C := C_g + C_l + C_\tau$, with $C_g := \{\langle p, \sigma_1, \omega_1\rangle\}$, $C_l := \{\langle p, \sigma_1, \omega_2\rangle\}$, and $C_\tau := \{\langle \omega_2 set, *, \omega_2\}$, which essentially means that $q2$ has proposition p set and we want to observe when ω_2 occurs as an output. Going through the described process, we receive the following Rebeca model($\mathcal{M}$, as defined in Section 15.5.3): $r_{sys} := \langle V_{sys} := \{q, p\}$, $M_{sys} := \{\sigma_1\}$,$K_{sys} := \{r_{env}\}\rangle$. $r_{env} := \langle V_{env} := \{\omega_2 set\}$, $M_{env} := \{req, \omega_1, \omega_2\}$,$K_{env} := \{r_{sys}\}\rangle$. Figure 15.3 shows a graphical representation of this example, while Listing 15.1 shows the resulting Rebeca code[13].

[13]Please note that in the code we use *state* for the state variable denoting q and that we spell the names of Greek letters in full instead of using their symbol.

```
reactiveclass Environment(3) {
    statevars {
        boolean omega2set;
    }
    knownrebecs {
        System system;
    }
    Environment(){
        self.req();
    }
    msgsrv req() {
        omega2set=false;
        int data = (0);
        switch(data) {
            case 0: system.sigma1(); break;
        }
    }
    msgsrv omega1(){
        self.req();
    }
    msgsrv omega2(){
        omega2set=true;
        self.req();
    }
}
reactiveclass System(3) {
    statevars {
        int state;
        boolean p;
    }
    knownrebecs {
        Environment environment;
    }
    msgsrv sigma1(){
        if(state==0) {
            state=1;
            p=true;
            environment.omega1();
        } else
        if(state==1) {
            state=0;
            p=false;
            environment.omega2();
        }
    }
}
main {
    Environment environment(system):();
    System  system(environment):();
}
```

Listing 15.1: Rebeca code from the illustrative example from Section 15.5.2.

15.5.5 Altering the Model

Since in Rebeca a model's behavior is defined similar to a programming language, altering it is trivial. Furthermore, Rebeca possess an operator for non-determinism (the ? operator). On the one hand, we can manually add any arbitrary fault that might be interesting to investigate and examine its impact on keeping the properties. On the other hand, this allows for re-introducing some non-deterministic behavior that is abstracted away during the learning process (see Section 15.2.2) [52]. By altering the template mentioned in Section 15.5.3, we just introduce a *timeout* server on the sender side and alter each input message server on the receiver to trigger a timeout with a certain probability[14]. If no timeout occurs, the logic continues. Otherwise, we reset the state to initial and call the timeout function on the sender, which sets a temporary proposition for *timeout*, to allow for checking for it. Since we can set the timeout probability to an arbitrary value, we can play with different scenarios to be checked. We also can convert the resulting Rebeca code back to a Mealy machine, so we possess a model that can contain timeouts and errors, just as the originally learned model would have[15]. Formally, we alter the Rebeca template by extending M_{env} with a *timeout* method, that sets a state variable $timeout \in V_{env}$ (to perform checks for timeouts if needed) and calls the *request* method. Further, we alter each method $m_{sys} \in M_{sys}$ to randomly (with a settable parameter determining the probability) reset the state identifier q to the initial state q_1 and call the *timeout* method ($\langle r_{sys}, r_{env}, timeout \rangle$), instead of the output method ($M_{env}(\omega)$ derived from $\lambda(q, \sigma)$.

15.5.6 Verifiying the Code

In order to assure the Rebeca code is faithful to the learned model, we check their equivalence. We generate the full state space of the Rebeca model. For that, we run the model checker without any properties to check, so, the model checker does not stop at a violation. This way the full state space is generated, which we export to a Linear Transition System (LTS) representation in the GraphViz format. Based on a previous work in [53], we then use an algorithm that collapses the state space LTS into a Mealy machine and check the bisimilarity of the converted Rebeca Mealy machine with the originally

[14] Realized with a random (?) pick of an integer

[15] If the resulting Rebeca code contains non-determinism, the resulting model is not technically a Mealy machine, but a *Mealy-styled non-deterministic state model, for it defines multiple transitions for a given input per state to represent the different possibilities.*

learned model. The Rebeca model checking tool (RMC) supports Linear Temporal Logic (LTL) to formulate properties. For security checking, these LTL form should describe security attributes.

15.6 Checking the Model

15.6.1 Defining Generic Properties

Since the requirements in Section 15.3 are very generic, we can also define generic properties to check them. The become more specific in conjunction with the CPMs (Section 15.5.1), which define the meaning of propositions used in the property in the context of a specific protocol[16]. For the sake of readability, we restrict our definitions of ACCESS to read operations but write, update, and similar operations can be defined the same way.

Authentication

For Requirement R1, we define an *authenticated state* as a state with the proposition AUTHENTICATED set to false, a when an access operation on a PROTECTED resource occurs, then the operation must not return a positive response (*access ok*).

Property P1: $\Box(\neg AUTH \wedge PROT \rightarrow \neg ACCESSOK)$[17].

