Dipl.-Ing. Andrea Pferscher, BSc

Automata Learning for Security Testing and Analysis in Networked Environments

DOCTORAL THESIS

to achieve the university degree of

Doktorin der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig
Institute of Software Technology (IST)

External Reviewer and Examiner
Prof. Dr. Martin Leucker
Institute for Software Engineering and Programming Languages
University of Lübeck

Graz, August 2023

Dipl.-Ing. Andrea Pferscher, BSc

Automatenlernen für das Testen und die Analyse der Sicherheit in vernetzten Umgebungen

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der technischen Wissenschaften

eingereicht an der

Technischen Universität Graz

Betreuer
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Aichernig
Institut für Softwaretechnologie (IST)

Externer Gutachter und Prüfer
Prof. Dr. Martin Leucker
Institute for Software Engineering and Programming Languages
Universität zu Lübeck

Graz, August 2023

Diese Arbeit ist in englischer Sprache verfasst.

Abstract

The Internet of Things (IoT) connects billions of devices and with the development of new communication technologies, the number of connected devices is expected to further grow. It is essential that in these networked environments, none of the devices introduces severe security issues. The challenge is that these networks consist of many heterogeneous components, where the insight into these components is limited. Therefore, we require techniques that allow us to automatically test and analyze black-box components for security issues. The goal of this thesis is to provide a holistic evaluation of the suitability of automata learning techniques for the automatic testing and analysis of security-critical aspects in networked environments.

Automata learning proved to be a useful tool to provide insights into the behavior of black-box systems. This thesis shows how automata learning can support the security analysis of networked environments from three perspectives: First, we evaluate the feasibility of automata learning in practice by learning behavioral models of communication protocol implementations. Second, we propose new learning techniques to overcome practical challenges such as non-deterministic behavior or multi-client communication. Third, we introduce a stateful black-box security testing technique that is based on automata learning.

Our evaluation considers popular communication protocols in the IoT such as Bluetooth Low Energy or the publish/subscribe protocol MQTT. Furthermore, we take into account the security-critical key exchange protocol IPsec-IKEv1 that is used in virtual private networks to establish secure encrypted communication. We present behavioral models of implementations for each of these protocols, where the learned behavioral models already show violations of the corresponding protocol specifications. To overcome the challenges experienced during learning, we propose new learning techniques. We explore that recurrent neural networks can be utilized to learn behavioral models on a given sample. Furthermore, we introduce a novel learning algorithm that learns an abstracted model of non-deterministic systems.

This thesis also introduces learning-based fuzzing as a new security testing technique that combines automata learning and fuzz testing. The symbiosis of these two techniques results in a stateful black-box fuzzing framework. We researched different strategies on how this technique can be applied in practice. Our results show that learning-based fuzzing reveals specification violations, and security and reliability issues in all investigated communication protocols.

Keywords: Automata Learning, Active Automata Learning, Passive Automata Learning, Fuzz Testing, Model-based Testing, Machine Learning, Recurrent Neural Networks, Bluetooth Low Energy, MQTT, Virtual Private Network.

Kurzfassung

Das Internet der Dinge (IdD) verbindet Milliarden an Geräten, und mit der Entwicklung neuer Kommunikationstechnologien wird die Anzahl an verbundenen Geräten voraussichtlich weiter ansteigen. In vernetzten Umgebungen ist es unerlässlich, dass keine Komponente Sicherheitslücken darstellt. Solche Netzwerke beinhalten die Herausforderung, dass sie unterschiedlichste Komponenten vereinen, bei denen der Einblick zu internen Strukturen limitiert ist. Aus diesem Grund benötigen wir Techniken, die uns eine automatische Analyse von Black-Box-Komponenten ermöglichen. Das Ziel dieser Arbeit ist eine ganzheitliche Evaluierung bezüglich der Eignung von Automatenlerntechniken für das die automatische Testung und Analyse sicherheitskritischer Aspekte in vernetzten Umgebungen vorzunehmen.

Automatenlernen hat sich als nützliches Werkzeug erwiesen, um Einblicke in das Verhalten von Black-Box-Systemen zu gewinnen. Diese Arbeit zeigt anhand von drei Perspektiven wie Automatenlernen die Sicherheitsanalyse in vernetzten Umgebungen unterstützen kann: Erstens evaluieren wir die Durchführbarkeit von Automatenlernen in der Praxis, indem wir Verhaltensmodelle von Implementierungen von Kommunikationsprotokollen lernen. Zweitens schlagen wir neue Lerntechnologien vor um Herausforderungen aus der Praxis wie nicht-deterministisches Verhalten und Multi-Client-Verbindungen zu berücksichtigen. Drittens führen wir eine zustandsbasierte Sicherheitstesttechnik ein, welche auf Automatenlernen für Black-Box-Systeme basiert.

Unsere Evaluierung betrachtet populäre Kommunikationsprotokolle im IdD wie Bluetooth Low Energy oder das Publish/Subscribe-Protokoll MQTT. Darüber hinaus berücksichtigen wir das sicherheitskritische Schlüsselaustauschprotokoll IPsec-IKEv1, das in virtuellen privaten Netzwerken verwendet wird, um eine sichere verschlüsselte Kommunikation aufzubauen. Wir präsentieren Verhaltensmodelle für jedes dieser Protokolle, welche bereits auf Verletzungen der entsprechenden Protokollspezifikation hinweisen. Um die Herausforderungen des Lernens zu überwinden, schlagen wir neue Lerntechniken vor. Wir entdecken, dass rekurrente neuronale Netzwerke dazu verwendet werden können, Verhaltensmodelle von gegebenen Daten zu lernen. Zusätzlich stellen wir einen neuen Lernalgorithmus vor, der ein abstrahiertes Modell von nicht-deterministischen Systemen lernt.

Diese Arbeit stellt auch lernbasiertes Fuzz-Testen als neue Sicherheitstesttechnik vor, die Automatenlernen und Fuzz-Testen kombiniert. Durch die Symbiose von Automatenlernen und Fuzzing entsteht ein zustandsorientiertes Fuzz-Test-Framework für Black-Box-Systeme. Wir präsentieren unterschiedliche Strategien, wie diese Technik in der Praxis angewendet werden kann. Unsere Ergebnisse zeigen, dass lernbasiertes Fuzz-Testen Spezifikationsverletzungen sowie Sicherheits- und Zuverlässigkeitsprobleme in allen untersuchten Kommunikationsprotokollen aufdeckt.

Schlagworte: Automatenlernen, Aktives Automatenlernen, Passives Automatenlernen, Fuzz-Testen, Modellbasiertes Testen, Maschinelles Lernen, Rekurrentes Neuronales Netz, Bluetooth Low Energy, MQTT, Virtuelles Privates Netzwerk.

Acknowledgements

I thank my supervisor Bernhard Aichernig, who introduced me to the topic of automata learning. I thank him for the support during the PhD, for the discussions about new ideas, and especially for the motivation to get the maximum out of it. I also thank Martin Leucker for reviewing this thesis and for serving as my external examiner.

I am also grateful for the helpful comments and editing proposals of this thesis by Martin Tappler, Florian Schwarz, Benjamin von Berg, Felix Wallner, and Benjamin Wunderling. Great thanks also go to Edi Muškardin, Maximilian Rindler, Konstantin Windisch, and Benjamin Wunderling, who supported this thesis with their Bachelor's and Master's theses, as well as master projects.

I am also grateful for the support of my colleagues, especially within our group consisting of Bernhard Aichernig, Edi Muškardin, Martin Tappler, Benjamin von Berg, and Felix Wallner. The discussions helped a lot to create new ideas and to highlight new aspects. I would also like to thank all my colleagues from the projects in which I participated for the good cooperation.

I would especially like to thank my partner, Florian, who has always stood by me throughout my PhD. Our time in our PhDs had ups and downs. Thank you for sticking together in the moments of stress and frustration, and for celebrating the successes with each other. I am also thankful for the sympathetic ears I have had from my friends and for sharing in the joy of successes achieved. Great thanks go to my family and especially to my parents, Helga and Josef, who have supported me unquestioningly on my journey through life and education. I thank them for their confidence in me that I will make it. Special thanks to my sister Martina and my brother-in-law Martin, who have always followed my achievements with great enthusiasm and joy.

This thesis is funded by the TU Graz LEAD project with the title "Dependable Internet of Things in Adverse Environments", by the *LearnTwins* project (No 880852) from the Austrian Research Promotion Agency (FFG), and by *AIDOaRt* project (grant agreement No 101007350) from the ECSEL Joint Undertaking (JU). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Sweden, Austria, Czech Republic, Finland, France, Italy, and Spain.

Andrea Pferscher
Graz, Austria, August 2023

Danksagung

Ich danke meinem Betreuer Bernhard Aichernig, der mir das Thema Automatenlernen nähergebracht hat. Ich danke für die Unterstützung während des Doktorats, für die Diskussionen über neue Ideen, und vor allem für die Motivation das Maximum herauszuholen. Ich danke außerdem Martin Leucker für die Begutachtung dieser Dissertation und für die Übernahme der Rolle als externer Prüfer.

Ich bin außerdem dankbar für die hilfreichen Kommentare und das Lektorat dieser Arbeit von Martin Tappler, Florian Schwarz, Benjamin von Berg, Felix Wallner und Benjamin Wunderling. Großer Dank gilt auch Edi Muškardin, Maximilian Rindler, Konstantin Windisch, und Benjamin Wunderling, die mit ihren Bachelor- und Masterarbeiten, sowie Masterprojekten diese Dissertation unterstützt haben.

Ich bin auch dankbar für die Unterstützung durch meine Kolleginnen und Kollegen, vor allem innerhalb unserer Gruppe bestehend aus Bernhard Aichernig, Edi Muškardin, Martin Tappler, Benjamin von Berg, und Felix Wallner. Die Diskussionen haben viel dazu beigetragen neue Ideen zu kreieren und neue Aspekte zu beleuchten. Ich danke außerdem allen Kolleginnen und Kollegen aus den Projekten, in denen ich mitgewirkt habe, für die gute Zusammenarbeit.

Ich danke vor allem meinem Partner, Florian, der mir während des gesamten Doktorats immer zur Seite gestanden ist. Unsere Zeit im Doktorat hatte Höhen und Tiefen. Danke, dass wir in den in den Momenten des Stresses und Frustration zusammengehalten haben und dass wir die Erfolge miteinander gefeiert haben. Ich bin auch dankbar für die offenen Ohren bei meinen Freundinnen und Freunden und für die Mitfreude bei erreichten Erfolgen. Großer Dank gilt meiner Familie und vor allem meinen Eltern, Helga und Josef, die mich auf meinem Lebens- und Bildungsweg bedingungslos unterstützt haben. Ich danke ihnen für das Vertrauen in mich das ich diesen Weg schaffen werde. Ein spezieller Dank gilt meiner Schwester Martina und meinem Schwager Martin, die immer mit großer Begeisterung und Freude meine Leistungen verfolgt haben.

Diese Arbeit wurde durch das TU Graz LEAD Projekt unter dem Titel *"Dependable Internet of Things in Adverse Environments"*, durch das Projekt *LearnTwins* Projekt (Nr. 880852) von der Österreichischen Forschungsförderungsgesellschaft (FFG), und durch das Project *AIDOaRt* (Zuwendungsvereinbarung Nr. 101007350) von dem ECSEL Joint Untertaking (JU) gefördert. Das JU erhält Unterstützung von dem Forschungs- und Innovationsprogramm Horizon 2020 der Europäischen Union und von Schweden, Österreich, Tschechien, Finnland, Frankreich, Italien, und Spanien.

Andrea Pferscher
Graz, Österreich, August 2023

Contents

List of Figures

List of Tables

List of Algorithms

List of Grammars

Abbreviations

BLE Bluetooth Low Energy.

BNF Backus-Naur form.

CPS cyber-physical system.

DFA deterministic finite automaton.

DTLS Datagram Transport Layer Security.

FSM finite state machine.

HTTP Hypertext Transfer Protocol.

IKE Internet Key Exchange.

IoT Internet of Things.

IPsec Internet Protocol Security.

ISAKMP Internet Security Association and Key Management Protocol.

MAT minimally adequate teacher.

MQTT Message Queuing Telemetry Transport.

MTU maximum transmission unit.

ONFSM observable non-deterministic finite state machine.

PAC probably approximately correct.

PSK pre-shared key.

PTA prefix-tree acceptor.

RERS Rigorous Examination of Reactive Systems.

RNN recurrent neural network.

RPC Remote Procedure Call.

RPNI Regular Positive Negative Inference.

SA Security Association.

SIG Special Interest Group.

SIP Session Initiation Protocol.

SMB Server Message Block.

SMTP Simple Mail Transfer Protocol.

SoC system on the chip.

SSL Secure Sockets Layer.

SUL system under learning.

SUT system under test.

TCP Transmission Control Protocol.

TFTP Trivial File Transfer Protocol.

TLS Transport Layer Security.

VM virtual machine.

VPN Virtual Private Network.

Chapter 1

Introduction

1.1 Motivation

In 2017, Texas Instruments published an article illustrating different use case scenarios of Bluetooth Low Energy (BLE) in an automotive system. The article advertises BLE as a technology to replace wired communication with wireless communication inside a vehicle. For example, BLE can be used to collect sensor data such as tire pressure, but also for accessing the car via a smartphone or key fob. In 2022, the NCC Group published an article [78] demonstrating that BLE can be exploited for relay attacks. In a relay attack, an attacker simulates trusted devices by relaying intercepted messages. This leads to security issues, as cars that incorrectly use BLE features for vehicle access can be opened without proper access. This exploit demonstrates that such an attack can be successfully executed on the Tesla Model 3 and Model Y. Such examples show that the misuse of BLE can cause severe security vulnerabilities.

To avoid scenarios such as the unprivileged access of a car, we require technologies that allow us to detect and mitigate issues in network environments. With the growth of the Internet of Things (IoT), this becomes a major challenge. The IoT connects billions of heterogeneous devices and with the development of new communication technologies such as 5G, the number of connected devices will grow even further [199]. The IoT comprises applications, e.g., for smart healthcare systems, smart homes, or smart energy grids [205]. In all of these applications, it is essential that they do not compromise the safety or security of the users.

Market predictions of the Bluetooth Special Interest Group project that especially the number of peripheral devices will grow [88]. For peripheral devices in particular, we do not have insights into the internal behavior of the devices. Thus, the devices appear as black boxes, where we can only execute inputs and observe the corresponding outputs. To ensure the safety and security of a user, we require techniques that automatically verify and test black-box systems.

This thesis proposes techniques that provide insights into the behavior of black-box components in networked environments. Furthermore, we extend these techniques to generate stateful security-testing techniques for black-box systems.

1.2 Automata Learning

Behavioral models are a useful tool to analyze systems. Models support techniques such as model-based testing [8, 30] or model-based verification [41]. Furthermore, they help to create a common understanding of complex systems. In practice, however, models are not always available. Creating models manually can be a tedious and error-prone process. Our goal is therefore to generate models automatically.

Automata learning is a technique to automatically generate behavioral models of black-box systems. The origins of automata learning lie in the work of Gold [73]. His work approaches the

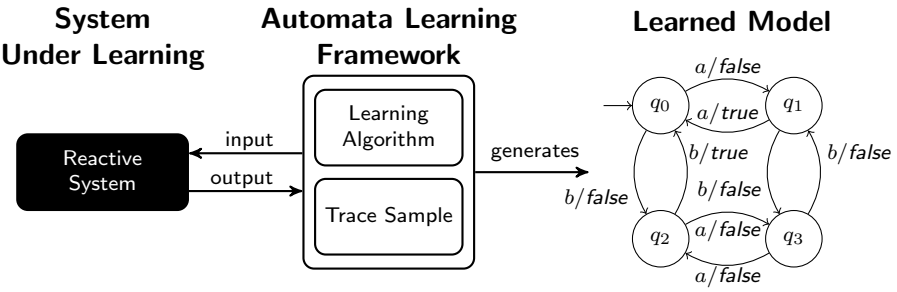


Figure 1.1: General automata learning framework for learning behavioral models of black-box systems. The figure is taken from Aichernig et al. [12].

NP-complete problem of identifying a finite state model with at most n states from a given set of data. Starting from this problem, the whole research area of automata learning has emerged.

Figure 1.1 illustrates the concept of automata learning for learning models of reactive systems. Automata learning allows us to learn a behavioral model of a black-box system. In this thesis, we learn behavioral models of communication protocol implementations. Communication protocols can be modeled as reactive systems, where we can execute inputs and observe outputs. Thus, we consider our system under learning (SUL) to be a reactive system. Based on a sample of input/output traces, the automata learning algorithm generates a behavioral model that represents the SUL. Depending on the origins of the sample, we distinguish between passive and active techniques. Passive learning creates a behavioral model from a given set of traces, whereas active learning actively queries the SUL to generate the sample.

This thesis considers both learning paradigms and evaluates their applicability for learning network protocols. We discuss challenges in learning real-world systems and propose techniques to overcome them. We present improvements for existing learning approaches but also introduce new learning techniques. Our proposed techniques also consider behavioral aspects such as nondeterministic observations. Besides classical learning approaches, we investigate the suitability of machine learning techniques for automata learning.

1.3 Fuzz Testing

Fuzz testing, frequently abbreviated as fuzzing, is a security testing technique that executes random inputs on the system under test (SUT). The random inputs include valid inputs as well as unusual and invalid inputs. The goal of fuzzing is to reveal unexpected behavior, where the unexpected behavior hints at security issues.

Miller et al. [119] were the first to call their tool for random testing a *fuzzer*. Their fuzzer successfully reveals reliability issues such as crash scenarios in UNIX utilities. Nowadays, fuzzing has proven itself as a popular security testing technique since it achieves a high success rate in finding vulnerabilities despite its ease of use. Fuzzers like AFL [201] found bugs in operating system libraries, web browsers, or even media players.

Fuzzing techniques are usually categorized based on the level of access to the SUT. The literature [71] usually categorizes fuzzing techniques into three categories: white-box, gray-box, and black-box fuzzing. White-box fuzzers require access to the source code, while black-box fuzzers can only execute inputs and observe outputs. Everything in between belongs to gray-box techniques.

This thesis considers black-box fuzzing techniques since components in network systems are often closed-source. The challenge in black-box fuzzing is to measure coverage and to determine which parts of the system have been fuzzed. Our proposed techniques overcome this problem, by considering an underlying behavioral model of the black-box system. This thesis discusses

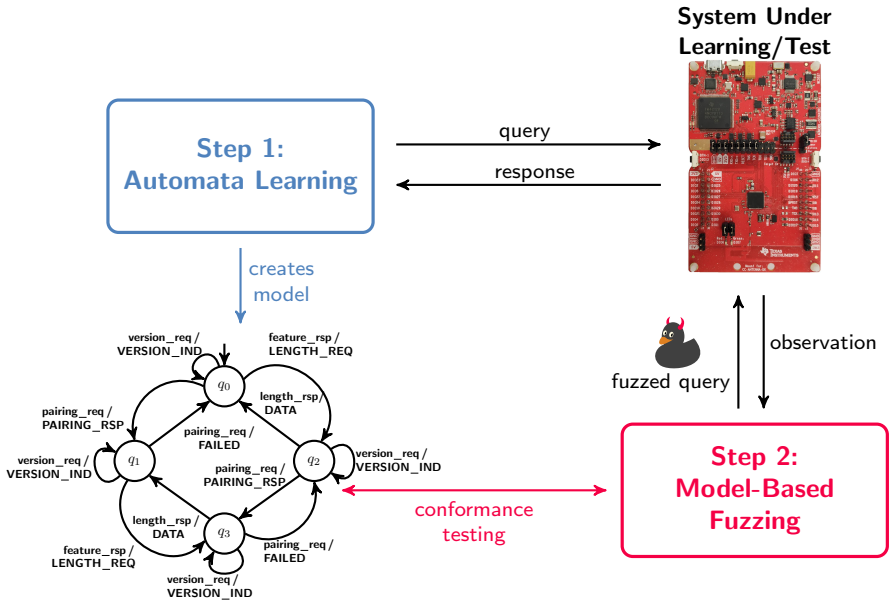


Figure 1.2: Two-step procedure of our learning-based fuzzing concept.

different model-based fuzzing techniques for communication protocols.

1.4 Protocol State Fuzzing

Protocol state fuzzing can be seen as a learning-based testing technique [8], where active automata learning is used to test communication protocol implementations. In contrast to other learning-based testing techniques for communication protocols such as for Message Queuing Telemetry Transport (MQTT) [170], protocol state fuzzing focuses more on testing security-critical aspects of communication protocols.

Protocol state fuzzing has successfully been applied to find issues in different communication protocols such as Transport Layer Security (TLS) [50], Secure Sockets Layer (SSL)/TLS [163], Datagram Transport Layer Security (DTLS) [63] and OpenVPN [47]. Protocol state fuzzing is considered a fuzzing technique since active automata learning executes inputs in states where the inputs might be unexpected in order to explore the state space. The goal of this technique is to reveal unexpected state transitions or states.

In this thesis, we will apply protocol state fuzzing techniques to different communication protocols. Our results show that the learned models demonstrate unexpected behaviors that violate the corresponding protocol specifications. The drawback of protocol state fuzzing is that simply executing inputs in an unexpected order may not be enough to detect unexpected behavior. Thus, we consider advanced fuzzing techniques to test for reliability issues and other unexpected behaviors that may be difficult to detect during the learning process.

1.5 Learning-Based Fuzzing

In this thesis, we introduce the concept of learning-based fuzzing. Learning-based fuzzing combines automata learning and model-based fuzzing techniques. Figure 1.2 depicts the learning-based fuzzing approach, which can be described in a two-step procedure. In the first step, we apply automata learning to learn a behavioral model of a black-box system. In the second step, we use the learned model as a basis for our model-based fuzzing technique. Our model-based fuzzing technique tests the conformance between the SUT and the learned model. The test cases

are generated using fuzzing techniques. Thus, they include unexpected and invalid inputs. The goal of our model-based fuzzer is to reveal behavioral differences between the behavioral model and the SUT.

This thesis introduces different learning-based fuzzing techniques. The proposed techniques are evaluated on popular communication protocols such as Message Queuing Telemetry Transport (MQTT), Bluetooth Low Energy (BLE), or the IPsec Internet Key Exchange (IPsec-IKEv1) protocol that is used in VPN servers. We present that our proposed techniques successfully reveal security issues in the implementations of all investigated protocols.

1.6 Problem Statement

This thesis provides a holistic evaluation of the suitability of automata learning for analyzing and testing security-critical aspects of communication protocols. Our evaluation approaches this problem from three different angles, formulated in three corresponding research questions.

First, we investigate whether active automata learning is suitable to learn behavioral models of real network components. If automata learning should be considered for real-world security analysis, it is important that automata learning works not only for simulated network environments, but also for learning behavioral models of protocol implementations on physical devices. Additionally, networked environments frequently consist of multiple clients interacting with each other. We also investigate if automata learning scales for such real-world scenarios. Overall, we approach the problem of whether automata learning is sufficient to learn expressive models of security-critical procedures. The challenge in this regard is to find an appropriate level of abstraction that enables learning but still provides an in-depth exploration of security-relevant aspects. We summarize these problems in research question **RQ 1**.

- **(RQ 1)** What are the challenges of learning behavioral models in networked systems?
  - **(RQ 1.1)** Does active automata learning perform well for learning communication protocol implementations on physical devices?
  - **(RQ 1.2)** Is automata learning useful to learn security-critical behavior?

Second, we evaluate alternatives for common learning techniques in order to improve the practical feasibility of automata learning. This problem can be considered as a follow-up question that addresses the findings obtained for **RQ 1**. For this purpose, we compare the two learning paradigms, active and passive learning, in terms of the effort required to learn adequate models of network protocols. Our evaluation not only considers passive learning algorithms from the literature. We also approach the question of whether machine-learning-based learning techniques represent an alternative to standard automata learning algorithms. Furthermore, we discuss if additional abstraction techniques enable the learning of large network systems, where we suspect that these systems do not always behave deterministically. All these techniques and challenges are discussed in **RQ 2**.

- **(RQ 2)** How can automata learning be improved for practical applications?
  - **(RQ 2.1)** Does passive learning represent an alternative to active learning?
  - **(RQ 2.2)** How to improve automata learning to make it feasible for different challenges in networked environments?

Third, we evaluate the feasibility of automata learning to support popular security testing techniques. To answer this question, we mainly focus on fuzz testing as a security testing technique. We investigate how automata learning can support fuzz testing, especially for black-box systems often found in networked environments. Furthermore, we evaluate if the presented

techniques are actually effective to uncover security vulnerabilities. In addition to fuzzing, we also approach the question if automata learning can be used to fingerprint black-box systems, which creates the risks to exploit system-specific vulnerabilities. **RQ 3** summarizes the aspects of this part of our evaluation.

- **(RQ 3)** Can automata learning support security testing techniques?
  - **(RQ 3.1)** How can black-box fuzzing techniques be extended with automata learning?
  - **(RQ 3.2)** Is learning-based fuzzing effective at revealing security issues?
  - **(RQ 3.3)** Can automata learning be used to fingerprint black-box devices?

1.7 Research Projects

The work performed toward this thesis is part of three different research projects: (1) Dependable Internet of Things in Adverse Environment, (2) LearnTwins, and (3) AIDOaRt. In the following, we describe the scope of the research projects and outline how the methods presented in this thesis contribute to the individual projects.

1.7.1 Dependable Internet of Things in Adverse Environments

The *Dependable Internet of Things in Adverse Environments* project was a university internal research project that unites researchers from the faculties of Computer Science and Biomedical Engineering, and Electrical and Information Engineering of Graz University of Technology. The project lasted from 2016 to 2022 and was divided into two main project phases. This thesis belongs to the second phase of the project, which lasted from 2019 to 2022. The goal of the project was to improve the level of dependability in the IoT in adverse environments. The project focused on foundational research in order to develop methods to make the IoT more reliable, secure and safe.

In the first phase of the project, my predecessor Martin Tappler applied automata learning for testing purposes in networked environments. To cope with environmental conditions in the IoT, his PhD thesis [169] proposes learning techniques for timed and stochastic behavior. I performed my Master's thesis [143] within the first phase of the project, proposing a metaheuristic-search-based learning algorithm for timed systems.

In the second phase of the project, the goal of the subproject to which this thesis belongs was to apply and improve the techniques of the first phase for security-critical environments. Thus, the goal of this thesis was to apply automata learning techniques to investigate security-critical aspects in the IoT. For this purpose, the thesis proposes learning techniques for communication protocols that are frequently used in the IoT such as MQTT and BLE. The thesis shows not only that automata learning can be used to learn behavioral models of physical devices, but also that automata learning can be extended to test the security-critical behavior of black-box systems.

1.7.2 LearnTwins

The *LearnTwins* project is a nationally-funded project between two academic partners, the Austrian Institute of Technology and the Graz University of Technology, and one industrial partner, the AVL List GmbH. The goal of the project is to develop methods to automatically generate digital twins of cyber-physical systems (CPSs). A digital twin is a virtual representation of a real-world system that simulates the behavior of the twinned system. Digital twins are useful since they provide insights into twinned systems, without requiring access to the real system.

Within this thesis, we investigated techniques to learn digital twins. A behavioral model of a system can be seen as a digital twin. Thus, automata learning represents a tool to automatically

learn digital twins. The project aims to compare and develop automata learning techniques for learning digital twins. This thesis presents machine-learning-based techniques to infer behavioral models that were developed together with our academic partner. Furthermore, based on case studies provided by our industrial partner, this thesis provides methods to learn security-critical components in automotive systems.

1.7.3 AIDOaRt

The *AIDOaRt* project is a European project including 31 partners from academia and industry of seven different countries. The goal of the project is to employ techniques based on artificial intelligence to test, model, monitor and develop CPSs.

Within this project, we collaborate closely with our national industrial partner AVL List GmbH. The goal of the collaboration is to develop security testing techniques for automotive components. This thesis contributes to this project goal by proposing learning-based fuzzing techniques. This research effort has been rewarded within the project. **Our learning-based fuzzing technique won the third edition of the project-internal hackathon**. Our presented hackathon challenges demonstrate how learning-based fuzzing can be used to learn and test the BLE interface of Automotive Grade Linux (AGL), where AGL is an open-source operating system for automotive software components.

1.8 Contributions and Publications

In the following, we summarize the contributions of this work. The list of contributions is supplemented by a list of main publications that form the basis for the content presented in this work. For all these main publications, I am considered a main author. For each of these publications, a statement of my actual contributions is included. In addition, this section enumerates the co-supervised Bachelor's and Master's theses and declares if they are part of this thesis. As a final part, this section provides a list of related publications for which either I was not the main author or the topic covered is out-of-scope for this thesis.

1.8.1 Contributions

- We demonstrate the practical applicability of automata learning by presenting a learning framework that learns behavioral models of physical black-box systems. The case studies also include the learning of components that are installed in a real car.
- We present that automata learning can be used to learn security-critical protocols such as the exchange of an encryption key.
- We report that exhaustive querying in active automata learning reveals security vulnerabilities and reliability issues in security-critical protocols.
- Our presented learned models of the considered communication protocols demonstrate violations of the corresponding protocol specification.
- Our learned models present differences in the communication protocol implementations. We show how these behavioral differences can be exploited to fingerprint the implementations.
- We compare the two main learning paradigms, active and passive learning, for their effectiveness in learning communication protocols.
- We extend state-of-the-art learning libraries by active automata learning algorithms that are more efficient in the number of performed queries.

- We introduce an abstraction technique that enables the efficient learning of non-deterministic systems.
- We present a passive learning technique that uses a constrained training technique for a recurrent neural network (RNN) model in order to learn a minimal finite state machine.
- We introduce a stateful black-box fuzzing concept based on automata learning and model-based fuzzing.
- We present different learning-based fuzzing techniques and evaluated their effectiveness in testing popular communication protocols. Our technique reveals several reliability issues and specification violations.
- We are committed to making our research and tools available to the public. The following frameworks are available online.
  - Active automata learning framework for Bluetooth Low Energy (BLE): [144]
  - Evaluation framework for the comparison of active and passive automata learning algorithms: [127]
  - Learning-based fuzzing framework for MQTT: [126]
  - Learning-based fuzzing framework for BLE: [145]
  - The improved version of the learning algorithm of Kearns and Vazirani (KV) and the improved version of the learning algorithm for abstracted non-deterministic systems are integrated into the public learning library AALpy [129]

1.8.2 Main Publications

1. ICTSS 2020: "*Learning Abstracted Non-deterministic Finite State Machines*" [146]. This paper presents an active learning algorithm for learning abstracted observable non-deterministic finite state machines (ONFSMs). The presented algorithm is evaluated on a case study on learning MQTT brokers interacting with multiple clients. The algorithm and the case study are presented in Chapter 8.

   I presented the paper at the virtual event of the ICTSS 2020 and the paper is published in the corresponding conference proceedings. I designed the algorithm, implemented the algorithm in Scala, set up the case study environment including all MQTT brokers and executed all reported experiments. I wrote all parts of the paper under the supervision of Bernhard Aichernig.

2. ICST 2021: "*Learning-Based Fuzzing of IoT Message Brokers*" [11]. The paper presents the learning-based fuzzing framework for fuzzing MQTT brokers. Chapter 9 discusses the grammar-based fuzzing technique and Chapter 10 provides the results of applying this technique on MQTT brokers.

   I presented the paper at the virtual event of ICST 2021. The paper was published in the corresponding proceedings. I implemented the learning interface. Edi Muškardin implemented the fuzzing component during his Master's project which I co-supervised. The idea to use grammar-based fuzzing came from my side. The paper was written in collaboration with Edi Muškardin and Bernhard Aichernig. The initial version was mostly written by myself except for some paragraphs in the experimental setup sections.

3. FM 2021: "*Fingerprinting Bluetooth Low Energy Devices via Active Automata Learning*" [147]. The paper presents the active automata learning case study on learning BLE devices. In addition, the paper discusses that active automata can be used to fingerprint black-box systems. Chapter 4 discusses the paper in detail.

I presented the paper at the virtual event of FM 2021. The paper was part of the corresponding proceedings. The setup of the learning interface and the Bluetooth Low Energy (BLE) devices, the conduction of the case study, and the writing of the initial version of the paper were done by myself. Under the supervision of Bernhard Aichernig, I revised the initial paper draft.

4. NFM 2022: "*Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning*" [148]. The paper introduces a learning-based fuzzing framework for BLE devices. The conducted case study revealed several issues in the investigated BLE devices. The fuzzing technique is presented in Chapter 9 and the corresponding case study in Chapter 10.

   I presented the paper at NFM 2022 in Pasadena, California. The paper was part of the corresponding conference proceedings. I designed the stateful fuzzing approach and implemented it. I conducted the case study and wrote the paper. I revised the paper according to the comments of Bernhard Aichernig.

5. FMAS 2022: "*Active vs. Passive: A Comparison of Automata Learning Paradigms for Network Protocols*" [13]. The paper compares active and passive automata learning techniques on their effectiveness to learn communication protocols. Chapter 6 presents the performed evaluation.

   I presented the paper at the workshop FMAS 2022 in Berlin, Germany. The paper is part of the workshop proceedings. The paper was written in collaboration with Edi Muškardin and Bernhard Aichernig. Edi Muškardin and I collaborated to set up the evaluation framework. I conducted the experiments except for the heatmap experiments. I wrote the majority of the paper, except for some paragraphs for the experimental setup.

6. SEFM 2022: "*Constrained Training of Recurrent Neural Networks for Automata Learning*" [12]. The paper introduces a passive automata learning approach that is based on training an RNN. Chapter 7 introduces the learning approach and presents the performed evaluation.

   I presented the paper at SEFM 2022 in Berlin, Germany. The paper is part of the respective conference proceedings. The work was done in close collaboration with Bernhard Aichernig, Sandra König, Cristinel Mateis, Dominik Schmidt, and Martin Tappler. I analyzed the early results of the developed approach and presented ideas on how to adapt the constrained training. For the evaluation, I generated the BLE data and all the active automata learning data for the case study. I wrote parts of the paper that affect my previous contributions and revised it according to the reviewer's comments.

7. FMDT 2023: "*Mining Digital Twins of a VPN Server*" [150]. The paper describes the active learning setup for learning behavioral models of IPsec IKEv1 protocol implementations. The learning setup and the learned models are presented in Chapter 5.

   I presented the paper at the workshop FMDT 2023 in Lübeck, Germany. The paper was published in the pre-proceedings of the workshop. A version for the post-proceedings has been submitted. The learning setup and experimental evaluation were performed within the Master's project of Benjamin Wunderling which I co-supervised. I advised on the active learning setup. I also extended the learning library AALpy by a version of the learning algorithm proposed by Kearns and Vazirani [96], which was applied in the learning setup. The paper was written in collaboration with Benjamin Wunderling. I wrote the version for the post-proceedings, taking into account the discussion at the workshop.

8. FMSD 2023: "*Fingerprinting and Analysis of Bluetooth Devices with Automata Learning*" [149]. This article presents an extended version of the FM 2021 paper [147]. The

article extends the paper by evaluating additional BLE devices and by a learning interface that enables the learning of the BLE pairing procedure. The extension is included in Chapter 4.

I set up the learning interface and conducted all experiments. The article was written by myself under the supervision of Bernhard Aichernig. I included the initial comments from the reviewers in a major revision of this article.

9. SoSyM: "*Learning Minimal Automata with Recurrent Neural Network*" [14] (under review, submitted February 2023). This article presents an extended version of the SEFM 2022 paper [12]. The extension includes an iterative approach that enables the learning of minimal automata with our RNN-based learning technique, even if the minimal number of states is unknown. Chapter 7 describes the performed extensions.

   The article was written in collaboration with Bernhard Aichernig, Sandra König, Cristinel Mateis, and Martin Tappler. I contributed to this article by proposing the idea of an iterative learning technique that incrementally decreases the maximum number of states, with the addition that minimization algorithms are used to further decrease the number of states. I described the initial version of the algorithm in the article. I submitted the article in February 2023.

1.8.3 Supervised Theses

1. Bachelor's thesis of Konstantin Windisch: "*Optimizations on Active Automata Learning for Observable Non-deterministic Finite State Machines*" [193]. The thesis discusses optimizations for the learning algorithms on ONFSMs implemented in the learning library AALpy [129]. The optimizations are presented in Chapter 8.

   The integration in the learning library AALpy of the learning algorithm that is presented in Chapter 8 was done by myself. I co-supervised the Bachelor's thesis of Konstantin Windisch in which the optimizations for the implementation in AALpy have been implemented. I advised the student on the learning algorithm, the implementation in AALpy, and the conducted evaluation. I reviewed the thesis and made suggestions for revision.

2. Bachelor's thesis of Maximilian Rindler: "*Implementing the Kearns-Vazirani Algorithm for Learning Deterministic Finite Automata in AALpy*" [155]. The Bachelor's thesis presents the implementation of the active automata learning algorithm proposed by Kearns and Vazirani [96] in the learning library AALpy. The thesis also proposes additional improvements for the learning algorithm. Chapter 3 discusses the implemented improvements.

   I co-supervised the Bachelor's thesis. I explained automata learning and the active learning algorithm. Furthermore, I suggested optimizations for the learning algorithm and proposed a setup for the evaluation. The implementation of the learning algorithm for deterministic finite automata (DFAs) built the basis for the extension to Mealy machines which was performed by myself. I reviewed the Bachelor's thesis and provided comments for the revision of the thesis.

3. Master's thesis of Benjamin Wunderling: "*Model Learning and Fuzzing of the IPsec-IKEv1 VPN Protocol*" [196] (not yet submitted). The Master's thesis presents that active automata learning can be used to learn protocols of VPN servers. Furthermore, the thesis discusses and applies learning-based fuzzing techniques. Chapter 5 presents the learning part, whereby the fuzzing approach is part of Chapter 9 and the corresponding case study is presented in Chapter 10.

   I co-supervised the Master's thesis. I advised on how active automata learning can be used to learn communication protocols and showed how to implement a mapper that can be

reused for fuzzing purposes. I recommended using a different learning algorithm in order to improve the performance. Furthermore, I extended the ideas for the applied fuzzing techniques. I reviewed the thesis and suggested revisions. The thesis is now submitted for review to Benjamin Wunderling's supervisor Bernhard Aichernig.

4. Bachelor's thesis of Valentina Wieser: "*Property-Based Testing of a Web Application using JavaScript*" [192]. The Bachelor's thesis describes the testing of REST APIs with the property-based testing framework FAST-CHECK[1].

   I co-supervised the Bachelor's thesis. I advised on the applied testing technique and on the properties that should be used. I reviewed the Bachelor's thesis and provided comments for the revision. The content of the Bachelor's thesis was not included in this thesis, since the topic of the Bachelor's thesis is out of scope.

1.8.4 Related and Other Publications

1. NFM 2020: "*From Passive to Active: Learning Timed Automata Efficiently*" [10]. The paper presents an active automata learning algorithm for learning timed automata. To overcome the large state space of timed systems, the algorithm applies a metaheuristic search. This paper presents an active version that implements a conformance testing technique for timed systems.

   I presented the paper at the virtual event of the NFM 2020. The main work was done as part of my Master's thesis [143], which was co-supervised by Martin Tappler. The content of my Master's thesis was part of the PhD thesis of Martin Tappler [169].

2. ATVA 2021: "*AALpy: An Active Automata Learning Library*" [128]. The tool paper presents the active automata learning library AALPY that is implemented in Python.

   The main contributor to this tool and paper was Edi Muškardin. I developed the active non-deterministic learning algorithms in the learning library and described the technique in the tool paper.

3. ISSE 2022: "*AALpy: an active automata learning library*" [129]. The article represents an extended version of the tool paper presenting AALPY [128].

   The main contributor to this article is Edi Muškardin. Similar, to the tool paper I contributed by writing the sections about learning of non-deterministic systems. Furthermore, I wrote the section on the BLE case study as an example of how AALPY can be used in practice.

4. FM 2023: "*A Systematic Approach to Automotive Security*" [55]. The paper proposes a methodology framework for the verification and validation of security risks in automotive systems.

   I contributed to this paper by advising on the usage of AALPY. Together with a subset of the authors of this paper, I set up a learning interface for an Electronic Control Unit (ECU) that uses the Unified Diagnostic Services (UDS) protocol. We learned a model that revealed a security vulnerability that is reported in the paper.

[1] https://github.com/dubzzz/fast-check

5. ICSE 2024: "*Learning and Repair of Deep Reinforcement Learning Policies from Fuzz-Testing Data*" [176] This paper introduces a reinforcement learning technique that benefits from demonstrations automatically generated using search-based fuzzing techniques. The paper also discusses that the presented technique can be used to repair existing reinforcement learning policies.

   This paper has just been accepted for ICSE 2024. I implemented the reinforcement learning technique and search-based fuzzing technique for the Minigrid experiments. I wrote parts of the paper describing the implemented fuzz testing technique and the corresponding sections on the Minigrid experiments.

1.9 Structure

The structure of the thesis is as follows. First, Chapter 2 provides background information that is required for the entire thesis. The background includes the formal definition of Mealy machines, which is the modeling formalism used in most of the case studies presented to describe the behavior of networked systems. We then provide background information on automata learning, including a discussion on active and passive automata learning algorithms from the literature that are applied in this thesis. As a last part, the background chapter introduces the three communication protocols that serve as case study subjects throughout the entire thesis: Message Queuing Telemetry Transport (MQTT), Bluetooth Low Energy (BLE), IPsec Internet Key Exchange (IPsec-IKEv1).

Chapter 3 discusses improvements for automata learning algorithms. This chapter provides an overview of algorithmic improvements such as counterexample processing and practical improvements such as caching data structures. We also present our improved version of the learning algorithm proposed by Kearns and Vazirani [96].

In Chapter 4, we present our case study on learning BLE devices. The chapter introduces the learning framework and discusses technical challenges and the achieved results on learning six different BLE devices. We also show how active automata learning can be used to fingerprint BLE devices. Furthermore, it provides an additional case study for learning the BLE devices used in a Tesla Model 3 and in the corresponding key fob to access the car.