Here, we define $AUTH$ for AUTHENTICATED, $PROT$ for PROTECTED, and $ACCESSOK$ for a positive response[18] (i.e., a successful read operation).

Confidentiality

For checking the confidentiality (Requirement R2), we assume the presence of unsecured and secured operations. For the *access* operation as used in Section 15.6.1, we therefore define two exemplary subsets: an UNSECURED read $UREAD$ and a SECURED read $SREAD$. Since $READ$ is a superset, in the respective context maps we then define that $READ$ will always be

[16] If a proposition used in a property is not defined in the respective CPM, we define it as *false*($\bot$) for the respective property.

[17] Afra does not support implications in the property file, therefore we use $\Box(\neg(\neg AUTH \wedge PROT) \vee \neg ACCESSOK)$, instead.

[18] Example for non-existing protocols: if we have a protocol with no protected resources, we set $PROT := \bot$, which means that reading is always allowed according to Property P1.

set along with $UREAD$ *or* $SREAD$ is set (using the same conditions, i.e., $c_{READ} \leftrightarrow c_{UREAD} \vee c_{SREAD}$). Given requirement R2, we define that an $UREAD$ may not succeed when $PROT$ is true. Please note that we fail to apply this property to the UDS case, since there is no encrypted communication on the CAN bus that servers as a medium for the UDS protocol[19].

Property P2: $\Box(PROT \rightarrow \neg\, UREADOK)$

Privilege Levels

We assume that there are areas that go beyond a basic protection level and that there additional protection measures (e.g., different type or level of authentication) have to be taken (Requirement R3). We define this higher protection level as *critical* resources ($CRIT$) and the additional authentication as privileged ($PRIV$). We further assume that since it is a higher security level *privileged* authentication ($PRIV$) also includes normal authentication ($AUTH$).

Property P3: $\Box((PRIV \rightarrow AUTH) \wedge (\neg PRIV \wedge CRIT \rightarrow \neg ACCESSOK))$

Key Validity

A property for using a valid key (Requirement R4) is straightforward, since it only has to define that an invalid key must not be accepted in the entire system.

Property P4: $\Box(\neg INVKEYOK)$

Depending on the protocol, we use propositions like $WRONGKEYOK$ or $OLDKEYOK$ that would both be seen as invalid keys.

15.6.2 Checking the Properties

This section defines and explains the CPMs for eMRTDs and UDS, along with defining some protocol-specific properties that allow for checking the protocol flow.

[19] While there are some proprietary and research solutions for encrypted CAN, none of them is widely adopted (i.e., used by a larger number of major OEMs). An industry standard gaining momentum, the AUTOSAR *Secure On-board Communication (SecOC)* [54], only provides authentication and integrity, but not confidentiality.

eMRTDs

For eMRTDs, we define $PROT$ as DF input in C_g of the CPM for eMRTDs (Table 15.1), since the ICAO standard defines the applications to be in a protected zone[20], while $AUTH$ is gained by performing a *basic authentication (BAC)* operation. As $ACCESSOK$ we define a successfully performed file selection in C_τ. This means that for P1 a BAC must be performed before selecting any file inside an application.

For $UREADOK$, we define an unsecured *read binary* operation. This means that for P2, only secured *read binary* operations are allowed.

We define $CRIT$ as a successful selection of data groups 2 or 3 in C_τ. These contain biometric data, which is why the ICAO standard requires and additional (particularly terminal) authentication. Since, we do not have an additional authentication method implemented, $PRIV$ is set to false ($\bot$). This means that P3 only holds when these sensitive data groups cannot be selected in the complete model.

Lastly, we define $INVKEYOK$ for P4 as *any* successful secured operation that has been carried out using an old key or an all-zero key in C_τ. We realize these by dedicated input symbols (see Section 15.4.1).

We also use a couple of complimentary properties for checking eMRTDs. SecureRead: $\Box(\neg(SREADOK \wedge \neg(DF \wedge AUTH \wedge EF)))$, which reads as a secure read operation (SREADOK) can only be successful if authenticated and a protected resource is selected ($DF \wedge EF$). This also creates the need for a normal read operation not to work on protected resources: PlainRead: $\Box(\neg(UREADOK \wedge (\neg EF \vee DF)))$, which reads as a successful plain read (READOK) cannot happen without a selected resource outside of LDS1 ($\neg EF \vee DF$). We can further specify a secure select process with the following property: SecureReadFollowsSecureSelect: $((\neg SREADOK)\ \mathcal{U}$ SSELEFOK$) \vee \Box(\neg SREADOK)$. Table 15.1 gives an overview of the CPM.

UDS

Authentication in UDS works over *SecurityAccess* and subsequent *SecurityAccessWithKey*. We therfore define the $AUTH$ proposition after the second step has been successfully exectued. As a canary for P1, we perform a reading operation on a protected resource. We choose to start the *Check Programming*

[20]Note: we only set DF for selecting LDS1, which corresponds to the eMRTD application. Technically, DF would also be set when selecting other applications (i.e., LDS2). However, we then would design the CPM to distinguish between DFs. This is not necessary in our case, since we only use one.