Next, we present a case study on learning VPN servers in Chapter 5. More specifically, we learn the IPsec Internet Key Exchange (IPsec-IKEv1) protocol. Within the case study, we also compare two different learning algorithms: an improved L^* version vs. an improved version of the learning algorithm proposed by Kearns and Vazirani.

In Chapter 6, we approach the question of how well classic state-merging-based passive learning performs on learning communication protocols. This chapter also discusses the potential for improving the L^*-variant, which was used in most of the case studies in this thesis.

Next, we introduce a novel passive learning algorithm in Chapter 7. We train an RNN model to predict the behavior and the structure of a Mealy machine. For this purpose, we propose an RNN architecture which we trained using a specific constrained loss function. The chapter evaluates the proposed learning technique on a typical benchmark set for RNN-based automata learning approaches and on BLE device data.

Chapter 8 introduces a learning algorithm for learning abstractions of non-deterministic systems. First, the chapter provides some background on the considered modeling formalism for non-deterministic systems. Afterwards, we introduce our proposed learning framework and algorithm. We evaluated the presented algorithm in a case study learning a communication protocol in a multi-client setup.

In Chapter 9, we propose the concept of learning-based fuzzing and introduce several methods that apply this concept. First, we provide some background on fuzz testing. Next, we present the learning-based fuzzing framework. We show that learning-based fuzzing can be combined with different techniques such as grammar-based fuzzing or search-based testing techniques.

We evaluate all our presented learning-based fuzzing methods in Chapter 10. The chapter includes case studies on the MQTT, BLE, and IPsec-IKEv1 protocol. Our presented results show that our techniques are effective in finding issues in the protocol implementations.

Chapter 11 discusses related work on learning communication protocols, enhancements for automata learning algorithms, RNN-based learning techniques and learning-based fuzzing techniques as well as other fuzzing techniques for the investigated communication protocols. We conclude the thesis in Chapter 12 with a summary, a final discussion based on the proposed research questions, and an outlook on future work.

Chapter 2

Background

2.1 Mealy Machines

Mealy machines are a commonly used modeling formalism for reactive systems. Moreover, many automata learning algorithms [89, 113, 162] have been extended for learning Mealy machines. Georg H. Mealy [117] introduces Mealy machines as finite state machines whose transitions are labeled with inputs and outputs. Formally, we define Mealy machines as follows.

Definition 1 (Mealy machine) *A Mealy machine $\mathcal{M}$ is a 6-tuple $\langle Q, q_0, I, O, \delta, \lambda \rangle$, where*

- *Q is the finite set of states,*
- *$q_0 \in Q$ is the initial state,*
- *I is the finite set of inputs,*
- *O is the finite set of outputs,*
- *$\delta \colon Q \times I \to Q$ is the state transition function, and*
- *$\lambda \colon Q \times I \to O$ is the output function.*

We define δ and λ as total functions. Consequently, $\mathcal{M}$ formalizes deterministic behavior, and δ and λ are defined for any input in any state $q \in Q$. In this way, we ensure testability for the model-based testing techniques applied later.

Let $s \in (I \times O)^*$ be a sequence of alternating inputs and outputs, which we call a *trace*. Let $s^I \in I^*$ and $s^O \in O^*$ be the corresponding input and output sequences. We denote the non-empty trace as $s^+ \in (I \times O)^+$, with the non-empty input sequence $s^{I^+} \in I^+$ and output sequence $s^{O^+} \in O^+$. We write $s = (i_0, o_0) \cdot \ldots \cdot (i_j, o_j) \cdot \ldots \cdot (i_{n-1}, o_{n-1})$, where $j, n \in \mathbb{N}$ and $0 \leq j < n$. Let ϵ denote an empty sequence. The concatenation of two sequences $s, s' \in (I \times O)^*$ is denoted by $\cdot$, i.e., $s \cdot s'$. We write $|\, s \,|$ for the length of a sequence s, which is equal to the number of input/output pairs n in a sequence. We extend δ and λ for sequences. Let $\delta^* \colon Q \times I^* \to Q$ be the corresponding state transition function that takes an input sequence instead of a single input and $\lambda^* \colon Q \times I^* \to O^*$ be the output function returning the output sequence. Let $\lambda^*(q, \epsilon) = \epsilon$ and $\delta^*(q, \epsilon) = q$ hold. We recursively define that $\delta^*(q, i_0 \cdot \ldots \cdot i_n) = \delta^*(\delta(q, i_0), i_1 \cdot \ldots \cdot i_n)$ and $\lambda^*(q, i_0 \cdot \ldots \cdot i_n) = \lambda(q, i_0) \cdot \lambda^*(\delta(q, i_0), i_1 \cdot \ldots \cdot i_n)$. For the sake of simplicity, we abbreviate δ^* and λ^* for the initial state q_0 by $\delta^*(q_0, i_0 \cdot \ldots \cdot i_n) = \delta^*(i_0 \cdot \ldots \cdot i_n)$ and $\lambda^*(q_0, i_0 \cdot \ldots \cdot i_n) = \lambda^*(i_0 \cdot \ldots \cdot i_n)$. A sequence $s^I \in I^*$ is called *access sequence* of a state q, if the state transition performed from the initial state q_0 yields state q, i.e., $\delta^*(s^I) = q$ holds.

A Mealy machine $\mathcal{M}$ defines a language $L(\mathcal{M})$, where $L(\mathcal{M})$ includes all traces that are observable in $\mathcal{M}$. We define two Mealy machines to be observable equivalent if they implement

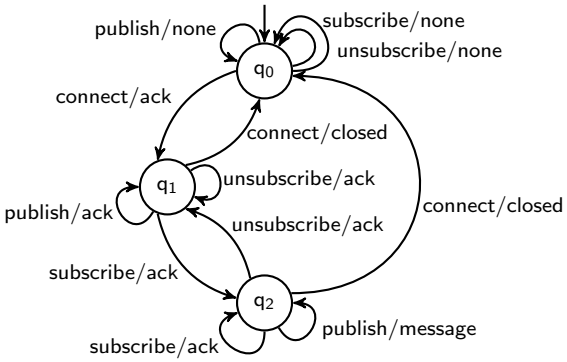


Figure 2.1: A Mealy machine representing a simple publish-subscribe protocol implementation.

the same language. In other words, two Mealy machines $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ and $\mathcal{M}' = \langle Q', q_0', I, O, \delta', \lambda' \rangle$ are equivalent iff no input sequence exists that produces a different output sequence. Thus, a counterexample to the equivalence between $\mathcal{M}$ and $\mathcal{M}'$ would be an input sequence s^I, where $\lambda^*(s^I) \neq \lambda'^*(s^I)$.

Example 1 (Mealy machine) *Figure 2.1 illustrates a Mealy machine that formalizes a simple implementation of a publish/subscribe protocol. In practice, an example of a publish/subscribe protocol in the IoT would be the MQTT protocol [22]. In this protocol, clients can connect to a central server. Clients can thereafter subscribe to receive published messages or publish messages themselves. An established connection can be terminated by another connection request.*

The Mealy machine depicted in Figure 2.1 has three states, via circles labeled with the unique state identifiers q_0, q_1, *and* q_2. *The initial state* q_0 *is indicated by an arrow with an empty source. The Mealy machine includes the inputs* $I = \{\mathsf{connect}, \mathsf{publish}, \mathsf{subscribe}, \mathsf{unsubscribe}\}$ *and the outputs* $O = \{\mathsf{ack}, \mathsf{closed}, \mathsf{message}, \mathsf{none}\}$. *A transition between two states is labeled with an input followed by a slash character ('/') and the corresponding observable output. For example, the input* connect *executed in state* q_0 *results in the output* ack *and the transition to state* q_1, *which corresponds to the output function* $\lambda(q_0, \mathsf{connect}) = \mathsf{ack}$ *and the state transitions function* $\delta(q_0, \mathsf{connect}) = q_1$ *respectively. A trace of this Mealy machine would be*

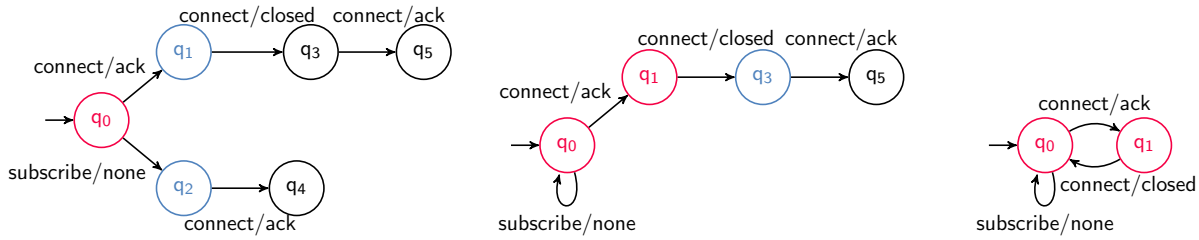
$$(\mathsf{connect}, \mathsf{ack}) \cdot (\mathsf{subscribe}, \mathsf{ack}) \cdot (\mathsf{publish}, \mathsf{message}) \cdot (\mathsf{connect}, \mathsf{closed}).$$

Mealy machines are a popular modeling formalism for communication protocols [50, 61, 62, 63, 168, 170]. Since many case studies in this thesis are based on communication protocols, we use Mealy machines as a preferred modeling formalism. However, the assumption of deterministic behavior may be too restrictive. Chapter 8 will introduce another modeling formalism that relaxes the assumption of deterministic behavior.

2.2 Automata learning

Automata learning is a technique for automatically generating a behavioral model that adequately formalizes a black-box system using a set of system observations. Formally, let $\mathcal{M}_{\text{SUL}}$ be an unknown Mealy machine representing a black-box system. The goal of automata learning is to identify a Mealy machine $\mathcal{M}$ that defines the same language as the SUL, i.e., $L(\mathcal{M}) = L(\mathcal{M}_{\text{SUL}})$. Gold [73] presents the first approaches to this problem in 1972.

In automata learning, two approaches are distinguished: Passive and active automata learning. Passive learning techniques identify a model from a given set of system data, whereas active techniques interact with the SUL to generate the data set for learning. The problem of



(a) Initial PTA generated from the two given traces in Example 2. Blue nodes indicate candidates for merging with red nodes.

(b) Automaton after merging node q_2 into q_0 of the PTA.

(c) Final merged automaton for the given traces.

Figure 2.2: Steps of the RPNI algorithm starting from the initial PTA to the final merged automaton. The example is modified from the example given by Aichernig et al. [13].

identifying a behavioral model with at most n states from a given data set is *NP*-complete [74]. Moreover, Angluin [16] shows that the number of required interactions in active learning is also exponential when only n is known.

Considering the hardness of automata learning, it can nevertheless be successfully applied in practice as it is shown in this thesis. Moreover, automata learning techniques have been extended to different types of modeling formalisms, so that this technique can be applied for application to various types of systems, e.g., timed systems [10, 75, 77, 171] or stochastic systems [32, 172, 173].

2.2.1 Passive Automata Learning

Passive automata learning algorithms learn a behavioral model given a sample. Commonly, passive learning techniques are based on state merging [32, 49, 138, 187] or are search-based [93, 102, 105, 171]. Other passive learning techniques utilize different methods such as the training of a recurrent neural network (RNN) [136, 137]. In this thesis, we will evaluate a state-merging based technique and introduce a new RNN-based approach in Chapter 7.

State-merging algorithms create a prefix-tree acceptor (PTA) from a given sample and then merge states so that the automaton still conforms to the sample. The learning algorithm terminates when no further states could be merged. The Regular Positive Negative Inference (RPNI) algorithm [49, 139] is an example of an algorithm that follows such a state-merging approach. RPNI requires a set of positive and negative traces. Positive traces contain behavior that shall be described by the learned automaton, while the behavior shown in negative traces must not be included. Let $L(\mathcal{M}_{\text{SUL}})$ be the language that is defined by an unknown Mealy machine $\mathcal{M}_{\text{SUL}}$. Positive traces are part of $L(\mathcal{M}_{\text{SUL}})$, while negative traces are not included.

Based on the positive traces, RPNI builds a PTA. The states of the PTA are then merged to create a finite automaton. A merge is valid if no negative trace can be generated by the merged automaton. Otherwise, if the merged automaton now includes negative examples, the merge is discarded and another merge is attempted. In this technique, additional labels are assigned to nodes to indicate which states are currently eligible for merging. These types of nodes are commonly assigned to a specific color, which can be seen as a classification of states. The classes distinguish between states that cannot be merged, states that are currently being considered for merging, and states that have not yet been considered. Variants of the RPNI algorithm can learn Mealy machines from input/output traces. In Mealy machine learning, states are merged when outputs for an input are the same. Tappler et al. [15] explain that learning Mealy machines from positive traces only is feasible, since other traces provide counterexamples showing different outputs for the same input. Incorrect merging would therefore violate the assumption that output behavior is deterministic.

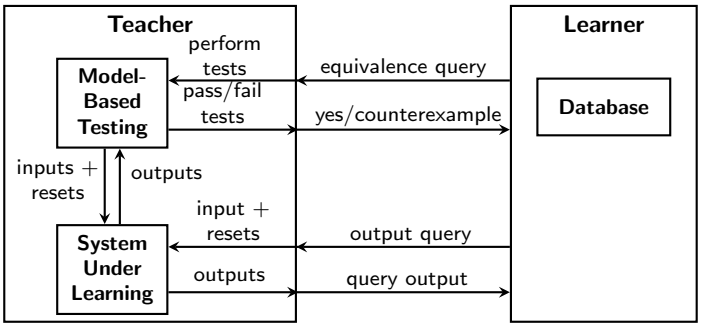


Figure 2.3: Angluin's [17] MAT framework with adaptions for learning reactive systems. The figure is based on the figures proposed by Smeenk et al. [164] and Tappler et al. [11].

Example 2 (RPNI) *Our goal is to learn a minimal Mealy machine that represents the following two traces.*

$$(1)\quad (\mathit{connect}, \mathit{ack}) \cdot (\mathit{connect}, \mathit{closed}) \cdot (\mathit{connect}, \mathit{ack})$$
$$(2)\quad (\mathit{subscribe}, \mathit{none}) \cdot (\mathit{connect}, \mathit{ack})$$

These two traces are part of the language defined by the Mealy machine depicted in Figure 2.1. Therefore, connect *and* subscribe *are input actions, and* ack, closed *and* none *are output actions. The PTA of these two traces is shown in Figure 2.2. The red states indicate the states that will be included in the final automaton, and the blue states are currently candidates for merging with the red states. For example, in the PTA shown in Figure 2.2a, the first step is to check if the blue state* q_1 *can be merged with the red state* q_0*. Merging is not possible since the input* connect *generates two different outputs:* ack *and* closed*. However,* q_2 *can be merged with* q_0*. Figure 2.2b shows the automaton after the merge. In the next step,* q_3 *can be merged into* q_0*, resulting in the final automaton presented in Figure 2.2c. Note that passive learning algorithms can only model behavior that is contained in the given data set. For example, the learned automaton presented in Figure 2.2c is not input enabled, since no trace was given that describes the behavior of input* subscribe *in state* q_1*.*

In addition to state-merging techniques, there are also search-based learning techniques [102, 108, 171]. Many of these approaches are based on evolutionary algorithms that optimize the solution using a fitness function. A third category of passive learning techniques based on the training of RNNs will be discussed in detail in Chapter 7.

2.2.2 Active Automata Learning

In active automata learning, we actively interact with the SUL to create the data set for learning. Angluin [17] proposed the L^* algorithm. The L^* algorithm was originally designed to generate a minimal deterministic finite automaton (DFA) that represents an unknown regular language. In her seminal work, Angluin introduced the minimally adequate teacher (MAT) framework, which is still the base for many active automata learning algorithms.

Minimally Adequate Teacher Framework

Figure 2.3 depicts an adapted MAT framework that distinguishes two members: the learner and the teacher. The learner wants to learn a behavioral model of a black-box SUL about which the teacher has knowledge. To learn the behavioral model, the learner asks the teacher questions. Note that the MAT framework requires that the SUL can be reset to the same initial state before each query. Thus, we assume that each query is executed from the initial state. In the MAT framework, we distinguish two types of queries: membership queries and equivalence

queries. Membership queries are asked to determine whether a given input sequence is part of the language. The learner stores the received answer in an internal database with a certain structure. In the original L^*, a table-based structure called *observation table* was used to store the queried data. Based on the queried data set, the learner then creates a hypothesis model. This hypothesis is a conjecture about the behavioral model that represents the unknown SUL.

In an equivalence query, the learner proposes the generated hypothesis to the teacher and asks if the hypothesis defines the equivalent behavior. The teacher then either responds that this conjecture is correct or returns a counterexample showing the behavioral differences between the provided hypothesis and the SUL. In the case of learning a DFA, a counterexample would be an input sequence that is accepted by the provided DFA but rejected by the SUL, or vice versa.

If the teacher provides a counterexample, the learner improves the hypothesis according to the provided counterexample. This may require asking further membership queries. The improved hypothesis is then proposed again to the teacher. This iterative procedure is repeated until the teacher no longer provides a counterexample and accepts the hypothesis as equivalent to the SUL. The last proposed hypothesis is then returned by the learner. Angluin [17] shows that under the assumption of a perfect teacher, the model can be learned by asking a polynomial number of queries.

Conformance Testing

In conformance testing, the goal is to test whether an implementation conforms to a given specification defined by a model. The literature [8, 27] discusses the similarities between conformance testing and automata learning. Both techniques aim to show the equivalence between a model and a black-box system. However, the problem in automata learning is inherently more difficult since the model is unknown. Under the assumption of a perfect teacher, who can tell the difference between the hypothesis and the SUL, the collection of performed queries would present a test suite for conformance testing. However, the assumption of a perfect teacher is not feasible in practice.

Considering the similarities between conformance testing and automata learning, we can formulate the equivalence oracle as a conformance testing problem. Hence, we want to find a test suite that can determine whether the SUL and the hypothesis are behavioral equivalent. In automata learning, we use model-based testing techniques to approximate whether the SUL conforms to the learned hypothesis. The test suite consists of a finite number of test cases. A test case *passes* if the behavior implemented by the SUL conforms to the learned model, otherwise, the test case *fails*. Formally, we write I **passes** tc and I **fails** tc if the implementation I passes and fails a test case tc, respectively. For conformance testing, we create a finite test suite. If all test cases in the test suite *pass*, it denotes that the learned hypothesis conforms to the SUL. Otherwise, a failed test case represents a counterexample to the equivalence between SUL and hypothesis. Utilizing Tretmans' implementation relation [180], as shown by Tappler [169], we define a conformance relation I **imp** H that is satisfied if an implementation I implements a specification H. The implementation relation is satisfied if all test cases pass. Let TC be a finite test suite, we define the following conformance relation

$$I \textbf{ imp } H \;\Leftrightarrow\; \forall\, tc \in TC \colon I \textbf{ passes } tc. \tag{2.1}$$

Note that this conformance relation can only assess behavioral equivalence between the learned hypothesis and the SUL depending on the generated test suite. In practice, there are several methods to generate such test suites for conformance testing. The generation of a test suite may depend on state or transition coverage, or pure randomness. There are also conformance testing techniques such as the W-Method [39, 186] that provide guarantees up to a fixed number of states with the drawback that the size of the test suite is exponential according to the number of states of the black-box system. Thus, deciding which model-based

testing technique to use to generate the test suite depends on the goal of automata learning. For example, if the model is only required for a first analysis or if it must provide any strict guarantees about conformance.

Learning Mealy Machines

The MAT framework has been extended for several modeling formalisms including Mealy machines [89, 113, 162]. Figure 2.3 shows the adaptions for learning Mealy machines. Instead of asking whether a word is a member of the language, the learner asks for the corresponding output sequences on a given input. Therefore, we write output queries instead of membership queries.

Based on the equivalence relation of two Mealy machines, a counterexample for the conformance between the hypothesis and the SUL would be an input sequence where the observed output sequences are different. Let $M = \langle Q, q_0, I, O, \delta, \lambda \rangle$ be the inferred Mealy machine and $\mathcal{M}_{\text{SUL}} = \langle Q', q_0', I', O', \delta', \lambda' \rangle$ the unknown Mealy machine that correctly represents the SUL. A counterexample for the conformance between $\mathcal{M}$ and $\mathcal{M}'$ is an input sequence s^I where the $\lambda^*(q_0, s^I) \neq \lambda'^*(q_0', s^I)$. With respect to Equation 2.1, a test case is a trace that includes for an input sequence the outputs defined in the learned hypothesis. A test case passes if the observed output sequence on the SUL is equivalent, otherwise, the test case fails.

Learning with L^*

The MAT framework serves as a base for the L^* algorithm. The learner in the L^* algorithm fills a database using a table-based structure. The table is called *observation table*.

Definition 2 (Observation Table) *An observation table $\mathcal{T}$ for learning a Mealy machine $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ is a triplet $\langle \Gamma, E, T \rangle$, with*

- *the prefix-closed set $\Gamma \subseteq I^*$,*
- *the suffix-closed set $E \subseteq I^+$, and*
- *the output mapping $T\colon\ \Gamma \times E \to O^*$.*

Furthermore, we define that $\Gamma = \Gamma_S \cup \Gamma_P$, where $\Gamma_S \cap \Gamma_P = \emptyset$ and $\Gamma_P \subseteq \Gamma_S \cdot I$. Each element $\gamma \in \Gamma$ identifies a row in the table. We write $\gamma \cong \gamma'$ for two rows $\gamma, \gamma' \in \Gamma$ that are equal, where $\forall e \in E\colon\ T(\gamma, e) = T(\gamma', e)$ holds. A row identifies a state in the automaton. If two rows $\gamma, \gamma' \in \Gamma$ are equal, they identify the same state. The set Γ_S contains at least one row for each state of the automaton learned so far. Note that an input sequence of $\gamma \in \Gamma$ leads to the corresponding states in the Mealy machine when executed from the initial state. Thus, representing an access sequence of the represented states. The set Γ_P is used to identify further transitions between the states. To identify the state and output transition for a state that is accessed by $\gamma \in \Gamma_S$ and input $i \in I$, we look up in Γ the concatenation $\gamma \cdot i$. The target state is then the state that is reached with $\gamma' \in \Gamma_S$ such that $\gamma \cdot i \cong \gamma'$ holds. The corresponding output can be looked up in the mapping $T(\gamma, i)$.

During learning, the table is filled with the outputs by performing output queries. We fill the values for the mapping T by querying each combination of $\gamma \cdot e$, where $\gamma \in \Gamma$ and $e \in E$. The suffix of the query output with length $|e|$ is then set for $T(\gamma, e)$.

Example 3 (Observation Table for Figure 2.1) *Table 2.1 shows an observation table that is generated during learning the Mealy machines of the publish/subscribe protocol that is depicted in Figure 2.1. The set Γ_S includes three input sequences, indicating the three states of the Mealy machine $\mathcal{M} = \langle Q, \mathsf{q}_0, I, O, \delta, \lambda \rangle$. For example, if we execute on the Mealy machine in Figure 2.1 the input sequence of the third row of the table $\gamma =$ connect $\cdot$ subscribe we would reach q_2, i.e.,*

Table 2.1: Observation table generated during learning the publish/subscribe protocol shown in Figure 2.1.

	Γ/E	E			
		connect	subscribe	publish	unsubscribe
Γ_S	ϵ	ack	none	none	none
	connect	closed	ack	ack	ack
	connect · subscribe	closed	ack	message	ack
Γ_P	subscribe	ack	none	none	ack
	publish	ack	none	none	none
	unsubscribe	ack	none	none	none
	connect · connect	ack	none	none	none
	connect · publish	closed	ack	ack	ack
	connect · unsubscribe	closed	ack	ack	ack
	connect · subscribe · connect	ack	none	none	none
	connect · subscribe · subscribe	closed	ack	message	ack
	connect · subscribe · publish	closed	ack	message	ack
	connect · subscribe · unsubscribe	closed	ack	ack	ack

$\delta^*(\mathsf{q}_0, \mathsf{connect} \cdot \mathsf{subscribe}) = \mathsf{q}_2$. *If we concatenate to the input sequence* γ *the input* connect*, the table shows us that we enter state* q_0 *since* $\mathsf{connect} \cdot \mathsf{subscribe} \cdot \mathsf{connect} \cong \epsilon$. *The rows map to the same state since for all entries in* E*, i.e.,* connect*,* subscribe*,* publish *and* unsubscribe*, the table defines the same outputs:* ack*,* none*,* none *and* none*, respectively. We observe the output* closed*, corresponding to our mapping* $T(\mathsf{connect} \cdot \mathsf{subscribe}, \mathsf{connect}) = \mathsf{closed}$.

Algorithm 1 shows an L^* implementation of the learner. Algorithm 1 takes as input the SUL and the input alphabet. Note that the learner only requires an interface to the SUL through which the learner can perform output queries and observe the corresponding outputs. Even though the algorithm learns a Mealy machine, the structure of this algorithm follows the basic structure of all L^*-based algorithms for different modeling formalisms.

The algorithm starts by initializing the observation table in Line 1, where $\Gamma_S = \{\epsilon\}$ and $E = I$. After that, the iterative learning procedure is performed from Line 2 to Line 19. First, the learner executes all membership queries to fill each cell in the observation table (Line 3). Then, Line 4 checks if the table fulfills two properties: closedness and consistency. The table is closed if $\forall \gamma_P \in \Gamma_P$, $\exists \gamma_S \in \Gamma_S \colon \gamma_P \cong \gamma_S$. In other words, the table is not closed if Γ_P contains a row that is not in Γ_S. If this is the case, we have explored a new state of our automaton that needs to be added to Γ_S. To close the table, the missing row $\gamma_P \in \Gamma_P$ is moved to Γ_S and Γ_P is extended by $\gamma_P \cdot i$, for each input $i \in I$. The observation table is made closed in lines 5–7. The table is consistent if $\forall \gamma, \gamma' \in \Gamma$, $\forall i \in I \colon \gamma \cong \gamma' \Rightarrow \gamma \cdot i \cong \gamma' \cdot i$. This is, if two rows are equal, i.e., denote the same state, then appending any input must result in the same state. We check consistency in Line 9. If the table is not consistent, then E is not sufficient to distinguish states appropriately. To make the table consistent, we need to extend E with an entry $i \cdot e$ with $e \in E$, where $\lambda^*(\gamma \cdot i \cdot e) \neq \lambda^*(\gamma' \cdot i \cdot e)$. The extension of E by $i \cdot e$ enables now to distinguish in the observation table the two states identified by γ and γ'. This is done in Line 10. In Line 11, we perform all output queries that are necessary to fill the empty cells of the updated observation table. After all output queries are performed to fill the empty cells of the table, and the table is finally closed and consistent, the algorithm generates a hypothesis in Line 14.

The hypothesis is then proposed to the equivalence oracle. In Line 15, the equivalence between the learned hypothesis and the SUL is checked. As explained in Section 2.2.2, we use model-based testing techniques to check if the behavior of the SUL conforms to the proposed hypothesis. The method *equivalence_query*$(\mathcal{M}, SUL)$ takes the current hypothesis and the interface to the SUL as input. The method returns two values: *verdict* and *cex*. The variable *verdict* is a Boolean variable that indicates whether the conformance relation is satisfied, i.e.,

Algorithm 1 L^* algorithm for Mealy machines

Input: black-box access to SUL SUL, input alphabet I
Output: learned Mealy machine $\mathcal{M}$

```
1:  T ← init_table(I)
2:  do
3:      T ← fill_table(T, SUL)
4:      while ¬(closed(T) ∧ consistent(T)) do
5:          if ¬closed(T) then
6:              T ← make_closed(T)
7:              T ← fill_table(T, SUL)
8:          end if
9:          if ¬consistent(T) then
10:             T ← make_consistent(T)
11:             T ← fill_table(T, SUL)
12:         end if
13:     end while
14:     M ← create_hypothesis(T)
15:     verdict, cex ← equivalence_query(M, SUL)
16:     if ¬verdict then
17:         T ← update_table(T, cex)
18:     end if
19: while ¬verdict
20: return M
```

whether all test cases evaluate to **pass**. In the case at least one test case evaluates to **fail**, the method assigns t the value *false* to the variable *verdict* and returns a counterexample *cex* that witnesses the behavioral difference between $\mathcal{M}$ and SUL. In the case of learning Mealy machines, *cex* is an input sequence that yields a different output sequence when executed on $\mathcal{M}$ and SUL. We check the value of *verdict* in Line 15, and if the proposed hypothesis does not conform to the SUL, we use the returned *cex* in Line 17 to extend the observation table. There are several techniques for extending an observation table with a found *cex*. In the original L^* algorithm, all prefixes of *cex* are added to the Γ set of the observation table. In other approaches, the suffixes of *cex* are added to the E set.

The condition in the while-loop in Line 19 again uses the variable *verdict*. The learning procedure is repeated until no counterexample is found for the conformance between the hypothesis and SUL, i.e., until *verdict* evaluates to true. Otherwise, we start again at Line 3 by performing further output queries required to improve the hypothesis according to the given counterexample. Once we found a conforming hypothesis, the algorithm returns the conforming hypothesis $\mathcal{M}$ in Line 20.

Learning with *KV*

A learner may use tree-based structures as an alternative to table based-structures. Kearns and Vazirani [96] introduced a tree-based learning algorithm that also uses the MAT framework to learn a behavioral model of a black-box system. In the remainder of this thesis, we refer to this learning algorithm proposed by Kearns and Vazirani as *KV*. This algorithm provides the basis for other table-based algorithms such as the TTT algorithm [89]. To define the *KV* algorithm, we first require a definition of distinguishing sequences.

Definition 3 (Distinguishing Sequence) *Let $\mathcal{M} = \langle Q, \mathsf{q}_0, I, O, \delta, \lambda \rangle$. We define an input sequence $s^I \in I^*$ as a distinguishing sequence for two states $q, q' \in Q$ if $\lambda(q, s^I) \neq \lambda(q', s^I)$ holds.*

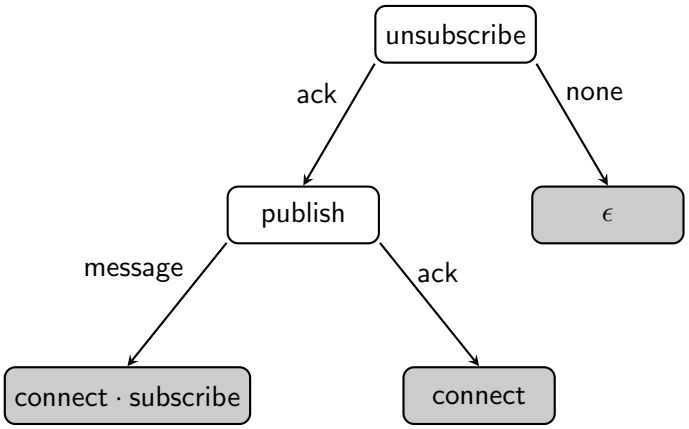


Figure 2.4: Example of a classification tree that can be used to create a Mealy machine representing the publish/subscribe protocol shown in Figure 2.1.

The learner in *KV* uses a *classification tree* to store the observations and construct the hypothesis. In the following, we will present how classification trees can be used to learn Mealy machines.

Definition 4 (Classification Tree) *A classification tree $\mathcal{T}_C$ is a pair $\langle N, root \rangle$, where N is the set of nodes and $root \in N$ is the root of the tree. A tree node $n \in N$ can be represented by a triplet $\langle l, C, \pi \rangle$ with input sequence $l \in I^*$ as the label of the node, a set of nodes $C \subset N$ denoted as children, and an output mapping for the children of the node $\pi \colon O^* \to N$, where O is the finite set of outputs. Let the* parent *of a node $n \in N$ be another node $n' = \langle l', C', \pi' \rangle \in N$, where $n \in C$. Each node in the tree has exactly one parent, except the root node which has no parent.*

The semantics of the nodes in the classification tree depends on the number of children. If the node has no children, i.e. is a leaf node, then the node defines a state in the automaton, where the label of the node is an access sequence to the state. Thus, the number of leaf nodes in the classification tree defines the number of states. The inner nodes, on the other hand, contain input sequences that are distinguishing sequences.

Example 4 (Classification Tree) *Figure 2.4 illustrates an example of a classification tree that is used to learn the publish/subscribe protocol shown in Figure 2.1. White nodes indicate inner nodes that contain a distinguishing sequence. Leaf nodes are colored in gray. The labels of these nodes define access sequences to the three different states. Note that the access sequences are identical to the set Γ_S shown in Table 2.1.*

Generation of a hypothesis. To derive a Mealy machine from a classification tree, *KV* uses the *sift* operation, which simulates the execution of an input sequence on a Mealy machine to determine the target state. Sifting an input sequence $s_I \in I^*$ through the classification tree starts at the root of the classification tree. Then, the label of the root node is appended to the input sequence to be sifted. Let $root = \langle l, C, \pi \rangle$ be the root node. For sifting s_I, we append to s^I the label of the root l which generates the input sequence $s^I \cdot l$. The learner then asks $s^I \cdot l$ as an output query to the SUL. The teacher returns an output sequence $s^O \in O^*$. The suffix of length $|l|$ of the output sequence is then used to determine which edge of the tree to follow. The same procedure is repeated with the next reached inner node, i.e., we again extend s^I with the label of the currently considered inner node. This procedure is repeated until a leaf node is reached. The leaf node reached corresponds to the state the input sequence would reach if executed on the Mealy machine.

Algorithm 2 *KV* algorithm for Mealy machines

Input: black-box access to SUL SUL, input alphabet I
Output: learned Mealy machine $\mathcal{M}$

```
1:  M ← create_initial_hypothesis(SUL, I)
2:  verdict, cex ← equivalence_query(M, SUL)
3:  if ¬verdict then
4:      T_C ← init_tree(SUL, cex)
5:  end if
6:  while ¬verdict do
7:      M ← create_hypothesis(SUL, T_C, I)
8:      verdict, cex ← equivalence_query(M, SUL)
9:      if ¬verdict then
10:         T_C ← update_tree(T_C, SUL, cex)
11:     end if
12: end while
13: return M
```

Example 5 (Sifting) *We want to sift* connect · publish *through the classification tree shown in Figure 2.4. We start at the root node labeled with* unsubscribe *and append it to the sifted input sequence. Querying the sequence* connect · publish · unsubscribe *on the SUL yields the output sequence* ack · ack · ack. *Since the last output is* ack*, we follow the corresponding transition and reach the inner node labeled with* publish*. We then query* connect · publish · publish *which generates the output sequence* ack · ack · ack*. This leads us to the leaf node labeled with* connect*, which corresponds to the state q_1 in the Mealy machine shown in Figure 2.1. We also see that $\delta^*(q_0, \text{connect} \cdot \text{publish}) = q_1$ holds.*

To generate a Mealy machine from a classification tree, we define the set of states based on the leaf nodes of the classification tree. To draw the transitions between the states, we sift each access sequence of the state with each input from the input alphabet, which then leads to the target state. The output label of the transition is determined by performing an output query.

***KV* algorithm.** The goal of the *KV* algorithm is to create a classification tree that allows us to derive a hypothesis that conforms to the SUL. Algorithm 2 defines the procedure of the *KV* algorithm. The algorithm starts in Line 1 with the creation of an initial hypothesis. The initial hypothesis consists of only one state, where all input transitions are self-loops. The corresponding output labels are queried using output queries. In Line 2, this initial hypothesis is then immediately proposed to the teacher by performing an equivalence query. The method *equivalence_query* is identical to the one described in Section 2.2.2. If the initial hypothesis does not conform to the behavior of the SUL, the counterexample is used to initialize the classification tree $\mathcal{T}_C$ in Line 4. The root of the tree is labeled with the input sequence that shows the difference between the initial hypothesis and the output produced by the SUL. The input sequence distinguishes two states. For this, we initialize the tree with two leaf nodes, which can be reached by the corresponding output sequence.

After the initialization procedure, we start the iterative learning procedure by executing the loop from Line 6 to Line 10. First, we construct in Line 7 a hypothesis from the classification tree, as described earlier in this section. We then test against the hypothesis the SUL in Line 8. If *verdict* is *false*, we use the counterexample in Line 10 to update the tree. A counterexample always shows that the current hypothesis does not have enough states to correctly model the behavior of the SUL. Therefore, new states and distinguishing sequences are added to the classification tree based on the counterexample. For more details on the algorithm, we refer the reader to the work of Kearns and Vazirani [96].

It should be noted that every sifting operation is likely to require several output queries and that these output queries may be repeated whenever a hypothesis is constructed. Therefore, the original KV algorithm uses a more concise data structure than L^*, but the number of (required) interactions with the SUL could be larger due to the repeated output queries. However, this problem can be easily solved by caching queries that have already been performed, as implemented in state-of-the-art learning libraries such as AALpy [129] or LearnLib [90].

Abstraction for Learning

For learning behavioral models of real systems, adaptions are required to make automata learning feasible. Earlier in this chapter, we showed that the perfect equivalence oracle is being replaced by model-based testing techniques. One problem that remains is that the complexity of automata learning depends on the state space of the SUL and the number of possible inputs.

In particular, when the input alphabet is large, many output queries must be performed and the underlying data structures grow immensely. For example, consider that the publish/-subscribe protocol shown in Figure 2.1 represents an abstracted Mealy machine. In the real system, a client can subscribe to various topics and publish messages containing any sequence of characters. Learning such a system with all possible inputs would not be feasible.

Cho et al. [38] introduced the concept of learning a Mealy machine considering an abstracted and therefore smaller input and output alphabet. In their work, they learn a model of a botnet server using an interface that translates abstract inputs provided by queries from the learner into concrete inputs that can be executed on the SUL. This interface then receives concrete outputs from the SUL and translates them into abstract outputs that are used by the learning algorithm. With this technique, the learned model formalizes the behavior of the SUL on a more abstract level.

Aarts et al. [7] extend this abstraction concept by making the applied abstraction state-dependent. For this purpose, they formalize a stateful mapper component that concretizes received abstract inputs from the learning algorithm to concrete inputs that can be executed on the SUL, where the abstraction performed depends on the current state of the mapper. Vice versa, this is performed for the received outputs from the SUL. For the definition of an abstraction component, we follow Definition 2 of Aarts et al. [2].

Definition 5 (Mapper) *Let a tuple $\mathcal{A} = \langle I, O, I^{\mathcal{A}}, O^{\mathcal{A}}, R, r_0, \Lambda_I, \Lambda_O, \Delta \rangle$ be an abstraction implemented by a mapper component, where*

- *I and O are finite sets of concrete inputs and outputs,*
- *$I^{\mathcal{A}}$ and $O^{\mathcal{A}}$ are finite sets of abstract inputs and outputs,*
- *R is a set of states,*
- *$r_0 \in R$ is the initial state,*
- *$\Lambda_I \colon R \times I \to I^{\mathcal{A}}$ is an abstraction function for concrete inputs,*
- *$\Lambda_O \colon R \times O \to O^{\mathcal{A}}$ is an abstraction function for concrete outputs, and*
- *$\Delta \colon R \times (I \cup O) \to R$ is a state transition function.*

Aarts et al. [7] stress that $\mathcal{A}$ behaves deterministically. Their proposed mapper component implements such an abstraction to concretize abstract input and abstract concrete outputs. While abstracting outputs is straightforward in $\mathcal{A}$, the concretization of inputs is not. To do this, the mapper takes an abstract input $i^{\mathcal{A}} \in I^{\mathcal{A}}$ from the learner and selects a concrete input $i \in I$ according to the input abstraction function Λ_I such that $\Lambda_I(r, i) = i^{\mathcal{A}}$ holds, where

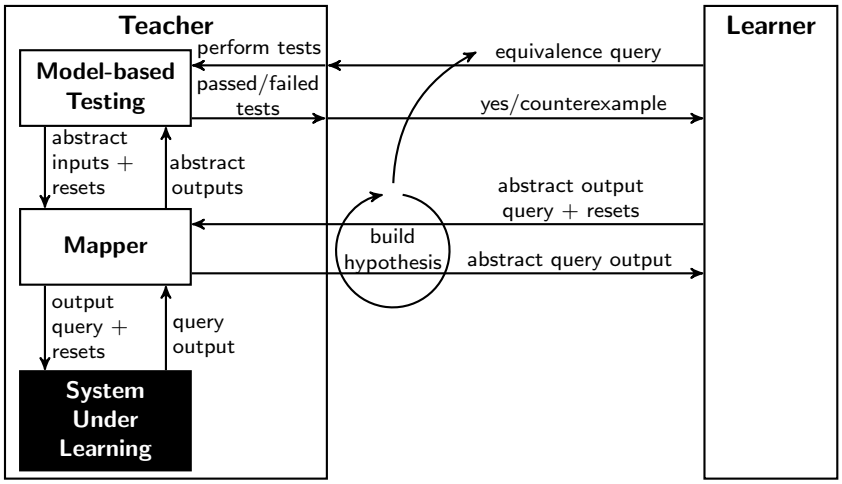


Figure 2.5: An extended MAT framework with a mapper component that translates abstract inputs into concrete inputs that can be executed on the SUL. Inversely, the received concrete outputs are translated into abstract outputs. The figure is based on the MAT framework presented by Aichernig et al. [8]

.

$r \in R$ is the current state of the mapper. Note that the mapper selects the concrete input non-deterministically. However, according to Aarts et al. [2], a mapper for learning Mealy machines must be designed to simulate deterministic behavior. Hence, the implementation of the mapper must ensure that the concretizations are chosen such that the observations are deterministic.

Figure 2.5 illustrates how the mapper can be included in the MAT framework. The mapper can be seen as a wrapper that encapsulates the interface of the SUL. Therefore, every interaction with the SUL is parsed and processed by the mapper. The mapper implements the concretization and abstraction function for translating the received inputs and outputs.