Dependencies (0xFF01) routine. This routine does not alter the firmware and is standardized [8]. Furthermore, it is industry practice that this route is protected by a *Security Access* routine [55]. Therefore we set both $PROT$ and $ACCESSOK$ as a c_τ when starting this routine. For P2, set perform a direct memory read operation to set $UREADOK$. Unfortunately, we don not have the knowledge of which memory are actually protected (as a workaround defining the CheckASWBit routine could also be defined as $PROT$). For P3, we define the *RequestDownload* routine as $CRIT$, as it allows to actually flash the device, while $PRIV$ his defined as *SecurityAcccess* with a higher level. Lastly for P4, $INVKEYOK$ is defined as *SecurityAcccess* with a wrong key (as an own input symbol). Table 15.2 shows the CPM for UDS.

15.7 Testing

With a property violation discovered by the learning and checking procedure, the only thing left is to verify the finding on the actual system. Due to relying on conformance testing for equivalence queries, the automata learning process maintains a residual risk for an inaccurate model. We therefore test the trace of a found violation with the life system to exclude false positives. If the actual system behavior matches the one predicted by the model, the violation is confirmed. Otherwise, the trace can be fed back to the learning system to refine the model. The verification of found traces is trivial, as the same tool set from the learning can be used for concretizing the input and interfacing with the SUL. Therefore, this combination can be seen as method for test case generation.

15.8 Evaluation

To show the practical usability and versatility of the described process, we evaluate it on two devices (an eMRTD and an automotive control unit), each speaking one of the communication protocols described earlier in this paper (NFC and UDS). Though the use cases are very different, in both cases the model checking is very similar. At the present abstraction level, we check generic security requirements (Section 15.3) using according LTL properties (Section 15.6.1).

Table 15.1: CPM for eMRTDs.

Proposition	Input	Output
Gains (C_g)		
AUTH	BAC	9000
DF, PROT	DF*	9000
EF	EF*	9000
CRIT	EF_DG2, EF_DG3	9000
PRIV[a]	TA	9000
Losses (C_l)		
EF, AUTH, PRIV, CRIT	DF	9000
AUTH, PRIV	EF*, *BIN, *REC	6*
CRIT	EF*	9000
Temporary (C_τ)		
UACCESSOK, ACCESSOK	SEL_EF*	9000
SSELEFOK, SACCESSOK, ACCESSOK	SSEL_EF*	9000
UREADOK, READOK	RD_BIN	9000
SREADOK, READOK	SRD_BIN	9000
SSELEFOK	SSEL_EF*	9000
INVKEYOK, WRONGKEYOK	WS*	9000
INVKEYOK, OLDKEYOK	OS*	9000

[a] Since, we do not have a Terminal Authentication implementation, this condition is hypothetical (will not be triggered).

15.8.1 Electronically Machine-Readable Travel Document

For the eMRTD use case, we scrutinize two different Austrian passports, on elder, expired and one current (see a process based on equivalence checking in [34]). The model was learned via LearnLib using the TTT algorithm (with binary closure) and a minimum input trace length of 40 symbols, a maximum

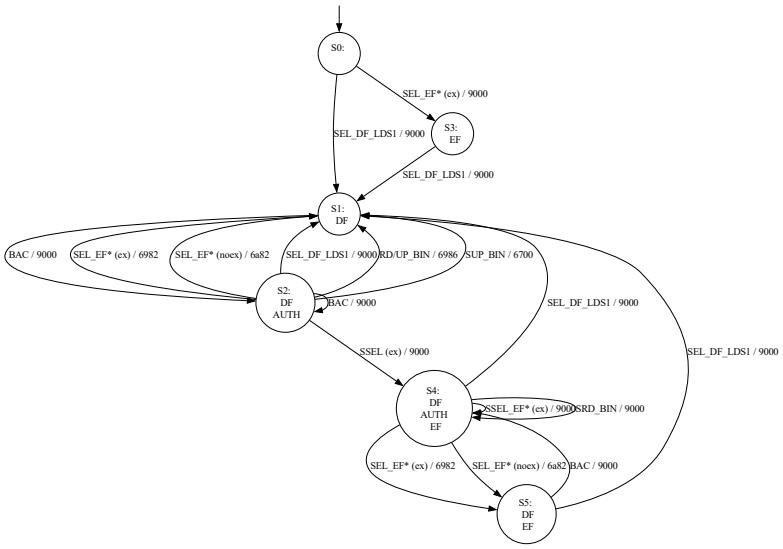

Figure 15.4: Simplified version of the annotated eMRDT. Self-loops that do not add to the understanding have been removed for readability. (S)SEL_EF* (ex) and (noex) denotes all elementary file selections of existing and non-existing files.

of 50, and 150 random walk conformance tests as equivalence oracle. Using the method described above, we translated the Mealy Machine into Rebeca code (see Listings 15.2 and 15.3 for parts of the model code for the environment and system agent, respectively). Using the CPM and properties as described in Section 15.6.2, we were able to verify that described security properties hold the scrutinized systems.

```
msgsrv req() {
   error=false;
   ureadok=false;
   sreadok=false;
   readok=false;
   sselefok=false;
   selefok=false;
   accessok=false;
   [..]
```

```
10    int data =?(0,1,2,[..],56);
11    switch(data) {
12       case 0: system.pp_sel_ef_ca_cvca(); break;
13       case 1: system.pp_sel_df_lds1(); break; [..]
14       case 22: system.pp_rd_bin(); break; [..]
15       case 28: system.pp_bac(); break; [..]
16       case 32: system.pp_ssel_ef_dg1(); break;[..]
17    }
18 }
19 [..]
20 msgsrv req_9000(int data){
21    switch(data) {
22       case 0: accessok=true;selefok=true;
23          uaccessok=true;break;
24       case 1: break; [..]
25       case 22: readok=true;ureadok=true;break; [..]
26       case 28: break; [..]
27       case 32: accessok=true;sselefok=true;
28          saccessok=true;break; [..]
29    }
30    self.req();
31 }
```

Listing 15.2: Example of generated eMRTD environment actor code.

```
1  msgsrv pp_rd_bin(){
2     if(state==0) {
3        state=0; environment.req_6986();
4     } else
5     if(state==1) {
6        state=1; environment.req_6986();
7     } else
8     if(state==2) {
9        auth=false; state=1; environment.req_6986();
10    } else
11    if(state==3) {
12       state=3; environment.req_9000(22);
13    } else
14    if(state==4) {
15       auth=false; state=5; environment.req_6982();
16    } else
17    if(state==5) {
18       state=5; environment.req_6982();
19    }
20 }
21 [..]
22 msgsrv pp_sel_df_lds1(){
23    if(state==0) {
24       df=true; prot=true; state=1; environment.req_9000(1);
```

```
25     } else
26     if(state==1) {
27        state=1; environment.req_9000(1);
28     } else
29     if(state==2) {
30        auth=false; state=1; environment.req_9000(1);
31     } else
32     if(state==3) {
33        df=true; prot=true; ef=false; state=1; environment.req_9000
       (1);
34     } else
35     if(state==4) {
36        auth=false; ef=false; state=1; environment.req_9000(1);
37     } else
38     if(state==5) {
39        ef=false; state=1; environment.req_9000(1);
40     }
41  }
42  [..]
43  msgsrv pp_bac(){
44     if(state==0) {
45        state=0; environment.req_6985();
46     } else
47     if(state==1) {
48        auth=true; state=2; environment.req_9000(28);
49     } else
50     if(state==2) {
51        state=2; environment.req_9000(28);
52     } else
53     if(state==3) {
54        state=3; environment.req_6985();
55     } else
56     if(state==4) {
57        state=4; environment.req_9000(28);
58     } else
59     if(state==5) {
60        auth=true; state=4; environment.req_9000(28);
61     }
62  }
63  [..]
64  msgsrv pp_ssel_ef_dg1(){
65     if(state==0) {
66        state=0; environment.req_6988();
67     } else
68     if(state==1) {
69        state=1; environment.req_6988();
70     } else
71     if(state==2) {
72        ef=true; state=4; environment.req_9000(32);
```

```
73      } else
74      if(state==3) {
75          state=3; environment.req_6988();
76      } else
77      if(state==4) {
78          state=4; environment.req_9000(32);
79      } else
80      if(state==5) {
81          state=5; environment.req_6988();
82      }
83  }
84  [..]
```

Listing 15.3: Example of generated eMRTD system actor code.

15.8.2 Automotive Electronic Control Unit

Our example use case for UDS is an automotive electronic control unit (ECU) from a major European Tier-1 supplier running in a vehicle from a Chinese car manufacturer. The model was learned via LearnLib using the TTT algorithm with a minimum input trace length of 20 symbols, a maximum of 50, and 50 random walks as equivalence oracle. Figure 15.5 shows the annotated Mealy machine (using the CPM in Table 15.2) of this ECU. We subsequently use the properties from Section 15.6.1 to check the model for security properties, along with the CPM for UDS (Section 15.6.2). We could verify P1 . For P2, we lack the knowledge which memory areas are actually protected. However, it is well known that the CAN bus (over which UDS communication runs) is completely unencrypted, while principally critical (safety) data can be exchanged [56]. Therefore, R2 is not met by the protocol itself, regardless of any implementation. To test P3, we lack a higher layer security access. However, scrutinizing critical functions (*RequestDownload*), we saw P3 to hold, as it were not possible to trigger with our current security level. P4 was found to be violated[21], as in the (authenticated) state S6 (see Figure 15.5), the system accepts a wrong key[22].

[21] According to the standard, wrong key should not be acknowledged, even though a wrong key does not mean the security level must be locked (de-authentication) [39].

[22] An issue known to us from previous works [48].

Table 15.2: CPM for UDS (SA means SecurityAccess).

Proposition	**Input**	**Output**
Gains (C_g)		
AUTH	SAWithKey	67
EXT	Extended	5003
PROG	Programming	5002
PRIV[a]	HLSAWithKey	67
Losses (C_l)		
EXT,PROG	Default	5001
EXT	Programming	5002
PROG	Extended	5003
AUTH	SA, SAwKey, SAwWrongKey	7f
AUTH	Session	50
Temporary (C_τ)		
INVKEYOK, WRONGKEYOK	SAwWrongKey	67
PROT	CheckASWBit	71
ACCESSOK	CheckASWBit	71
UACCESSOK	CheckASWBit	71
CRIT	RequestDownload	74
UREADOK	Read*	62

[a] Since, we do not have higher level security access, this condition is hypothetical (will not be triggered).