Example 6 (Mapper) *Assuming that Figure 2.1 represents an abstracted model of the underlying SUL, we can define an example mapper for learning this model. Consider that different publish messages can be sent to specific topics. Let* publish(*topic*, *message*) *be a publish message, where topic identifies a topic filter and message the published message. Both are sequences of Unicode characters. A concrete publish message would be* publish(`office/temperature`, `10.5`)*. Similar to publish messages, subscribe and unsubscribe messages can be performed for a specific topic. Therefore, we consider* subscribe(*topic*) *and* unsubscribe(*topic*) *as additional concrete inputs, where topic is again an arbitrary sequence of characters. Corresponding to the automaton in Figure 2.1, the abstract inputs would be* $I^{\mathcal{A}} = \{$connect, publish, subscribe, unsubscribe$\}$ *and the abstract outputs* $O^{\mathcal{A}} = \{$none, ack, message$\}$*. The concrete inputs would be* $I = \{$connect, publish(*topic*, *message*), subscribe(*topic*), unsubscribe(*topic*)$\}$ *and the outputs* $O = \{$none, ack, *message*$\}$*, where message and topic are any character sequences. The mapper then translates the inputs and outputs straightforwardly, e.g., the abstract input* publish *would be translated into an instance of* publish(*topic*, *message*)*. To receive messages of subscribed topics, the mapper must keep track of the subscribed topic. Hence, the mapper considers different states for each subscribed topic.*

The used mappers in this thesis are manually created. However, in the literature [4] there are also concepts for the automatic generation of abstractions.

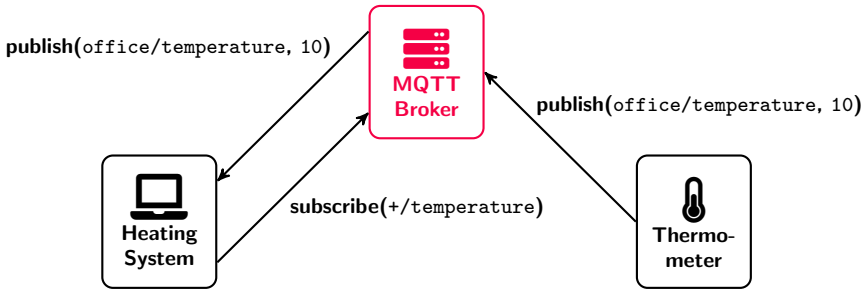


Figure 2.6: An application scenario for the MQTT protocol in a smart home scenario with two clients. The heating system subscribes to all temperature topics. The thermometer in the office sends publish messages including the measured temperature. The broker is responsible for distributing the messages in the MQTT network.

2.3 Communication Protocols

The following chapter introduces the main case study subjects that are used to evaluate the methods presented in this thesis. Since the main focus of this thesis is on developing methods for increasing the dependability in the IoT, we focus on communication protocols that are popular in the IoT, e.g., Message Queuing Telemetry Transport (MQTT) and Bluetooth Low Energy (BLE), but we also examine protocols that have a large impact on securing an IoT network, like VPN protocols.

2.3.1 Message Queuing Telemetry Transport (MQTT)

The Message Queuing Telemetry Transport (MQTT) protocol [22] is a widely used publish/-subscribe protocol. The protocol is especially popular in the IoT due to its lightweight design.

The basic MQTT setup considers two members: an MQTT broker and an MQTT client. A broker is a central unit responsible for managing connections and subscriptions of clients and distributing published messages. Thus, the broker is essential for the functionality of an MQTT network. Clients can connect to brokers, subscribe to specific topics, and publish messages on specific topics.

Figure 2.6 illustrates a typical application scenario for the MQTT protocol for a home automation scenario. In this application, a heating system listens for published temperature messages. A thermometer publishes its measurements for the office space. The broker is responsible for forwarding the received published messages to all subscribed clients.

The MQTT protocol organizes topics in a level-based structure. Where levels are separated by a slash ('/'). For example, Figure 2.6 shows that the thermometer publishes to the topic `office/temperature`, where the topic filter has two levels: `office` and `temperature`. The MQTT client in the heating system subscribes to the topic filter `+/temperature`, where `+` is a wildcard character for a single level. Therefore, the heating system receives all messages on topics that are published to topics that have two levels and the last level is `temperature`.

The reliability and robustness of an MQTT network stand and fall with the correct functionality of an MQTT broker. Since the MQTT broker is the central node in an MQTT network, it must behave according to the specification, does not introduce any security issues, and is not vulnerable to malicious clients. Therefore, the presented methods target the analysis of MQTT brokers.

2.3.2 Bluetooth Low Energy (BLE)

Bluetooth is a popular wireless short-range communication protocol. With the introduction of Bluetooth version 4.0, the standard offers an optional low-energy implementation that enables

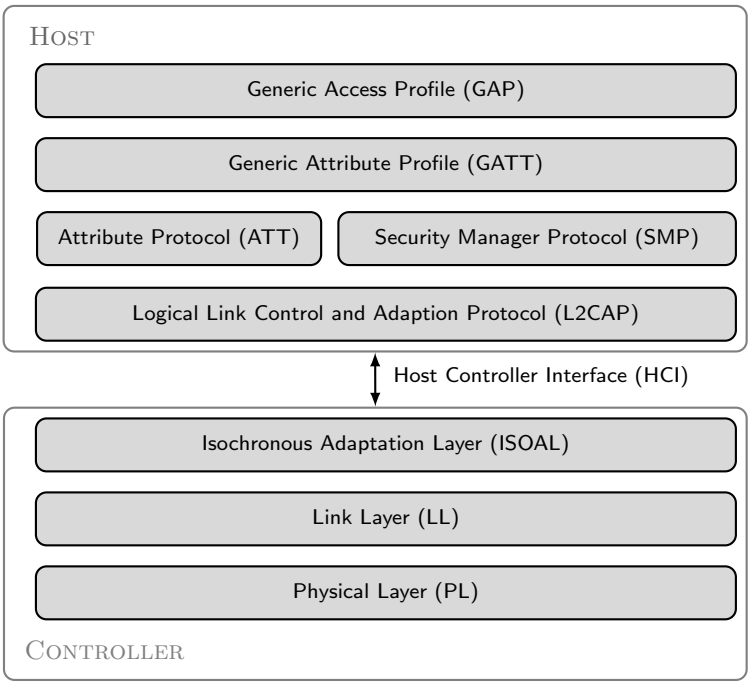


Figure 2.7: The different layers of the BLE stack. This figure is adapted from the illustration provided by the Bluetooth SIG [194].

the usage of Bluetooth as a communication protocol in the IoT. The Bluetooth core specification [87] distinguishes between Bluetooth Basic Rate (BR)/Enhanced Data Rate (EDR), also known as Bluetooth Classic, and Bluetooth Low Energy (BLE), both of which are independent protocols. Bluetooth Classic is mainly used for streaming larger amounts of data, such as audio data. BLE has more a lightweight design and enables wireless short-range communication for low-energy devices, such as sensors. In addition, BLE has other features such as broadcasting, mesh network communication, and location.

The Bluetooth Special Interest Group (SIG) [88] reports that 4.9 billion Bluetooth devices were shipped in 2022 and expects the number of annual shipments to increase to 7.6 billion Bluetooth devices by 2027. The growth is primarily due to the increasing number of peripheral devices using Bluetooth Classic and BLE, or BLE only. In the remainder of this thesis, we will focus only on BLE since it has more applications in the IoT.

Figure 2.7 depicts the different layers of the BLE protocol stack. The BLE protocol stack distinguishes between a host and a controller component. Technical reports from the Bluetooth SIG [194] explain that the host is usually implemented by an operating system, while the controller is implemented on a system on the chip. The stack consists of different layers, where the lower layers are part of the controller component, and higher layers are implemented in the host component. The controller and the host communicate through a host controller interface (HCI). Each layer has a different responsibility. For example, the Link Layer (LL) manages the various states of the BLE device, such as scanning for other devices, advertising, or establishing a connection. The Attribute Protocol (ATT) manages attributes of the communication such as the maximum transmission unit (MTU), and the Security Manager Protocol (SMP) is used to establish an encrypted connection.

Figure 2.8 describes the connection procedure between two BLE devices. The devices have different roles: One is referred to as the central device and the other is referred to as the peripheral device. In the remainder of this thesis, we will refer to the central device as *central* and to the peripheral device as *peripheral*. The peripheral broadcasts advertisements, indicating that is ready to connect to another device. Therefore, the link layer of the peripheral is in the

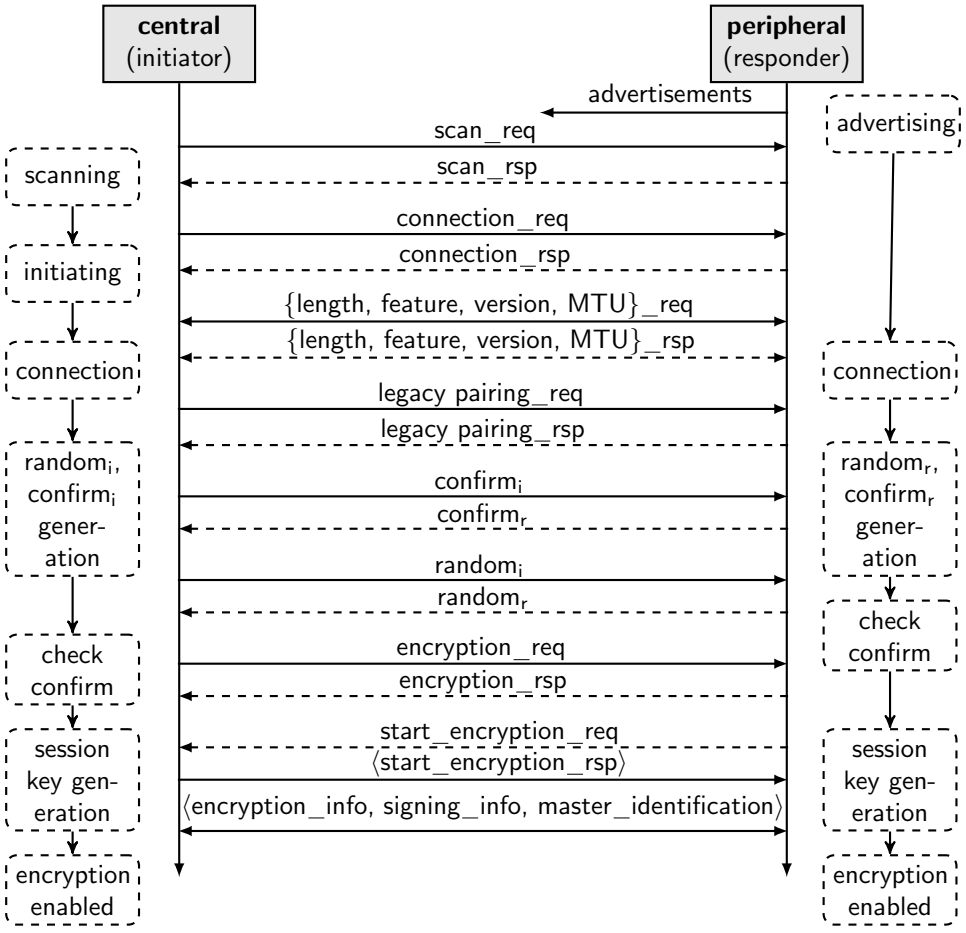


Figure 2.8: The message sequence diagram shows the BLE connection and pairing procedure. The central device initiates a connection to a peripheral that sends advertisements. After the two devices are connected, the pairing procedure starts, where they exchange keys to establish an encrypted communication. The exchanged messages in parenthesis ($\langle \ldots \rangle$) indicate that communication is encrypted. The figure is taken from Pferscher and Aichernig [149].

advertising state. The central enters the *scanning* state by searching for these advertisements. When such advertisements are found, the central initiates the connection by sending a connection request. When the peripheral is ready for a connection, it responds with a connection response. Both devices are then in the *connection* state. Then the central and the peripheral negotiate some parameters such as the Bluetooth version used, the MTU, or other characteristics of the connection. Note that not only the central can send parameter requests, but also the peripheral can send requests to which the central must respond. The order of these parameter requests is not fixed. After all requested parameters have been negotiated, the pairing procedure can begin. A connection can be terminated by both communicating parties by performing a termination indication termination_ind.

The pairing procedure is used to establish encrypted communication. For this purpose, the central and the peripheral exchange keys. In this pairing procedure, the central is usually referred to as the *initiator*, and the peripheral as the *responder*. The pairing procedure is initiated by the central with a pairing request that contains the parameters of the pairing procedure. The pairing request specifies, e.g., the type of pairing procedure to be performed, the length of the encryption key, and the authentication methods. The Bluetooth standard distinguishes between two different pairing modes: legacy and secure pairing.

The legacy pairing procedure is shown in Figure 2.8, where the central and the peripheral exchange values to establish a session key that is then used to encrypt the communication. For this purpose, each of them calculates a confirmation value (confirm). The confirmation value is based on previous connection parameters, a random value, and a temporary key that is initially set to zero. Then the central and the peripheral exchange their confirmation values. First, the peripheral checks the confirmation value of the initiator $\mathsf{confirm}_i$ considering also the received random value random_i. If the confirmation value can be recalculated by the responder, it forwards its own random value random_r to the initiator. The initiator then checks $\mathsf{confirm}_r$ and sends its part of the session key to the peripheral. The peripheral then responds with its part and also sends a request to start the encryption (start_encryption_req), which must be responded to with an encrypted response (start_encryption_rsp). All the messages are encrypted with AES-CCM [124]. Figure 2.8 shows encrypted messages with messages in angle brackets $\langle \ldots \rangle$. Then, further sensitive messages are exchanged over an encrypted communication channel. Encryption can be terminated with the pause encryption request (pause_encryption_req). However, legacy pairing is not required to include authentication and is vulnerable to man-in-the-middle attacks.

In secure pairing, the connected devices first exchange public keys and then use the Diffie-Hellman [52] key exchange procedure to establish secure encrypted communication. Additional methods for authentication can be used that require the other device to actively confirm that devices are to be paired, e.g., by entering or confirming a short sequence of numbers.

In the remainder of this thesis, we will analyze the behavior of the peripheral devices, since insight into these devices is usually limited. Therefore, the goal is to analyze the behavior of black-box devices. Furthermore, the role of the peripheral is more testable since the central controls the initiation of a connection and the pairing procedure.

2.3.3 IPsec Internet Key Exchange (IKEv1)

A VPN is an artificial network that enables secure communication over an insecure communication channel [135]. VPNs are used not only to prevent eavesdropping but also to allow external access to internal network resources. During the COVID-19 pandemic, when people were forced to stay home, access to an internal network became important for many companies and institutions overnight. Feldmann et al. [59] investigate the internet traffic during the COVID-19 pandemic, and their results showed that the amount of VPN traffic within the regular working hours approximately doubled between February 2020 and March 2020. In addition, their results show that VPN traffic decreased again after the first lockdown, but remained at a higher level than before the pandemic.

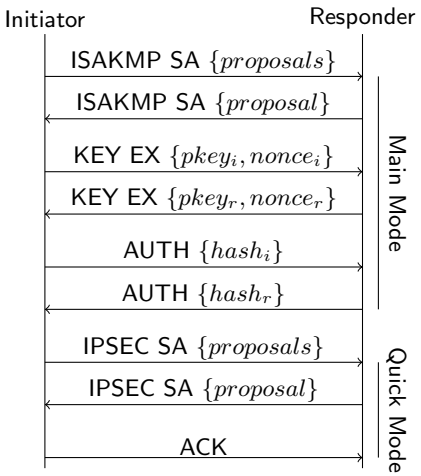


Figure 2.9: The message sequence diagram shows the messages exchanged between an initiator and a responder to establish encrypted communication according to the IKEv1 specification [33]. The figure is taken from Pferscher et al. [150].

VPNs enable secure communication by using authenticated encryption schemes. One way to establish secure communication is to use protocols defined in the Internet Protocol Security (IPsec) protocol suite. IPsec includes several protocols that define how encryption and authentication can be established securely. In this thesis, we focus on the Internet Key Exchange (IKE) protocol which is part of the IPsec protocol suite. The IKE protocol defines how to exchange keys that are later used to encrypt communication.

The IKE protocol exists in two versions: IKEv1 [33] and IKEv2 [95]. IKEv1 defines the first version of the IKE protocol. Due to its complicated setup, a second version, IKEv2, was introduced. Even though IKEv2 is the recommended version, IKEv1 is still used. For example, many routers of the popular *FRITZ!Box* router series from the German company AVM solely support IKEv1 [70]. In this thesis, we only consider IKEv1, since the setup is more complicated and thus more error-prone. Therefore, it provides a more interesting target for our methodology.

Figure 2.9 shows the message sequence diagram for establishing a secure communication channel according to the IKEv1 protocol. IKEv1 distinguishes between two communicating members: the initiator and the responder. In practice, the initiator could be represented by a VPN client, and the responder is represented by a VPN server. IKEv1 can be divided into two phases. The first phase is called *main mode* and is followed by the second phase, the *quick mode*. The goal of the main mode is to share information to exchange an encryption key based on the Internet Security Association and Key Management Protocol (ISAKMP) protocol. The initiator first proposes a set of Security Associations (SAs). An SA is a specification that describes which encryption algorithms should be applied in which configuration. The responder then replies to the request by sending a single SA with the selected configuration.

The Diffie-Hellman key exchange procedure [52] is based on the negotiated ISAKMP SA. For this purpose, both the initiator and the responder generate a key and exchange the public parts of their generated key and their used nonces. The final step of the main mode is authentication, where the initiator and the responder exchange hash values that are generated from previously sent messages and the shared secret generated using Diffie-Hellman. Authorization can be based on either a pre-shared key (PSK) or certificates.

From now on, all subsequent communication is encrypted based on the key generated in the main mode. In IKEv1, the communicating parties now enter the quick mode, which is used to share keying material for subsequent protocols in the IPsec protocol suite, e.g., for the Authentication Header (AH) or Encapsulating Security Payload (ESP) protocols. This mode starts similar to the main mode, where the initiator proposes a set of SAs, and the responder

replies with an accepted SA proposal. Finally, the initiator sends a hash value of the previous messages, which serves as a confirmation for the SA provided.

Chapter 3

Efficient Automata Learning

Declaration of Resources

This chapter provides additional background on existing work. Section 3.1 describes improvements for the L^* algorithm proposed by Rivest and Schapire [156]. Section 3.2 discusses caching strategies in existing automata learning libraries. Section 3.3 introduces specific improvements for the KV algorithm that were implemented and evaluated in the Bachelor's thesis of Maximilian Rindler [155], which was co-supervised by the author of this thesis.

The following chapter describes improvements in the application of efficient automata learning. These improvements are mainly aimed at reducing the interaction with the system under learning (SUL). In practice, communication with the SUL is considered expensive in terms of the time required to execute inputs and reset the SUL. In particular, for learning communication protocols reducing the number of queries is beneficial since messages may be lost or arrive delayed. Thus, the overall goal is to minimize communication with SUL without comprising the expressiveness of the learned behavioral model.

Interaction with the SUL is required for active automata learning. Therefore, only improvements for active algorithms are discussed in this chapter. Furthermore, the following improvements are only applicable to learning deterministic systems. Specific improvements for active learning of non-deterministic systems are addressed separately in Chapter 8.

First, general improvements for L^* [17] and KV [96] are presented in this chapter. For this purpose, Section 3.1 and Section 3.2 briefly explain improvements that are already available in state-of-the-art automata learning libraries such as AALpy [129] or LearnLib [90]. Section 3.3 presents specific improvements for KV that have been implemented specifically in AALpy.

3.1 Counterexample Postprocessing

Rivest and Schapire [156] introduce an automata learning algorithm that learns a model without requiring the SUL to be reset during learning. Even though their resetless learning algorithm is not applied in this thesis, many state-of-the-art learning algorithms implement the improvements to the L^* algorithm proposed also in the work of Rivest and Schapire.

Their proposed improvements are based on the fact that a counterexample received from the teacher indicates that the provided hypothesis does not have enough states to model the behavior adequately. Let $(\Gamma_S \cup \Gamma_P, E, T)$ be the observation table maintained by the learner. In the original L^* algorithm, the counterexample and its prefixes would be added to the Γ_S set to extend the state space. In contrast, Rivest and Schapire suggest adding a distinguishing sequence to the E set. That the provided counterexample must contain such a distinguishing sequence follows from the fact that executing it on the hypothesis and on the SUL leads to

different outputs. However, adding all suffixes of the received counterexample to the E set may add redundant information. Hence, we want to post-process the received counterexample to obtain a shorter distinguishing sequence.

We assume that we learn a hypothesis in the form of a Mealy machine $\mathcal{H} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ and let $\mathcal{M} = \langle Q', q_0', I, O, \delta', \lambda' \rangle$ be the unknown Mealy machine representing the SUL. The provided counterexample is an input sequence $c \in I^*$ showing that $\lambda^*(q_0, c) \neq \lambda'^*(q_0, c)$. The goal is to find a partition of a given counterexample $c = u \cdot a \cdot v$ such that v is a sufficient distinguishing sequence, with $u, v \in I^*$ and $a \in I$. Let $s \in I^*$ be the access sequence of a state $q \in Q$ reached by executing u on the hypothesis $\mathcal{H}$, i.e., $\delta^*(q_0, u) = \delta^*(q_0, s) = q$. The access sequence of this state can be retrieved from Γ_S. Using the same technique, we obtain $s' \in I^*$ which is the access sequence of $q' \in Q$ where $\delta^*(q_0, u \cdot a) = \delta^*(q_0, s') = q'$. The input sequence v is a distinguishing sequence if $\lambda'^*(q_0, s \cdot a \cdot v) \neq \lambda'^*(q_0, s' \cdot v)$. To find such a partition of c, a binary search can be applied. The shares are then executed on the SUL until a sufficient partition is found. Then all suffixes of v are added to E.

This approach can also be applied to KV, where the access sequences are retrieved from the leave nodes of the classification tree. The retrieved distinguishing sequence v is then added to the classification tree as an internal node.

3.2 Caching

State-of-the-art learning libraries such as LearnLib [90] and AALpy [129] implement caching techniques to reduce the number of queries performed during active automata learning on the SUL. The cache represents a data structure where performed queries and observations are stored. Instead of repeatedly executing the same queries or parts of them on the SUL, the corresponding results are looked up in the cache. In this way, the number of queries executed on the SUL can be reduced. Reducing queries is especially essential when query execution takes a long time, resetting the system is tedious, or query execution is not reliable. Hence, learning real systems benefits greatly from such caching techniques.

For learning Mealy machines, caching strategies simply store the executed output queries and their corresponding query output in specific data structures. For deterministic systems, the repetition of queries or their prefix is not required, since the output sequence must be the same. However, especially for larger systems, it becomes important that queries are stored efficiently to remain memory efficient. For example, the learning library AALpy stores all performed queries in a tree-based data structure.

In the MAT framework, the cache represents an additional component implemented by the learner that checks for each requested output query, if it already exists in the cache. If it is not, the query is executed on the SUL and then added to the cache. In learning is done using an abstraction layer, as described in Section 2.2.2, the cache stores abstracted output queries.

Later in this thesis, we will utilize the cache for checking and handling non-deterministic behavior.

3.3 *KV* Improvements

Section 2.2.2 presents the learning algorithm proposed by Kearns and Vazirani [96]. The advantage of KV over L^* is that KV uses a more concise data structure to distinguish states. However, Aichernig et al. [15] show that KV requires a lot of interaction with the SUL, especially when the counterexamples are long and the state space is large. In their work, they compared different learning algorithms and conformance testing techniques implemented in the automata learning library LearnLib [90].

The learning library AALpy [129] provides an improved version of the learning algorithm KV since version v1.3.0. The improvements over the classical implementation can be divided into

the following three categories: (1) reuse of counterexamples, (2) processing of counterexamples, and (3) caching mechanisms.

3.3.1 Counterexample Reuse

The algorithmic design of the KV algorithm usually implies more learning rounds than for L^*. A learning round is always terminated by an equivalence query, which we implement with conformance testing. Since performing conformance testing may require executing a large set of output queries, we aim to minimize the number of required rounds of conformance testing. For this purpose, our KV implementation slightly adapts the treatment of counterexamples.

A counterexample is an input sequence that shows a different output sequence when executed on the SUL and the hypothesis provided. In the original KV implementation, we look for the first input in the counterexample that reveals the behavioral difference. A new hypothesis is then created and a new equivalence query is performed. However, the previously provided counterexample may still contain another input showing that the hypothesis is still non-conforming. Instead of querying a new counterexample, we adapt the hypothesis as long as it conformance to the previously provided counterexample. In this way, we might save rounds of conformance testing.

3.3.2 Counterexample Postprocessing

Similar to L^*, KV also benefits from shorter counterexamples. Long counterexamples tend to add longer distinguishing suffixes to the classification tree used in KV. Consequently, long distinguishing sequences in the classification tree lead to long input sequences that must be queried during the sifting procedure. For this, the counterexample postprocessing proposed by Rivest and Schapire [156] as described in Section 3.1 can help to shorten the received counterexamples. As a result, input sequences of the inner nodes of the classification tree are shorter and sifting requires fewer executions of inputs.

3.3.3 Caching

AALpy implements three different caches for KV. In addition to the cache described in Section 3.2, two additional caches are added: the *sifting cache* and the *output-query cache*.

The *sifting cache* aims to reduce the number of queries required while sifting an input sequence through the classification tree. Each time a hypothesis is constructed, many sifting operations are required to define the state transition function, with each sifting operation including multiple output queries. The sifting cache is used to avoid sifting operations for input sequences that always lead to the same leaf node. For this purpose, the algorithm stores additionally a mapping from input sequences to leaf nodes. With this mapping, cached input sequences can be directly assigned to a state in the automaton and do not need to be sifted again. The sifting cache for the corresponding nodes must be updated whenever leaf nodes get a new distinguishing sequence as a parent.

In addition to state transitions, the outputs of the output functions must also be repeatedly queried. To avoid these query repetitions, the *output-query cache* is used. Even though these output queries would be stored in the tree-based caching structure as presented in Section 3.2, the operations in the tree still require more resources than a simple lookup. Therefore, the output-query cache maps input sequences to their last output in order to quickly obtain the output for the transition. Note that this cache in combination with other caching mechanisms does not necessarily decrease the interaction with the SUL, but rather reduces the runtime of the algorithm.

Chapter 4

Learning of Bluetooth Low Energy Devices

Declaration of Resources

This chapter is based on the paper *"Fingerprinting Bluetooth Low Energy Devices via Active Automata Learning"* [147] presented at *FM 2021* and the article *"Fingerprinting and Analysis of Bluetooth Devices with Automata Learning"* published in the journal *"Formal Methods in System Design"* [149] in May 2023. The presented work also received support from Maximilian Schuh, who gave advice on the Bluetooth hardware. The authors of the framework SweynTooth [67] also supported this work by the provision of an open source BLE interface.

4.1 Introduction

In this chapter, we want to outline an automata learning framework that learns Bluetooth Low Energy (BLE) stack implementations from real physical devices. Automata learning has proven itself as a convenient tool to learn communication protocols like (D)TLS [50, 63], TCP [61], SSH [62], MQTT [170], or the 802.11 4-way handshake of the WiFi protocol [168]. However, less work exists that learned communication protocol implementation on real hardware. Instead, software components have commonly been simulated in artificial environments.

Bluetooth is a popular protocol for wireless short-distance communication. As stated in Section 2.3.2, the number of annually shipped Bluetooth devices will grow to 7.6 billion devices. In particular, the number of low-energy devices that use BLE for wireless communication will grow [88]. This has also been indicated by a technical report from Texas Instruments Inc. [103] which advertises BLE as a replacement for wired communication in vehicles. In their given scenario, BLE is used, e.g., for car access, personalized configurations, or to collect sensor data.

Based on this scenario presented by Texas Instruments Inc. [103], we motivate the development of approaches that enable the automatic analysis and testing of BLE devices. In a vehicle, many heterogenous components communicate with each other. Usually, these components are developed by third-party suppliers, where the insight into the component is limited. For example, it might not be known which BLE device is built into a car's side mirror sensor. To ensure that such components do not introduce any faults, automatic testing and verification techniques are required.

Behavioral models are a useful tool for system analysis. They form the basis for model-based testing and verification technique. However, the availability of accurate models is usually limited. Creating models manually is a tedious and error-prone process, which must be repeated each time the system is updated. In the case of the Bluetooth protocol, the specification [87]

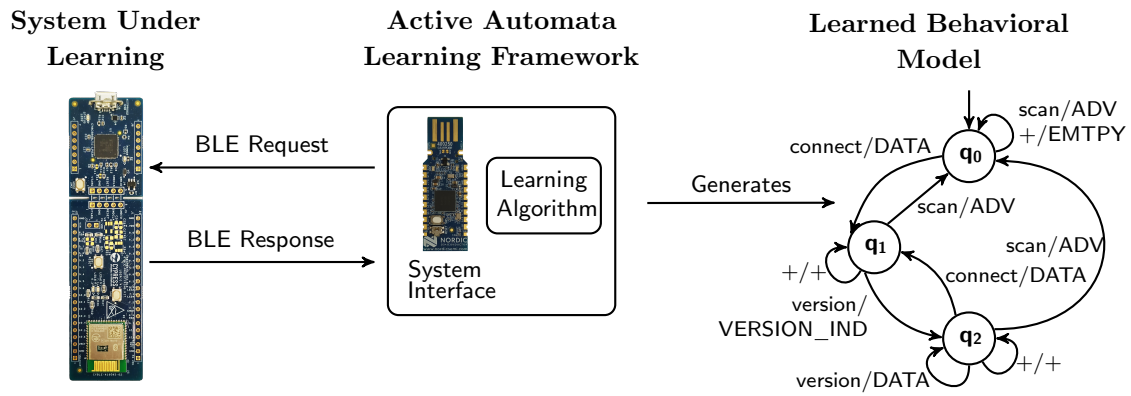


Figure 4.1: General learning framework for learning behavioral models of BLE devices. A black-box BLE device serves as SUL.

has more than 3 000 pages, but still allows for a lot of freedom in the actual implementations.

This chapter shows how active automata learning can be used to automatically identify behavioral models of BLE devices. Figure 4.1 gives an overview of the learning setup for BLE devices. This chapter introduces the developed active automata learning framework. The active automata learning framework consists of a learning interface that allows sending BLE requests to the SUL. The SUL itself is a BLE device from which we identify the behavioral model that represents the BLE stack implementation. The received BLE responses are then used by the learning algorithm to construct the behavioral model. We evaluate this learning setup on six different BLE devices. The learned models show behavioral differences within all the models. Furthermore, we also observe that one device crashes when executing a sequence of valid BLE packets.

The chapter is organized as follows. Section 4.2 explains the methodology for automatically learning behavioral models of BLE devices. Section 4.3 presents the evaluation performed on six BLE devices, including different levels of the BLE stack. Furthermore, we show how the learned models can be used to fingerprint black-box devices. Finally, Section 4.4 concludes this chapter with a summary and a discussion of the results.

4.2 Learning Setup

This section describes the developed automata learning framework to learn behavioral models of BLE devices. Figure 4.2 shows the following five components of the learning framework and their interaction with each other: (1) the learning algorithm, (2) the learning interface, (3) the mapper, (4) the BLE central device, and (5) the BLE peripheral device. The learning algorithm (1) generates the behavioral model using active automata learning. The learning algorithm retrieves the necessary information from the learning interface for this purpose. The learning interface (2) is responsible for taking care of any unexpected observations received from the device, e.g., non-deterministic behavior due to message loss. The mapper (3) is responsible for abstraction and interacts with the BLE central device. The BLE central device (4) is controlled by the framework to send and receive BLE packets from the BLE peripheral device. The BLE peripheral device (5) is the SUL from which we learn the behavioral model. In the remainder, we refer to the central device as *central* and to the peripheral device as *peripheral*.

In Section 2.3.2, we provided a message sequence diagram (Figure 2.8) describing the connection and pairing procedure between two BLE devices. For this case study, we learn the behavioral model of the peripheral, since it is easier to control the central that initiates the connection and pairing procedure. The message sequence diagram in Figure 2.8 shows 13 different

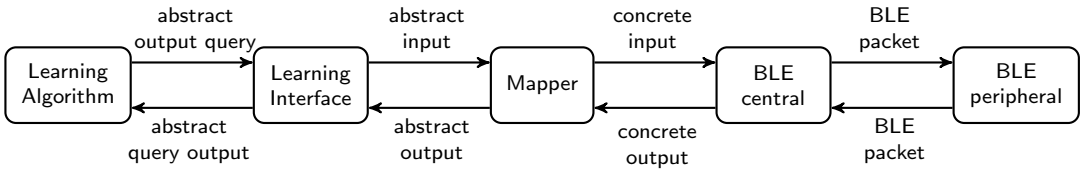


Figure 4.2: Our presented learning framework for learning BLE devices distinguishes five components.

BLE packets that can be considered as inputs for learning. Learning real systems, especially if the communication is wireless, would hardly be feasible with such a large input alphabet. The biggest problem, in this case, is that long sequences of unexpected inputs can cause timeouts, where the peripheral enters and remains in a standby state. For learning the connection and pairing procedure of BLE devices, this can especially be a problem, e.g., if we perform many inputs that belong to the connection procedure during the pairing procedure. To solve this problem, we separate the input alphabet in a logical way, where the first subset considers the inputs for the connection procedure and the second considers the inputs for the pairing procedure. Both models show the initialization of the pairing procedure, which is a behavioral overlap that connects both models. In this way, we learn two models, one formalizing the connection behavior and the other the pairing behavior.

4.2.1 Learning Algorithm

For learning behavioral models of BLE devices, we use active automata learning algorithms. The connection and pairing procedure is represented by a request/response scheme, which corresponds to the behavior of a reactive system. Additionally, we assume that the BLE devices behave deterministically. As motivated in Section 2.3.1, we can model the deterministic behavior of reactive systems with Mealy machines.

In the BLE protocol, requests can be performed by the initiator of the connection but also by the responder. The message sequence diagram in Figure 2.8 shows that the peripheral can also send requests for connection parameters that must be answered by the central. No reaction to these requests from the peripheral could limit the observable state space. Consequently, we also include response messages as inputs for learning the behavioral model. These responses are treated as normal inputs, meaning that they are performed even if they are not requested.

We applied the improved version of the L^* algorithm [17] for Mealy machines with the counterexample processing of Rivest and Schapire [156]. This learning setup was chosen according to the results of Aichernig et al. [15]. They recommend this setup as one that requires fewer queries but still learns correctly.

For testing the equivalence between the learned hypothesis and the SUL, we use conformance testing as described in Section 2.2.2. To generate the finite test suite, we use an approach that provides state coverage but also includes randomness to enable the exploration of new behavior. For this, we build sequences that traverse to particular states and then execute a random sequence of inputs. For reaching a particular state, we can use the access sequences that are derived during active learning. The number of random walks per state is defined by $n_{\text{test}} \in \mathbb{N}$ and the length of the random input sequence by $n_{\text{len}} \in \mathbb{N}$.

The learning algorithm interacts only with the learning interface. This interaction consists of the learning algorithm sending output queries to the learning interface. Then, the learning algorithm receives the corresponding query outputs back from the learning interface.

4.2.2 Learning Interface

The learning interface is responsible for providing the learning algorithm with *representative* query outputs from the received output queries. Representative, when learning BLE devices, means that the query outputs do not contain any outputs that show communication errors between the central and the peripheral device. If such a communication error is detected, the learning interface should take countermeasures to generate the correct query output.

In addition, learning behavioral models of physical devices should require as few queries as possible. The interaction with the SUL might be expensive in terms of time required to reset the SUL, execute an input and observe an output. Additionally, reducing the number of interactions is useful since every performed request introduces the risk that BLE packets might get lost or arrive delayed, which introduces non-deterministic behavior. To reduce the interaction with the SUL, we use the proposed improvements by Rivest and Schapire [156]. Furthermore, we also save every performed query in a cache as described in Section 3.2. Moreover, the cache is also used to handle non-deterministic behavior.

The tasks of the learning interface can be divided into two areas: Ensuring a reliable reset of the SUL and handling non-deterministic behavior.

Reliable Reset. The learning algorithm L^* requires that the SUL is reset to the same initial state before performing a query, i.e., an output query is always assumed to be performed starting from an initial state. This requirement can be challenging when learning behavioral models of physical devices. A manual reset after each query would be tedious and time consuming, which would hamper the applicability and feasibility of the learning approach. A physical or hard reset for BLE devices would mean that either a reset button on the device is pushed or the power supply is removed and added again.

Since such a hard reset is infeasible, we reset a connection by sending BLE requests from the central to the peripheral. According to the BLE specification [87], a termination request (`termination_request`) should abort an established connection. The central performs a termination request to reset the connection to the peripheral after executing the last input of a query. Another method to reset a connection would be that the central scans again for advertisements, i.e., it sends a scan request to the peripheral. We assume that the peripheral returns to the *advertising* state after a connection has been reset.

Some of the tested devices enter a *standby* state, after a certain amount of time without any interaction. This can be a problem in learning, where long input sequences are sent, but no valid connection to the peripheral has been established. To avoid such timeouts, we always establish a valid connection between the central and the peripheral after a reset has been performed. For establishing this connection, the central performs a scan and connection request and immediately aborts the connection by sending another scan request to the peripheral. We expect to receive a response to each of these requests, which allows us to check whether the peripheral is still reachable.

We call the event where we do not receive a response to a connection request a *connection error*. In case of a connection error, we repeat the establishment of a connection at most $n_{\text{error}} \in \mathbb{N}$ times. If no connection can be established within n_{error}, we ask the user to perform a hard reset on the device.

Modern learning libraries provide an interface that enables the implementation of resets via methods that are executed before and after performing an output query. The methods are called `pre` and `post` respectively. For learning the pairing procedure, we additionally perform in the `pre` method all the requests that are necessary before the pairing procedure could be started. This means the central initiates a connection and responds to all requests from the peripheral. To reset the pairing procedure, we have to consider if the communication is encrypted. In this case, we additionally send a request to pause the encryption. Afterwards, we terminate the connection in any case using a termination request.

Handling Non-Determinism. One challenge in learning behavioral models of communication protocols is that the observed behavior can be non-deterministic. In cases where the SUL behaves non-deterministically, countermeasures are required to deal with the non-deterministic observations. For learning BLE devices, we assume that the devices behave, in general, deterministically. Our case study shows that this assumption is problematic for some of the investigated BLE devices. Furthermore, in Chapter 8 we will discuss an implementation that considers a more relaxed assumption about non-deterministic behavior for learning communication protocols.

In the literature, non-deterministic behavior is a common problem in learning communication protocols. Commonly proposed strategies [63, 170] consider the definition of proper timeouts to wait for messages. Another approach [61] is to simply repeat queries a certain number of times and then select the most commonly observed output. In the work of Fiterău-Broştean et al. [63], they also introduce the concept of validating the observed value against cached values. Following a similar idea, we also set timeouts for responses and use cached values to detect non-determinism.

Example 7 (Non-deterministic Behavior in BLE) *Consider that the central sends the following input sequence to the peripheral*

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

A possible expected output sequence on this input sequence would be

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

So the central scans for advertisements from the peripheral, the peripheral acknowledges the connection request but sends a feature request to the central. The central then answers to the feature request, where DATA *refers to an empty BLE packet that is sent as a kind of keep-alive message. Finally, the peripheral accepts the pairing request. However, we might observe the following output sequence if the feature response from the central got lost*

$$\mathsf{ADV} \cdot \mathsf{FEATURE\_REQ} \cdot \mathsf{DATA} \cdot \mathsf{PAIRING\_FAILED}.$$

In this case, the pairing request was rejected since the previous parameter negotiation was not completed. However, the difference in the output sequence was not directly evident from the lost message, since the peripheral always sends keep-alive messages.

Repeating each output query multiple times would have a huge impact on the runtime. Therefore, we prefer to repeat the query only in cases where we detect non-determinism. For this purpose, we utilize the caching structure provided by the used learning library. The used cache is organized in a tree-based structure, where nodes are labeled with outputs. In the case we do not observe the expected output defined by the node, we repeat the query $n_{\text{cache}} \in \mathbb{N}$ times and then pick the most frequent output as the label for the node. However, to ensure that our algorithm still terminates, we allow nodes to be updated a maximum number of observed non-deterministic errors $n_{\text{nondet}} \in \mathbb{N}$.

The learning interface interacts with the learning algorithm to process received output queries and provide the corresponding query outputs. Furthermore, the interface interacts with the mapper component, sending single inputs that should be executed on the SUL and receiving the corresponding outputs.

4.2.3 Mapper

Learning a behavioral model of a BLE device considering all possible BLE packets would not be feasible in a certain amount of time. To overcome this problem, we introduce an abstraction

in the form of a mapper component as explained in Section 2.2.2. Hence, the learned model represents the behavior on a more abstract level.

The mapper translates abstract inputs it receives from the learning interface into concrete BLE packets that can be sent from the central to the peripheral. The mapper receives the corresponding responses from the central, where it then forwards the response from the peripheral. The response from the peripheral may contain several BLE packets. Hence, the mapper receives a list of packets, which the mapper translates into one concrete output. For this purpose, the received list is reduced to a set, which is then concatenated into a single string in alphabetical order. This postprocessing is beneficial for possible non-deterministic behavior due to a different order of transmissions.

For learning BLE devices, we consider the following abstract input alphabet for the connection procedure

$$I_C^{\mathcal{A}} = \{\mathsf{scan\_req}, \mathsf{connection\_req}, \mathsf{length\_req}, \mathsf{length\_rsp}, \mathsf{feature\_req}, \mathsf{feature\_rsp}, \mathsf{mtu\_req}, \mathsf{version\_req}, \mathsf{legacy\_pairing\_req}\},$$

and the following abstract input alphabet for learning the pairing procedure

$$I_P^{\mathcal{A}} = \{\mathsf{legacy\_pairing\_req}, \mathsf{confirm}, \mathsf{random}, \mathsf{encryption\_request}, \mathsf{start\_encryption\_rsp}\}$$

For generating the concrete BLE packets, the packet manipulation library Scapy [158] has been used. For the concretization of BLE packets, we consider the default values of the library.

Example 8 (Concretization for BLE) *The abstract input* length_req *would be translated to the following concrete BLE packet following the* Scapy *syntax* BTLE/BTLE_DATA/BTLE_CTRL/LL_LENGTH_REQ(*max_tx_bytes*, ...)*, where the packet* LL_LENGTH_REQ *has field values such as max_tx_bytes which is translated to a concrete value. The concrete values are selected based on the predefined values in* Scapy*, which conform to standard values that are commonly accepted by the peripheral to establish a valid connection and pairing.*

Example 9 (Abstraction for BLE) *After the central sends a concrete packet to the peripheral, the peripheral responds with several BLE packets. The central then forwards a list of received BLE packets to the mapper. For example, the central provides the mapper with the following list of BLE packets*

[BTLE/BTLE_DATA,
BTLE/BTLE_DATA/BTLE_CTRL/LL_LENGTH_REQ(...),
BTLE/BTLE_DATA,
BTLE/BTLE_DATA,
BTLE/BTLE_DATA/L2CAP_Hdr/ATT_Hdr/ATT_Exchange_MTU_Request(...),
BTLE/BTLE_DATA,
BTLE/BTLE_DATA]

The mapper then translates this list into the following abstract output "ATT_Exchange_MTU_Request|ATT_Hdr|BTLE|BTLE_CTRL|BTLE_DATA|
L2CAP_Hdr|LL_LENGTH_REQ" *and forwards the abstract output to the learning interface.*

Example 9 shows that the mapper provides rather long output strings. For the sake of simplicity, we usually abbreviate the abstract output to shorter strings in the remainder of this thesis. However, the translation to longer strings in the learned models should be straightforward. For example, instead of writing the long output of Example 9, we simply write ATT_Exchange_MTU_Request|LL_LENGTH_REQ instead.

In addition, we require the mapper to be stateful for learning the pairing procedure. For this purpose, the mapper keeps track if encryption is enabled or disabled. Hence, BLE packets are encrypted if encryption is enabled and are decrypted before they are sent to the learning interface. Furthermore, the mapper stores all intermediate generated and received values to generate valid responses and the corresponding encryption key.

4.2.4 BLE Central

The central device is another BLE device that communicates with the peripheral. To send manually crafted packets from the mapper to central, we flashed a BLE device with custom firmware. The firmware and the driver implementation are taken from the SweynTooth [67] repository.

After sending a BLE packet, the central collects responses from the peripheral. The central collects packets by listening for packets that are addressed to the central. Since the peripheral may respond with more than one BLE packet, the central collects a list of BLE packets. First, the central listens $n_{\text{min}}^{\text{rsp}} \in \mathbb{N}$ times for responses. The driver then checks if the list contains any *convincing* response. We denote a response as *convincing* if it contains any other packets then BTLE|BTLE_DATA. If the response is convincing after listing $n_{\text{min}}^{\text{rsp}}$ times, the central sends the list of received BLE packets to the mapper. Otherwise, it increases the number of listening attempts $n_{\text{max}}^{\text{rsp}}$. After $n_{\text{max}}^{\text{rsp}}$ it returns the received list of packets to the mapper.

4.2.5 BLE Peripheral

The peripheral represents our SUL. According to the message sequence diagram shown in Figure 2.8, we assume that the peripheral sends advertisements and accepts connection requests. After receiving a resetting BLE packet, e.g., a termination or scan request, we assume that the peripheral returns to the advertising state.

Learning the behavioral model of the peripheral device is motivated by two reasons. First, peripheral devices represent components that represent utilities for achieving a main task, e.g., a wireless mouse. Peripherals are mostly components from third-party resources, where the insight is limited. Hence, we identify black-box systems, in order to test and evaluate a whole ecosystem consisting of many heterogenous components. Second, the learning setup is easier to control by the central device, since the central initiates the connection and the pairing procedure.

4.3 Evaluation

The following section describes the performed evaluation on learning six different BLE devices. First, we provide details about the concrete learning setup. Then, we provide the results for learning the connection procedure, followed by the results for the pairing procedure. The implemented framework, the learned models, and the learning results are available **online** [144]. We conclude this section by presenting a case study on learning the behavioral model of the BLE device built into a Tesla Model 3 and its key fob.

4.3.1 Learning Setup

For learning, the learning library AALpy [129], version 1.0.1, was used. The Python library implements state-of-the-art learning algorithms including the L^* algorithm with the improvement of Rivest and Schapire. To create a smooth integration of all components, we base the whole implementation on Python 3. More concretely, we used Python 3.9.0 for the performed evaluation.

The cache used in the learning interface is based on the cache implementation from AALpy, but was adapted to react to non-deterministic behavior. For the generation and parsing of con-

Table 4.1: The investigated BLE devices in the conducted case study.

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

crete BLE packets, we used the Python library Scapy [158] (version 2.4.4). To parse all packets some modifications to the used version of Scapy were necessary. The applied modification should be available starting from Scapy v2.4.5.

As the central device, we used the Nordic nRF52840 Dongle and the Nordic nRF52840 Development Kit. We flashed the central devices with the firmware provided by the SweynTooth repository [67].

4.3.2 BLE devices

Table 4.1 and Figure 4.3 show the six investigated BLE devices. For the case study, devices from different manufacturers, but also different devices from the same manufacturer were selected. We also investigate the behavior of BLE devices built into well-known hardware, like the Raspberry Pi 4. All the system on the chip (SoC) implement the Bluetooth 5 standard [87] and run an example application that starts in the advertising state. If possible the preinstalled applications from the manufacturers were used. Otherwise, we flashed an example application provided by the manufacturer's software development kits. In the remainder of this thesis, we refer to the devices by their SoC name.

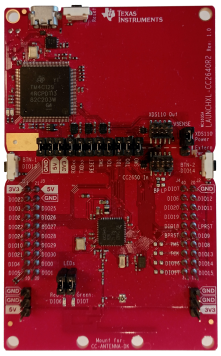
4.3.3 Connection-Procedure Evaluation

For learning the connection procedure, we learn the behavior on the upper part of the message sequence diagram depicted in Figure 2.8 including the start of the pairing procedure. For learning, we set the parameters $n_{\text{error}} = n_{\text{cache}} = n_{\text{nondet}} = 20$. For checking equivalence, we use the equivalence oracle `StatePrefixEqOracle` of AALpy, which implements a model-based testing technique that provides state coverage in combination with randomized input sequences. We adapted the implementation of the equivalence oracle to deal with non-deterministic behavior and connection errors. We set $n_{\text{test}} = n_{\text{len}} = 10$.

The central listens for $n_{\text{min}}^{\text{rsp}} = 10$ and $n_{\text{max}}^{\text{rsp}} = 20$ attempts in order to get a response from the peripheral. For the nRF52832 this configuration had to be adapted since the device responded slower than the other devices. Therefore, we increase the parameters to $n_{\text{min}}^{\text{rsp}} = 20$ and $n_{\text{max}}^{\text{rsp}} = 30$. To quickly check if the device is still reachable during the reset procedure, we set $n_{\text{min}}^{\text{rsp}} = 5$ and $n_{\text{max}}^{\text{rsp}} = 50$. The resetting action itself can be executed quickly, since there is no need to check the output. Hence, we set for the termination_ind the parameters $n_{\text{min}}^{\text{rsp}} = n_{\text{max}}^{\text{rsp}} = 1$.

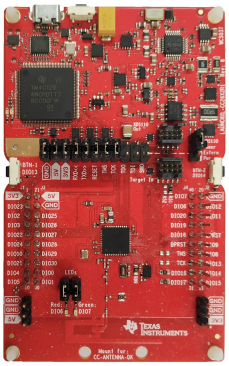
All experiments were executed on an Apple MacBook Pro 2019 with an Intel Quad-Core i5 operating at 2.4 GHz and with 8 GB RAM, running macOS Catalina (version 10.15.7).

Table 4.2 shows the results for learning the connection procedure. First, the table only presents the results for five out of six devices. One device, CC2640R2, was not learnable with the presented setup since it always behaved non-deterministically, even with the conducted countermeasures. We will later present how we still managed to come up with a learning setup that enables deterministic learning for the CC2640R2. Furthermore, we also had to slightly adapt the learning setup for the CC2652R1 and the CYW43455 due to reliability issues in establishing a connection. Hence, the behavioral model is learned starting from an already established connection. Therefore, the `pre` method is extended by a scan_req and connection_req.



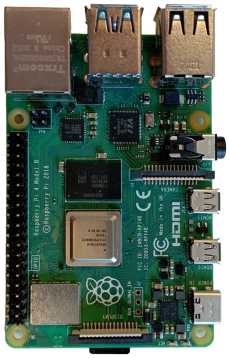
(a) LAUNCHXL-CC2640R2, SoC: CC2640R2

(b) LAUNCHXL-CC2650, SoC: CC2650



(c) LAUNCHXL-CC26X2R1, SoC: CC2652R1

(d) CY8CPROTO-063-BLE, SoC: CYBLE-416045-02



(e) Raspberry Pi 4 Model B, SoC: CYW43455

(f) decaWave DWM1001-DEV, SoC: nRF52832

Figure 4.3: The investigated BLE devices in the performed case study.

Table 4.2: Learning results for the performed evaluation on learning the connection procedure implemented by the different BLE devices. One device could not be learned due to non-deterministic behavior.

	CC2650	**CC2652R1**†	**CYBLE-416045-02**	**CYW43455**†	**nRF52832**
# States	5	4	3	16	5
Total time in minutes (min)	23.61	5.57	12.00	65.12	126.24
Learning (min)	18.22	3.46	9.41	51.07	73.80
Conformance checking (min)	5.39	2.11	2.59	14.05	52.44
# Output queries	405	196	243	784	405
# Output query steps	1542	588	747	3136	1459
# Conformance tests	59	44	32	164	50
# Conformance test steps	626	467	344	1958	580
# Connection errors	526	-	292	-	459
# Non-deterministic outputs	5	1	0	3	1

The learned models have between three and 16 states. This already shows the behavioral differences between the investigated devices. The learning runtime is between 5.6 minutes and 2.1 hours. Each model could be learned within one learning round, i.e., only one equivalence query was required. Even for models with the same state space the runtime differs noticeably. For example, both CC2650 and nRF52832 have five states, but learning the nRF52832 took more than five times longer. This difference occurs due to extended waiting time for responses required when learning the nRF52832. The results in Table 4.2 also underline the need for countermeasures against connection errors and non-deterministic errors to learn a behavioral model of a BLE device.

Even though CC2650 and nRF52832 have the same state space, they still show behavioral differences. Figure 4.4 and Figure 4.5 show the simplified learned model of the CC2650 and nRF52832 respectively. The red transitions highlight one difference between the two learned models: the CC2650 and the nRF52832 react differently on unrequested length responses (length_rsp). The CC2650 remains in the same state, whereas the nRF52832 returns to the initial state, which means the connection is terminated. According to BLE standard [87], both behaviors are permitted. This also shows that the specification leaves some freedom in the actual implementation of the BLE stack.

Moreover, a manual analysis of the learned models shows a violation of the BLE specification. Figure 4.6 depicts a simplified version of the learned model from the CC2652R1. The red edges of the model show that a version request is always answered by a version response. This is a violation of the following statement in the BLE specification [87]:

"If the Link Layer receives an LL_VERSION_IND PDU and has already sent an LL_VERSION_IND PDU then the Link Layer shall not send another LL_VERSION_IND PDU to the peer device."

Note that the initial state in the depicted model of the CC2652R1, Figure 4.6, considers that the central and the peripheral are already connected. We made this assumption for the CC2652R1 and for the CYW43455. For the CYW43455, establishing a large number of connections was not reliable. The CYW43455 non-deterministically rejects connection requests. For the CC2652R1, a deterministic pattern was observed where no connection was established. The device stops sending advertisements after receiving two connection requests, even though the connections are terminated. For example, the following input sequence would trigger a stop of sending advertisements:

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

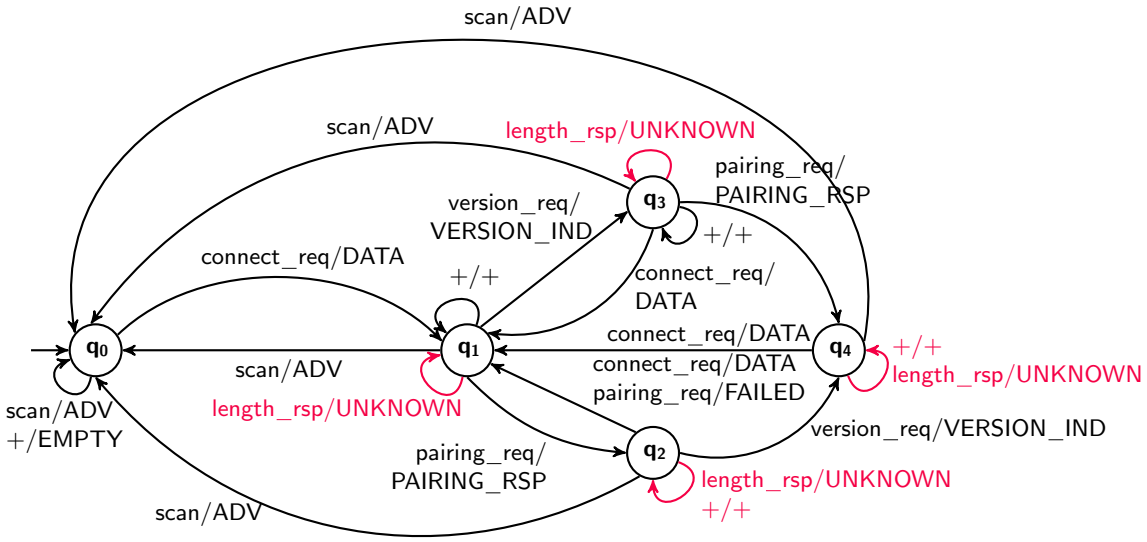


Figure 4.4: Learned model of the CC2650. The red transitions highlight the behavioral difference to the nRF52832 (Figure 4.5) for a non-requested length_rsp. For simplification, some inputs and outputs are abbreviated with a '+' symbol.

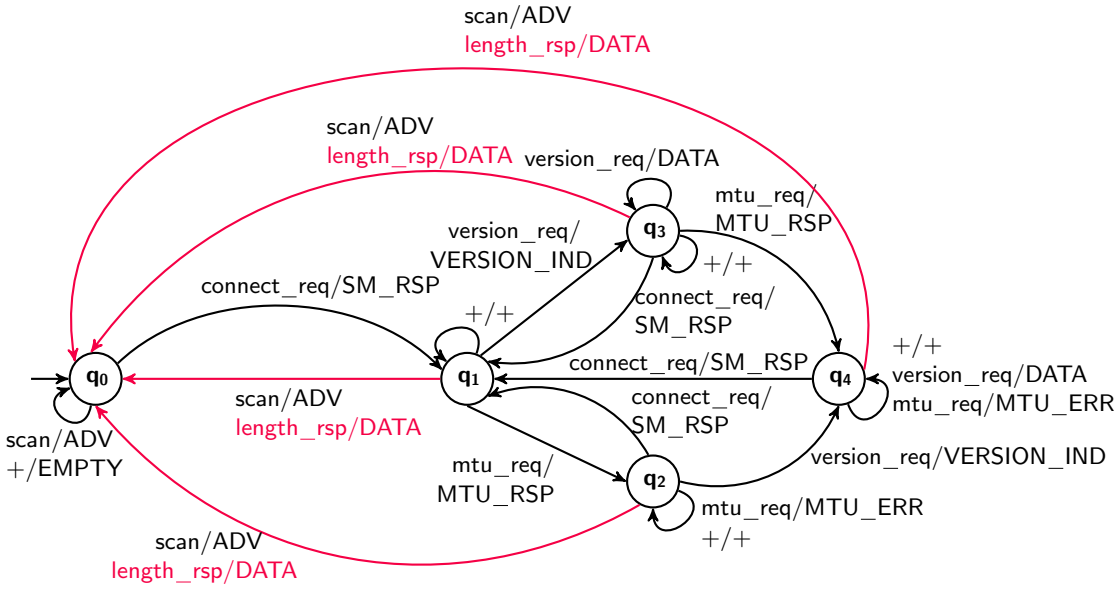


Figure 4.5: Learned model of the nRF52832. The red transitions highlight the behavioral difference to the CC2650 (Figure 4.4) for a non-requested length_rsp. For simplification, some inputs and outputs are abbreviated with a '+' symbol.

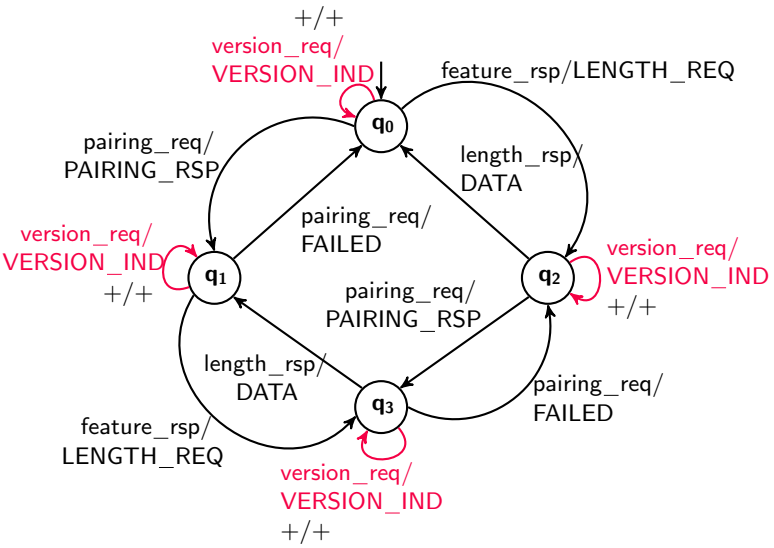


Figure 4.6: Learned model of the CC2652R1. For simplification, some inputs and outputs on the transitions of the learned model are abbreviated with a '+' symbol. The red transitions show a violation of the BLE specification [87], since the device always responds to VERSION_REQ, where the BLE specification states that an answer should be only received once.

Table 4.3: Learning results for learning the CC2640R2 which behaved non-deterministically. The device could be learned by reducing the input alphabet by one input. We show the results on three setups where each excludes one input.

	no pairing_req	**no** length_req	**no** feature_req
# States	6	11	11
Total time (min)	26.40	47.57	40.29
Learning time (min)	16.94	30.73	28.29
Conformance checking time (min)	9.46	16.84	11.70
# Output queries	384	705	704
# Output query steps	1474	3143	3143
# Conformance tests	61	115	111
# Conformance test steps	712	1406	1371
# Connection errors	449	822	821
# Non-deterministic outputs	1	10	2

Only by performing a hard reset, does the device start sending advertisements again. Hence, we remove the connection_req and scan_req from the input alphabet and establish a connection before querying the SUL.

We also encountered problems in learning the CC2640R2, since this device always behaved non-deterministically. The following input sequence always leads to non-deterministic observations:

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

Earlier in the learning process the received query output corresponds always to the following output sequence:

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

Later in learning the received query output sequence changes to the following output sequence:

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

Still, we managed to learn the behavioral models of this device. For this, we define different experiment setups. In each experiment setup, we removed one of the nine inputs of the input

Table 4.4: Learning results for the performed evaluation on learning the pairing procedure implemented by the different BLE devices. Only a subset of the devices were learned, since not all devices have pairing mode activated.

	CC2640R2	**CC2650**	**CYW43455**
# States	11	10	6
Total time (min)	133.01	312.37	52.72
Learning time (min)	116.95	201.34	38.83
Conformance checking time (min)	16.06	111.03	13.89
# Output queries	487	453	223
# Output query steps	3142	2869	1012
# Conformance tests	110	100	60
# Conformance test steps	273	601	287
# Non-deterministic outputs	133	80	29
# Cache updates	1	3	0
# Hard resets	6	11	0

alphabet. We defined three different setups for the CC2640R2. With this approach, three variations of behavioral models representing the CC2640R2 could be learned. Table 4.3 shows the results for learning the CC2640R2. One experimental setup does not consider the pairing_req, one without the length_req, and the third one without the feature_req. The learned models have between six and eleven states, where the required time for learning scales with the state space. We see that considering the pairing_req for learning leads to almost twice as many states as without considering it. The results also show that the number of connection errors is higher compared to the results presented in Table 4.2. For the experiment that does not consider the length_req, we still observe a lot of non-deterministic behavior compared to the other experiments.

4.3.4 Pairing-Procedure Evaluation

For learning the pairing procedure, we consider all BLE packets that are required to establish an encrypted communication using the legacy pairing method. According to Figure 2.8, we include all the BLE packets starting from the execution of a pairing request.

For our case study on the pairing procedure, we selected three devices from our previously performed case study: CC2640R2, CC2650, and CYW43455. These devices accept a legacy pairing request and allow an efficient exchange of messages. For creating all the information required to establish an encrypted communication, we adapted the security manager interface provided by the SweynTooth [67] repository. Since this C/C++ interface uses the Linux BLE stack implementation BlueZ [152], executing the learning experiments on a Linux-based system was more convenient for this case study. Hence, we executed all experiments for learning the BLE pairing procedure on an Ubuntu 20.04.2 LTS running on an HP EliteBook 840 G2 with an Intel i5-2000 operating at 2.2 GHz with 16 GB RAM. Note that also the experiments presented in Section 4.3.3 could have been run on an Linux-based system, but we decided to use a more state-of-the-art hardware setup instead.

The parameter configuration was changed for learning the pairing procedure, since the learning interface implements a different error-recovery strategy. This strategy involves that the device can be hard reset in the case of encountering too many connection errors or errors due to non-deterministic faults. The interface allows only a small number of connection errors and errors due to non-deterministic behavior. For this, we set $n_{\text{error}} = 5$, $n_{\text{cache}} = 3$, and $n_{\text{nondet}} = 3$. In case the maximum number of errors is reached, the user is asked to hard reset the device or to abort the learning procedure.

Table 4.4 presents the results on learning the pairing procedure of three different devices. Again, all three learned models were different, ranging from a state space between six and eleven

states. Learning took between 52.7 minutes and 5.2 hours. We again see a large difference in the runtime between models with the same state space, CC2640R2 vs. CC2650. The table also provides information about the number of performed updates in the cache, which indicates the number of replacements of observed and cached outputs. Second, the table also lists the number of performed hard resets. Connection errors are not reported since the connection request is not considered here.

Hard resets were required for learning the CC2640R2 and the CC2650. The CYW43455 could be learned without any user interaction. The causes for the required hard resets were different. The CC2640R2 stops accepting pairing requests after an unknown number of received BLE packets. In order to again accept pairing requests, the device needs to be hard reset when the maximum number of non-deterministic errors is reached.

We observe a different reason for hard resetting the CC2650 during learning: After executing the learning experiment multiple times, we observe that the device stops responding when it receives a certain input sequence. The device crashes on the following input sequence, where inputs surrounded by $\langle \ldots \rangle$ indicate encrypted messages:

$$\begin{aligned} &\mathsf{connection\_req} \cdot \mathsf{pairing\_req} \cdot \mathsf{confirm} \cdot \mathsf{random} \cdot \\ &\mathsf{encryption\_req} \cdot \langle \mathsf{pause\_encryption\_req} \rangle \cdot \mathsf{terminate\_ind}. \end{aligned}$$

The central performs a valid key exchange procedure in order to establish an encrypted communication. To confirm that the encrypted communication channel is successfully established, the peripheral expects to receive an encrypted start_encryption_rsp. However, the peripheral instead sends an encrypted request to terminate the encryption, $\langle \mathsf{pause\_encryption\_req} \rangle$. Also after performing a termination indication, the CC2650 does not return to advertising and requires to be hard reset. This shows again that an unexpected sequence of valid BLE packets can make the devices unreachable.

Figure 4.7 shows the learned behavioral model of the pairing procedure implemented by the CYW43455. The learned model strictly follows the procedure depicted in the message sequence shown in Figure 2.8. All other received messages are basically ignored. The transition colored in red indicates the start of the encrypted communication.

Figure 4.8 shows the learned model of the CC2640R2. We see a clear difference between the learned model of the CYW43455 depicted in Figure 4.7. The red transition between state q_3 and q_4 indicates again the start of encrypted communication. First, we see that a pairing request resets the existing pairing procedure until the encryption is enabled. Furthermore, we see a loop between the states q_2 and q_3, which shows that the confirm value replaces confirm values that have already been sent, which in turn must be followed by the corresponding random share. The models are also different after encryption is enabled. We see in the states q_9 and q_{10} that a new pairing request can be sent, and every second pairing request is accepted. If in any case a second enc_req is received, the device stops responding to any further requests as shown in state q_7.

4.3.5 Fingerprinting

We want to use the learned models to detect which BLE device or which BLE stack implementation is used. Our results show that all learned models were different for each device. Hence, the learned models can be used to fingerprint the considered devices. Instead of comparing each model with the others, we derive one input sequence that shows unique behavior when executed on each device. For this purpose, we consider the models generated during learning the connection procedure, to make this approach applicable to all devices.

To create a fingerprinting sequence, we can use the outputs stored in the observation table, which is generated during learning. By comparing the rows of the generated observations tables, we can derive a sequence that fingerprints the devices. Table 4.5 presents all the outputs of the different devices that can be observed after an initial connection has been established. The table

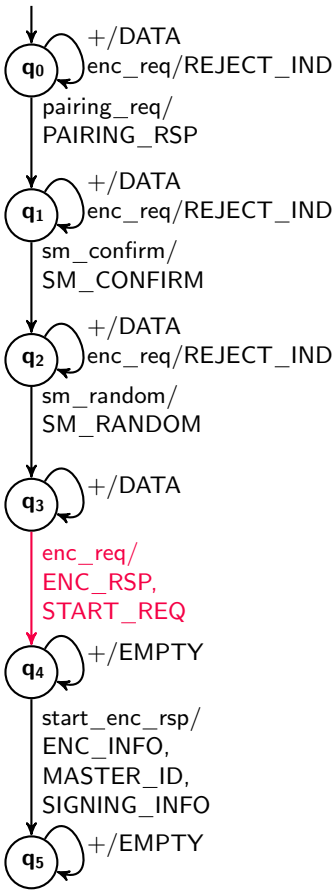


Figure 4.7: Learned model of the CYW43455 pairing procedure. For simplification, some inputs and outputs are abbreviated with a '+' symbol. The red transition indicates the start of encrypted communication.

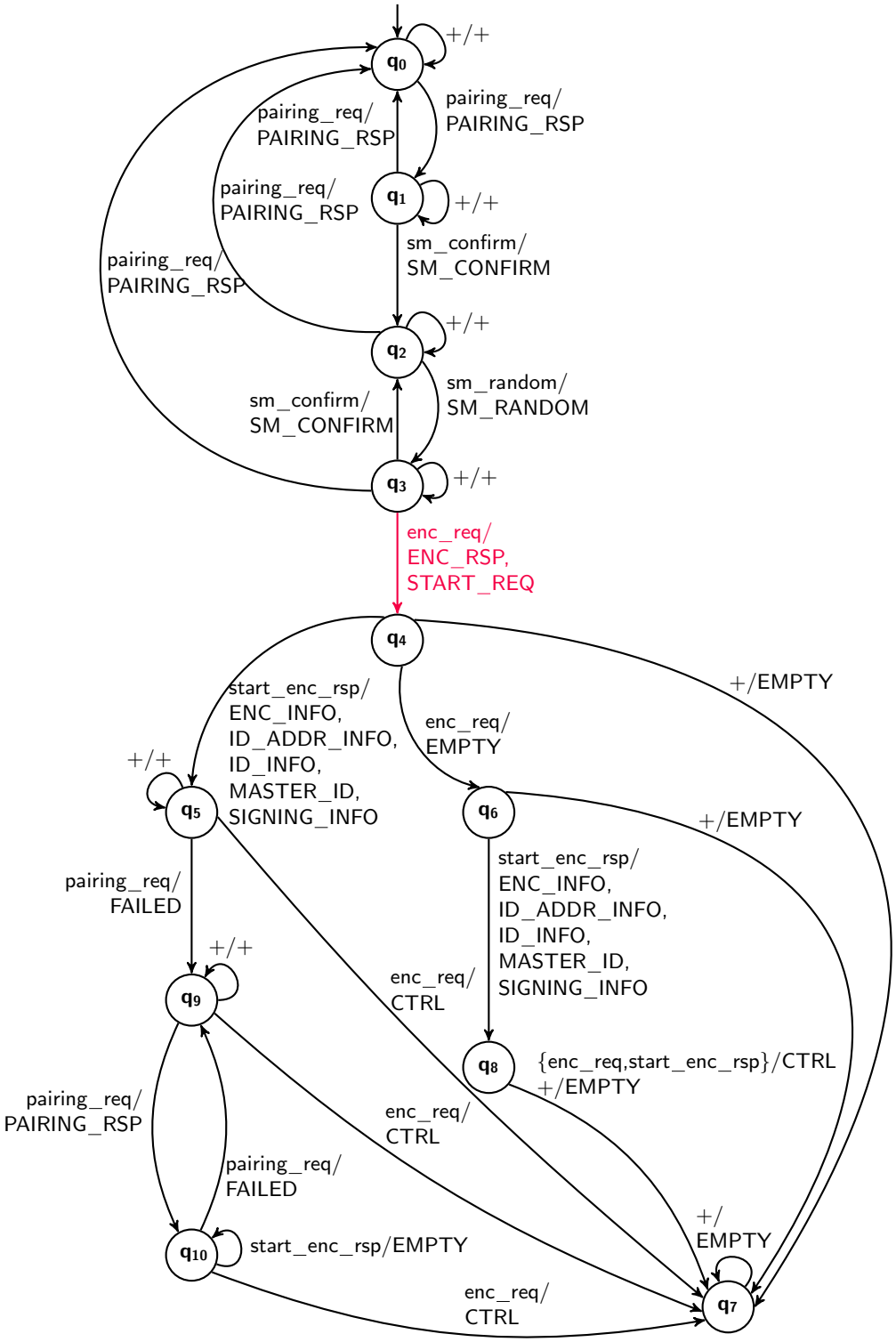


Figure 4.8: Learned model of the CC2640R2 pairing procedure. For simplification, some labels on the transitions of the learned model are abbreviated with a '+' symbol. The red transition indicates the start of encrypted communication.

Table 4.5: Different outputs on each input symbol after a previous connection_req has been performed. The different outputs enable us to distinguish the different devices and create a fingerprint sequence.

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

shows that already performing a feature_rsp enables to distinguish four out of six devices. To distinguish the remaining two devices, either a version_req, length_req, or length_rsp can be performed.

We can now concatenate these inputs into a sequence that yields a unique output sequence for every device. We assume that scan_req resets a connection and that connection_req establishes a connection. Hence, one possible fingerprinting sequence can be the following input sequence

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

Executing this input sequence on each of the investigated devices leads to a different output sequence. For example, the observed output on the nRF52832 would be

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

and on the CC2650 it would be

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

even though both learned models have the same number of states. We see that these two devices are already distinguishable after performing a connection request. However, to distinguish from the other devices, the whole output sequence would be necessary. Note that there also exists other input sequences that enable fingerprinting the devices. The outputs presented in Table 4.5 already show several different possibilities.

Considering that the fingerprinting sequence is relatively short, we assume that generating such a sequence by random sampling would also be sufficient for the set of investigated devices. However, the models only need to be learned once and enable the generation of new fingerprinting sequences without requiring additional interaction with the SUL. The manual analysis should then be replaced by similarity checks based on model-based testing techniques as proposed by Lee and Yannakakis [104] or Tappler et al. [170].

4.3.6 Case Study on Tesla Model 3

The previously presented case study considers mostly devices on specific development kits. To show that our BLE learning approach is not only applicable for BLE development kits, we also evaluate a more advanced black-box scenario.

For another practical evaluation, we consider a case study from the automotive industry. Texas Instruments Incorporated [103] proposes several use case scenarios for the usage of BLE in a vehicle. One application scenario is the keyless access to a car via BLE. One car manufacturer that implements such keyless access via BLE is Tesla, Inc. for their Tesla Model 3 and the Tesla Model Y. The car can be unlocked via an app on the mobile phone, but also by a key fob that uses BLE.

If BLE is used for car access, it is essential that the protocol is correctly implemented and used. Otherwise, a risk exists that the car might be unlocked by an unprivileged person. The

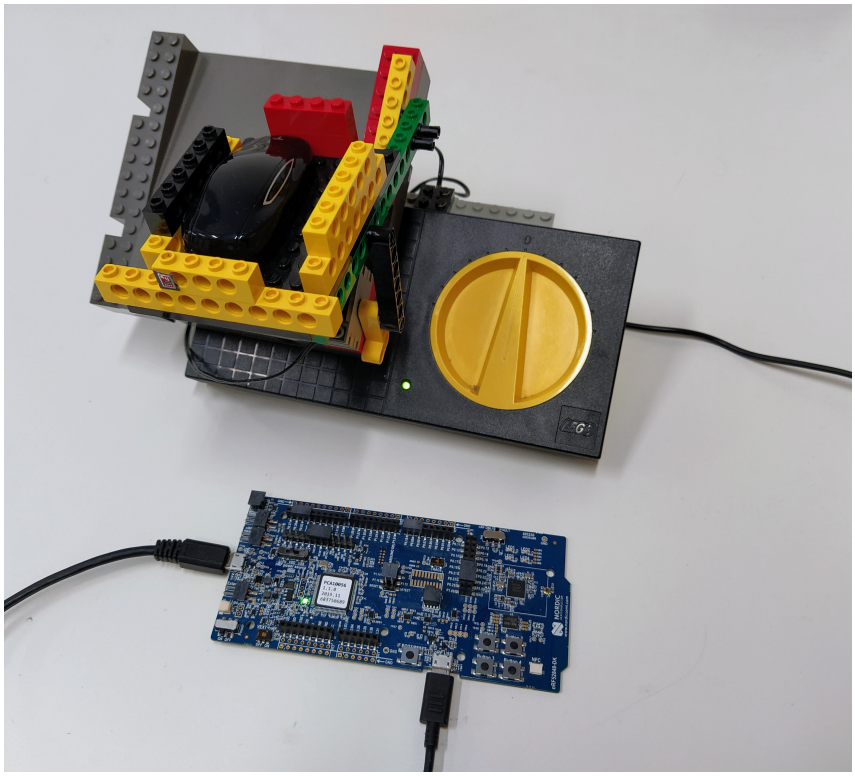


Figure 4.9: Advanced learning setup for key fob. We use Lego® to keep the key fob constantly in motion. This prevents the key fob from stopping to send advertisements.

NCC group [78] showed in 2022 that this is a real threat. They present a relay attack that enabled an attacker to access a Tesla Model 3 and Y.

In the remainder of this section, we describe how our BLE learning setup can be used to learn the behavioral model of the BLE devices implemented in a Tesla Model 3 and in its key fob.

Learning setup. To cope with the more difficult conditions under which the experiments have to be carried out, we decided to slightly adapt the learning setup: For learning the model of a BLE device built into a car, the biggest problem was time, since the car was only available for a few hours. To deal with this problem, we decided to switch the learning algorithm to *KV* [96]. Using *KV* has the advantage that the algorithm generates more intermediate hypotheses. Therefore, in case the connection is lost or any other environmental influences abort the current learning procedure it is easier to restart the learning procedure from the last learned hypothesis. We apply the *KV* implementation of AALpy version 1.3.0, which considers all improvements as discussed in Chapter 3.

We also used this setup for learning the Tesla Model 3/Y key fob. The reason, however, for using this adapted framework was different: One big problem in the communication with the BLE key fob was that the key fob responds very slowly to performed requests. Hence, we had to increase the parameters $n^{\text{rsp}_{\text{min}}}$ and $n^{\text{rsp}_{\text{max}}}$ to the values 60 and 80 respectively. At the same time, waiting too long may cause the current connection to be terminated. An additional problem was that the key fob must be moved to avoid entering a standby state in which the key fob stops sending advertisements. Moving the key fob manually during learning is tedious. Hence, we extended our learning setup by Lego® components that continuously move the key fob. Figure 4.9 shows the learning setup with Lego®.

By observing the BLE traffic around a Tesla Model 3, we can determine the hardware address of the BLE device since the Tesla Model 3 continuously sends advertisements. This can be simply

Table 4.6: Learning results for learning the BLE devices of the Tesla Model 3 and Tesla Model 3/Y key fob.

	Tesla Model 3	**Tesla Model 3 Key Fob**
# States	11	11
# Learning round	8	7
# Learning restarts	1	2
Total time (min)	86.94	-
# Output queries	323	307
# Conformance tests	155	87

done by using a mobile phone and a corresponding application such as the *nRF Connect for Mobile* application provided by Nordic Semiconductor ASA [21]. Note that this does not require access into the car itself. The same can be done with the Tesla Model 3/Y key fob, since it also sends BLE advertisements.

Learning results. Table 4.6 shows the results of learning the Tesla BLE devices. Both models have eleven states and are behavioral equivalent to each other. Figure 4.10 illustrates the simplified learned models. Learning the model of the Tesla Model 3 took approximately 86.94 minutes, where we had to restart the learning procedure once after the first iteration generated a model with 10 states. For learning the model of the key fob, we had to restart the learning procedure twice. We do not provide any time measurements since the restart was required due to an unreliable connection to the device.

To make the approach better comparable to the previously presented learning results of the other BLE devices, we also learned a model of the Tesla Model 3 using the previous learning setup based on L^* and a reduced input alphabet. The reduced input alphabet contains the following five inputs

$$I'^{\mathcal{A}}_C = \{\mathsf{scan\_req}, \mathsf{connection\_req}, \mathsf{feature\_rsp}, \mathsf{version\_req}, \mathsf{legacy\_pairing\_req}\}.$$

Using this reduced input alphabet, we managed to learn a model with only six states. Learning took 33.1 minutes and required a total of 237 output queries and 1462 input steps. We observed 282 connection errors and eight times non-deterministic behavior during learning. These results show that learning a model of BLE devices that are built into automotive components is indeed possible and feasible in a finite amount of time.

The Tesla BLE models show many similarities with the CC2640R2. For example, similar to the CC2640R2 the BLE device in the Tesla remembers previously performed version requests and answers them immediately after responding to the initial request of the peripheral device. However, the device in the Tesla initially sends a feature request, whereas the CC2640R2 sends a length request. Hence, executing the fingerprinting sequence that we presented in the previous section would produce a unique output sequence for the Tesla BLE devices. A closer look at the concrete transmitted packets of the Tesla Model 3 reveals that the advertising messages mimic an Apple iBeacon. However, by investigating further transmitted packets, we observe that the BLE device used in the Tesla Model 3 contains the official company identifier of Texas Instruments Inc., which aligns with the found similarities to the CC2640R2 that is also manufactured by Texas Instruments.

4.4 Conclusion

This chapter presented a case study on learning behavioral models of BLE devices via active automata learning. For this purpose, we introduced a learning framework that allows us to interact with a black-box BLE device by sending and receiving BLE packets. Our results show

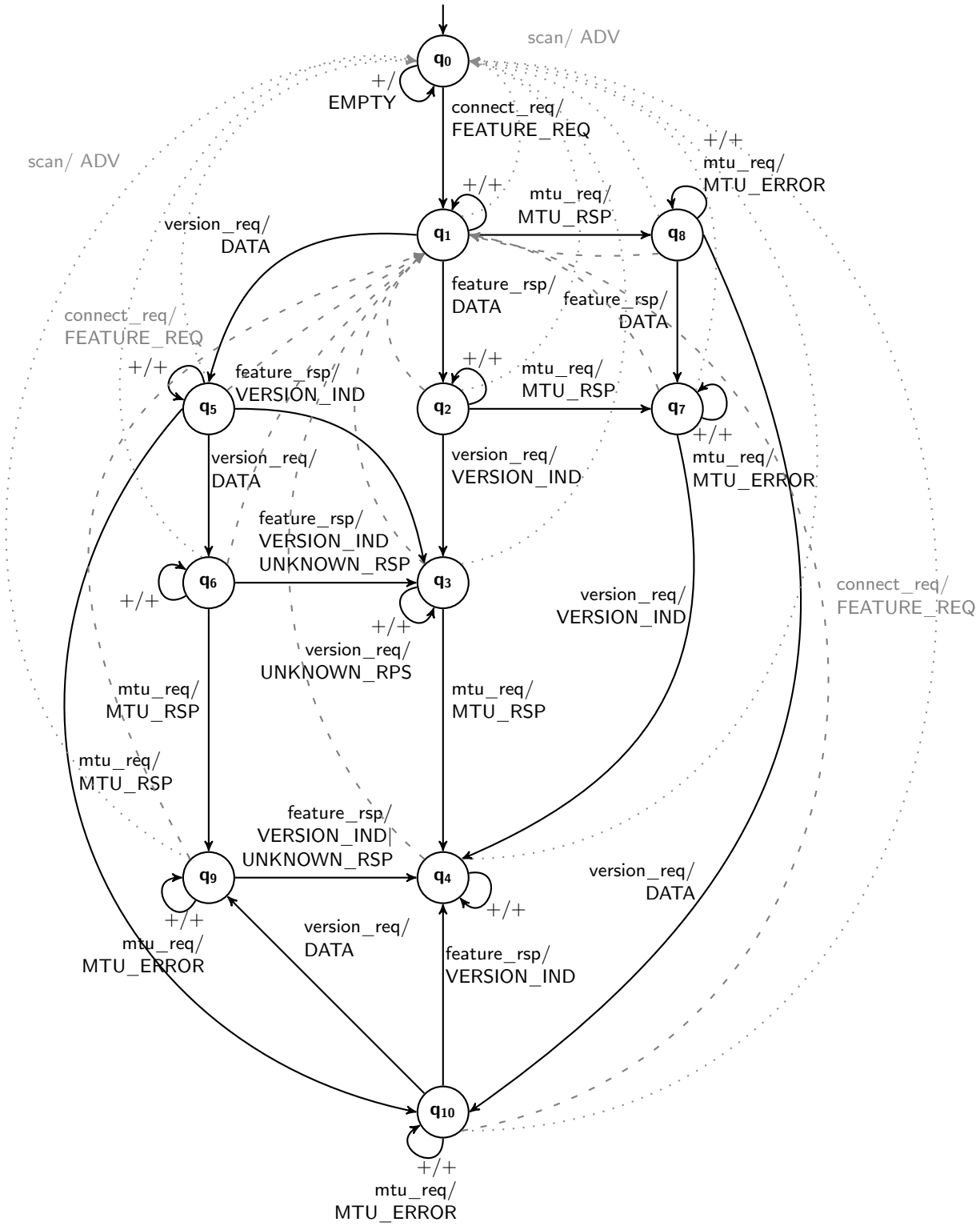


Figure 4.10: Learned model of the BLE device built into the Tesla Model 3. To provide a clear illustration, we grouped inputs and outputs by the '+' symbol. Moreover, we color resetting transitions to the state q_0 and q_1 with gray colors, where transitions are dotted and dashed respectively. Note that the model for the learned model of the key fob is equivalent except for the renaming of states.

that learning wireless protocol implementations on physical devices with our proposed learning setup is indeed possible. However, we faced several challenges in doing so. BLE packets might get lost or arrive delayed. To overcome these problems, we introduced countermeasures to ensure reliable but still efficient learning. We showed that the learned models reveal violations of the BLE specification. Furthermore, the models show behavioral differences between all investigated devices. We showed that the learned models can be used to generate input sequences that enable the fingerprinting of BLE devices. For real-world systems, the possibility to create a fingerprint can be critical in the case that there exist known security vulnerabilities for certain devices. We also presented a case study on learning models of the BLE devices built into a Tesla Model 3 and its key fob. In doing so, we experienced several challenges but managed to learn behavioral models.