15.9 Related Work

This work is based on previous work on model learning and model checking [48, 34, 53]. Peled et al. [19] have provided a very influential paper regarding black box testing that combines automata learning with model checking. They learn a model (using Angluin's algorithm) create a cross product with a Büchi automaton created from the negate of an LTL property to check. If the accepted language is empty, the property is satisfied. Otherwise, the contained (non-empty) words it serve as a counterexamples that will be tested against a true system model. If these tests yield the same results, the counterexample poses a property violation. Otherwise, it serves as a counterexample for refining the model. They propose both an off-line and on-the-fly approach. The for-

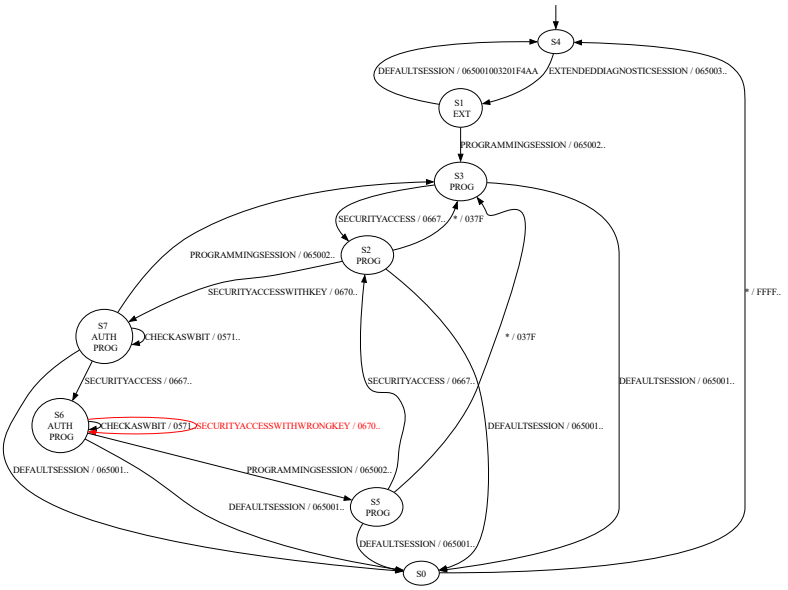

Figure 15.5: Simplified annotated Mealy machine of the learned ECU model. Self-loops that do not add to the understanding have been removed and the output truncated for readability. The star (*) denotes any other input not explicitly stated.

mer uses a fully learned (i.e., a full run-through of Angluin's algorithm) model that is checked. The latter incorporates the checking into the learning process and stops it, if a violation is found, potentially greatly reducing the runtime. Groce et al. [57] advance the former approach by integrating the model checking deeper in the learning algorithm for an efficiency boost. Shijubo et al [58] expand Peled's approach by introducing strengthened specifications, that tend to find counterexamples that earlier detect divergence between the learned and the actual model, boosting the learning process' efficiency. We refrain from using on-the-fly checking because we want to fully investigate the model for any violation of a defined set of properties, and therefore not stop at the first found violation, as the purpose is to investigate already implemented real-world sys-

tems for all incorrect behavior. [23] While our work is based on a similar idea as the off-line variant, we extended the approach by annotating Mealy machines with atomic propositions creating Kripke-like structures that we turned into Rebeca code for model checking. That way, we implemented all the phases for going through learning, checking, and testing real-world examples. In our case, however, we automated the complete process by combining (a) the learner(s) (using LearnLib [10]), (b) an automated way of generating Kripke structures and translating it into a checkable model (described in Section 15.5) and (c) a model checker translates this model and (the negation) LTL properties to Büchi automata, creates a product, and check its language for emptiness [9] with generic properties (Section 15.6). This improves both the usability of the approach and its applicability to real-world (cyber-pyhsical) systems. Neider and Jansen used a symbolic approach to learn and model-check DFAs (but not Mealy or More machines) [59]. Similarily, Fiterau-Brostean et al. [60, 61] also used a symbolic approach (using the NuSMV model checker). Translating the learned model to Rebeca, our approach in contrast provides more advanced possibilities to manipulate the model, including re-introducing abstracted (e.g. non-deterministic) behavior (see Section 15.5.5), as they also stated in their work that ironing out timing issues were a major engineering problem. Altering the Rebeca template, it is trivial to re-introduce timeouts (non-deterministically) to check a protocol's behavior under such conditions. This allows for more advanced checking possibilities. Furthermore, our approach using CPMs does not only allow for a more explicit and comprehensible annotation of states with propositions, but also using generic properties and a clean separation between state proposistions (visible in the annonated Mealy machine) and pure transition outputs (in C_τ and τ states).

Other approaches [62, 15, 63] combine automata learning with bisimulation-based equivalence checking, which need a specification model to compare the learned automaton with. Our approach needs generalized properties. We do not know of an approach that translates a learned model into a modeling language allowing for both direct property checking and manipulating the model to simulate scenarios.

[23] While the Rebeca model checker also stops with a counterexample when a property violation is found, we can, in contrast to on-the-fly-checking, check the other properties as well on the complete model and therefore determine if the system violates multiple properties. Furthermore, many properties do not actually require a modal operator – e.g., Property 1 uses a single $\Box$ operator, which means it should hold globally and a further distinction is not necessary. We can check these with a simple script running over a Rebeca-generated full state space, enabling us to find all occurrences of a violation.

15.10 Conclusion

In this paper, we presented an approach to combine automata learning with model checking for cyber-physical systems. We defined *Context-based Proposition Maps (CPMs)* that provide an annotation mechanism to annotate Mealy machines inferred by Automata Learning with propositions. As a result, we received annotated Mealy machines that combine attributes of Mealy machines and Kripke structures. We subsequently translated the annotated Mealy machine into Rebeca code by formally defining a template using two actors: a receiver modeling the learned system and a sender modeling an external actor that interacts with the system. Starting from four high-level security requirements (authentication, confidentiality, privilege levels, and key validity), we defined generic LTL properties to check models for security. Protocol-specific CPMs provided the context to assure the generated Rebeca code provides all propositions that are used in these properties. This way, each property can be checked with regard to the protocol the examined cyber-physical system runs on. We presented a case study with systems from different domains (a passport and and automotive control unit) speaking different protocols (NFC and UDS) to show the versatility of the approach. We were able to verify the security properties for all requirements on to different passports. On the automotive control unit, we could verify two of them (authentication and privilege levels), while one of them (confidentiality) is intrinsically not provided by the protocol and the last one (key validity) was not met (property violated) on the examined unit.

15.10.1 Outlook

Further research directions to lift the approach to a larger scale lie in creating adapters for other different protocols and creating other generalized LTL properties to check different aspects of security. Additionally, we also investigate utilizing an LLM to generate LTL properties from threat models to check their respective implementations' Mealy machines. Also we can improve the learning process by using timing [64] and by extending the model with unknown inputs using input symbol mutation; if the unknown input triggers new behavior, we can dynamically extend the input alphabet (as described here [65]). This allows for a more holistic testing.

Acknowledgment

This work has received funding from the European Union under the Horizon Europe programme under grant agreement No. 101168438 (project INTACT) as well as from the Swiss State Secretariat for Education, Research and Innovation and UK Research and Innovation under grant No. 10126241. We further acknowledge the support of the Swedish Knowledge Foundation via the industrial doctoral school RELIANT under grant No. 20220130. The authors want to thank the anonymous reviewers of the Computers & Security Journal, as well as of the International Conference on Runtime Verification for their valuable input. We further want to thank Thomas Grandits for helping to formulate requirements and properties more understandable.

Bibliography

[1] M. H. ter Beek, R. Chapman, R. Cleaveland, H. Garavel, R. Gu, I. ter Horst, J. J. A. Keiren, T. Lecomte, M. Leuschel, K. Y. Rozier, A. Sampaio, C. Seceleanu, M. Thomas, T. A. C. Willemse, and L. Zhang, "Formal Methods in Industry," *Form. Asp. Comput.*, vol. 37, pp. 7:1–7:38, Dec. 2024.

[2] F. Vaandrager, "Model learning," *Communications of the ACM*, vol. 60, pp. 86–95, Jan. 2017.

[3] I. Oklo, "Maximum Credible Accident Methodology," Tech. Rep. Oklo-2021-R19-NP, Oklo, Inc., 2021.

[4] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, pp. 1045–1079, Sept. 1955.

[5] S. Kripke, "Semantical Considerations on Modal Logic," *Acta Philosophica Fennica*, vol. 16, pp. 83–94, 1963.

[6] M. Sirjani, "Rebeca: Theory, Applications, and Tools," in *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures* (F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, eds.), vol. 4709 of *Lecture Notes in Computer Science*, pp. 102–126, Springer, 2006.

[7] 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.

[8] International Organization for Standardization, "Road Vehicles – Unified diagnostic services (UDS) – Part 1: Specification and requirements," ISO/SAE Standard 14229-1, International Organization for Standardization, 2020.

[9] E. Khamespanah, M. Sirjani, and R. Khosravi, "Afra: An Eclipse-Based Tool with Extensible Architecture for Modeling and Model Checking of Rebeca Family Models," in *Fundamentals of Software Engineering* (H. Hojjat and E. Ábrahám, eds.), (Cham), pp. 72–87, Springer Nature Switzerland, 2023.

[10] M. Isberner, F. Howar, and B. Steffen, "The Open-Source LearnLib," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), Lecture Notes in Computer Science, (Cham), pp. 487–495, Springer International Publishing, 2015.

[11] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937.

[12] J. von Neumann, "The General and Logical Theory of Automata," in *Systems Research for Behavioral Science*, pp. 1–41, Routledge, 1968.

[13] M. O. Rabin and D. Scott, "Finite Automata and Their Decision Problems," *IBM Journal of Research and Development*, vol. 3, pp. 114–125, Apr. 1959.

[14] E. M. Clarke, "The Birth of Model Checking," in *25 Years of Model Checking: History, Achievements, Perspectives* (O. Grumberg and H. Veith, eds.), pp. 1–26, Berlin, Heidelberg: Springer, 2008.

[15] M. Tappler, B. K. Aichernig, and R. Bloem, "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.