(RQ 1.1) Does active automata learning perform well for learning communication protocol implementations on physical devices?

In actively learning different BLE stack implementations on physical devices, we faced several challenges: First, active learning required an interface that enabled active communication with the SUL. Furthermore, efficient mechanisms were required to reset the SUL. Another challenge in learning the behavior of physical devices, especially when learning wireless communication protocol implementations, was that messages could be lost or arrive delayed. Thus, error-recovery mechanisms were needed to deal with possible non-deterministic observations. Our results showed that creating such a learning interface was indeed possible: We learned behavioral models of all eight investigated BLE devices. However, some devices could only be learned with a reduced input alphabet. To enable learning in a limited amount of time, we learned the BLE connection and pairing procedure separately, which keeps the input alphabet small.

(RQ 1.2) Is automata learning useful to learn security-critical behavior?

We learned the BLE pairing procedure of the BLE devices. The pairing procedure defines a security-critical key exchange procedure. The exchanged key is used to establish encrypted communication. The learned models showed behavioral differences, especially when receiving unexpected messages. Some devices terminated the ongoing pairing procedure and reset to a previous state, while other devices ignore unexpected messages. We also observed that a device stopped responding to requests when the key exchange procedure was unexpectedly terminated. Active automata thus provided useful insights into differences between implementations and tested unexpected input sequences that uncovered reliability issues.

(RQ 3.3) Can automata learning be used to fingerprint black-box devices?

We learned seven different models for eight different devices for the BLE connection procedure. Only the two learned models for the BLE devices used in the Tesla components were equal to each other. The discovered behavioral differences allowed us to fingerprint the individual devices. Furthermore, we showed how the data structure that are generated during learning can be used to generate fingerprinting sequences. The fingerprinting sequence was an input sequence, where the execution of this input sequence led to different output sequences on each of these devices.

Chapter 5

Learning of IPsec-IKEv1 VPN Servers

Declaration of Resources

This chapter is based on the paper *"Mining Digital Twins of a VPN Server"* [150] presented at *FMDT 2023* and on the Master's Thesis of Benjamin Wunderling *"Model Learning and Fuzzing of the IPsec-IKEv1 VPN Protocol"* [196]. The Master's Thesis of Benjamin Wunderling was co-supervised by the author of this thesis. For more details on the content presented in this chapter, we refer to the underlying Master's Thesis of Benjamin Wunderling.

5.1 Introduction

A Virtual Private Network (VPN) is a method that enables secure communication over an insecure channel. In addition, VPN can be used for remote access to internal network resources, e.g., for the remote control of the heating system in a home automation system. Therefore, the security of the used VPN is also critical for the security of an IoT network. To ensure security, it is essential to verify that the VPN implementation used conforms to the standard and does not introduce security issues. The challenge is that many VPN implementations are closed-source, which requires black-box testing techniques.

Learning-based testing of communication protocols has numerous success stories as shown by examples from the literature [50, 61, 62, 63, 168, 170] and by the case study on BLE presented in the previous chapter. Automata learning has been applied to different VPN protocol implementations. For example, Daniel et al. [47] learning-based tested OpenVPN implementations, and Guo et al. [80] learned and model checked IPsec-IKEv2 protocol implementations.

In the following, we complete the VPN learning case studies by learning IPsec-IKEv1 protocol implementations. Albeit IKEv1 is the predecessor of IKEv2, IKEv1 is still supported and used as it is shown by real-world setups [70]. Since IKEv2 was introduced as the successor of IKEv1 due to IKEv1's complicated setup, we stress the necessity of automatic verification techniques for IKEv1-based systems. This helps to test setups when only IKEv1 is supported.

In addition, the IKEv1 case study is used to evaluate the required amount of interaction with the SUL to actively learn a behavioral model. In practice, the goal is to learn a behavioral model with a minimum number of interactions. For our evaluation, we compare two active learning algorithms in the case study on IKEv1: L^* and KV. The algorithms are compared based on their runtime, and the number of queries and inputs that need to be executed on the SUL to learn a conforming behavioral model.

First, this chapter introduces the setup for learning IKEv1 in Section 5.2. Section 5.3 then

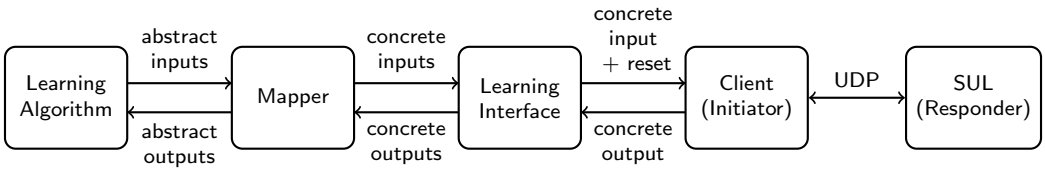


Figure 5.1: The active automata learning setup for learning a model of an IPsec-IKEv1 implementation.

presents the results of learning two IKEv1 server implementations followed by a discussion of one bug that was found during learning.

5.2 Learning Setup

The learning setup for learning behavioral models of IKEv1 protocol implementations consists of five components. Figure 5.1 shows the five components that are similar to those used for learning BLE devices, as described in Chapter 4. In contrast to the BLE learning setup, the components for learning VPN are arranged slightly differently. Thus, the mapper and the learning interface are interchanged. Again, the learning interface was used to handle irregularities in the learning procedure. The components were swapped since unexpected behavior was easier to detect and to handle with concrete packets than with abstracted inputs.

Learning algorithm. For learning IKEv1, we evaluated two different learning algorithms: the L^* algorithm [17] and the KV algorithm [96]. Both algorithms use all the improvements described in Chapter 3. The exchange of the learning algorithm does not require any other changes in the learning setup than the change of the algorithm in the learning algorithm component. This shows the flexibility of the modular learning setup. Both learning algorithms implement the equivalence oracle with a conformance-testing technique that provides state coverage in combination with randomly selected inputs.

The learning algorithm learns a behavioral model using an abstracted input alphabet since learning with all possible IKEv1 packet instantiations would not be feasible. The abstracted input alphabet used for learning is set to $I^{\mathcal{A}} = \{\mathsf{main\_sa}, \mathsf{main\_key\_ex}, \mathsf{main\_authenticate}, \mathsf{quick\_sa}, \mathsf{quick\_ack}\}$. The learning algorithm then provides abstract output queries to the mapper and receives the corresponding abstract query outputs from the mapper.

Mapper. The mapper component is used as described in Section 2.2.2. Thus, it translates abstract inputs into concrete inputs and received concrete outputs into abstract outputs. Similar to the BLE learning setup, we use the Python library Scapy [158] to parse and generate IKEv1 packets. Considering the messages shown in Figure 2.9, the following mapping from abstract inputs to concrete inputs applies:

- main_sa → ISAKMP SA $\{proposals\}$,
- main_key_ex → KEY EX $\{pkey_i, nonce_i\}$,
- main_authenticate → AUTH $\{hash_i\}$,
- quick_sa → IPSEC SA $\{proposals\}$, and
- quick_ack → ACK,

where the fields in *italics* are concretized by the mapper.

The output abstraction follows the standard packet identifiers used in Scapy. However, Scapy v.2.4.5 could not parse all received ISAKMP packets. Therefore, a manual extension of Scapy was needed to parse and generate all required ISAKMP packets. If no response is returned from the SUL, the mapper returns the output NONE, and all received error messages are mapped to the abstract output ERROR_NOTIFICATION.

Since the purpose of the IKEv1 protocol is to establish an encrypted communication channel, the mapper must store all values required to successfully perform a Diffie-Hellman key exchange [52] and an encrypted communication. The mapper also stores a Boolean flag indicating whether the current communication is encrypted, and shares this information with the learning interface to decrypt and encrypt messages.

Learning interface. The learning interface identifies and handles non-deterministic behavior. Initial learning experiments have shown that models cannot be reliably learned due to non-deterministic errors. Moreover, repetitions of the learning procedure yield different models. To overcome these issues, several countermeasures were taken. First, we simply increase the waiting time to receive responses. Already increasing the waiting time from one second to two seconds was sufficient to reduce the number of different models. However, we still occasionally learned different models, even though the variation of different models was smaller. A manual analysis of the learned models revealed that the learning of different models was caused by retransmitted messages.

To learn the same model in each repetition of the experiment, we adapted our learning interface to handle these retransmitted messages. In general, our goal is to learn models with each input defined for each state. To this end, the learning algorithm queries each input for each state, albeit this input might be unexpected at this stage of the communication protocol. In the examined IKEv1 implementations, these unexpected inputs trigger the retransmission of messages. We conjecture that the SUL assumes that messages have been lost since the initiator is sending inputs from an earlier stage of the communication protocol. Therefore, the SUL reacts by retransmitting previous messages. This is an acceptable behavior according to RFC 2048 specification [33]. The problem in learning was that these retransmitted messages occurred randomly and did not follow a deterministic pattern.

Albeit this type of non-deterministic behavior could be interesting for fingerprinting different IKEv1 implementations, we desire to have deterministic behavior for testing purposes. To avoid non-deterministic behavior, our learning setup provides an option to filter out these resent messages. When this option is enabled, the learning interface removes received retransmitted messages and forwards only the remaining received messages to the mapper.

Detecting whether a message has been retransmitted is simple since each message has a unique numeric identifier. The learning interface can determine whether the received message is retransmitted by checking if this message identifier has already been received. For this purpose, the learning interface stores all identifiers from all messages received so far. By filtering messages in this way, we can learn the same model for almost all learning repetitions. There was only one exception where we could still observe non-deterministic behavior. In this case, the non-deterministic behavior occurred due to a bug in a Python library used for Diffie-Hellman key exchange. In Section 5.3.5, we will discuss the revealed bug in more detail.

The learning interface is also responsible for resetting the SUL before performing a new output query. We assume that the connection can be reset during the main mode via an ISAKMP error notification message. After entering the quick mode, the ISAKMP delete message can be used to reset the SUL to its initial state. Later in the case study on learning concrete IKEv1 implementations, we see that this reset method does not work for all examined implementations. It seems that one of the IKEv1 servers ignores all received error notification messages. Since the communication could not be reset by sending packets, it was necessary to reset the VPN server via SSH commands. In practice, this could hamper the learning setup for black-box components

as the privileges to execute commands via SSH may not be present on the SUL host. However, for a local or custom setup, we assume that these access rights are present. Therefore, with this reset method, the black-box assumption remains as we do not need access to the source code.

Client (initiator). The client is used to communicate with the SUL, where the client is the initiator in the IKEv1 protocol. For this purpose, the client implements a network socket that sends and receives packets via UDP. We assign the socket a specific IP address and port over which it sends and receives messages. The client uses a preset identifier, the initiator cookie, to avoid any unexpected behavior due to randomized identifiers. The client then sends byte streams via the socket and the received packets are then forwarded to the learning interface, which post-processes them.

SUL (responder). The SUL represents the responder in the IKEv1 protocol. Therefore, we learn a model that formalizes the behavior of the responder, which corresponds to the role of a VPN server. The server is configured to run at a certain IP address and to expect incoming requests. Furthermore, the setup disables unique identifiers for established connections. Otherwise, consecutive ISAKMP SA messages would always trigger a new instance of a connection. In addition, the configuration considers the usage of pre-shared keys (PSKs) for authentication, 256-bit AES-CBC for encryption, SHA-1 for hashing, and 2048-bit new Modular Exponential (MODP) Diffie-Hellman group for key exchange.

Note that the learning algorithm, mapper, learning interface, and client run on one virtual machine (VM), while the SUL runs on a second VM. The VMs are configured to have isolated communication, i.e., they can communicate with each other but no other external communication is possible.

5.3 Evaluation

5.3.1 Case Study Subjects

For the case study on learning behavioral models of IKEv1 implementations, we considered two different implementations: strongSwan [166] and libreswan [107]. The strongSwan implementation is a fork of the FreeS/WAN implementation, whereas libreswan is a fork of Openswan, which also originates from FreeS/WAN. Hence, FreeS/WAN, which is no longer maintained, is a common ancestor of both strongSwan and libreswan. Both implementations are available for Linux, Android, FreeBSD, and OS X based operating systems. For this case study, we evaluated strongSwan U5.9.5/K5.15.0-25-generic and libreswan 3.32/5.15.0-41-generic. The detailed configuration of the servers is given in the Master's thesis of Wunderling [196].

5.3.2 Environmental Setup

For our learning setup, we require two virtual machines that can communicate with each other. For this purpose, we used two VMs with Ubuntu 22.04 LTS running in VirtualBox 6.1. Both VMs run on the same host computer: a Dell XPS 15 with an Intel® Core™ i7-1195G7 CPU running at 2.9 GHz and 32 GB of memory using Ubuntu 20.04.6 LTS. For reliable communication without any external confounding factors, we only allowed communication between the two VMs in an internal network, where each VM has a fixed IP address. To make the communication fast, we used the para-virtualized network adapter. For learning a model of libreswan, it was also necessary to enable an SSH connection between the VMs.

For learning, we used the learning library AALpy version 1.2.9, which implements the learning algorithms L^* and KV. As improvements for learning, we enabled the Rivest and Schapire

counterexample processing and caching. As an equivalence oracle, we used the StatePrefixEqOracle which generated input sequences that access every state in the provided hypothesis model, and then append a random sequence of inputs. The number of input sequences per state was set to ten, and also the number of appended random inputs is set to ten.

5.3.3 Learning Results for strongSwan

In this section, we present the learning results for learning the strongSwan implementation. For this purpose, we will first discuss the results of learning without filtering retransmitted messages as discussed in Section 5.2. Then, we will show the model learned with retransmitted messages filtered out. The evaluation on strongSwan is concluded by comparing the L^* and KV algorithms in terms of runtime and required interaction with the SUL.

Learned models with retransmissions. Repeating the learning experiments with L^* shows differences between the final learned models of strongSwan. We consider two automata to be equivalent if they are isomorphic to each other. In approximately 80% of the learning experiments performed, the algorithm generates one of two distinct models. We refer to these two models as *retransmission model I* and *retransmission model II*. The remaining 20% of the learned models represent other automata that are not isomorphic to either *retransmission model I* or *retransmission model II*.

Figure 5.2 illustrates *retransmission model I*, where red transitions indicate that a retransmitted message has been received, and orange edges indicate the probable cause for receiving retransmitted messages. The model shows that retransmitted messages are only observable after entering the quick mode in the IKEv1 protocol. The retransmitted messages seem to be triggered by initiating the quick mode with quick_sa, followed by an input that is usually required only in the main mode, e.g., main_authenticate, main_sa, and main_key_ex. We see cyclic behavior where retransmission can always be repeated either by performing further main-mode inputs or by performing quick_sa.

Figure 5.3 shows the second frequently generated model by applying the learning algorithm. Compared to the model shown in Figure 5.2, a slightly different retransmission behavior can be observed, where messages from the main mode do not trigger further retransmissions after they have been observed twice.

In general, retransmissions are allowed by the IKEv1 specification [33]. However, since they occur in a non-deterministic manner, the learned model becomes impractical for conformance testing purposes. The behavior of retransmitted messages can be very specific to each implementation. Therefore, this behavior might reveal which IKEv1 implementation is being used. In this case, we should learn such models for later fingerprinting scenarios, as we discussed in Chapter 4 for BLE devices.

Learned models without retransmissions. Figure 5.4 depicts the learned model when filtering out retransmitted messages as explained in Section 5.2. We refer to this model as the *base model*. In this setup, each repetition of a learning experiment resulted in a model that is isomorphic to the base model. In all three models, we see a clear separation between the main mode and the quick mode. While the main mode is completed by sending a main_authenticate message in state q_2 and the quick mode is started by sending a quick_sa message in state q_3. Figure 5.4 presents a much clearer description of the quick mode in contrast to the figures 5.2 and 5.3.

Learning metrics. Table 5.1 shows the results for learning the models with KV and L^*. As expected, KV requires more equivalence queries than L^*. Note that equivalence queries are implemented via conformance testing. The results show that the number of states learned

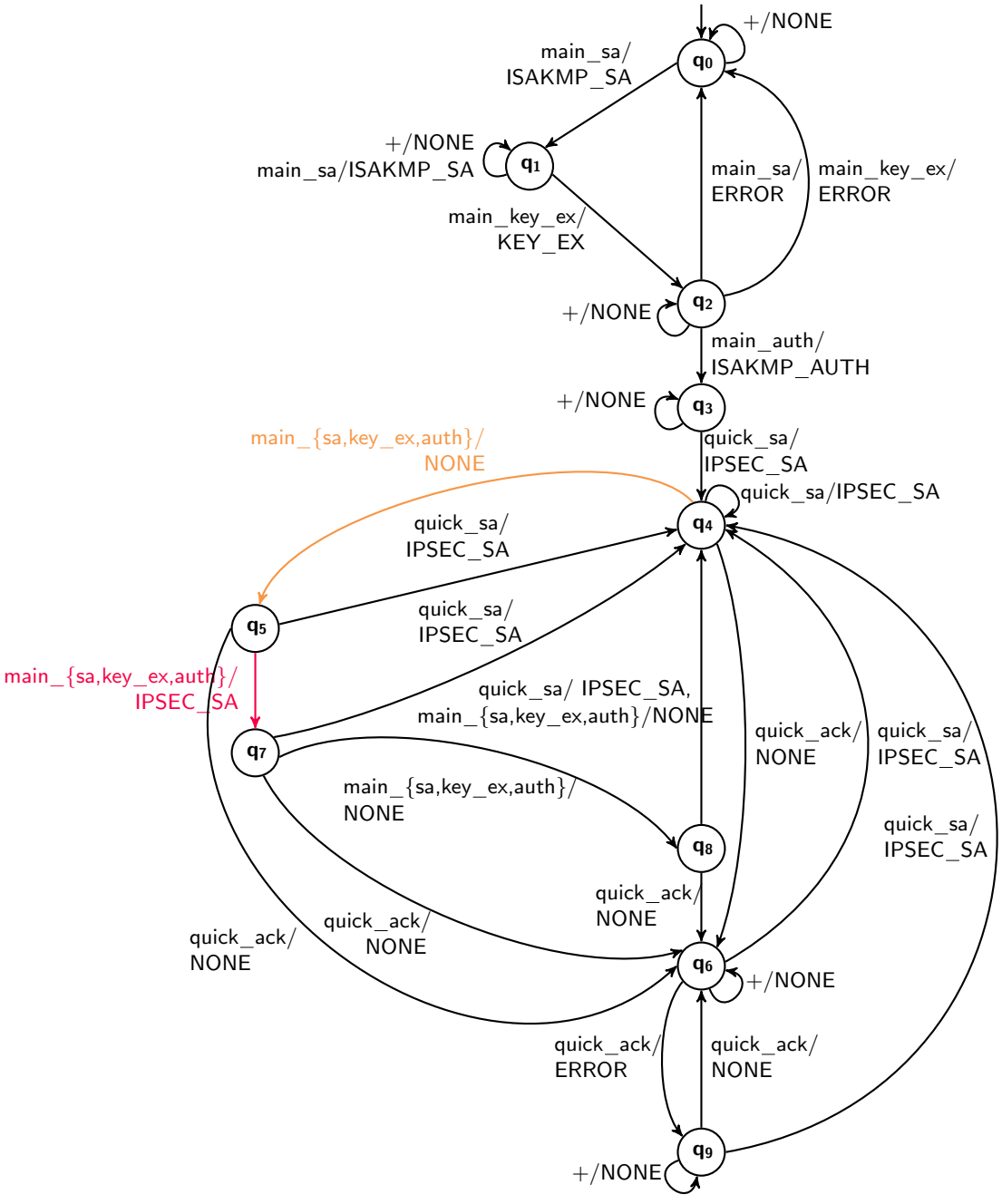


Figure 5.2: First commonly learned model including retransmitted messages (*retransmission model I*) of the strongSwan implementation. For simplification, some labels on the transitions of the learned model are abbreviated with a '+' symbol.

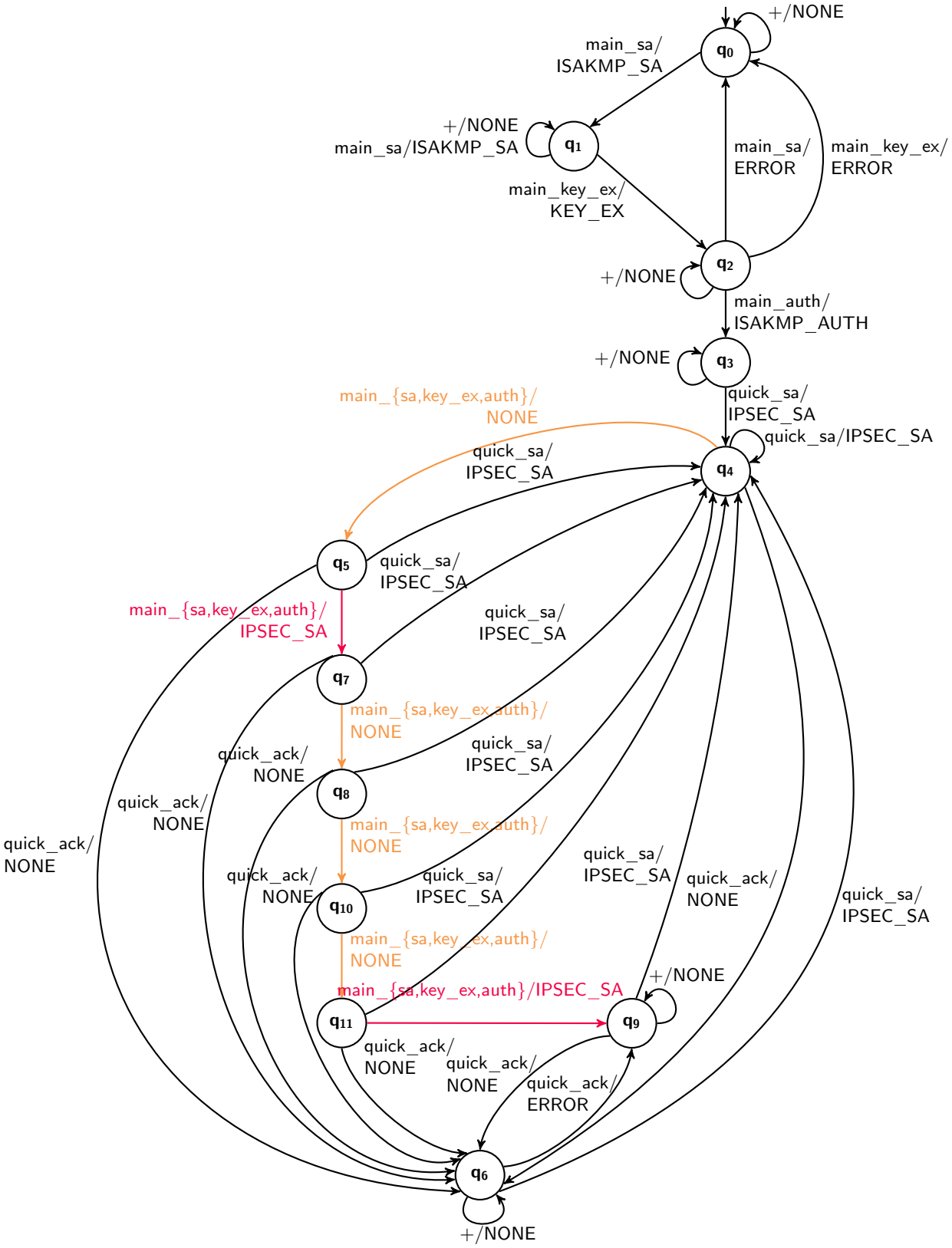


Figure 5.3: Second commonly learned model including retransmitted messages (*retransmission model II*) of the strongSwan implementation. For simplification, some labels on the transitions of the learned model are abbreviated with a '+' symbol.

Table 5.1: Learning results for the performed case study on learning IKEv1 protocol implementations of strongSwan.

	Retransmission model I		**Retransmission model II**		**Base model**	
	L^*	KV	L^*	KV	L^*	KV
# Learning rounds	5	7	5	9	2	4
# States	10	10	12	12	6	6
Total time in minutes (min)	84.9	51.5	125.3	75.1	27.5	20.2
Output queries (min)	58.2	41.6	89.9	64.8	15.0	11.5
Conformance checking (min)	26.8	25.9	35.4	35.4	12.6	12.4
# Output queries	462	171	522	215	177	78
# Output query steps	2922	2047	4225	2219	856	676
# Conformance tests	100	100	120	120	60	60
# Conformance testing steps	1379	1826	1745	2219	747	934

directly influences the runtime and the number of output queries performed. Since we use a state-coverage-based conformance testing technique, the number of performed conformance tests is identical for both learning algorithms. Comparing the overall runtime, KV always performs better than L^*. Learning the largest model in terms of the number of states, *retransmission model II*, took 75.1 minutes with KV, while learning the *base model* took only 20.2 minutes. These results show that learning an IKEv1 implementation is feasible in a reasonable time budget.

Comparison of learning algorithms. In active automata learning, it is usually preferable to reduce the number of required interactions with the SUL as much as possible, especially when we are learning real-world applications. Therefore, we want a learning algorithm that requires a small number of queries to learn correctly. As shown in the Bachelor's thesis by Rindler [155], the improved KV implementation in AALpy requires fewer queries than L^* for learning DFAs.

This case study compared the learning algorithms, L^* and KV, for learning Mealy machines of real-world applications. Table 5.1 provides the results of the experiments for L^* and KV. Due to the selected conformance testing technique, the experiments are subject to randomness. To allow a fair comparison, we repeated each learning experiment 20 times and report the average. We can see that KV requires twice as many learning rounds, but overall KV is 1.36 faster than L^*. We can also see that KV can aid in reducing the number of resets. Assuming that the system is reset before executing each query, L^* requires 237 resets, which is the sum of output queries and conformance tests. KV, on the other hand, requires only 138 resets. There is no difference in the number of executed inputs, 1 603 for L^* vs 1 610 for KV. Most of the inputs for KV are performed during the equivalence check. Choosing a different equivalence oracle could further reduce this number.

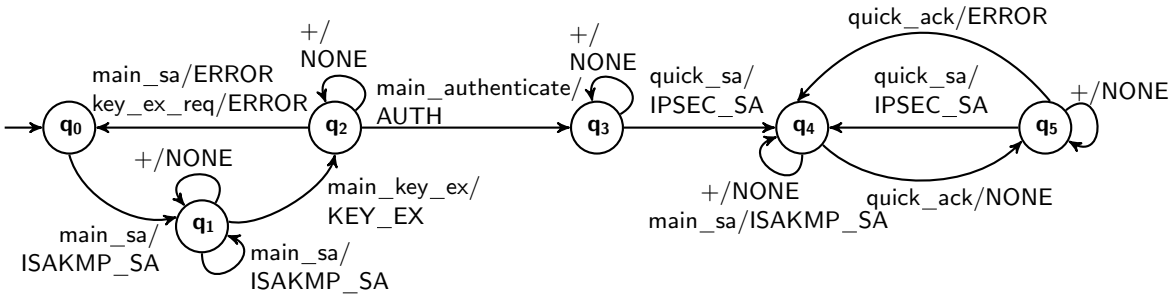


Figure 5.4: Learned base model of the strongSwan implementation. For simplification, some labels on the transitions of the learned model are abbreviated with a '+' symbol.

Table 5.2: Learning results for the performed case study on learning IKEv1 protocol implementations of LIBRESWAN. For learning LIBRESWAN, the retransmission filtering was enabled. No runtime measurements are provided due to the more complex reset.

	Base model	
	L^*	KV
# Learning rounds	1	2
# States	4	4
# Output queries	100	36
# Output query steps	350	321
# Conformance tests	40	40
# Conformance testing steps	460	660

5.3.4 Learning Results for libreswan

As a second case study, we considered the IKEv1 implementation of LIBRESWAN [107]. We again started by learning the model without filtering retransmitted messages. Unlike STRONGSWAN, LIBRESWAN already retransmits messages in the main mode. Therefore, the described behavior of the resulting models varies much more. Due to the unreliability of learning with retransmitted messages, we only provide the base model where the learning framework filters out retransmitted messages. To filter out the retransmitted messages in the main mode, additional checking of hashes and nonces was required, since the message identifiers in this mode are not sufficient.

The learned model of the LIBRESWAN implementation is shown in Figure 5.5. The shown model of LIBRESWAN is learned considering the same setup as for learning the base model of STRONGSWAN. However, we see that the LIBRESWAN model has only four states. The behavior is rather linear, following the message sequence diagram of IKEv1 presented in Figure 2.9. The model also shows that unexpected messages are ignored by the implementation, while in STRONGSWAN some trigger a reset to the initial state of the current mode.

The laissez-faire behavior is also shown by the aspect that initiating the quick mode does not change the behavior even when repetitively performing a quick_sa or ack_quick message.

Table 5.2 shows the learning results for learning the base model of LIBRESWAN with KV and with L^*. Note that no runtime data is provided due to the more complex reset method. Similar to the comparison of the learning algorithms for STRONGSWAN, we see that KV requires 1.84 times fewer queries than L^*, but overall KV executed more queries. Again, the results show that most of the inputs were executed during conformance testing, which could be reduced by using a different conformance testing technique.

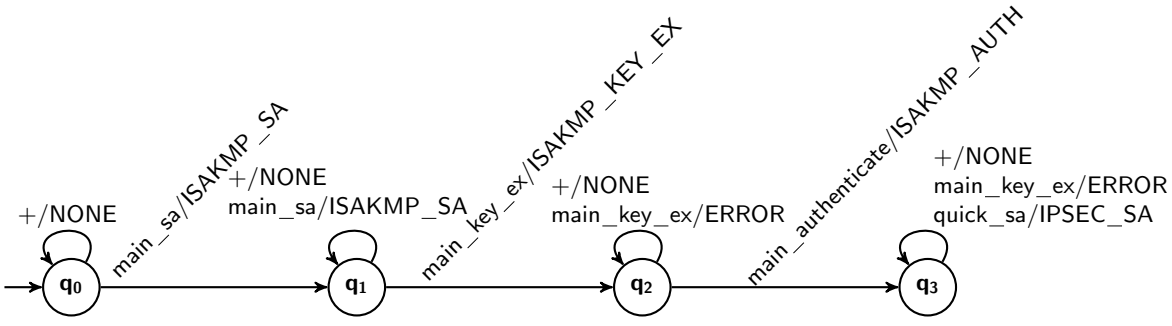


Figure 5.5: Learned base model of the LIBRESWAN implementation. For simplification, some labels on the transitions of the learned model are abbreviated with a '+' symbol.

5.3.5 Diffie-Hellman Library Bug

The case study conducted on the VPN protocol showed again that active automata learning is a successful tool for detecting bugs in an implementation. While learning behavioral models of IKEv1 implementation, we found a bug in a Python library called `py-diffie-hellman`[1], which we used to generate the values for the Diffie-Hellman key exchange.

```
1       def generate_private_key(self, key_bits: int = 540) -> bytes:
2           private_key = os.urandom(key_bits // 8 + 8)
3           self.set_private_key(private_key)
4           return self.get_private_key()
5
6       def set_private_key(self, key: bytes) -> None:
7           self._private_key = int.from_bytes(key, byteorder="big")
8           self._public_key = pow(2, self._private_key, self._prime)
```

Listing 5.1: Original code copied from the `py-diffie-hellman`

The indicator for the presence of a problem was that we occasionally failed to learn a behavioral model due to non-deterministic behavior. After debugging the inputs that caused non-deterministic behavior, we received a different response when our generated private key started with a zero byte. Further investigation revealed that the Python library we used incorrectly converts a byte stream into an integer. Listing 5.1 shows an excerpt of the original library's code. First, a random byte stream is generated, which is then set as the private key in Line 2. In Line 7, the byte stream is converted to an integer used to generate the public key in the next line.

The problem is that the type conversion in Line 7 ignores null bytes at the beginning of the byte stream. This not only leads to inconsistent key generation but also to security issues as the private key becomes shorter, with shorter keys being easier to bruteforce. Learning was able to expose this bug, as a large number of different inputs were generated during the learning procedure. The amount of concretized inputs was sufficient to occasionally generate byte streams with initial zeros.

5.4 Conclusion

We presented a learning setup for learning behavioral models of IKEv1 implementations. We show that learning is feasible, but setting up the learning framework is not straightforward. Several challenges were encountered in setting up a reliable learning framework. First, the existing packet manipulation libraries had to be extended since not all packet types were supported. Second, we experienced non-deterministic behavior for different reasons. Some of this non-deterministic behavior could be reduced by increasing the message waiting time. Another measure was to manipulate the received output from the SUL. The main problem was that unexpected input triggered the non-deterministic retransmission of messages. By filtering out these retransmitted messages, we were able to reliably learn models of two IKEv1 implementations. Using the learning framework developed, we also revealed a security issue in a Python library used to generate the values for the Diffie-Hellman key exchange.

By developing this learning framework for the IKEv1 protocol, we extend the series of learning frameworks [47, 80] for VPN protocols. In addition, we evaluated the impact of the learning algorithm regarding the required interactions with the SUL. For this purpose, we compared the two improved versions of L^* and KV. The results of the comparison show that KV requires fewer queries, which also reduces the number of resets required. Since resets can be tedious when learning real-world applications, learning settings can benefit from using KV. However, due to the larger number of required equivalence queries, KV may execute longer input sequences on

[1] https://github.com/TOPDapp/py-diffie-hellman

the SUL. However, Aichernig et al. [15] have shown that the total number of inputs performed by learning algorithms also depends on the conformance testing technique used. Therefore, it might be useful to also consider other conformance testing techniques in future work.

(RQ 1) What are the challenges of learning behavioral models in networked systems?

When learning behavioral models of IPsec-IKEv1 implementations, we again observed non-deterministic behavior due to delayed messages. This problem could be easily fixed by increasing the timeout for messages. Another challenge we observed when learning IPsec-IKEv1 implementations was that the protocol behaved non-deterministically when previous messages were resent. In active automata learning, we typically test every input at each state. Thus, during the learning process, we might repeat inputs that resulted in messages being non-deterministically retransmitted by the SUL. We overcame this problem by filtering out these retransmitted messages.

(RQ 1.2) Is automata learning useful to learn security-critical behavior?

IPsec-IKEv1 is a security-critical protocol since it defines the key exchange procedure for encrypting communications via a VPN. We succeeded in learning the behavioral models of two IPsec-IKEv1 implementations. Our results were to some extent similar to the case study on learning behavioral models of BLE devices that we presented in Chapter 4. Again, we saw differences in the implementations where the responses to unexpected messages were different. In addition, the extensive testing performed during active automata learning revealed a security issue in the Python library used to implement the Diffie-Hellman key exchange.

(RQ 3.3) Can automata learning be used to fingerprint black-box devices?

We again found that the learned models of the IPsec-IKEv1 implementations were different, even though both investigated implementations originated from the same implementation. Hence, the learned models could again be used to fingerprint the implementations.

Chapter 6

Active vs. Passive Automata Learning

Declaration of Resources

This chapter is based on the paper *"Active vs. Passive: A Comparison of Automata Learning Paradigms for Network Protocols"* [13] presented at *FMAS 2022*.

6.1 Introduction

This chapter compares the two paradigms of automata learning: active and passive. Chapter 4 and Chapter 5 show that active learning for real systems requires an elaborate setup in order to enable a reliable and fault-tolerant active interaction. For example, queries have to be repeated or received messages must be post-processed. The implementation of these countermeasures often requires domain knowledge. The disadvantage of active learning is that all these countermeasures must be performed during learning. During learning, it might be unclear whether we observed new behavior or whether environmental conditions led to incorrect observations.

In passive learning, we learn behavioral models from a given sample. Even though we experience the same challenges in the collection of this sample, they do not require an immediate reaction at runtime. For passive learning, we can pre-process the sample in order to remove traces or investigate suspicious behavior. The advantage is that this pre-processing can be done offline, without access to the SUL.

In the following, we assess active and passive learning techniques considering different aspects. We will base our evaluation on the following research challenges (RCs):

- **(RC 1)** Can passive learning based on a random sample compete with active learning?
- **(RC 2)** Does the considered active automata learning algorithm generate an optimal sample?
- **(RC 3)** Can random sampling support active automata learning?

(RC 1). As a first aspect, we assess if random samples of a specific size are sufficient to cover behavioral aspects of the SUL, where the specific size is set in relation to the number of queries active learning requires. When learning real systems, it might be easier to first generate a random sample and then use this sample to learn a behavioral model. Hence, we investigate whether a random sample similar in size to the number of queries required by active learning is sufficient, and if not, how large such a sample must be in order to learn the same model as in active learning.

(RC 2). As a second aspect, we assess the efficiency of active automata learning. This research question is motivated by the challenges experienced in the presented case studies using active learning. To create a fault-tolerant active learning setup, we needed to implement countermeasures that often required the repetition of queries. Thus, we aim to reduce the number of interactions to a minimum. For this purpose, we evaluate the L^* algorithm [17] with improvements of Rivest and Schapire [156]. In most of the case studies throughout this thesis, we used this learning algorithm. In this chapter, we assess the potential improvements in terms of performed queries.

(RC 3). As a last aspect, we asses if random samples could reduce the number of interactions in active learning. In Chapter 3, we discussed that active learning algorithms can be extended by caching data structures that aim to reduce the number of queries performed on the SUL. In RC 3, we evaluate if we can reduce the number of interactions with the SUL when we initialize the cache with a random sample.

Our previous chapters show that automata learning can successfully be applied to learn communication protocol implementations. Our following evaluation focuses specifically on assessing passive and active learning algorithms in terms of their suitability for learning communication protocols. Thus, our provided answers to our presented research questions only consider this context.

6.2 Methodology

The following section introduces the evaluated learning algorithms and their setup. Since we investigate reactive systems for this evaluation, we learn Mealy machines in all scenarios. In addition, we explain our applied method to compare the learned Mealy machines.

6.2.1 Learning Setup

Our comparison considers active and passive learning algorithms. We base our evaluation on the algorithm implementations provided by the learning library AALpy [129] written in Python. The learning library implements state-of-the-art learning algorithms and conformance testing techniques. Initially, AALpy mainly provided active learning algorithms. For this comparison, AALpy was extended by an implementation of a passive learning algorithm for deterministic systems. Passive learning algorithms for deterministic systems are available as of AALpy version 1.2.8. Next, we present the used setup of our evaluated learning algorithms.

Active learning. For active learning, we evaluated the L^* algorithm variant for Mealy machines as presented by Shahbaz and Groz [162]. The algorithm includes the improved counterexamples processing presented by Rivest and Schapire [156] as described in Section 3.1. Additionally, we consider the caching procedure introduced in Section 3.2 to be enabled.

The equivalence oracle is approximated using conformance testing techniques. For conformance testing, we apply a model-based testing technique that provides state coverage in combination with random exploration. The test suite for conformance testing comprises sequences that visit every state n_{walk} times. Each access sequence to a state is then extended by n_{step} random inputs. In all our experiments, we set $n_{\text{walk}} = 25$ and $n_{\text{step}} = 30$.

We selected this setup for the comparison since it was applied in most of the presented case studies in this thesis based on the recommendation of Tappler et al. [15] and the available learning algorithms in AALpy.

Passive learning. We base our passive evaluation on the learning algorithm RPNI which we introduced in Section 2.2.1. Passive learning algorithms create a behavioral model from a given sample. This provided sample might be incomplete in the sense that it misses behavioral aspects of the SUL. To deal with this problem, the RPNI implementation in AALpy provides two strategies. We assume that our learned Mealy machines are input complete. In case the sample misses an input for a specific state, AALpy either uses self-looping transitions or transitions to a sink state to model the missing behavior. Self-loop transitions are transitions, where the source and the target state are equal. For modeling transitions to a sink state, we add an additional artificial state, which then represents the target state for transitions of undefined inputs in a specific state. The output for these transitions is a default output, indicating that this is an artificial transition added to make the Mealy machine input complete. For our setup, we use the sink state option, since the additional sink state immediately indicates that the data was incomplete.

6.2.2 Sample Generation

Active learning sample. The sample from active learning serves as a basis for comparison. Let S_{L^*} be the set of traces that includes all the output queries and their corresponding query outputs that are performed during active learning with the previously described L^* setup for learning Mealy machines. The set also includes all the queries that are performed during the equivalence check. We assume that S_{L^*} is sufficient for L^* to learn the minimal ground truth Mealy machine $\mathcal{M}$ of the SUL.

Optimized sample. We also want to evaluate the size of the optimal data set as it would be sufficient for active learning algorithms such as L^* to learn the ground truth automaton. Due to the incremental nature of L^*, the algorithm queries redundant information. Hence, the set of performed queries S_{L^*} commonly includes several queries that are prefixes of other queries. Considering Definition 2, which defines the observation table as it is used in the L^* algorithm, we already see that the prefix-closed definition of the set Γ introduces redundancy. Furthermore, the improved version of L^* also initializes the E set with the whole input alphabet. However, not all inputs are required for all systems to distinguish states.

Example 10 (Redundant Information in E) *Table 2.1 shows the observation table that L^* generates when learning the Mealy machine presented in Figure 2.1. The E set contains the whole input alphabet, but to distinguish the three states the subset* $\{\mathsf{connect}, \mathsf{publish}\}$ *would be sufficient. By minimizing the E set to this set, many queries that are necessary to fill the other two columns could be avoided.*

For our case study, we compare the size of S_{L^*} with an optimal set $S_{\mathcal{M}}$. To generate $S_{\mathcal{M}}$, we use the ground truth Mealy machine $\mathcal{M}$ that represents the SUL with a minimal number of states. Using the W-Method [39, 186], we calculate the characterization set. We then manually initialize the E set with the characterization set and execute the L^* algorithm as described. We then post-process the sample generated by L^*, where we remove all queries that are prefixes of other queries. We denote this post-processed optimal sample as $S_{\mathcal{M}}$.

Random sample. We evaluate passive learning based on randomly generated samples. Algorithm 3 describes the generation of a random sample S_R. The random sample consists of a list of input/output traces. The algorithm takes as input the size of the sample n_{data}, the minimum length n_{min} of a trace, and the maximum length of a trace n_{max}. Additionally, we require black-box access to the SUL such that inputs can be executed and outputs observed. We represent the SUL by a Mealy machine $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$. The algorithm returns a list of random traces S_R.

Algorithm 3 Generation of a set of random input/output traces.

Input: data set size n_{data}, minimum length $n_{\min}$, maximum length $n_{\max}$, SUL $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$
Output: random sample $S_R \subset (I \times O)^*$

```
1:  S_R ⟵ [ ]
2:  for i ← 1 to n_data do
3:      n_len ← random_integer(n_min, n_max)
4:      s^I ⟵ [ ]
5:      for j ← 1 to n_len do
6:          s^I ⟵ s^I · random(I)
7:      end for
8:      s^O ⟵ λ*(q_0, s^I)
9:      S_R ⟵ append(S_R, trace(s^I, s^O))
10: end for
```

Algorithm 3 iteratively adds random sequences to the S_R until the size of S_R is equal to n_{data}, where S_R is initialized to the empty list. For each trace, we pick a random length n_{len} within $[n_{\min}, n_{\max}]$ in Line 3. We then create a random input sequence of length n_{len}. In Line 6 we randomly select an input from the input alphabet and append it to the currently generated input sequence s^I. Afterwards, we execute s^I on the SUL to receive the corresponding output sequence s^O in Line 8. Let $\mathit{trace}\colon I^* \times O^* \to (I \times O)^*$ be a function that generates a trace of alternating inputs and outputs from a given input and output sequence of the same length. Line 9 then appends the randomly generated trace to S_R.

For our evaluation, we consider the following four different random samples:

(Sample 1) size $|S_{L^*}|$. This sample aims to represent a set with similar properties in terms of size and length as the sample S_{L^*}. The size of the random sample S_R is equal to the size of S_{L^*}, i.e., $|S_R| = |S_{L^*}|$. For defining the length of the traces, we set the parameters $n_{\min}$ and $n_{\max}$ based on the trace lengths in S_{L^*}. The shortest query in S_{L^*} considers only one input. Thus, we set $n_{\min} = 1$. Let $\overline{n_{L^*}}$ be the average trace length of S_{L^*}. We set $n_{\max} = \lfloor 2 \cdot \overline{n_{L^*}} - 1 \rceil$, where $\lfloor x \rceil$ is the closest natural number to the real number $x \in \mathbb{R}$, where a distance of 0.5 is associated with the higher natural number.

(Sample 2) size $2 \cdot |S_{L^*}|$. This sample is equal to **Sample 1** except the number of traces is doubled, i.e., $|S_R| = 2 \cdot |S_{L^*}|$.

(Sample 3) length $[|Q|, 2 \cdot |Q|]$. This sample includes traces whose length is aligned to the size of the ground truth Mealy machine $\mathcal{M}$. We set $n_{\min} = |Q|$ and $n_{\max} = 2 \cdot |Q|$, where Q is the set of states. The size of the sample is again $|S_{L^*}|$.

(Sample 4) sufficient traces. The size and length of this sample are set so that the randomly generated sample is sufficient to learn a Mealy machine that is behavioral equivalent to the ground truth. To create such a sample, we iteratively increase the size of the sample and the length of the trace. To do so, we created several samples, increasing either the size or the length, or both.

6.2.3 Result Evaluation

We define an equivalence metric to measure the behavioral equivalence between a learned automaton and the ground truth automaton. Similar to active learning, we use conformance

Table 6.1: Properties of the Mealy machines modeling BLE devices and MQTT implementations. Full models are available online [127].

BLE			**MQTT**		
SUL	$\|Q\|$	$\|I\|$	*SUL*	$\|Q\|$	$\|I\|$
CC2640 [no feature_req]	11	8	ActiveMQ	18	9
CC2640 [no pairing_req]	6	8	emqtt	18	9
CC2650	5	9	HBMQTT	17	9
CC2652R1	4	7	Eclipse Mosquitto	18	9
CYBLE-416045-02	3	9	VerneMQ	17	9
CYW43455	16	7			
nRF52832	5	9			

testing techniques to check for behavioral equivalence. We define a conformance relation that is similar to Equation 2.1. Let $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ be the learned Mealy machine and $\mathcal{M}_{\text{SUL}} = \langle Q_{\text{SUL}}, q_{0_{\text{SUL}}}, I, O, \delta_{\text{SUL}}, \lambda_{\text{SUL}} \rangle$ the ground truth Mealy machine. We then define a test suite T that consists of a set of test cases, where each test case is based on a trace $t \in \mathcal{L}(\mathcal{M}_{\text{SUL}})$. We say that $\mathcal{M}$ conforms to $\mathcal{M}_{\text{SUL}}$, if the following equation holds:

$$\mathcal{M} \textbf{ imp } \mathcal{M}_{\text{SUL}} \Leftrightarrow \forall\, t \in T \colon \mathcal{M} \textbf{ passes } t. \tag{6.1}$$

We say that $\mathcal{M}$ **passes** t if $\lambda^*(t^I) = t$ holds, where t^I is the input sequence of the trace t. For our evaluation, we considered two different methods to generate a test suite. In both methods, we approach a test suite size of approximately 10 000 traces.

Our first test suite is a randomly generated sample. For the generation of a random sample, we follow Algorithm 3. We set the sample size to $n_{\text{data}} = 10\,000$. We set the length of the randomly generated traces based on the number of states of the considered SULs. We introduce our SULs in Section 6.2.4. We set the minimum length $n_{\text{min}} = 3$, which is equal to the fewest number of states. The maximum length is set to $n_{\text{max}} = 32$, which is twice the largest number of states.

For generating the second test suite, we consider the underlying ground truth model $\mathcal{M}$ of the corresponding SUL to generate a coverage-based test suite. The generation of the coverage-based test suite is similar to the approach that we use in active learning for the substitution of the equivalence oracle. We described the coverage-based generation technique in the active learning setup in Section 6.2.1. For the evaluation of the results, we set the number of walks to every state $n_{\text{walk}} = \lceil \frac{10\,000}{|Q|} \rceil$ and the length of the random suffix of trace $n_{\text{len}} = 10$, where Q is the set of states of the minimal Mealy machine representing the corresponding SUL.

6.2.4 Case Study Subjects

We compare active and passive learning techniques based on their performance in learning communication protocols. The investigated protocols are BLE and MQTT. The purpose of this evaluation is to compare the different learning paradigms. Hence, we decided to consider the already learned automata as SUL instead of the real implementation, which makes it possible to generate large samples in a feasible amount of time.

Table 6.1 provides an overview of the considered case study subjects. For all SULs, we state the size of the input alphabet $|I|$ and the number of states $|Q|$ of the minimal Mealy machine. For BLE, we considered the six BLE devices of our case study presented in Chapter 4. Since we learned three different models for the CC2640, we considered only two variants that cover the difference in the state space between the variants adequately. For MQTT, we consider learned models of MQTT broker implementations that are available in the benchmark set in the *Automata Wiki* [131]. The MQTT broker models originate from a case study on learning-based testing MQTT brokers performed by Tappler et al. [170]. We selected a subset of the provided MQTT broker models. The selected Mealy machines model a two-client setup with the *will*

message as an additional input. A client sends a will message to the broker and the broker then distributes the will message when the client disconnects. This advanced setup increases the state space of the MQTT models noticeably. All models of this case study are available online [127].

6.3 Evaluation

In the following, Section 6.3.1 describes the experimental setup in which we performed all experiments. Afterwards, we provide the results of our evaluation in Section 6.3.2. Section 6.3.3 concludes with a discussion on the proposed research challenges of this chapter.

6.3.1 Experimental Setup

We performed all experiments except for the results presented in Figure 6.1 on an Apple MacBook Pro 2019 with an Intel Quad-Core i5 running at 2.4 GHz and with 8 GB memory. The experiments shown in Figure 6.1 were run on a Dell Latitude 5410 with Intel Core i7-10610U running at 2.3 GHz and with 16 GB memory. The source code and the automata considered for this case study are available **online** [127].

6.3.2 Results

In the following, we provide the results of our evaluation. All experiments were repeated five times. Unless otherwise stated, the following tables contain the average results. The standard deviation is indicated in curly brackets ('{...}').

Table 6.2 and Table 6.4 present the results of actively learning the BLE and MQTT automata. The tables provide the number of performed output queries and conformance tests, with the corresponding number of input steps performed. Furthermore, we show the number of learning rounds, which is equal to the number of equivalence queries performed during learning. The sum of queries corresponds to the size of the sample S_{L^*} and $\overline{n_{\text{len}}}$ shows the average length. Additionally, the table provides the size of the optimized sample that would be generated by L^* under the condition that the characterization set and state prefixes are known. We also state the average length of the traces in the optimized sample.

Table 6.3 and Table 6.5 present the passive learning results for BLE and MQTT respectively. The tables include the results for the previously described random samples: **Sample 1**, **Sample 2** and **Sample 3**. For each sample, we provide the sample size n_{data} and the average trace length $\overline{n_{\text{len}}}$. We then provide the conformance metrics based on the random sample and on the sample that provides state coverage. The provided conformance metrics state the percentage of passed test cases on the corresponding test suites. As a last metric, we state the absolute number of correctly learned models within the five repetitions. Thus, if the number of correct models is equal to 5, this would mean that in all repetitions a model is learned that is 100% conforming to the ground truth.

For the evaluation of **Sample 4**, Figure 6.1 provides an overview of the influence of the sample parameters to create a sufficient sample. The heatmaps show on the x-axis the sample size, where we constantly increase the factor that is multiplied with $|S_{L^*}|$. The y-axis provides the average trace length. The values in the heatmap are color coded. The darker the green, the higher the conformance, corresponding to the provided numerical value. We present these values for one BLE SUL, the CC2640 (no `feature_req`), and for one MQTT SUL, the Eclipse Mosquitto broker.

Table 6.2: Active learning results of the BLE case study. We repeated each experiment five times. Since all devices could be learned within one learning round, there is no deviation between the repetitions. All BLE automata could be learned correctly.

	CC2640 no feature_req	**CC2640** no pairing_req	**CC2650**	**CC2652R1**	**CYBLE-416045-02**	**CYW43455**	**nRF52832**
# Output queries	704	384	405	196	243	784	405
# Output queries steps	3136	1472	1458	588	729	3136	1458
# Conformance tests	275	150	125	100	75	400	125
# Conformance tests steps	8925	4775	3950	3100	2325	12800	3950
# Learning rounds	1	1	1	1	1	1	1
Sum queries	979	534	530	296	318	1184	530
Sum steps	12061	6247	5408	3688	3054	15936	5408
$\overline{n_{len}}$	12.32	11.70	10.20	12.46	9.60	13.46	10.20
# Optimized queries	312	129	123	50	50	388	123
Optimized $\overline{n_{len}}$	4.55	3.9	3.65	3.08	3.04	4.13	3.65

Table 6.3: Passive learning results of the BLE case study. We repeated each experiment five times. The standard deviation is given in curly brackets '{...}'.

		CC2640 no feature_req	**CC2640** no pairing_req	**CC2650**	**CC2652R1**	**CYBLE-416045-02**	**CYW43455**	**nRF52832**
Sample 1 (size $\lvert S_{L^*}\rvert$)	n_{data}	979.00 {0.00}	534.00 {0.00}	530.00 {0.00}	296.00 {0.00}	318.00 {0.00}	1184.00 {0.00}	530.00 {0.00}
	$\overline{n_{\text{len}}}$	12.64 {0.25}	11.46 {0.32}	9.84 {0.30}	12.69 {0.37}	9.47 {0.23}	13.54 {0.20}	10.04 {0.24}
	Conformance (random) %	99.92 {0.01}	99.93 {0.02}	99.89 {0.06}	100.00 {0.00}	99.93 {0.06}	100.00 {0.00}	99.94 {0.03}
	Conformance (coverage) %	99.80 {0.02}	99.86 {0.03}	99.82 {0.08}	100.00 {0.00}	99.89 {0.08}	100.00 {0.00}	99.85 {0.09}
	Correct model	0	0	0	5	1	5	0
Sample 2 (size $2 \cdot \lvert S_{L^*}\rvert$)	n_{data}	1958.00 {0.00}	1068.00 {0.00}	1060.00 {0.00}	592.00 {0.00}	636.00 {0.00}	2368.00 {0.00}	1060.00 {0.00}
	$\overline{n_{\text{len}}}$	12.49 {0.16}	11.62 {0.15}	10.02 {0.20}	12.33 {0.24}	9.61 {0.16}	13.55 {0.10}	10.10 {0.15}
	Conformance (random) %	99.97 {0.01}	99.98 {0.03}	99.98 {0.02}	100.00 {0.00}	100.00 {0.00}	100.00 {0.00}	99.97 {0.02}
	Conformance (coverage) %	99.92 {0.04}	99.96 {0.06}	99.96 {0.04}	100.00 {0.00}	100.00 {0.00}	100.00 {0.00}	99.93 {0.06}
	Correct model	0	2	2	5	5	5	0
Sample 3 (length $[\lvert Q\rvert, 2 \cdot \lvert Q\rvert]$)	n_{data}	979.00 {0}	534.00 {0}	530.00 {0}	296.00 {0}	318.00 {0}	1184.00 {0}	530.00 {0}
	$\overline{n_{\text{len}}}$	16.55 {0.05}	9.03 {0.07}	7.46 {0.09}	5.99 {0.07}	4.49 {0.10}	24.08 {0.12}	7.54 {0.07}
	Conformance (random) %	99.93 {0.04}	99.87 {0.04}	99.87 {0.06}	99.81 {0.14}	99.79 {0.10}	100.00 {0.00}	99.91 {0.03}
	Conformance (coverage) %	99.83 {0.09}	99.76 {0.07}	99.76 {0.08}	99.86 {0.10}	99.72 {0.11}	100.00 {0.00}	99.81 {0.04}
	Correct model	0	0	0	1	0	5	0

Table 6.4: Active learning results of the MQTT case study. We repeated each experiment five times. The standard deviation is given in curly brackets '{...}'. All MQTT automata could be learned correctly.

	ActiveMQ	**emqtt**	**HBMQTT**	**Eclipse Mosquitto**	**VerneMQ**
# Output queries	5890.40 {695.77}	5900.60 {1834.84}	4009.40 {533.83}	4056.40 {846.08}	4356.20 {976.87}
# Output queries steps	55771.60 {14265.88}	62134.80 {33276.46}	33176.40 {8277.56}	34335.80 {10979.61}	33850.80 {12344.85}
# Conformance tests	450 {0.00}	450 {0.00}	425 {0.00}	450 {0.00}	425 {0.00}
# Conformance tests steps	14721.2 {70.33}	14654.2 {24.69}	13794 {16.41}	14737 {88.12}	13834.4 {87.33}
# Learning rounds	5.20 {1.30}	5.20 {1.30}	3.60 {0.55}	4.00 {1.00}	5.20 {1.10}
Sum queries	6340.40 {695.77}	6350.60 {1834.84}	4434.40 {533.83}	4506.40 {846.08}	4781.20 {976.87}
Sum steps	70492.80 {14208.67}	76789.00 {33275.29}	46970.40 {8282.37}	49072.80 {10998.34}	47685.20 {12364.07}
$\overline{n_{len}}$	11.04 {1.18}	11.77 {2.21}	10.55 {0.86}	10.85 {1.00}	9.90 {0.66}
# Optimized queries	1450	1450	959	1015	959
Optimized $\overline{n_{len}}$	6.26	6.26	5.91	6.05	6.05

6.3.3 Discussion

We discuss the presented results based on the following three research challenges:

- **(RC 1)** Can passive learning based on a random sample compete with active learning?
- **(RC 2)** Does the considered active automata learning algorithm generate an optimal sample?
- **(RC 3)** Can random sampling support active automata learning?

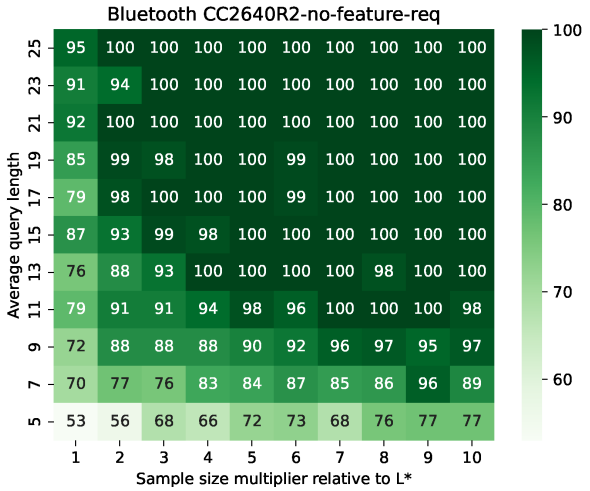
(RC 1) *Can passive learning based on a random sample compete with active learning?* Table 6.3 and Table 6.5 show that passive learning achieved high conformance values, where the lowest value is 99.72% conformance. This indicates that passive learning approximated the correct behavior very well with the same resources as active learning.

However, passive learning was not sufficient to achieve 100% conformance in most cases. We observe a slight difference between the BLE and MQTT results. Table 6.3 shows that passive learning could learn a conforming automaton. For the CC2652R1 and the CYW43455, the same sampling budget as for active learning was sufficient to learn a conforming model in all repetitions. For some experiments, we occasionally observe conforming solutions especially if we increase the sample size. However, in approximately two-thirds of the experiments, passive learning did not learn a conforming model.

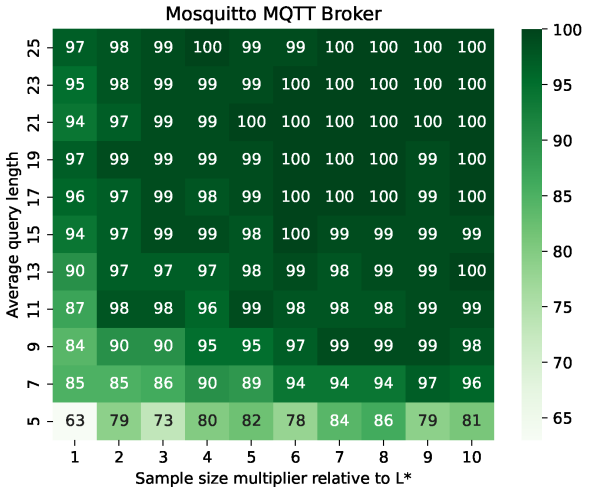
Table 6.5 shows, that for the MQTT case study, passive learning never managed to learn a conforming model. Even though the achieved conformance is never below 99.9%, except for one setup on the HBMQTT. A closer look at the learned models reveals that the passively learned models have much more states than the minimal solution of the SUL. For example, the learned models on the HBMQTT have on average 64.4 states, whereas the minimal ground truth has 17. This shows that the learned automata achieve good behavioral conformance, but due to the

Table 6.5: Passive learning results of the MQTT case study. We repeated each experiment five times. The standard deviation is given in curly brackets '{...}'.

		ActiveMQ	**emqtt**	**HBMQTT**	**Eclipse Mosquitto**	**VerneMQ**
Sample 1 (size $\lvert S_{L^*}\rvert$)	n_{data}	6340.00 {0.00}	6351.00 {0.00}	4434.00 {0.00}	4506.00 {0.00}	4781.00 {0.00}
	$\overline{n_{\text{len}}}$	11.00 {0.11}	12.00 {0.05}	10.46 {0.08}	10.98 {0.06}	10.07 {0.10}
	Conformance (random) %	99.97 {0.01}	99.96 {0.01}	99.85 {0.01}	99.95 {0.00}	99.95 {0.02}
	Conformance (coverage) %	99.95 {0.02}	99.94 {0.02}	99.79 {0.02}	99.91 {0.01}	99.92 {0.04}
	Correct model	0	0	0	0	0
Sample 2 (size $2 \cdot \lvert S_{L^*}\rvert$)	n_{data}	12680.00 {0.00}	12702.00 {0.00}	8868.00 {0.00}	9012.00 {0.00}	9562.00 {0.00}
	$\overline{n_{\text{len}}}$	11.02 {0.06}	11.99 {0.03}	10.54 {0.03}	11.00 {0.04}	9.94 {0.05}
	Conformance (random) %	99.98 {0.01}	99.97 {0.01}	99.98 {0.01}	99.98 {0.01}	99.98 {0.01}
	Conformance (coverage) %	99.97 {0.01}	99.97 {0.01}	99.96 {0.02}	99.96 {0.02}	99.97 {0.02}
	Correct model	0	0	0	0	0
Sample 3 (length $[\lvert Q\rvert, 2 \cdot \lvert Q\rvert]$)	n_{data}	6340.00 {0.00}	6351.00 {0.00}	4434.00 {0.00}	4506.00 {0.00}	4781.00 {0.00}
	$\overline{n_{\text{len}}}$	26.99 {0.05}	27.00 {0.02}	25.52 {0.02}	27.00 {0.15}	25.45 {0.12}
	Conformance (random) %	99.97 {0.01}	99.97 {0.01}	99.99 {0.01}	99.95 {0.06}	99.97 {0.02}
	Conformance (coverage) %	99.96 {0.02}	99.96 {0.02}	99.97 {0.01}	99.94 {0.06}	99.97 {0.02}
	Correct model	0	0	0	0	0



(a) Conformance heatmap of different sample sizes for CC2640 (no feature_req).



(b) Conformance heatmap of different sample sizes for Eclipse Mosquitto.

Figure 6.1: The heatmaps show the achieved conformance (%) to corresponding SUL of the different random sample sizes for one BLE device and one MQTT broker.

missing data, many states could not be merged, which leads to larger automata. Automata with more than 60 states might be hard to interpret for humans.

The heatmaps presented in Figure 6.1 show that generating a sufficient data set with random sampling requires a large sample compared to the sample generated by active learning. Therefore, passive learning might be sufficient to approximate high-conforming solutions, but learning the minimal automaton with a classical implementation of RPNI, requires large random samples.

(RC 2) *Does the considered active automata learning algorithm generate an optimal sample?*

The results of RC 1 show that learning with RPNI requires a large random sample in order to learn the minimal conforming automaton. If the goal is to learn a 100% conforming automaton, active learning might still be the better choice in terms of the number of performed queries. However, the setup of an active learning interface can be tedious and reliable learning might require the repetition of queries. Hence, we aim to apply active learning techniques that require as few interactions as possible with the SUL. Table 6.2 and Table 6.4 compare the number of queries that are actually required by L^* with the optimal number of queries. On average S_{L^*} is 4.45 times larger than the optimized sample for the BLE case study and 4.55 times larger for the MQTT case study. We observe a similar trend in the comparison of the actual average trace length versus the optimal average trace length. In practice, it would not be feasible to reduce the number of queries to the optimal sample size as this would require a perfect equivalence oracle. However, we still assume that the achieved results by L^* can be further improved since the queries performed by L^* include a lot of redundancy.

(RC 3) *Can random sampling support active automata learning?* Our results in RC 2 show that the applied active approach requires a lot more queries than an optimal approach would require. This raises the question how active learning, especially L^*-based techniques, can be improved to reduce the required interaction with the SUL. Due to the iterative nature of L^*, we assume that many queries are prefixes of later performed queries. To overcome this, we want to investigate whether a pre-initialization of the query cache might help to reduce this redundancy in the queries.

We initialize the cache that is used in the tested L^* variant by a randomly generated sample.

The randomly generated sample fulfills the same properties as **Sample 1**. Our results show that initializing the cache in this way does little to save queries. On average, 27% of the queries were cached in the BLE examples, leaving 73% to be executed on the SUL. For MQTT, we observe similar results, where on average only 10.6% of the queries were cached. Thus, even if we initialize the cache with a random sample that has the same size as the sample required to learn the system, only a minority of queries could be looked up in the cache.

6.4 Conclusion

This chapter presented a comparison between active and passive learning paradigms. The generation of a fault-tolerant learning interface in active learning can be tedious. The presented case studies on learning communication protocols in Chapter 4 and Chapter 5 showed that several countermeasures had to be implemented in order to deal with delayed or lost packets.

In this chapter, we examined whether the effort to develop such an interface is justified. First, we evaluated if passive learning on random samples is sufficient to adequately model the system behavior. Our results showed that random samples of the same size as the sample generated during active learning achieve quite high behavioral conformance. Nevertheless, the minimal confirming automaton could not be learned in most cases. The main problem was that the random sample had to be very large to adequately cover the behavior of the SULs. Passive learning algorithms that can deal with incomplete data might be a possible solution to overcome this problem. In the next chapter, we will present a novel RNN-based architecture to solve this problem.

Given the requirement to learn a behavioral model that is equal to the SUL, we rather accept the effort of creating a reliable learning interface for active learning. For this purpose, we also compared the effort required by the used L^* variant in terms of performed queries with the optimal sample L^* would require under perfect conditions. The results demonstrated that the difference is significant. Even if the reduction to an optimal sample would be infeasible in practice, we still assume that improvements are possible. Even the initialization with a random sample did not reduce the number of required queries noticeably.

Another direction would be to change the considered active learning algorithm. Tree-based algorithms such as *KV* [96] or *TTT* [89] might help to reduce redundancy in the performed queries by inferring the characterization set using different techniques. The results of the case study on VPN presented in Chapter 5 also pointed in this direction.

(RQ 2.1) Does passive learning represent an alternative to active learning?

To compare the passive and active learning paradigms, we investigated whether passive learning can achieve competing results with a similar budget as active learning. Our results showed that the passively learned models achieved high conformance scores compared to the ground truth models, where the underlying sample for passive learning was randomly generated. However, to achieve 100% conformance, large random samples would have been required to cover the whole behavior of the SUL. Thus, the active learning efforts were still worthwhile. When active learning is not possible, passive learning with a random sample provides a highly accurate approximation.

(RQ 2.2) How to improve automata learning to make it feasible for different challenges in networked environments?

Typically, the goal of active automaton learning is to learn a conforming behavioral model with as little interaction as possible. To reduce the number of interactions required for active learning, we initialized the internal cache with a random sample. However, it turned out that random samples based on the active learning budget covered only the minority of queries that had to be performed.

Chapter 7

Automata Learning with Recurrent Neural Networks

Declaration of Resources

This chapter is based on the paper *"Constrained Training of Recurrent Neural Networks for Automata Learning"* [12] presented at *SEFM 2022* and the article *"Learning Minimal Automata with Recurrent Neural Networks"* [14] submitted to the journal *"Software and Systems Modeling"* in February 2023.

7.1 Introduction

The previous chapter shows that active learning techniques are preferable compared to passive learning techniques based on random samples. Classic passive learning techniques such as RPNI require a large randomly generated sample to learn a behavioral model that adequately represents the SUL. However, active learning is not always feasible or requires a comprehensive setup. For this purpose, we require passive learning techniques that better generalize on sparse samples.

Kleene [101] already showed in 1951 that neural networks can simulate finite state machines that define regular languages. recurrent neural networks (RNNs) are well-suited to model sequential behavior. However, trained RNNs often miss interpretability, where their predictions depend on large real-valued vectors. Behavioral models are usually better interpretable by humans, which is beneficial for the behavioral analysis of black-box systems. Furthermore, Khmelnitsky et al. [98] show that the learned models enable the application of automated techniques for verifying RNNs using model-checking techniques.

In this chapter, we want to investigate whether machine learning techniques can be used not only to simulate the behavior of finite state machines, but also to predict their structure, as is done by automata learning techniques. Gold [73] shows that the problem of inferring a behavioral model with at most k states is *NP*-complete. This chapter approaches this problem. For this purpose, we propose an RNN-based framework that infers Mealy machines from a given sample of input and output traces. We design an RNN architecture that allows us to simulate the state-transition and output function of a Mealy machine. The presented RNN architecture applies a special regularization term for training the RNNs. For learning a minimal finite state representation, we propose an approach that iteratively converges towards a minimal finite state representation.

We evaluate our approach on theoretical and practical examples. These examples have been widely used in the past to evaluate RNN-based learning techniques. In a second case study, we investigate the practical applicability of our approach, by training an RNN on BLE traces. Our results show that accurate models can be learned even with a small random sample. However,

we also show that the size of the SUL presents a non-negligible challenge.

7.2 Background

This section provides background on the RNN architecture under consideration. In addition to the background provided in Chapter 2 on Mealy machines (Section 2.1), this section discusses further concepts that are particularly relevant to our RNN-based learning technique.

7.2.1 Recurrent Neural Network (RNN)

RNNs are popular for the simulation of sequential data such as time series data. Hence, we consider them sufficient to model the behavior of reactive systems. In the following, we consider a *vanilla* RNN architecture with hidden layers. A vanilla RNN predicts a corresponding output on a given input. In a vanilla RNN, the output of the hidden layer is then fed forward as an additional input to the next input in a sequence.

We define the hidden state h_t at the t^{th} step of a sequence as

$$h_t = af(i_t W_{hi} + h_{t-1} W_{hh} + b_h). \tag{7.1}$$

The hidden state is calculated using an activation function *af*. In the literature, popular activation functions are the hyperbolic tangent ($tanh$) or rectified linear unit ($ReLU$). The activation function takes as parameters the current input vector i_t and the previous hidden layer h_{t-1}. Additionally, the activation function considers the weights W_{hi}, W_{hh}, and the bias b_h.

We define the output of an RNN $\hat{o}_t$ to be

$$\hat{o}_t = g(W_o h_t + b_o). \tag{7.2}$$

The output $\hat{o}_t$ is again defined via an activation function g, but for the output prediction usually the functions *softmax* and *hardmax* are applied. The output function takes as parameters the hidden layer together with the weights W_o and b_o. The weights used in the activation functions of the hidden layer and output are optimized during training to predict the correct outputs.

For the training of an RNN, we use a sample of input/output traces S. We slightly adapt our notation of traces presented in Section 2.1, to better fit it into an RNN architecture. Let $\mathbf{s}_j \in S$ be a trace of alternating inputs and outputs, where $\mathbf{i}_j$ is the corresponding input sequence and $\mathbf{o}_j$ is the output sequence of the trace $\mathbf{s}_j$. By slightly relaxing the notation, we define that a sequence $\mathbf{s}_j \in S$ can also be represented by a pair $(\mathbf{i}_j, \mathbf{o}_j)$.

The RNN produces for a given input sequence $\mathbf{i}_j$, the output sequence $\hat{\mathbf{o}}_j$. For the training of the RNN, we consider the difference between the predicted output by the RNN $\hat{\mathbf{o}}_j$ and the actual outputs $\mathbf{o}_j$. We calculate the difference based on the cross entropy between $\hat{\mathbf{o}}_j$ and $\mathbf{o}_j$. To decrease the difference, we optimize the weights in the activation functions. For this purpose, we consider a gradient-descent-based optimization.

7.2.2 Addendum to Mealy Machines

In the following, we extend our preliminary remarks to Mealy machines. For this purpose, we introduce an encoding of Mealy machines for use in common machine learning frameworks. We discuss techniques for determining bounds on the state space for Mealy machines. Finally, we introduce the conceptual idea of state minimization in Mealy machines.

Encoding of Mealy machines. To conform to standard RNN architectures, we introduce an encoding for Mealy machines into a vector-based data structure. Let $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$ be a Mealy machine as defined in Section 2.1. We encode the finite set of states Q, inputs I, and outputs O using a one-hot encoding π. In a one-hot encoding, exactly one value in an

n-dimensional vector is set to one, whereas all other values are set to zero. The number of dimensions n of the vector depends on the size of the set that is encoded. For example, for encoding the finite set of inputs, the size of the vector corresponds to the number of inputs, i.e., $|I|$. We use the same one-hot encoding to encode each input and output in the given sample on which we train the RNN model.

Example 11 (Encoding of Mealy machines.) *The Mealy machine presented in Figure 2.1 defines the finite set of states* $Q = \{\mathsf{q}_0, \mathsf{q}_1, \mathsf{q}_2\}$*, the input alphabet* $I = \{\mathsf{connect}, \mathsf{publish}, \mathsf{subscribe}, \mathsf{unsubscribe}\}$ *and the output alphabet* $O = \{\mathsf{ack}, \mathsf{closed}, \mathsf{message}, \mathsf{none}\}$*. The following sets can be represented in the following one-hot encoding:*

- $\pi(Q) : \{\begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\end{pmatrix}\}$,
- $\pi(I) : \{\begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\1\end{pmatrix}\}$, *and*
- $\pi(O) : \{\begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\1\end{pmatrix}\}$,

where the indexes in the sets stay unchanged. Thus, for $i \in I$*,* $i = I[argmax(\pi(i))]$ *holds, where argmax function returns the index of the vector element with the highest value. For example, the input* $\mathsf{connect}$ *in* I *would be encoded as* $\pi(\mathsf{connect}) = \begin{pmatrix}1\\0\\0\end{pmatrix}$.

Bounding of Mealy machines. In Section 2.2.1, we explained that many passive learning algorithms initially build a prefix-tree acceptor (PTA) based on a provided sample. State-merging algorithms such as RPNI [49] then merge states in the PTA to create a minimal automaton. Hence, the number of states in the PTA defines an upper bound for the number of states the learned automaton can define based on the provided sample. For example, based on the PTA shown in Figure 2.2a an upper bound for the number of states of the learned Mealy machine would be six since the PTA defines six states for the sample provided in Example 2.

State minimization. Berg et al. [26] discuss that the L^* algorithm by construction learns a minimal automaton of a regular language. This property of the L^* algorithm is based on the Myhill-Nerode theorem [132]. This theorem defines equivalence classes for strings in a regular language, where the number of states of a minimal DFA is equivalent to the number of equivalence classes. Steffen et al. [167] provide a translation of this theorem for Mealy machines. Let two input sequences $i, i' \in I^*$ be in the same equivalence class, then

$$\forall z \in I^* \colon \lambda^*(i \cdot z) = \lambda^*(i' \cdot z) \tag{7.3}$$

holds. Similar to DFAs, every state in a minimal Mealy machine refers to one equivalence class. An equivalence class $[i]$ for a state $q \in Q$ includes all input sequences that lead to the state q when executing the input sequence i in the initial state. The initial state corresponds to the equivalence class with the empty sequence $[\epsilon]$. Hopcroft et al. [82] show, for DFAs that correctly define a regular language but are not minimal in the number of states, that there exists a partition of Q that groups states into the same equivalence class. Based on Equation 7.3, we

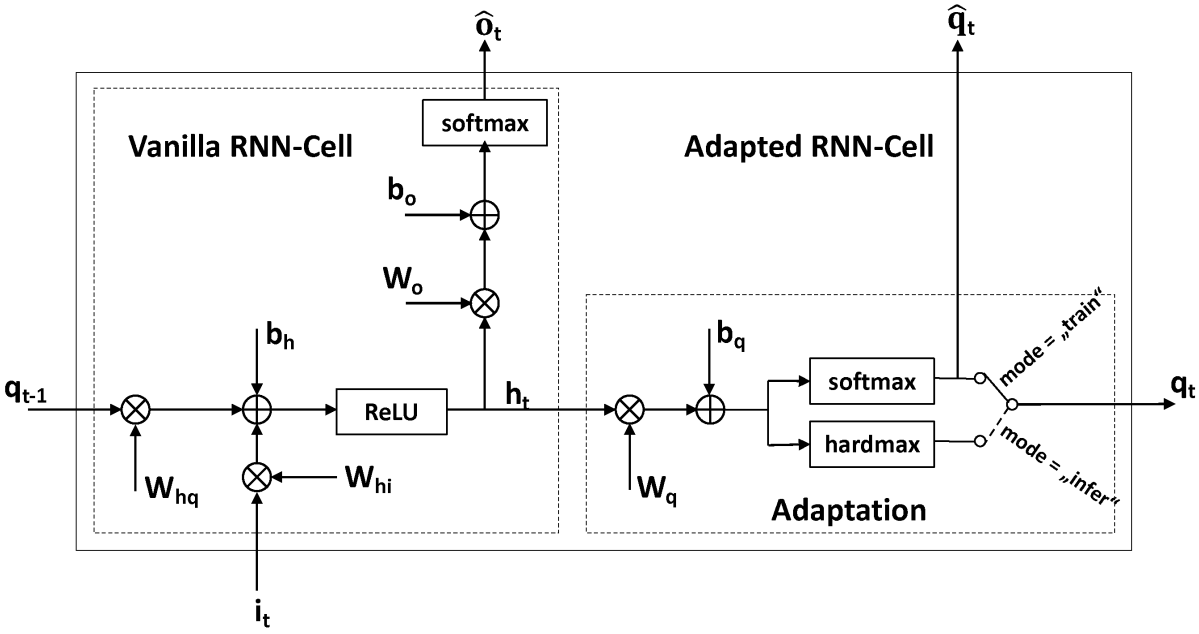


Figure 7.1: Adapted RNN cell that takes as inputs the t^{th} input of a sequence and the prediction of the $(t-1)^{\text{th}}$ state. A step in the RNN model can be made in the `train` or `infer` mode.

can apply this partitioning principle also for Mealy machines. The partition is a set of state sets, where each set is denoted as blocks B. Each block in the partition corresponds to an equivalence class. Note that the intersection between the blocks is the empty set. We define state transitions for an input $i \in I$ and a block $b \in B$ by $\delta([b], i) = [b \cdot i]$. Based on this construction, we can build a minimal Mealy machine. Note that every minimal Mealy machine $\mathcal{M}$ that represents a regular language is isomorph to any other minimal Mealy machine $\mathcal{M}'$ that represents the same language. Hopcroft [81] shows that minimization algorithms can be defined that take $O(n \log n)$ time.

7.3 Method

The goal of our RNN-based learning technique is to create a Mealy machine $\mathcal{M}$ that defines an unknown regular language based on a given sample S. Let $\mathcal{M}_{\text{SUL}}$ be an unknown Mealy machine that represents a language $\mathcal{L}(\mathcal{M}_{\text{SUL}})$. Let k define an upper bound for the number of states of $\mathcal{M}$. The given sample only includes positive traces, i.e., $S \subseteq \mathcal{L}(\mathcal{M}_{\text{SUL}})$. Under the assumption that S sufficiently represents $\mathcal{L}(\mathcal{M}_{\text{SUL}})$, our approach should satisfy $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}_{\text{SUL}})$, where $\mathcal{M}$ has at most k states.

In the following section, we present our RNN-based training approach. First, we introduce our developed RNN architecture that predicts the outputs on a given input sequence. We modified the classical vanilla RNN architecture in such a way that the feed-forward input corresponds to the next-state prediction simulating a state transition in a Mealy machine. The learning is based on a two-step procedure: First, we train the RNN model based on the given sample of traces. Second, we derive from the trained RNN model a Mealy machine with at most k states. We require for the learning procedure that an upper bound k is provided. Since in a black-box scenario the minimal number of states might be unknown, we show an algorithm that iteratively decreases the upper bound in order to learn a minimal Mealy machine.

7.3.1 Architecture

Figure 7.1 illustrates our RNN architecture. We modified the classical vanilla RNN architecture in such a way that we do not feed forward the hidden layer h_{t-1}. Instead, we extended the RNN cell by an adaption that predicts the next state similar to the state transition function of a Mealy machine. The state prediction is then fed forward to the next step in the execution of the current sequence. Hence, for a step in the adapted RNN cell, we provide an input i_t at a particular step t of an input sequence $\mathbf{i}$ and the state prediction q_{t-1} from the previous step. Note that our RNN cell distinguishes two modes of operation: "'`infer`" and "'`train`". Figure 7.1 depicts the following equations that define our RNN cell:

$$h_t = af(q_{t-1}W_{hq} + i_tW_{hi} + b_h) \qquad af \in \{tanh, ReLU\} \tag{7.4}$$

$$\hat{o}_t = softmax(h_tW_o + b_o) \tag{7.5}$$

$$\hat{q}_t = softmax(h_tW_q + b_q) \tag{7.6}$$

$$q_t = \begin{cases} softmax(W_qh_t + b_q) & \text{if mode="}\texttt{train}\text{"} \\ hardmax(W_qh_t + b_q) & \text{else (i.e. mode="}\texttt{infer}\text{")} \end{cases} \tag{7.7}$$

For the activation of the hidden layer, we either select the hyperbolic tangent ($tanh$) or the rectified linear unit ($ReLU$). The calculation of the hidden layer considers the current input and the previous state prediction. The goal of training is that the hidden layer encodes the state transition at a specific step of a provided input sequence by combining the input with the state information. The output prediction $\hat{o}_t$ is normalized by the *softmax* function. Note that the correct output o_t is provided in the currently executed trace. Hence, we use the prediction $\hat{o}_t$ only to calculate the difference to the actual output o_t based on the cross entropy of the two vectors. We use gradient-based optimization to adjust the weights so that the prediction matches the actual output. To approach a one-hot encoding for state predictions, our optimization also considers the cross entropy between $\hat{q}_t$ and $hardmax(\hat{q}_t)$. In the next section, we explain the training of the RNN model, including details of the loss function.