[16] E. M. Clarke, O. Grumberg, D. Peled, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[17] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, Nov. 1987.

[18] F. Vaandrager, "Model learning," *Communications of the ACM*, vol. 60, pp. 86–95, Jan. 2017.

[19] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black Box Checking," in *Formal Methods for Protocol Engineering and Distributed Systems: FORTE XII / PSTV XIX'99 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX) October 5–8, 1999, Beijing, China* (J. Wu, S. T. Chanson, and Q. Gao, eds.), IFIP Advances in Information and Communication Technology, pp. 225–240, Boston, MA: Springer US, 1999.

[20] L. Kruger, S. Junges, and J. Rot, "State Matching and Multiple References in Adaptive Active Automata Learning," in *Formal Methods* (A. Platzer, K. Y. Rozier, M. Pradella, and M. Rossi, eds.), (Cham), pp. 267–284, Springer Nature Switzerland, 2025.

[21] E. Muškardin, B. K. Aichernig, I. Pill, A. Pferscher, and M. Tappler, "AALpy: An active automata learning library," *Innovations in Systems and Software Engineering*, vol. 18, pp. 417–426, Sept. 2022.

[22] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon, "Libalf: The Automata Learning Framework," in *Computer Aided Verification* (T. Touili, B. Cook, and P. Jackson, eds.), (Berlin, Heidelberg), pp. 360–364, Springer, 2010.

[23] B. Russell, "Mathematical Logic as Based on the Theory of Types," *American Journal of Mathematics*, vol. 30, no. 3, pp. 222–262, 1908.

[24] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, Apr. 2008.

[25] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*, pp. 46–57, Oct. 1977.

[26] E. Allen Emerson and A. Prasad Sistla, "Deciding branching time logic: A triple exponential decision procedure for CTL," in *Logics of Programs* (E. Clarke and D. Kozen, eds.), (Berlin, Heidelberg), pp. 176–192, Springer, 1984.

[27] M. M. Jaghoori, A. Movaghar, and M. Sirjani, "Modere: The model-checking engine of Rebeca," in *Proceedings of the 2006 ACM Symposium*

on Applied Computing, SAC '06, (New York, NY, USA), pp. 1810–1815, Association for Computing Machinery, Apr. 2006.

[28] M. Sirjani, E. A. Lee, and E. Khamespanah, "Model Checking Software in Cyberphysical Systems," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1017–1026, July 2020.

[29] C. Menard, M. Lohstroh, S. Bateni, M. Chorlian, A. Deng, P. Donovan, C. Fournier, S. Lin, F. Suchert, T. Tanneberger, H. Kim, J. Castrillon, and E. A. Lee, "High-performance Deterministic Concurrency Using Lingua Franca," *ACM Trans. Archit. Code Optim.*, vol. 20, pp. 48:1–48:29, Oct. 2023.

[30] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, "Modeling and Verification of Reactive Systems using Rebeca," *Fundamenta Informaticae*, vol. 63, pp. 385–410, Dec. 2004. Publisher: SAGE Publications.

[31] 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.

[32] International Organization for Standardization, "Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange," ISO/IEC Standard "7816-4", International Organization for Standardization, 2020.

[33] International Civil Aviation Organization, "Machine Readable Travel Documents – Part 9: Deployment of Biometric Identification and Electronic Storage of Data in MRTDs (Eigth Edition)," ISO/IEC Standard 9303-9, International Civil Aviation Organization, 2021.

[34] S. Marksteiner, M. Sirjani, and M. Sjödin, "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, (New York, NY, USA), pp. 1–8, Association for Computing Machinery, July 2024.

[35] J. P. Anderson, "Computer Security Technology Planning Study," Technical Report ESD-TR-73-51-VOL-1, National Technical Information Service, 1972.

[36] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, Sept. 1975.

[37] C. Namprempre, "Secure Channels Based on Authenticated Encryption Schemes: A Simple Characterization," in *Advances in Cryptology — ASIACRYPT 2002* (Y. Zheng, ed.), (Berlin, Heidelberg), pp. 515–532, Springer, 2002.

[38] International Civil Aviation Organization, "Machine Readable Travel Documents – Part 10: Logical Data Structure (LDS) for Storage of Biometrics and Other Data in the Contactless Integrated Circuit (IC)orage of Data in MRTDs (Eigth Edition)," IACO Doc 9303-10, International Civil Aviation Organization, 2021.

[39] International Organization for Standardization, "Road Vehicles – Unified diagnostic services (UDS) – Part 1: Specification and requirements," ISO/SAE Standard 14229-1, International Organization for Standardization, 2020.

[40] L. P. Binamungu and S. Maro, "Behaviour driven development: A systematic mapping study," *Journal of Systems and Software*, vol. 203, p. 111749, Sept. 2023.

[41] S. Brandner, "Key Words for Use in RFCs to Indicate Requirement Levels," RFC 2119, RFC Editor, 1997.

[42] B. Leiba, "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words," RFC 8174, RFC Editor, 2017.

[43] International Civil Aviation Organization, "Machine Readable Travel Documents – Part 1: Introduction," IACO Doc 9303-1, International Civil Aviation Organization, 2021.