We execute an input sequence in this RNN model either in the "`train`" or "`infer`" mode. In the "`train`" mode, we base the activation function for the state vector q_t on the *softmax* function. During training, we continuously enforce the RNN model to approach a one-hot encoding for the state prediction. However, at the same time, we want to avoid getting stuck in a local maximum due to wrong state predictions. We assume that the *softmax* allows us to improve state predictions iteratively during training. To simulate the behavior of a Mealy machine, we execute an input sequence on the RNN cell in the "`infer`" mode. In the "`infer`" mode, state predictions are made using the *hardmax*. Note that the size of the vector q_t creates an upper bound for the number of states the learned Mealy machine could define.

7.3.2 Training and Automaton Extraction

Forward pass. Algorithm 4 describes the procedure of simulating an input sequence on the RNN model M. We denote this procedure as *forward pass*, which takes as input an input sequence $\mathbf{i}$ and the mode in which the RNN model should be executed, i.e., "`train`" or "`infer`". The forward-pass procedure then returns a pair ($\hat{\mathbf{o}}$, $\hat{\mathbf{q}}$) which includes the sequence of output and state predictions respectively.

The forward pass implements the RNN architecture as explained in the previous section. The algorithm starts in Line 1 to initialize the sequences of output vectors $\hat{\mathbf{o}}$ and state predictions $\hat{\mathbf{q}}$ with the empty list. In Line 2, we initialize the current state q with the one-hot-encoded initial state $\mathtt{q}_0$. Thus, every forward pass starts at the initial state.

We iterate through each input of the provided input sequence $\mathbf{i}$ starting at Line 3. From Line 4 to Line 11, we implement our RNN architecture based on the equations that we introduced

Algorithm 4 Model forward pass $M(\mathbf{i}, mode)$

Input: Input sequence $\mathbf{i}$, Forward pass mode $\in \{$"`train`", "`infer`"$\}$
Output: Pair $(\hat{\mathbf{o}}, \hat{\mathbf{q}})$ of predicted outputs and automaton states, respectively

1: $\hat{\mathbf{o}}, \hat{\mathbf{q}} \leftarrow [\,], [\,]$
2: $q \leftarrow \pi(\mathsf{q}_0)$
3: **for** $t \leftarrow 0$ **to** $\#steps(\mathbf{i})$ **do**
4: $\quad h = af(qW_{hq} + i_t W_{hi} + b_h)$ ▷ $af \in \{ReLU, \tanh\}$
5: $\quad \hat{o}_t \leftarrow softmax(W_o h + b_o)$
6: $\quad \hat{q}_t \leftarrow softmax(W_q h + b_q)$
7: $\quad$ **if** $mode =$ "`train`" **then**
8: $\quad\quad q \leftarrow \hat{q}_t$
9: $\quad$ **else** ▷ $mode =$ "`infer`"
10: $\quad\quad q \leftarrow hardmax(W_q h + b_q)$
11: $\quad$ **end if**
12: **end for**
13: **return** $(\hat{\mathbf{o}}, \hat{\mathbf{q}})$

in the previous section. Note that we assume that the activation function *af* for the hidden layer in Line 4 is globally set depending on the currently executed experiment. We either consider *tanh* or *ReLU* as activation functions.

By iterating through the input sequence, we collect in each step the output prediction $\hat{o}_t$ (Line 5) and the state prediction $\hat{q}_t$ (Line 6), where both are normalized using the *softmax* function. Hence, we enforce a categorization of the predicted values, where each value is between zero and one and the values of the vector sum up to one. Depending on the mode in which the RNN model is executed, we either feed-forward the *hardmax* or *softmax* prediction of the next state.

RNN training. Algorithm 5 describes the training procedure of our RNN model. The goal of training is to learn the weights used in the equations that define our RNN model to make accurate predictions about output and state behavior. In classic RNN-based training approaches, we aim to avoid overfitting the training data to make general predictions on unknown inputs. However, for inferring a deterministic behavioral model, we explicitly target overfitting. More precisely, we train the RNN model until it correctly predicts all outputs of the given sample. Note that for training, we do not consider any state behavior from the unknown Mealy machine $\mathcal{M}_{\text{SUL}}$. We only consider the output behavior provided by the given sample.

Algorithm 5 takes as input the RNN model M and the sample S of input and output traces. We train the RNN model for a maximum of $\#epochs \in \mathbb{N}$ iterations. Furthermore, we consider a constant factor $C \in \mathbb{R}$ that regularizes the influence of state predictions in the loss function. The algorithm returns the trained RNN model M and a Boolean variable *converged* indicating whether training has converged. Our algorithm converges if M correctly predicts all outputs.

The algorithm starts with the initialization of variables. To optimize our weights in the activation functions, we use the *Adam* optimizer [100] which implements a stochastic gradient-descent-based optimization. In Line 1, we initialize *Adam* with its standard configuration. We train the RNN model for a maximum number of epochs, given by $\#epochs$. In each epoch, we iterate through every trace in the provided sample S starting in Line 4. We first perform the forward pass in the RNN model given the current input sequence $\mathbf{i}$ as described in Algorithm 4. For training, we forward pass $\mathbf{i}$ in the training mode. Based on the returned predicted outputs $\mathbf{o}_{\text{tr}}$ and states $\mathbf{q}_{\text{tr}}$, we then calculate the loss in Line 6. The loss term includes the cross entropy between the output predictions $\mathbf{o}_{\text{tr}}$ and the actual outputs $\mathbf{o}$ from the current trace in the sample. Furthermore, we want to enforce the prediction of the next state to conform to a one-

Algorithm 5 RNN training $train(M, S)$

Input: Initialized RNN model M, Training sample $S = \{(\mathbf{i}_1, \mathbf{o}_1), \ldots, (\mathbf{i}_m, \mathbf{o}_m)\}$, $\#epochs$, Regularization factor C
Output: Pair of Boolean variable *converged* indicating accuracy and trained RNN model M

1: $optimizer \leftarrow Adam(M)$
2: $converged \leftarrow \bot$
3: **for** $i \leftarrow 1$ **to** $\#epochs$ **do**
4: **for** $(\mathbf{i}, \mathbf{o}) \in S$ **do**
5: $\hat{\mathbf{o}}_{\text{tr}}, \hat{\mathbf{q}}_{\text{tr}} \leftarrow M(\mathbf{i}, \text{“}\texttt{train}\text{”})$
6: $loss \leftarrow cross\_entropy(\mathbf{o}, \hat{\mathbf{o}}_{\text{tr}}) + cross\_entropy(hardmax(\hat{\mathbf{q}}_{\text{tr}}), \hat{\mathbf{q}}_{\text{tr}}) \times C$
7: $loss.backward()$
8: $optimizer.step()$
9: **end for**
10: $acc_{\text{inf}} \leftarrow 0$
11: **for** $(\mathbf{i}, \mathbf{o}) \in S$ **do**
12: $\hat{\mathbf{o}}_{\text{inf}}, \hat{\mathbf{q}}_{\text{inf}} \leftarrow M(\mathbf{i}, \text{“}\texttt{infer}\text{”})$
13: $acc_{\text{inf}} \leftarrow acc_{\text{inf}} + accuracy(\mathbf{o}, \hat{\mathbf{o}}_{\text{inf}})/|S|$
14: **end for**
15: **if** $acc_{\text{inf}} = 100\%$ **then**
16: $converged \leftarrow \top$
17: **break**
18: **end if**
19: **end for**
20: **return** $(converged, M)$

hot encoding. For this, we extend our loss function by a regularization term that considers the cross entropy between the state predictions $\mathbf{q}_{\text{tr}}$ and the *hardmax* of the same predictions. The influence of this regularization term can be controlled by the constant C. Line 7 computes then the gradients during the backward pass. We optimize the weights of our RNN model in Line 8.

After training the RNN model on all traces in S, we evaluate the trained model. For the evaluation, we again iterate through the whole sample in Line 11 and simulate each trace on M. This time, however, we perform the forward pass in the `infer` mode (Line 12). By doing so, we evaluate how well the trained RNN model simulates a Mealy machine. We define the accuracy of the RNN model based on the output prediction and the actual output in the trace. We accumulate all accuracy values in Line 15. In the case that the model predicts all output values correctly, it would achieve 100% accuracy. In Line 15, we check if all outputs have been predicted correctly. If this is the case, we set the Boolean variable *converged* to true and terminate the training procedure. Otherwise, we continue training the RNN model for another epoch. Training is repeated for a maximum number of epochs.

Automaton extraction. Algorithm 6 describes the second step in our RNN-based learning method, in which we extract the Mealy machine from the trained RNN model.

The extraction algorithm takes as input the trained RNN model M and the sample S, and returns the generated Mealy machine $\mathcal{M}$. The algorithm starts by extracting the input and output sets in Line 1 from S. Line 2 initializes the finite set of states Q with the initial state $\mathtt{q}_0$.

We then extract the state transitions and output transitions by executing the sample on the trained RNN model. From Line 3 to Line 18, we iterate through each trace in the sample. First, we forward pass the input sequence on M using the “`infer`” mode to simulate the behavior of a Mealy machine. Next, in Line 6, we iterate through each step of the current trace. For each step in the trace, we extract the index of the predicted next state in Line 7. The input and output

Algorithm 6 Automaton extraction from RNN $extract(M, S)$

Input: Trained RNN model M, Training data $S = \{(\mathbf{i}_1, \mathbf{o}_1), \ldots, (\mathbf{i}_m, \mathbf{o}_m)\}$
Output: Mealy machine $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$

```
1:  I, O ← inputs(S), outputs(S)
2:  Q = {q_0}
3:  for trace ← (i, o) ∈ S do
4:      ô, q̂ ← M(i, "infer")
5:      state_index ← 0
6:      for t ← 0 to #steps(trace) do
7:          state_index' ← argmax(q̂_t)
8:          i, o ← argmax(i_t), argmax(o_t)
9:          if o = argmax(ô_t) then
10:             q, q' ← q_state_index, q_state_index'
11:             Q ← Q ∪ q'
12:             inp, out ← I[i], O[o]
13:             δ(q, inp) ← q'
14:             λ(q, inp) ← out
15:         else
16:             break
17:         end if
18:         state_index ← state_index'
19:     end for
20: end for
21: return M
```

index are taken directly from the sample (Line 8). In Line 9, we check if the predicted output is equal to the corresponding output in the trace. If this is the case, we define the source and target state, $\mathbf{q}$ and $\mathbf{q}'$ respectively. After updating the set of states Q, we extract the input and output based on the given alphabet in Line 12. As a last step, we extend the state transitions function δ and output function λ in Line 13 and Line 14 respectively.

If the predicted output does not match the output in the currently considered trace, we terminate the iteration through this trace and proceed to the next trace in the sample. After iterating through each trace, we return the generated Mealy machine $\mathcal{M}$.

7.3.3 Learning Minimal Automaton

The previous chapter introduces the two-step procedure, where we first train an RNN model and then extract from the trained model a Mealy machine. Note that the initialization of the RNN model requires an assumption about the number of states. In practice, such an assumption can hardly be made. To overcome this problem, we propose an iterative procedure that continuously decreases the maximum number of states until a fixpoint is reached.

Algorithm 7 describes our fixpoint algorithm in order to learn a minimal automaton. The algorithm requires as input an integer $\#runs$, which defines a maximum budget for reducing the assumed upper bound of states. For training the RNN model, we require a sample of traces. Our fixpoint algorithm distinguishes two strategies for finding a minimal Mealy machine. We distinguish between the best-effort and exhaustive strategy. By using the best-effort strategy, we immediately reduce the considered upper bound of states in case we found an accurate model. The exhaustive strategy, instead, always uses the maximum budget to find a minimal solution and continues in the next iteration of the fixpoint algorithm with the lowest solution found in the previous iteration. A fixpoint is reached if no further reduction in the number of states can be found. In this case, the algorithm returns the minimal found automaton A_{min} and a Boolean

Algorithm 7 Minimal automaton learning *fixpoint*(S, *strategy*)

Input: Trials budget $\#runs$, Training dataset $S = \{(\mathbf{i}_1, \mathbf{o}_1), \ldots, (\mathbf{o}_m, \mathbf{o}_m)\}$, Minimization strategy $\in \{$"`bestEffort`", "`exhaustive`"$\}$

Output: Pair of best learned automaton $A_{\min}$, and Boolean variable $approved_{\min}$ indicating whether a fixpoint has been reached

```
1:  A_min ← PTA(S)
2:  repeat
3:      states_min ← states_n(A_min)
4:      approved_min ← ⊥
5:      for i ← 1 to #runs do
6:          M ← RNN(states_min)
7:          converged, M ← train(M, S)
8:          if converged then
9:              A ← extract(M, S)
10:             A ← minimize(A)
11:             if states_n(A) < states_n(A_min) then
12:                 A_min ← A
13:             else
14:                 approved_min ← ⊤
15:             end if
16:             if strategy = "bestEffort" then
17:                 break
18:             end if
19:         end if
20:     end for
21: until states_min = states_n(A_min)
22: return (A_min, approved_min)
```

variable $approved_{\min}$ that indicates if the minimal solution can be reproduced, i.e., if the fixpoint has been reached.

Algorithm 7 starts by determining an initial upper bound. To determine the initial upper bound, we use the approach that we introduced in Section 7.2.2. Based on the given sample, we build a PTA that serves as the initial minimal automaton $A_{\min}$ from which we derive the upper bound. We then search for the minimal automaton in an iterative manner from Line 3 to Line 20. Note that in general any upper bound greater or equal to the minimum number of states of the ground truth Mealy machine can be set at this point.

In Line 3, the fixpoint iteration starts by setting the upper bound based on the minimal automaton found so far. The automaton $A_{\min}$ represents a Mealy machine and can also be written as $A_{\min} = \langle Q, \mathtt{q}_0, I, O, \delta, \lambda \rangle$. The function $states_n$ returns the number of states of the provided automaton, which is equal to $|Q|$. We start a finite number of iterations, where we try to approve the fixpoint or search for an automaton representation with fewer states. In each iteration, we train an RNN model using the given sample. For this purpose, we then initialize the RNN model based on the current upper bound $states_{\min}$ in Line 6. Afterwards, we start the training as explained in Algorithm 5. The training algorithm returns the trained model M and a Boolean variable *converged* that indicates if M achieved 100% accuracy in predicting the outputs of the given sample. If this is the case, we call Algorithm 6 to extract the Mealy machine A in Line 9.

In Line 10, we minimize the learned model A. Our rationale for this step is that our proposed RNN-based learning technique does not provide guarantees for learning a minimal model, as is the case with algorithms such as L^*. Therefore, it is possible that with an assumed upper bound

Table 7.1: Overview on the investigated Tomita grammars.

Grammar	Description	# States
Tomita 1	strings of the form 1^*	2
Tomita 2	strings of the form $(1\ 0)^*$	3
Tomita 3	strings that do not include an odd number of consecutive 0 symbols following an odd number of consecutive 1 symbols	5
Tomita 4	strings without more than 2 consecutive 0 symbols	4
Tomita 5	strings with an even number of 0 and even number 1 symbols	4
Tomita 6	strings where the difference between the numbers of 0s and 1s is divisible by three	3
Tomita 7	strings of the form $0^*1^*0^*1^*$	5

of states larger than the size of the minimal model, the RNN model may use a larger state space to model the behavior, while the learned model is still accurate. To minimize these models, we apply a minimization algorithm for Mealy machines as explained in Section 7.2.2. The costs for the minimization procedure are negligible, especially compared to the *NP*-complete problem of inferring a behavioral model with n states from a given sample.

Line 11 then compares if the minimized model A has fewer states than the so far found most minimal solution A_{min}. If this is the case, A replaces A_{min} in Line 12. Otherwise, if A does not represent an even more minimal solution, we approve that the fixpoint is reached. In the case of the best effort strategy, we terminate in both cases the current set of runs, whereas the exhaustive continues the current iteration with the upper bound $states_{\text{min}}$ to search for a solution with fewer states. The algorithm starts a new set of runs in case we found in the previous iteration a Mealy machine with fewer states. Otherwise, the algorithm terminates if no further minimization has been performed.

7.4 Case Studies

The following section presents case studies that evaluate our RNN-based learning technique. For our case study, we consider theoretical and practical examples. For our evaluation of theoretical examples, we consider a set of regular languages that have frequently been used for benchmarking RNN-related learning techniques [53, 130, 136, 190]. For the evaluation of practical applicability, we provide the approach with traces monitored on a subset of the investigated BLE devices from the BLE case study presented in Chapter 4. In these case studies, we first evaluate whether our approach functions in principle by evaluating it under perfect conditions. Perfect conditions mean that the minimal number of states is known and that a sample covers all behavioral aspects of the SUL. Hence, the sample fulfills similar properties as the one that we used to evaluate the L^* algorithm in Chapter 6. We slightly relax this assumption in another evaluation, where we provide a random sample but still under the assumption that the minimum state number is given. In the second part of our evaluation, we then consider the more realistic assumption that the minimal number of states is unknown. For this purpose, we evaluate our iterative approach to learn minimal automata on a comprehensive and a random sample.

7.4.1 Evaluation

We evaluate our approach on the Tomita grammars [178] and on three BLE devices.

Tomita Grammars. Table 7.1 provides an overview of the considered Tomita grammars. The benchmark set includes seven different regular languages. The languages are defined on a binary alphabet that only considers the characters '*0*' and '*1*'. The benchmark contains examples that are frequently mentioned in the literature such as Tomita 5 which is equal to the running example in Angluin's article explaining the L^* algorithm [17]. The Tomita grammars

are commonly represented as DFAs, but we consider for our RNN architecture Mealy machine equivalents. The outputs in these equivalents define whether a string is part of the language, indicated by '*true*' or '*false*'. Tomita grammars are well known in the literature [53, 130, 136, 190] to evaluate RNN-related automata learning approaches.

Bluetooth Low Energy (BLE). As a second case study subject, we consider a subset of the BLE devices that we investigated in Chapter 4. The subset includes the following three system on the chips (SoCs) of the six devices presented in Table 4.1:

- CC2650 (5 states),
- CYBLE-416045-02 (3 states), and
- nRF52832 (5 states).

We select these three SoCs since they were learnable considering the whole input alphabet. All the BLE devices run the same application as stated in Table 4.1. For our experimental evaluation, we extended the input alphabet by one input. Hence, the input alphabet includes ten different inputs. We consider as additional input the termination indication, denoted by termination_req. The additional input does not increase the number of states of the ground truth models and behaves equally to the scan request. We add this input since we use it in active automata learning to reset the device. By adding this input, we can reuse traces generated by our active automata learning setup.

7.4.2 Sample Generation

Active automata learning (AAL) sample. To provide a sample that is considered to be complete in the sense that it covers all behavioral aspects of the SUL, we collected all the queries performed during active learning. Hence, the sample is considered to be complete in such a way that an active learning algorithm could learn the behavioral model from the set of traces. For active learning, we applied the L^* algorithm with the improvements proposed by Rivest and Schapire. The sample also includes the traces performed by the equivalence oracle in order to test conformance between the SUL and the provided hypothesis. The used equivalence oracle provides state coverage for the tested hypothesis in combination with randomized inputs. Thus, the sample is equal to the *active learning sample* described in Section 6.2.2. As presented in Chapter 6 the set of performed queries by L^* is not minimal and might include redundant behavioral information. However, we assume that a larger sample with possible redundant information is beneficial for our evaluation. To actively learn the behavioral models, we used the learning library AALpy [129]. We refer to the sample generated by active automata learning as AAL sample.

We generate the traces for the BLE experiments by actively learning the real BLE devices. As previously explained, we consider this time an input alphabet that is extended by the termination request. Thus, the alphabet includes ten different inputs. For this purpose, we reused the learning setup as proposed in Chapter 4. Our sample contains all the traces that are executed on the SUL. Since we collected traces from the real device, we might observe non-deterministic behavior. For this purpose, we post-process the sample in order to remove traces that show non-deterministic behavior.

Random sample. For the generation of the random sample, we generate n_{data} random walks through the SULs starting at the initial state. The random sample generation is similar to the generation procedure defined in Algorithm 3 with the exception that we assume that our random sample is a set of traces instead of a list, since we aim to avoid redundant traces. Hence, we only add a randomly generated trace to the sample if it is unique compared to all other traces

Table 7.2: Learning results on Tomita grammars, where the minimal number of states is given. In all cases, we learned the correct automaton.

		Sample Size	Trace Length (avg) {std}	RNN af	$\#hl$	C	$\#e$	t (sec)
Tomita 1	AAL	41	8.4 {4.5}	ReLU	1	0.001	5	3
	Random	10	6.2 {3.4}	ReLU	1	0.001	9	1
Tomita 2	AAL	73	9.5 {5.3}	tanh	1	0.001	12	14
	Random	100	6.7 {2.3}	tanh	1	0.001	4	5
Tomita 3	AAL	111	10.4 {4.9}	ReLU	2	0.001	79	233
	Random	500	7.8 {1.9}	ReLU	2	0.001	28	305
Tomita 4	AAL	83	10.4 {5.2}	tanh	1	0.001	34	53
	Random	50	5.8 {2.5}	tanh	1	0.001	11	5
Tomita 5	AAL	91	9.3 {4.7}	ReLU	1	0.001	16	26
	Random	50	6.2 {2.8}	ReLU	1	0.001	24	15
Tomita 6	AAL	68	8.6 {4.6}	ReLU	1	0.001	5	5
	Random	20	5.5 {2.5}	ReLU	2	0.001	12	3
Tomita 7	AAL	115	10.4 {5.1}	ReLU	1	0.001	21	44
	Random	50	6.5 {2.3}	tanh	1	0.001	31	21

in the sample. For the random walks on the BLE devices, we aim to simulate a complete BLE session. Hence, each randomly generated trace includes as a last input a termination request, indicating the end of a BLE session.

7.4.3 Learning with Given States

This section presents the results of the case study, where we assume that the number of minimal states is known. Hence, we can initialize the RNN architecture in such a way that the dimension of the state vector is equal to the size of the minimal automaton representation.

All experiments are performed on a Dell Latitude 5501 with an NVIDIA GeForce MX150 and an Intel Hexa-Core I7-9850H operating at 2.60 GHz with 32 GB memory running Windows 10. Throughout all experiments, we set the number of neurons in a single hidden layer to 256, and the learning rate of the Adam optimizer to 0.001.

Table 7.2 presents the results on learning the Tomita grammars with our RNN-based learning technique given the AAL and random sample. We present our results based on the sample size and the average trace length with the standard deviation (std). In addition, we provide the parameters on the used RNN-architecture, i.e., the applied activation function (af), the number of hidden layers ($\#hl$), and the regularization factor C. We present also performance results by providing the number of required epochs (#e), and the required runtime (t) in seconds (sec).

The results are promising since we managed for all Tomita grammars to learn the correct Mealy machine. This was possible with the AAL sample and with the random sample. We checked the correctness of the learned models with a bisimilarity check to the ground truth automaton. That the learned Mealy machines are minimal is given by the constraint that the state vector admits only a one-hot-encoding for the minimal number of states. We hyper-tuned the parameters of the learning framework to learn the correct automaton in as few episodes and in as short a time as possible. We managed to learn a correct solution within 4 to 79 epochs. The shortest learning attempt took only one second, whereas the longest required a bit more than 5 minutes. To our surprise, we were able to learn correctly in most cases with fewer and shorter random samples than the AAL sample. Exceptions to this observation are Tomita 2 and Tomita 3. Most of the time we used the same RNN parameter setup for one grammar independent of the sample generation method.

Table 7.3 provides the same overview of the result for learning the BLE devices. The structure and notation of the table are equal to Table 7.2, except that the runtime is stated in minutes (min) and seconds (sec). For conducting the BLE experiments, we used the same RNN setup for

Table 7.3: Learning results on BLE experiments, where the minimal number of states is given.

		Sample Size	Trace Length (avg) {std}	RNN af	#hl	C	#e	t (min:sec)
CYBLE-416045-02	AAL	272	4.0 {1.0}	ReLU	1	0.01	5	0:12
	Random	100	11.5 {8.6}	ReLU	1	0.01	29	0:59
CC2650	AAL	473	5.1 {2.1}	ReLU	1	0.001	162	13:30
	Random	1000	11.8 {7.8}	ReLU	1	0.001	42	14:48
nRF52832	AAL	447	4.6 {1.1}	ReLU	1	0.001	21	1:30
	Random	1500	12.1 {8.1}	ReLU	1	0.001	21	11:18

each experiment. For training the RNN, however, we consider different values for the constant C that allows us to steer the influence on the used regularization term in the applied loss function. In contrast to the learning setup for Tomita grammars, we considered the fact that the BLE traces always describe a completed BLE session, where the last input resets the connections to the initial state. Hence, we extended the loss function to consider the cross entropy between the prediction of the last state and the initial state. Again, our proposed learning algorithm manages for all BLE devices to learn a Mealy machine that is isomorph to the ground truth. However, the BLE learning results present a different outcome on the random sample. In all cases, the traces in the random sample are on average longer than the AAL samples. Furthermore, in two out of three cases, we required more random samples to learn correctly. This correlates with our evaluation comparing active and passive techniques presented in Chapter 6.

We assume that the differences between the Tomita and BLE results relate to the complexity of the BLE example. Even though the number of states for both case studies is within the same range, we observe a large difference in the size of the input and output alphabet. Hence, we require more random samples to cover the behavior adequately.

7.4.4 Learning Minimal Automata

Our previous results show that our RNN framework is capable of learning the correct Mealy machine. However, we also examine whether this prerequisite of knowing the minimum number of states can be neglected. For this purpose, we evaluate our presented iterative algorithm for learning minimal automata. The evaluation again considers as case study subjects the seven Tomita grammars and the three BLE devices. Again, our evaluation compares the results of the AAL sample and the random sample.

Our achieved results depend on two types of randomness: the randomness in generating the samples and the randomness in training the RNN model. Hence, we repeat each experiment ten times and present the average values. The parameter setup for each experiment is equal to the one used for the corresponding experiment in Section 7.4.3. Due to a large number of experiments, we ran the following experiments on a scientific cluster based on Intel® Xeon® Gold 6230R CPU at 2.10 GHz and Ubuntu 20.04.

In the evaluation of our minimization algorithm, we observe four different outcomes:

1. Minimal automaton learned and fixpoint approved
2. Training does not converge within the given budget
3. Learned automaton is not minimal
4. Incorrect automaton due to missing data

Minimal automaton learned and fixpoint approved (Outcome 1). In this case, we learn the minimal automaton and the achievement of the fixpoint can be approved. Note that this takes in our setup at least two iterations in our minimization algorithm. In the first iteration, we minimize the number of states and in the second iteration, we approve the fixpoint.

Training does not converge within the given budget (Outcome 2). The training algorithm does not converge, which means that the RNN model cannot be trained within the given budget to predict all outputs correctly.

Learned automaton is not minimal (Outcome 3). The learned automaton is not minimal and cannot be further minimized. In this case, the RNN learned a non-conforming solution, but the sample does not include any long enough counterexample that reveals the wrong behavior. Thus, the learned model does not generalize the actual behavior well enough.

Incorrect automaton due to missing data (Outcome 4). The randomly generated sample misses behavior. In this case, the learned model conforms to the sample, but the sample is not sufficient to model the correct minimal automaton.

Table 7.4 and Table 7.5 present the results of our evaluation for learning minimal automata with our RNN-based learning technique. We present the results using the following metrics, where *std* always denotes the standard deviation of the averaged values:

- *#e (strategy).* Given budget based on the number of epochs (#e) and the strategy applied, where "*be*" denotes the best effort and "*ex*" the exhaustive strategy.
- *Learned (%).* Percentage of learning attempts where an automaton could be learned within the given budget.
- *Learned min. (%).* Percentage of the learned automata that are minimal and correct.
- *Learned wrong. (%).* Percentage of the learned automata that are not minimal and cannot be minimized.
- *Sample incompl. (%).* Percentage of the learned automata that cannot be correctly learned since the sample misses behavior.
- *Initial state bound (avg)* {*std*}. Average of the initial sizes of the state vectors in the first iteration based on the PTA generated from the provided sample.
- *#Iteration (avg)* {*std*}. Average of iterations performed to reach the fixpoint.
- *#Iteration (avg)* {*%*}. Average of the maximum number of reduced states via minimizing the learned automaton, and the percentage of applied state reductions.
- *#Time [min:sec] (avg)* {*std*}. Average runtime of the learning algorithm.

Table 7.4 provides the results of the Tomita experiments. The results show that when the training converges and we were able to extract an automaton, we always learned the correct minimal automaton using the best-effort strategy. The results also indicate that we always learned the minimal solution within the minimum number of iterations, except for Tomita 1, where learning on the random sample required three iterations twice. In the majority of the cases, we could reduce the number of iterations by minimizing the extracted automaton from the RNN. We observe that the RNN could significantly reduce the number of states within the first iteration, where the learned automaton is correct, but most of the time not minimal. By minimizing the learned automata, we could commonly approve the fixpoint in the next iteration.

For all Tomita experiments, where we did not achieve 100% correctness, since training did not converge within the given budget epochs (Outcome 2). Given the budget of 100 epochs, our training performed worse for Tomita 5 and Tomita 7 on the AAL sample. Hence, we repeated our approach with a budget of 200 epochs, which was most of the time sufficient to learn the correct minimal iterations within two iterations. Learning took between 4 seconds and approximately 43.5 minutes.

Table 7.4: Learning results on Tomita grammars, where the minimal number of states is unknown.

Use Case		**#e** (strategy)	**Learned** (%)	**Learned min.** (%)	**Learned wrong** (%)	**Sample incompl.** (%)	**Initial state bound** (avg) {std}	**#Iteration** (avg) {std}	**#States reduced** (avg) {%}		**Time** [min:sec] (avg) {std}	
Tomita 1	AAL	100 (be)	100	100	0	0	207 {0}	2 {0}	1	{90}	36	{33}
	Random	100 (be)	100	100	0	0	44 {9}	2.2 {0.4}	1	{60}	4	{2}
Tomita 2	AAL	100 (be)	100	100	0	0	390 {0}	2 {0}	1	{100}	2:18	{1:22}
	Random	100 (be)	100	100	0	0	227 {17}	2 {0}	1	{90}	45	{43}
Tomita 3	AAL	100 (be)	90	100	0	0	545 {0}	2 {0}	1.43	{78}	43:28	{23:15}
	Random	100 (be)	100	100	0	0	740 {11}	2 {0}	3	{90}	43:01	{37:01}
Tomita 4	AAL	100 (be)	80	100	0	0	459 {0}	2 {0}	-	{0}	20:33	{31:59}
	Random	100 (be)	80	100	0	0	135 {7}	2 {0}	1.14	{88}	5:27	{3:31}
Tomita 5	AAL	100 (be)	10	100	0	0	436 {0}	2 {0}	-	{0}	3:42	{0}
	Random	100 (be)	100	100	0	0	135 {7}	2 {0}	1	{40}	4:42	{3:29}
	AAL	200 (be)	80	100	0	0	436 {0}	2 {0}	1.88	{100}	37:22	{39:28}
Tomita 6	AAL	100 (be)	100	100	0	0	308 {0}	2 {0}	2.4	{100}	2:03	{1:50}
	Random	100 (be)	100	100	0	0	70 {8}	2 {0}	1.6	{50}	24	{20}
Tomita 7	AAL	100 (be)	60	100	0	0	564 {0}	2 {0}	-	{0}	3:35	{12:30}
	Random	100 (be)	80	100	0	0	135 {7}	2 {0}	1	{40}	3:51	{2:28}
	AAL	200 (be)	100	100	0	0	564 {0}	2 {0}	1	{40}	31:12	{18:29}

Table 7.5: Learning results on BLE devices, where the minimal number of states is unknown.

Use Case		**#e (strategy)**	**Learned** (%)	**Learned min.** (%)	**Learned wrong** (%)	**Sample impcompl.** (%)	**Initial state bound** (avg) {std}		**#Iteration** (avg) {std}	**#States reduced** (avg) {%}		**Time** [min:sec] (avg) {std}	
CYBLE-416045-02	AAL	200 (be)	100	100	0	0	547	{0}	2.4 {0.5}	1	{100}	1:30	{31}
	Random	200 (be)	90	67	0	33	946	{57}	2.6 {0.7}	1	{67}	24:30	{36:35}
CC2650	AAL	200 (be)	100	10	90	0	1180	{0}	3.3 {0.7}	3	{10}	57:08	{1:06:56}
	Random	200 (be)	90	78	11	11	9564	{161}	2.6 {0.7}	4	{78}	13:02:51	{11:22:21}
	AAL	200 (ex)	80	87.5	12.5	0	1180	{0}	2.8 {0.4}	4.4	{87.5}	3:40:51	{42:21}
nRF52832	AAL	200 (be)	100	100	0	0	896	{0}	2.7 {0.5}	1.67	{100}	6:09	{4.02}
	Random	200 (be)	100	100	0	0	14109	{400}	2.1 {0.3}	1.67	{100}	3:58:09	{2:37:52}

Table 7.5 presents the results of learning the minimal automaton of the BLE devices. Similar to the results, where the minimal number of states is given, we observe more difficulties in learning these case study subjects. Even though we could learn in most cases an automaton, the automaton was sometimes different from the correct minimal automaton. This problem was commonly observable when learning the CC2650 with the AAL sample. For this example, we learned an automaton that was not minimal and not be further minimized (Outcome 3). Thus, the learned automaton generalizes wrongly on the provided sample. We overcome this problem by switching to the exhaustive strategy, where we search for more solutions. With the exhaustive strategy, we were able to learn the correct minimal automaton of CC2650 in most cases.

We observe a second problem in the experiments that use a random sample. For these experiments, the generated random sample does not cover the entire behavior of the SUL in all cases. In this case, we learned an automaton that conforms to the provided sample but not to the ground truth automaton (Outcome 4). We also observe that learning took in the worst case significantly longer requiring more than 13 hours for learning. However, we also observe fast learning on the CYBLE-416045-02 where the average runtime was 1.5 minutes.

Both tables show that within a small number of iterations, the minimal solution could be learned. We also see that our RNN architecture does not hinder an immediate reduction of the state space. Thus, we could learn an approximate minimal solution even if the state vector is large.

7.5 Conclusion

This chapter evaluated the applicability of machine learning techniques for mining finite state machines. We approached the problem of learning a Mealy machine with at most k states from a given sample. For this purpose, we presented a passive learning technique that utilizes an RNN architecture to learn behavioral models of reactive systems. Our proposed RNN model not only predicts the output behavior but also the state behavior as similar to state transition functions in Mealy machines. We trained the model using a special regularization term that forced the state prediction towards a single state. From the trained RNN model we then simply extracted a Mealy machine by replaying the provided sample. In addition, we presented an extension of this RNN-based approach to learn minimal automata.

The results of our case study showed promising results, especially when the number of states is known. In our case studies, we evaluated our approach on a benchmark with theoretical examples and on samples of BLE traffic from three different devices. We were able to learn the correct automata for all the case studies we evaluated. We could learn the minimal automata for the theoretical examples, even when the number of states is initially unknown. Learning the BLE devices without the given number of states was more challenging. We suspect that the cause of the observed problems is the larger size of the models with respect to the larger input and output alphabet. However, we proposed an exhaustive learning strategy that allows the generation of the correct model. In practice, we recommend following this exhaustive strategy when resources are available and the input and output alphabets are larger. However, this also requires multiple iterations of training the RNN model, which can be time-consuming.

(RQ 2.1) Does passive learning represent an alternative to active learning?

Our novel RNN-based learning technique provided promising results for learning behavioral models based on a given sample. If the number of states is known, we managed to learn all investigated case study subjects. The results also showed that a small random sample was sufficient to learn the correct behavioral model, especially for SULs with a small input and output alphabet. However, for systems with a larger alphabet, more random samples were required. We also proposed strategies to solve the problem that the number of states had to be known. We could find a setup for all examples where the majority of learned models represented minimal automata that were isomorphic to ground truth automata when the final number of states was unknown.

Chapter 8

Learning Abstracted Non-Deterministic Finite State Machines

Declaration of Resources

This chapter is based on the paper *"Learning Abstracted Non-deterministic Finite State Machines"* [146] presented at *ICTSS 2020*. The proposed optimizations were implemented in the learning library AALpy [129] by Konstantin Windisch as part of his Bachelor's Thesis [193], which was co-supervised by the author of this thesis.

8.1 Introduction

The following chapter provides a learning algorithm for non-deterministic systems. The presented algorithm approaches two problems that hamper the application of automata learning in practice: (1) large input and output alphabets, (2) non-deterministic observations due to timed or stochastic behavior.

In the previously presented case studies on learning communication protocols, we observed both problems. Usually, communication protocols consider a large input and output alphabet as the packets can contain arbitrary character sequences. We overcome the problem of a large input and output alphabet by learning with abstracted input and output symbols instead. To the best of our knowledge, Cho et al. [38] were the first that applied such an abstraction to learn network protocols. However, finding such an alphabet abstraction is not always straightforward since a too-coarse abstraction could lead to non-deterministic behavior.

To overcome the second problem, we developed in the previous case studies domain-specific learning setups to filter out messages or to adapt response timeouts. However, some systems still behaved non-deterministically as shown in the BLE case study presented in Chapter 4.

In the following, we present a learning algorithm that creates an observable non-deterministic finite state machine (ONFSM) of a black-box system. To make the learning of large systems feasible, we introduce a two-layer-based abstraction technique. Our learning algorithm follows Angluin's L^*-based learning approach [17], but extends the MAT framework by two abstraction layers.

We evaluate our presented learning algorithm for learning the behavioral models of five different MQTT brokers. Tappler et al. [170] show that learning behavioral models of MQTT brokers is feasible. However, they limit their case study to learning setups where one or two clients interact with the MQTT broker. We show that our learning technique enables learning in a multi-client setup within a feasible amount of time. We compare our technique to a setup

that follows an approach similar to that presented by Tappler et al. [170]. The results show that our method creates a concise generalization within a reasonable amount of time, especially when we compare our method to the runtime required by a setup similar to Tappler et al. [170].

The chapter is organized as follows. First, we provide a formal definition of ONFSMs and slightly adapt the definition of observation tables. In Section 8.3, we then introduce our learning framework, the applied abstraction concepts, and introduce our learning algorithm. Section 8.5 presents the case study on learning five different MQTT brokers in a multi-client setup. Section 8.6 introduces briefly the integration of the presented learning algorithm into the learning library AALpy. We conclude this chapter in Section 8.7.

8.2 Background

8.2.1 Observable Non-deterministic Finite State Machine (ONFSM)

ONFSMs are a special type of non-deterministic input/output automata, where the state transition function defines exactly one state for a triplet of a given state, an input, and an output.

Definition 6 (Observable Non-deterministic Finite State Machine) *We define an ONFSM $\mathcal{M}$ as a five-tuple $\langle Q, q_0, I, O, \delta \rangle$, where*

- *Q is the finite set of states,*
- *q_0 is the initial state,*
- *I is the finite set of inputs,*
- *O is the finite set of outputs, and*
- *$\delta : Q \times I \times O \to Q$ is the state transition function.*

We define ONFSM as input enabled, where every input $i \in I$ is defined for every state $q \in Q$ at least once. We call a non-deterministic finite state machine *observable* if a triplet $(q, i, o) \in Q \times I \times O$ defines exactly one target state $q' \in Q$ in the state transition function δ. Note that a pair $(q, i) \in Q \times I$ can be defined multiple times for a state q and based on the output $o \in O$ it is observable which transition is chosen. The definitions of input and output sequences and traces correspond to those for Mealy machines in Section 2.1.

8.2.2 Observation Table for ONFSMs

To learn ONFSMs, we adapt Definition 2 that defines observation tables.

Definition 7 (Observable Table for ONFSMs) *We define an observation table $\mathcal{T}$ as triplet $\langle \Gamma, E, T \rangle$, with*

- *the prefix-closed set $\Gamma \subseteq (I \times O)^*$,*
- *the suffix-closed set $E \subseteq I^+$, and*
- *the mapping $T \colon \Gamma \times E \to 2^{O^+}$.*

Similar to Definition 2, we divide Γ into two distinct subsets Γ_S and Γ_P, with $\Gamma = \Gamma_S \cup \Gamma_P$ and $\Gamma_S \cap \Gamma_P = \emptyset$. For our learning algorithm on ONFSMs, we define $\Gamma_S \in (I \times O)^*$. In difference to Definition 2, we divide also Γ_P in two sets with $\Gamma_P = \Gamma_{P_S} \cup \Gamma_{P'}$, where $\Gamma_{P_S} = \Gamma_S \cdot (I \times O)$, $\Gamma_{P'} = \Gamma \cdot (I \times O)$ and $\Gamma_{P_S} \cap \Gamma_{P'} = \emptyset$. We write $\gamma \cong \gamma'$ if two rows $\gamma, \gamma' \in \Gamma$ are equal in $\mathcal{T}$, where γ, γ' are equal, iff $\forall e \in E \colon\ T(\gamma, e) = T(\gamma', e)$ holds. Another distinction to Definition 2 is the mapping T which maps values of Γ and E to sets of output sequences.