[44] International Civil Aviation Organization, "Machine Readable Travel Documents – Part 11: Security Mechanisms for MRTDs (Eigth Edition)," IACO Doc 9303-11, International Civil Aviation Organization, 2021.

[45] I. Cervesato, C. Meadows, and D. Pavlovic, "An encapsulated authentication logic for reasoning about key distribution protocols," in *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pp. 48–61, June 2005. ISSN: 2377-5459.

[46] T. Lauser and C. Krauß, "Formal Security Analysis of Vehicle Diagnostic Protocols," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, (New York, NY, USA), pp. 1–11, Association for Computing Machinery, Aug. 2023.

[47] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), Lecture Notes in Computer Science, (Cham), pp. 307–322, Springer International Publishing, 2014.

[48] M. Ebrahimi, S. Marksteiner, D. Ničković, R. Bloem, D. Schögler, P. Eisner, S. Sprung, T. Schober, S. Chlup, C. Schmittner, and S. König, "A Systematic Approach to Automotive Security," in *Formal Methods* (M. Chechik, J.-P. Katoen, and M. Leucker, eds.), vol. 14000 of *Lecture Notes in Computer Science*, (Cham), pp. 598–609, Springer International Publishing, 2023.

[49] F. D. Garcia, G. de Koning Gans, and R. Verdult, "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.

[50] S. Marksteiner, M. Sirjani, and M. Sjödin, "Using Automata Learning for Compliance Evaluation of Communication Protocols on an NFC Handshake Example," in *Engineering of Computer-Based Systems* (J. Kofroň, T. Margaria, and C. Seceleanu, eds.), vol. 14390 of *Lecture Notes in Computer Science*, (Cham), pp. 170–190, Springer Nature Switzerland, 2023.

[51] PEAK-System Technik GmbH, "PCAN-USB FD – User Manual," Tech. Rep. 2.2.0, PEAK-System Technik GmbH, 2024.

[52] M. Sirjani, E. Lee, Z. Moezkarimi, B. Pourvatan, B. Johansson, S. Marksteiner, and A. Papadopoulos, "Actors for Timing Analysis of Distributed Redundant Controllers," in *Gul Agha's festschrift*, pp. 1–27, Springer, 2025.

[53] S. Marksteiner and M. Sjödin, "Black-box protocol testing using Rebeca and Automata Learning," in *Rebeca for Actor Analysis in Action* (E. Lee, M. R. Mousavi, and C. Talcott, eds.), vol. 15560 of *Lecture Notes in*

Computer Science, pp. 212–235, Cham: Springer International Publishing, 2025.

[54] AUTOSAR, "Specification of Secure Onboard Communication," Classic Platform R24-11 654, AUTOSAR, 2024.

[55] Visach, Virjjn, and Vishor, "Flash Bootloader OEM – Vector UDS (SLP3)," Technical Reference Version 5.2.1, Vector Informatik, 2024.

[56] M. Wolf, A. Weimerskirch, and C. Paar, "Secure In-Vehicle Communication," in *Embedded Security in Cars: Securing Current and Future Automotive IT Applications* (K. Lemke, C. Paar, and M. Wolf, eds.), pp. 95–109, Berlin, Heidelberg: Springer, 2006.

[57] A. Groce, D. Peled, and M. Yannakakis, "Adaptive Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems* (J.-P. Katoen and P. Stevens, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 357–370, Springer, 2002.

[58] J. Shijubo, M. Waga, and K. Suenaga, "Efficient Black-Box Checking via Model Checking with Strengthened Specifications," in *Runtime Verification* (L. Feng and D. Fisman, eds.), (Cham), pp. 100–120, Springer International Publishing, 2021.

[59] D. Neider and N. Jansen, "Regular Model Checking Using Solver Technologies and Automata Learning," in *NASA Formal Methods* (G. Brat, N. Rungta, and A. Venet, eds.), (Berlin, Heidelberg), pp. 16–31, Springer, 2013.

[60] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining Model Learning and Model Checking to Analyze TCP Implementations," in *Computer Aided Verification* (S. Chaudhuri and A. Farzan, eds.), (Cham), pp. 454–471, Springer International Publishing, 2016.

[61] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of SSH implementations," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, (New York, NY, USA), pp. 142–151, Association for Computing Machinery, July 2017.

[62] M. Schuts, J. Hooman, and F. Vaandrager, "Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial

Experience Report," in *Integrated Formal Methods* (E. Ábrahám and M. Huisman, eds.), (Cham), pp. 311–325, Springer International Publishing, 2016.

[63] S. Marksteiner, D. Schögler, M. Sirjani, and M. Sjödin, "Learning single and compound-protocol automata and checking behavioral equivalences," *International Journal on Software Tools for Technology Transfer*, vol. 27, pp. 35–52, Feb. 2025.

[64] F. Vaandrager, R. Bloem, and M. Ebrahimi, "Learning Mealy Machines with One Timer," in *Language and Automata Theory and Applications* (A. Leporati, C. Martín-Vide, D. Shapira, and C. Zandron, eds.), Lecture Notes in Computer Science, (Cham), pp. 157–170, Springer International Publishing, 2021.

[65] D. Schögler, "An Automata Learning Framework for Bluetooth Low Energy," Master's thesis, Graz University of Technology, Graz, Austria, 2023.