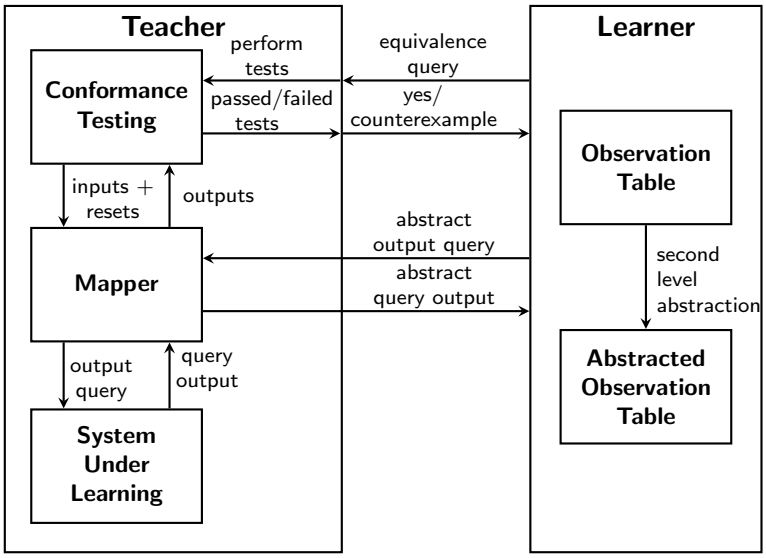


Figure 8.1: Modified MAT framework for learning abstracted ONFSMs.

8.3 Method

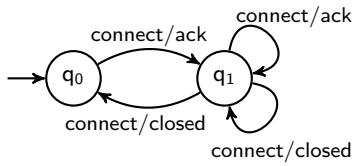
In the following, we introduce our learning algorithm for ONFSMs. Our learning algorithm follows an L^*-based learning approach to learn abstractions of non-deterministic systems. First, we provide an overview of the modifications of the MAT framework, followed by a more detailed description of the modifications. Afterwards, we describe the procedure of our learning algorithm.

8.4 Learning Framwork

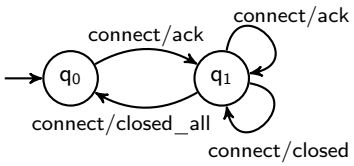
Figure 8.1 illustrates our modified MAT framwork. Following the classical MAT framework structure, we consider two members: a teacher and a learner. The learner queries the teacher to learn a behavioral model of the SUL. We refer to Section 2.2.2 for further details on learning reactive systems with the MAT framework.

We extend the MAT by two levels of abstraction: the *first-level abstraction* and the *second-level abstraction*. The first-level abstraction refers to the abstraction of the concrete input and output alphabet. This kind of alphabet abstraction was first proposed by Cho et al. [38]. Aarts et al. [7] formalize this abstraction technique and extend the MAT framework by a mapper component. The mapper component translates abstract inputs into concrete inputs and concrete outputs into abstract outputs. Using the abstracted input and output alphabet, the learner then learns a behavioral model on a more abstract level.

On top of the first-level abstraction, we propose a second-level abstraction. The learner only considers the abstracted input and output alphabet. Thus, the observation table contains abstract inputs and outputs. Our second-level abstraction introduces a second observation table that represents an abstraction of the other observation table. In the remainder of this chapter, we refer to the observation table of the second-level abstraction as *abstracted observation table*, whereas the observation table of the first-level abstraction is addressed as *standard observation table*. Based on the abstracted observation table the learner then explores the state space and performs output queries in order to create an abstracted hypothesis.



(a) Non-deterministic finite state machine modeling an abstraction of a connection procedure. The model is not observable non-deterministic.



(b) Non-deterministic finite state machine, where the output alphabet is refined so that the state transitions are observable non-deterministic.

Figure 8.2: Abstracted representation of multi-client connection procedure based on the protocol presented in Figure 2.1.

8.4.1 First-Level Abstraction

We apply the first-level abstraction to reduce the input and output alphabet. For this purpose, we follow the approach discussed in Section 2.2.2. In previous chapters, we showed that the abstraction of the concrete inputs and outputs made learning communication protocols such as BLE feasible.

For learning, we consider an abstract input alphabet $I^{\mathcal{A}}$ and an abstract output alphabet $O^{\mathcal{A}}$. The mapper translates abstract inputs from the learner to concrete inputs of the input alphabet I. It receives concrete outputs of the output alphabet O from the SUL and forwards the abstracted outputs to the learner. Consequently, the learner maintains the observation table based on the abstracted input and output alphabet. In Definition 7, we replace I by $I^{\mathcal{A}}$ and O by $O^{\mathcal{A}}$.

Finding an appropriate abstraction for a concrete input and output alphabet is not straightforward. Aarts et al. [7] define a mapper as well-designed if the learner could learn a deterministic Mealy machine. Hence, a too-coarse abstraction by the mapper component can cause non-deterministic observations. To overcome these non-deterministic observations, the mapper has to be refined. A different approach is to learn non-deterministic models. The literature [56, 97, 140] proposes different learning algorithms for ONFSMs. However, the mapper design must still ensure that an observable non-deterministic model can be learned.

Example 12 (First-Level Abstraction for Multi-client Connection Protocol)
Chapter 2 introduces a publish/subscribe protocol in Example 1. For now, we only consider the connection procedure of this protocol, but in a multi-client setup. The concrete input alphabet is $I = \{\mathsf{connect}(id) | id \in \mathbb{N}\}$*, where id is a unique identifier for each client. Similar to the input alphabet, we consider the concrete output alphabet* $O = \{\mathsf{ack}(id), \mathsf{closed}(id) | id \in \mathbb{N}\}$*. The protocol defines that every client receives an acknowledge message (*ack*) when the client connects. If an already connected client connects again the connection is terminated and the client receives the output* closed*.*

Figure 8.2a presents an abstracted version of the connection protocol, where the abstract input alphabet is $I^{\mathcal{A}} = \{\mathsf{connect}\}$ *and abstract output alphabet is* $O^{\mathcal{A}} = \{\mathsf{ack}, \mathsf{closed}\}$*. The initial state* q_0 *indicates that no client is connected, hence we only observe* ack *as output. At least one client is connected in the state* q_1*. Thus, we either observe* ack *if another client connects or* closed *if an already connected client reconnects. If the last connected client reconnects we enter again state* q_0*. The non-deterministic finite state machine that is shown in Figure 8.2a is not observable non-deterministic since we cannot distinguish in state* q_1 *if an observed* closed *leads to state* q_0 *or if we remain in state* q_1*.*

Figure 8.2b shows an ONFSM that represents the same connection protocol. To provide observable non-determinism, we change the output of the transition from q_1 *to* q_0 *with the input* connect *to* closed_all*. To learn an ONFSM, we have to change the abstraction of the output alphabet. We change the abstract output alphabet to* $O^{\mathcal{A}} = \{\mathsf{ack}, \mathsf{closed}, \mathsf{closed\_all}\}$*, where the*

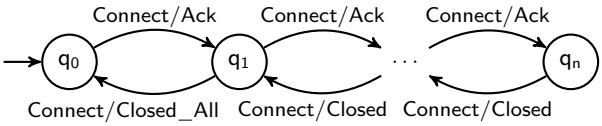


Figure 8.3: ONFSM of the multi-client connection protocol that would be generated by common ONFSM learning algorithms [56, 97, 140] using the proposed first-level abstraction of Example 12. The number of states of the learned model depends on the number of considered clients.

additional abstract output closed_all *indicates that all clients have disconnected. We implement such an abstraction via a stateful mapper that maintains an internal counter for the number of connected clients.*

8.4.2 Second-Level Abstraction

The second-level abstraction aims to further reduce the state space. Even if we learn non-deterministic systems and design the mapper such that we can learn an ONFSM, we might still observe the problem that the learned model does not generalize as expected. In particular, when the mapper performs abstractions based on a counting scheme, the counting behavior is directly reflected in the state space, since the observations depend on the value of the counter. Hence, common learning algorithms for ONFSMs from the literature [56, 97, 140] would learn ONFSMs that model the underlying counter.

Example 13 (Learning with First-Level Abstraction)
Common learning algorithms for ONFSMs [56, 97, 140] learn with the abstraction proposed in Example 12 an ONFSM as it is shown in Figure 8.3, where the number of states depends on the number of considered clients. Thus, q_1 *identifies the state where one client is connected. In state* q_n*, n clients would be connected.*

To generate a more general model, we add a second-level abstraction. The second-level abstraction aims to abstract outputs by adding another mapping to group abstract observations. We define the surjective function $\mu\colon O^{\mathcal{A}} \to O^{\mathcal{A}'}$, where $O^{\mathcal{A}'}$ is an abstraction of the abstract output alphabet $O^{\mathcal{A}}$. Let $\mu^*\colon O^{\mathcal{A}^+} \to O^{\mathcal{A}'^+}$ be an extension of μ for output sequences, which returns for an abstract output sequence $o_0 \cdot \ldots \cdot o_n \in O^{\mathcal{A}^+}$ the second-level abstracted output sequence $\mu(o_0) \cdot \ldots \cdot \mu(o_n) \in O^{\mathcal{A}'^+}$.

Example 14 (second-level abstraction for Multi-client Connection Protocol)
An abstraction function for the second-level abstraction of the multi-client connection protocol introduced in Example 12 can be defined as $\mu\colon \{\{\mathsf{ack}\} \to \mathsf{ack}, \{\mathsf{closed}, \mathsf{closed\_all}\} \to \mathsf{closed}\}$.

For our second-level abstraction, we extend now the learner to consider a second observation table, which we denote as *abstracted observation table*. We define the abstracted observation table as follows:

Definition 8 (Abstracted Observation Table) *We define the abstracted observation table* $\mathcal{T}^{\mathcal{A}'}$ *as a quadruplet* $\langle \Gamma, E, T^{\mathcal{A}'}, \mu \rangle$*, with*

- *the prefix-closed set* $\Gamma \subseteq (I^{\mathcal{A}} \times O^{\mathcal{A}})^*$,
- *the suffix-closed set* $E \subseteq I^{\mathcal{A}^+}$,
- *the abstraction mapping* $T^{\mathcal{A}'}\colon \Gamma \times E \to 2^{O^{\mathcal{A}'^+}}$, *and*

Table 8.1: Standard observation table of the multi-client connection protocol of Example 12.

	$\Gamma \backslash E$	connect
Γ_S	ϵ	ack
	(connect, ack)	ack, closed_all
Γ_P	(connect, ack)(connect, ack)	ack, closed
	(connect, ack)(connect, closed_all)	ack

Table 8.2: Abstracted observation table of Table 8.1.

	$\Gamma \backslash E$	connect
Γ_S	ϵ	ack
	(connect, ack)	ack, closed
Γ_P	(connect, ack)(connect, ack)	ack, closed
	(connect, ack)(connect, closed_all)	ack

- *the second level output abstraction function* $\mu \colon O^{\mathcal{A}} \to O^{\mathcal{A}'}$.

We write $\gamma \cong_{\mathcal{A}'} \gamma'$ *if two rows* $\gamma, \gamma' \in \Gamma$ *are equal in* $\mathcal{T}^{\mathcal{A}'}$*, where* γ, γ' *are equal, iff* $\forall e \in E \colon T^{\mathcal{A}'}(\gamma, e) = T^{\mathcal{A}'}(\gamma', e)$

Example 15 (Abstraction of Observation Table) *Table 8.1 shows an intermediate standard observation table that the learner generates on the first-level abstraction while learning the multi-client connection protocol, which we introduced in Example 12. The standard observation table contains in its cells sets of observable outputs. For example, for the row that is identified by* (connect, ack)*, we can observe for the input* connect *either the output* ack *or* closed_all*. Table 8.2 shows then the abstracted observation table of Table 8.1 using the abstraction function* μ *presented in Example 14. We see that the observations* closed *and* closed_all *are replaced by abstracted observation* closed*. Note that the outputs in the traces of* Γ *stay unchanged.*

Based on the abstracted observation table the learner then queries the SUL and constructs an abstracted ONFSM using the abstracted and standard observation table. Note that the transitions in the hypothesis still define inputs and outputs on the first-level abstracted alphabet. In the next section, we explain how these abstraction levels are used in the definition of a learning algorithm to learn abstracted ONFSMs.

8.4.3 Learning Algorithm

Algorithm 8 describes the learning procedure for learning abstracted ONFSMs. The learning algorithm describes an L^*-based learning approach as presented in Algorithm 1, but extends it by maintaining the abstracted observation table $\mathcal{T}^{\mathcal{A}'}$.

Similar to L^*, our learning algorithm only constructs a hypothesis if the observation table satisfies certain properties. We check the properties of closedness and consistency on the abstracted observation table. For our learning, we need an additional property called *completeness*, for which the standard observation table is additionally required. In the following, we define the three properties: closedness (1), consistency (2), and completeness (3).

Closedness (1). In principle, the closedness check is similar to the one we defined in Section 2.2.2. We define the abstracted table $\mathcal{T}^{\mathcal{A}'} = \langle \Gamma = \Gamma_S \cup \Gamma_P, E, T^{\mathcal{A}'}, \mu \rangle$ as closed if $\forall \gamma_P \in \Gamma_P$, $\exists \gamma_S \in \Gamma_S \colon \gamma_P \cong_{\mathcal{A}'} \gamma_S$. Note that we check closedness only on the abstracted observation table, and we do not require that the standard observation table is closed.

To make the abstracted observation table closed, we move every $\gamma \in \Gamma_P$ to Γ_S where for every $\gamma' \in \Gamma_S$ the equation $\gamma \ncong_{\mathcal{A}'} \gamma'$ holds. Furthermore, we extend for every γ that we moved to Γ_S the set Γ_P by $\gamma \cdot (i, o)$ for every pair $(i, o) \in (I^{\mathcal{A}} \times O^{\mathcal{A}})$. The pairs $(i, o) \in (I^{\mathcal{A}} \times O^{\mathcal{A}})$ are derived for every combination of input $i \in I^{\mathcal{A}}$ and $o \in T(\gamma, i)$.

Example 16 (Closedness of Abstracted Observation Table) *Table 8.3 shows the standard observation table that is initially created and filled when learning the connection protocol as presented in Example 14. Table 8.3 represents the corresponding abstracted observation table*

Table 8.3: Observation table of Example 12 after the first output queries.

	$\Gamma \backslash E$	connect
Γ_S	ϵ	ack
Γ_P	(connect, ack)	ack, closed_all

Table 8.4: Abstraction of Table 8.3 using the mapping defined in Example 12.

	$\Gamma \backslash E$	connect
Γ_S	ϵ	ack
Γ_P	(connect, ack)	ack, closed

which is abstracted with the abstraction function μ given in Example 14. Based on our definition of closedness, Table 8.3 is not closed since Γ_P identifies a row that is not part of the rows identified by Γ_S. The row identified by (connect, ack) *is not part of the rows defined by Γ_S. To make Table 8.3 closed, we need to move the row* (connect, ack) *to Γ_S and extend Γ_P by the rows* (connect, ack) $\cdot$ (connect, ack) *and* (connect, ack) $\cdot$ (connect, closed_all)*. Table 8.1 and Table 8.2 show the closed and filled version, where Table 8.2 is the abstracted version. Note that Table 8.1 would not be closed in other learning algorithms for ONFSMs, but for learning abstracted ONFSMs we are only interested in the closedness of the abstracted observation table (Table 8.2).*

Consistency (2). For the consistency check, we have to extend our definition of consistency to input/output sequences. We denote the abstracted observation table $\mathcal{T}^{\mathcal{A}'} = \langle \Gamma = \Gamma_S \cup \Gamma_P, E, T^{\mathcal{A}'}, \mu \rangle$ as consistent if for all $\gamma, \gamma' \in \Gamma$ where $\gamma \cong_{\mathcal{A}'} \gamma'$ holds, there exists no input/output pair $(i, o) \in (I^{\mathcal{A}} \times O^{\mathcal{A}})$ such that $\gamma \cdot (i, o) \ncong_{\mathcal{A}'} \gamma' \cdot (i, o)$.

We follow the idea by Niese [133] to make observation tables for learning Mealy machines consistent. By doing so, we extend E by an input sequence $e \cdot i \in I^{\mathcal{A}+}$, where $e \in E$ and $i \in I^{\mathcal{A}}$ given that for two traces $\gamma, \gamma' \in \Gamma$ and an output $o \in O^{\mathcal{A}}$ the equations $\gamma \cong_{\mathcal{A}'} \gamma'$, $\gamma \cdot (i, o) \ncong_{\mathcal{A}'} \gamma' \cdot (i, o)$ and $T^{\mathcal{A}'}(\gamma \cdot (i, o), e) \neq T^{\mathcal{A}'}(\gamma' \cdot (i, o), e)$ hold.

Completeness (3). The completeness check is different from other L^*-based learning algorithms and is required in order to define all transitions in the learned ONFSM. We build the state space of the hypothesis on the abstracted observation table. To define all transitions, we might need to extend the abstracted observation table to make the hypothesis complete in the number of transitions.

For the completeness definition, we require two auxiliary functions: $pre(s^I)$, and $trace(s^I, s^O)$. Let $pre(s^I)$ be a function that takes an input sequence $s^I \in I^*$ as input and returns all prefixes of the s^I including s^I itself. We define $trace(s^I, s^O)$ as a function that takes an input sequence $s^I \in I^*$ and an output sequence $s^O \in O^*$ of equal length as input and generates an alternating input/output sequence, i.e., a trace.

To assess whether the abstracted observation table is complete, we take a row identified by $\gamma \in \Gamma_S$ and select another row $\gamma' \in \Gamma$, where the rows are equal in the abstracted observation table, i.e., $\gamma \cong_{\mathcal{A}'} \gamma'$, but not in the standard observation table, i.e., $\gamma \ncong \gamma'$. We then identify the input sequences $e \in E$ such that $T(\gamma, e) \neq T(\gamma', e)$. Furthermore, we select two outputs sequences $s^{O^+}, s'^{O^+} \in O^{\mathcal{A}^+}$ where $s^{O^+} \in T(\gamma, e)$ and $s'^{O^+} \in T(\gamma', e)$ and $s^{O^+} \neq s'^{O^+}$ but $\mu^*(s^{O^+}) = \mu^*(s'^{O^+})$. The table is complete if $\forall t \in pre(\gamma' \cdot trace(e, s'^{O^+})) : t \in \Gamma$ holds. To make the table complete, we need to add the missing prefixes $pre(\gamma' \cdot trace(e, s'^{O^+}))$ to Γ_P.

Example 17 (Completeness of Abstracted Observation Table) *Table 8.2 shows an intermediate version of the abstracted observation table that is closed and consistent. As a last step, we check if the table is complete. The completeness check also considers the corresponding standard observation table (Table 8.1). First, we determine all equal rows in the abstracted observation table (Table 8.2) and then check if the rows are not equal in Table 8.1. The rows* (connect, ack) *and* (connect, ack) $\cdot$ (connect, ack) *are equal in Table 8.2, but they are not in Table 8.1. Hence, we have to take the entry in E that shows the difference in Table 8.1, which for*

Table 8.5: Final observation table of the connection protocol of Example 12. The input connect is abbreviated by conn.

	$\Gamma \backslash E$	connect
Γ_S	ϵ	ack
	(conn, ack)	ack, closed_all
Γ_P	(conn, ack)(conn, ack)	ack, closed
	(conn, ack)(conn, closed_all)	ack
	(conn, ack)(conn, ack)(conn, closed)	ack, closed_all

Table 8.6: Final abstracted observation table generated from Table 8.5 The input connect is abbreviated by conn.

	$\Gamma \backslash E$	connect
Γ_S	ϵ	ack
	(conn, ack)	ack, closed
Γ_P	(conn, ack)(conn, ack)	ack, closed
	(conn, ack)(conn, closed_all)	ack
	(conn, ack)(conn, ack)(conn, closed)	ack, closed

this example can only be connect. *As a next step, we check if all prefixes* $\gamma \cdot trace(i^+, o^+)$ *are included in* Γ*, where* γ *is one of the selected row indices and* i^+ *is the input sequence from the* E *set and* o^+ *is one output sequence in* $T(\gamma, i)$*. For this example, Table 8.2 is not complete since the prefix* (connect, ack) · (connect, ack) · (connect, closed) *is not included in* Γ*. To make the table complete, we add this prefix to* Γ_P*. Table 8.6 presents the complete abstracted observation table.*

Based on these definitions, we define our learning algorithm for abstracted non-deterministic ONFSMs. Algorithm 8 describes the learning procedure. The algorithm takes as input the abstract input alphabet $I^{\mathcal{A}}$, the abstraction function μ for the second-level abstraction, and a black-box access to the SUL to query it. The algorithm returns an abstracted ONFSM $\mathcal{M}$.

In Line 3, the algorithm starts by initializing the observation table, where $\Gamma_S = \epsilon$ and $E = I^{\mathcal{A}}$. From Line 3 to Line 28, we perform the iterative learning procedure, where the standard observation table is filled by performing output queries. We then apply the second-level abstraction function μ to generate the abstracted observation table. Based on the observation tables, we then create a hypothesis and check if it conforms to the SUL. This procedure is repeated until we find a conforming hypothesis.

In Line 4, we fill the standard observation table. For this, we fill all cells in the table, i.e. we define the mappings in T. Furthermore, we extend Γ_P such that all rows in Γ_S have in Γ_P an extension with every input/output combination. Line 5 creates the abstracted observation table from the standard observation table. The function $abstract\_table(\langle \Gamma, E, T \rangle, \mu)$ takes as input the observation $\langle \Gamma, E, T \rangle$ and the abstraction function μ to generate the abstracted observation table $\langle \Gamma, E, T^{\mathcal{A}'}, \mu \rangle$, where the first level abstracted outputs in T are replaced by second level abstracted outputs defined by μ in order to create the mapping $T^{\mathcal{A}'}$. Note that the observation tables always share the same sets Γ and E.

Starting at Line 6, we then check if the abstracted observation table is closed, complete and consistent. From Line 7 to Line 21, we then follow the same pattern for every property. We check if the respective property holds, i.e., if the table is closed (Line 7), complete (Line 12), and consistent (Line 17). If one of the properties does not hold we always follow the same pattern: (1) we fix the table such that the property is satisfied, (2) fill the observation table, and (3) generate the abstracted observation table. If this is done for every property, we return to Line 6 and check if all properties hold. This procedure is repeated until all properties hold.

In Line 23, we then construct the hypothesis $\mathcal{M}$ based on both observation tables. We describe the hypothesis construction for creating an abstracted ONFSM later in this section. Line 24 performs an equivalence query, which returns a pair $(verdict, cex)$, where *verdict* indicates if $\mathcal{M}$ conforms to the SUL, and *cex* contains an input sequence that reveals the behavioral difference between $\mathcal{M}$ and the SUL. The Boolean variable *verdict* evaluates to *true* if $\mathcal{M}$ conforms to the SUL, otherwise it evaluates to *false*. We will later explain how the equivalence check can be implemented for learning abstracted ONFSMs.

In the case that the *verdict* equals false, we update the table by the provided counterexample in Line 26. For the counterexample processing, we distinguish two different types of counterexamples. The first type reveals a missing transition for an input in a state in which an output of the same equivalence class already exists. In this case, we add all prefixes of the trace that leads

Algorithm 8 Learning algorithm using an abstracted observation table

Input: input alphabet $I^{\mathcal{A}}$, equivalence class mapping μ, black-box access to *sul*
Output: ONFSM $\mathcal{M}$

1: $\langle \Gamma, E, T \rangle \leftarrow init\_table(I^{\mathcal{A}})$
2: verdict $\leftarrow \bot$
3: **do**
4: $\quad \langle \Gamma, E, T \rangle \leftarrow fill\_table(\langle \Gamma, E, T \rangle, SUL)$
5: $\quad \mathcal{T}^{\mathcal{A}'} \leftarrow abstract\_table(\langle \Gamma, E, T \rangle, \mu)$
6: $\quad$ **while** $\neg(closed(\mathcal{T}^{\mathcal{A}'}) \wedge consistent(\mathcal{T}^{\mathcal{A}'}) \wedge complete(\mathcal{T}^{\mathcal{A}'}, \langle \Gamma, E, T \rangle))$ **do**
7: $\quad\quad$ **if** $\neg closed(\mathcal{T}^{\mathcal{A}'})$ **then**
8: $\quad\quad\quad \Gamma \leftarrow make\_closed(\mathcal{T}^{\mathcal{A}'})$
9: $\quad\quad\quad \langle \Gamma, E, T \rangle \leftarrow fill\_table(\langle \Gamma, E, T \rangle, SUL)$
10: $\quad\quad\quad \mathcal{T}^{\mathcal{A}'} \leftarrow abstract\_table(\langle \Gamma, E, T \rangle, \mu)$
11: $\quad\quad$ **end if**
12: $\quad\quad$ **if** $\neg complete(\mathcal{T}^{\mathcal{A}'}, \langle \Gamma, E, T \rangle)$ **then**
13: $\quad\quad\quad \Gamma \leftarrow complete\_table(\mathcal{T}^{\mathcal{A}'}, \langle \Gamma, E, T \rangle)$
14: $\quad\quad\quad \langle \Gamma, E, T \rangle \leftarrow fill\_table(\langle \Gamma, E, T \rangle, SUL)$
15: $\quad\quad\quad \mathcal{T}^{\mathcal{A}'} \leftarrow abstract\_table(\langle \Gamma, E, T \rangle, \mu)$
16: $\quad\quad$ **end if**
17: $\quad\quad$ **if** $\neg consistent(\mathcal{T}^{\mathcal{A}'})$ **then**
18: $\quad\quad\quad E \leftarrow make\_consistent(\mathcal{T}^{\mathcal{A}'})$
19: $\quad\quad\quad \langle \Gamma, E, T \rangle \leftarrow fill\_table(\langle \Gamma, E, T \rangle, SUL)$
20: $\quad\quad\quad \mathcal{T}^{\mathcal{A}'} \leftarrow abstract\_table(\langle \Gamma, E, T \rangle, \mu)$
21: $\quad\quad$ **end if**
22: $\quad$ **end while**
23: $\quad \mathcal{M} \leftarrow create\_hypothesis(\langle \Gamma, E, T \rangle, \mathcal{T}^{\mathcal{A}'}, I^{\mathcal{A}})$
24: $\quad verdict, cex \leftarrow equivalence\_query(\mathcal{M}, SUL)$
25: $\quad$ **if** $\neg verdict$ **then**
26: $\quad\quad \langle \Gamma, E, T \rangle \leftarrow update\_table(\langle \Gamma, E, T \rangle, cex)$
27: $\quad$ **end if**
28: **while** $\neg verdict$
29: **return** $\mathcal{M}$

to this state to the set Γ_P. If the provided counterexample does not belong to the first type, we need to add new states. In this case, we follow the approach of El-Fakih et al. [56] and add all suffixes of the provided counterexample to E. Based on the updated observation table, we start a new iteration to fill the observation tables and construct a new hypothesis. This procedure is repeated until a conforming ONFSM could be learned.

Hypothesis creation. Algorithm 9 describes the procedure to generate an abstracted ONFSM from the observation table $\mathcal{T}$ and the abstracted observation table $\mathcal{T}^{\mathcal{A}'}$. The algorithm returns the created ONFSM $\mathcal{M}$. The generation of $\mathcal{M}$ starts by the initialization of its states in Line 1 and Line 2. For keeping the construction simple, we identify the states via their access sequence provided in the set Γ_S. After the hypothesis is constructed, we can map these sequences to numeric state identifiers.

For the construction of the ONFSM, we traverse through all states in Γ_S. In Line 4, we select all rows $\Gamma_{\gamma S}$ that are in the abstracted observation table equal to the currently considered state. We then traverse through the rows $\gamma \in \Gamma_{\gamma S}$ and add for each input in the input alphabet $i \in I^{\mathcal{A}}$ all transitions for all outputs that are defined in the mapping $T(\gamma, i)$ of the observation table $\mathcal{T}$. In Line 8, we concat the current input/output pair $(i, o) \in (I^{\mathcal{A}} \times O^{\mathcal{A}})$ to the row γ in order to get

Algorithm 9 Creation of an ONFSM *create_hypothesis*$(\mathcal{T}, \mathcal{T}^{\mathcal{A}'}, I^{\mathcal{A}})$

Input: observation table $\mathcal{T} = \langle \Gamma = \Gamma_S \cup \Gamma_P, E, T \rangle$, abstracted observation table $\mathcal{T}^{\mathcal{A}'} = \langle \Gamma, E, T^{\mathcal{A}'}, \mu \rangle$, abstract input alphabet $I^{\mathcal{A}}$
Output: ONFSM $\mathcal{M} = \langle Q, q_0, I^{\mathcal{A}}, O^{\mathcal{A}}, \delta \rangle$

```
1:  q_0 ← ϵ
2:  Q ← Γ_S
3:  for all γ_S ∈ Γ_S do
4:      Q_γS ← {γ | γ ∈ Γ ∧ γ ≅_A' γ_S}
5:      for all γ ∈ Q_γS do
6:          for all i ∈ I^A do
7:              for all o ∈ T(γ, i) do
8:                  γ' ← γ · (i, o)
9:                  if γ' ∈ Γ then
10:                     δ(γ_S, i, o) ← γ'_S, where γ'_S ∈ Γ_S ∧ γ'_S ≅_A' γ'
11:                 end if
12:             end for
13:         end for
14:     end for
15: end for
16: return M
```

the target state γ'. We then check if $\gamma' \in \Gamma$ holds. For a complete abstracted observation table, this condition should be satisfied if no other equal row defines this behavior for the respective input/output pair. Line 10 then defines the state transition in the state transition function δ. The source state is the currently considered state $\gamma_S \in \Gamma_S$, the transition is labelled with the input/output pair (i, o) and the target state γ'_S, where γ'_S is a row in Γ_S which is equal to the row γ' in the abstracted observation table $\mathcal{T}^{\mathcal{A}}$, i.e., $\gamma'_S \cong_{\mathcal{A}'} \gamma'$ holds. Based on this approach, we can define all state transitions of the ONFSM.

The function returns the constructed ONFSM $\mathcal{M} = \langle Q, q_0, I^{\mathcal{A}}, O^{\mathcal{A}}, \delta \rangle$, where the abstracted input alphabet $I^{\mathcal{A}}$ is given and the output alphabet can be directly derived from the collected observations.

Example 18 (Hypothesis Creation) *The abstracted observation table, Table 8.6, is closed, complete and consistent. Therefore, we can create an ONFSM. The set Γ_S defines two states:* $\mathsf{q_0} = \epsilon$, *and* $\mathsf{q_1} = (\mathsf{connect}, \mathsf{ack})$. *To construct the transitions, we look up the target states in the abstracted observation table, and the corresponding output labels in the standard observation table. This leads to the following transitions:* $\delta(\mathsf{q_0}, \mathsf{connect}, \mathsf{ack}) = \mathsf{q_1}$, $\delta(\mathsf{q_1}, \mathsf{connect}, \mathsf{ack}) = \mathsf{q_1}$, $\delta(\mathsf{q_1}, \mathsf{connect}, \mathsf{closed}) = \mathsf{q_1}$, *and* $\delta(\mathsf{q_1}, \mathsf{connect}, \mathsf{closed\_all}) = \mathsf{q_0}$. *Using this mapping, we construct the ONFSM that is depicted in Figure 8.2b.*

Conformance testing for abstracted ONFSMs. For active automata learning based on the MAT framework, we require an equivalence oracle that tells the learner if the conjectured hypothesis is equivalent to the behavior of the SUL. Similar to learning Mealy machines, ONFSM learning techniques aim to learn an ONFSM $\mathcal{M}$ that fulfills that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}_{\text{SUL}})$, where the language $\mathcal{L}(\mathcal{M})$ includes all the traces that can be defined by $\mathcal{M}$, and $\mathcal{M}_{\text{SUL}}$ is the unknown ONFSM representation of the SUL. Since we learn an ONFSM that represents a more abstract solution we do not test for trace equivalence. Instead, we test for trace inclusion since the learned hypothesis represents an underspecification of an implementation. Therefore, we define the conformance relation for learning as follows:

$$\mathcal{L}(\mathcal{M}_{\text{SUL}}) \subseteq \mathcal{L}(\mathcal{M}). \tag{8.1}$$

We say that $\mathcal{M}$ conforms to $\mathcal{M}_{\text{SUL}}$ if all traces of $\mathcal{M}_{\text{SUL}}$ are included in the set of traces $\mathcal{M}$. A counterexample represents a trace that is not included in $\mathcal{L}(\mathcal{M})$. In other words, this means that $\mathcal{M}_{\text{SUL}}$ produces an output sequence for an input sequence that cannot be observed when executing the same input sequence on $\mathcal{M}$. In practice, we cannot assume that we have a perfect equivalence oracle that returns all traces of a black-box SUL that are not included in the provided hypothesis. Hence, we follow a similar approach as for deterministic systems and approximate trace inclusion via conformance testing techniques. Our test suite consists of a finite set of input sequences. A test case **passes** if all output sequences that are generated by $\mathcal{M}_{\text{SUL}}$ are also observable when executing the same input sequence on $\mathcal{M}$. Thus, a test case **fails** if $\mathcal{M}_{\text{SUL}}$ produces an output sequence that cannot be observed on $\mathcal{M}$. Note that the conformance check compares traces that are abstracted on the first-level abstraction.

8.5 Evaluation

We evaluate our proposed learning algorithm for learning real communication protocol implementations in a multi-client setup. The following section presents adaptions to our presented learning algorithm to make it feasible in practice. Afterwards, we provide the experimental setup in which we conducted the case study followed by a discussion of our case study results. Additionally, the discussion includes a comparison to a classic learning setup.

8.5.1 Practical Considerations

Learning of ONFSMs is challenging due to the underlying non-deterministic behavior of the SUL. Hence, the execution of one input sequence yields different output sequences. Learning algorithms for ONFSMs [56, 97, 140] assume that the *all-weather condition* [120] holds. Thus, the algorithms require that all query outputs can be observed after repeating an output query a finite number of times. In practice, such an assumption implies that a large number of queries must be executed to learn non-deterministic systems. To make learning feasible in practice, we adapt our presented learning algorithm. We propose the following two adaptions: (1) adaptive repetitions of queries, and (2) shrinking of the observation table.

Adaptive repetitions of queries (1). Similar to other learning algorithms for ONFSMs, we repeat each query a finite number of times, where the number of repetitions is defined by a constant $n_q \in \mathbb{N}$. The number of repetitions depends on the underlying SUL and on the considered first-level abstraction. We propose to align n_q to the proportion between abstract inputs to concrete inputs. For example, considering the running example of the previous section on a multi-client connection protocol, we can align n_q to the number of connected clients.

In addition, some systems might behave only non-deterministic for a certain subset of inputs, whereas other inputs show deterministic behavior. Hence, for deterministic inputs, it does not make sense to repeat them. In this case, we assign each entry in the observation table an *observation score* $s \in \mathbb{R}_{\geq 0}$ that indicates how often the output changes. We define $s \in [0, 1]$, where 0 indicates the highest chance to observe a new output on the next execution, and 1 indicates that the chance to observe a new output is very low. We use s to steer the number of repetitions of a query, where $s = 1$ indicates that the query should not be repeated another time. During learning, we adapt s by the constants $s_{\text{inc}}, s_{\text{dec}} \in \mathbb{R}_{\geq 0}$, where s_{inc} increases the value of s and s_{dec} decreases the value of s. These values allow us to adapt the number of repeated queries according to the amount of non-deterministic behavior observable on the SUL.

Instead of executing all repetitions of a query at once, we continuously repeat the queries according to our adaptive query repetition function that we introduced earlier. Therefore, each time we fill the table, we repeat each query that has an observation score of less than one n_q times and afterwards adapt the score s.

Shrinking of the observation tables (2). Using our adaptive query approach implies that entries of the observation table might change during the learning procedure. The update of the entries in the table has the consequence that rows in the table might get similar to others or vice-versa differ from previously similar states. In principle, this does not reflect a problem for our proposed learning approach, but we might perform unnecessary queries. We aim to avoid any entry in the table that is not necessary to model an ONFSM since every entry requires several query repetitions. To keep the observation table as small as possible, we shrink the table if possible. Shrinking means that we remove unnecessary rows from the table that do not show additional behavior. We shrink the table by removing similar rows defined by Γ_S under the consideration that Γ_S remains prefix-closed. If we identify two equal rows, i.e., $\gamma, \gamma' \in \Gamma_S$, where $\gamma \cong \gamma'$, we remove the row with the longer trace in Γ_S. Let γ be the trace that identifies the row that should be removed, then $|\gamma| \geq |\gamma'|$. If we remove the row that is indexed by γ, we also check if we can remove any rows that represent prefixes of γ in Γ_S and if Γ_P contains any extensions $\gamma \cdot (i, o)$ with $i \in I$ and $o \in O$ of the removed row that are not further required.

Note that these adaptions to weaken the all-weather assumption are not only applicable in our learning technique for learning abstracted ONFSM. They can also be directly transferred to other ONFSM learning approaches.

8.5.2 Case Study Subjects

For the following case study, we base our evaluation on five different MQTT broker implementations. We provide a more detailed description of the MQTT protocol in Section 2.3.1. Tappler et al. [170] present that learning MQTT broker implementations that interact with two clients using standard learning algorithms is feasible but the size of the input alphabet increases noticeably. In this case study, we present results on learning abstractions of MQTT broker implementations in a multi-client setup and compare them to the learning results achieved by learning algorithms of state-of-the-art learning libraries.

We consider the following five MQTT broker implementations:

- Eclipse Mosquitto 1.6.8[1],
- ejabberd 20.3,[2]
- EMQ X v4.0.0[3],
- HiveMQ 2020.2[4], and
- VerneMQ 1.10.0[5],

All of the considered MQTT brokers implement the MQTT v5.0 standard [22]. To avoid any external influences, we installed all brokers locally and performed the communication via a local network.

For our case study, we consider that a broker communicates with five different MQTT clients that can connect and disconnect, publish, subscribe, and unsubscribe to topics. The MQTT client implementation is based on the client proposed by Tappler et al. [170].

8.5.3 Learning Setup

The considered learning setup considers four components: (1) the learning algorithm, (2) the mapper, (3) the learning interface, and (4) the system under learning.

[1] https://mosquitto.org/
[2] https://www.ejabberd.im/
[3] https://github.com/emqx/emqx
[4] https://github.com/hivemq/hivemq-community-edition
[5] https://github.com/vernemq/vernemq

Learning algorithm (1). Learning is based on our proposed learning algorithm for learning abstracted ONFSMs. For the first-level abstraction, we consider the abstract input alphabet $I^{\mathcal{A}} = \{\mathsf{Connect}, \mathsf{Disconnect}, \mathsf{Subscribe}, \mathsf{Unsubscribe}, \mathsf{Publish}\}$. The abstracted inputs are translated to the following concrete alphabet $I = \{\mathsf{Connect}(\mathit{client}), \mathsf{Disconnect}(\mathit{client}),$ $\mathsf{Subscribe}(\mathit{client}, \mathit{topic}), \mathsf{Unsubscribe}(\mathit{client}, \mathit{topic}), \mathsf{Publish}(\mathit{client}, \mathit{topic}, \mathit{message}))\}$, where *client* always identifies the client who sends the packet, *topic* is a concrete topic name on which clients publish, subscribe or unsubscribe, and *message* is a concrete messages clients publish. The concretization for *client* is based on a random selection from the corresponding pool of clients. The concretization of *topic* is aligned to currently considered topics and *message* is randomly generated.

The concrete output is the set of received packets of every client after a client sends a packet. The translation to abstract outputs is straightforward by the packet name, where all distinct packets are concatenated. However, to enable the learning of an ONFSM, we distinguish between the states where no client is connected and no client is subscribed. For these cases, we extend the abstract output alphabet by the outputs Closed_all, Unsuback_all and Closed_Unsuback_all, where Closed_all indicates that all clients are disconnected, Unsuback_all that all clients are unsubscribed and Closed_Unsuback_all that the last subscribed client disconnects. Hence, the abstracted output alphabet is $O^{\mathcal{A}} = \{\mathsf{Connack}, \mathsf{Puback}, \mathsf{Suback}, \mathsf{Unsuback}, \mathsf{Puback\_Publish}, \mathsf{Closed\_all}, \mathsf{Unsuback\_all}\}$.

The learning algorithm component includes the second-level abstraction. For learning the MQTT protocol, we define the second-level abstraction function as follows:

$$\begin{aligned}\mu = \{&\{\mathsf{Closed}, \mathsf{Closed\_all}, \mathsf{Closed\_Unsuback\_all}\} \to \mathsf{Closed},\\ &\{\mathsf{Unsuback}, \mathsf{Unsuback\_all}\} \to \mathsf{Unsuback},\\ &\{\mathsf{Connack}\} \to \mathsf{Connack},\\ &\{\mathsf{Puback}\} \to \mathsf{Puback},\\ &\{\mathsf{Suback}\} \to \mathsf{Suback},\\ &\{\mathsf{Puback\_Publish}\} \to \mathsf{Puback\_Publish}\}.\end{aligned} \tag{8.2}$$

For the repetition of queries, we set $n_q = 10$ and we initialize $s = 0.9$, where $s_{\text{inc}} = 0.2$ and $s_{\text{dec}} = 0.1$. Our conformance testing technique is based on 2 000 random walks, with a stop probability of 0.25 for each trace.

Mapper (2). The purpose of the mapper component is to translate inputs from the first-level abstraction into concrete inputs that can be executed on the SUL. The same is done for received concrete outputs which the mapper receives directly from the different clients. The mapper then translates these concrete outputs to abstract outputs according to the first-level abstraction. To perform abstractions such that an ONFSM can be learned, the mapper needs to be stateful in the sense that it keeps track of the number of connected and subscribed clients. The mapper concretizes matching topic names of the subscribe, unsubscribe and publish messages. Furthermore, the mapper implements a cache that saves concretizations of abstract input sequences to reproduce the corresponding output sequence. This especially helps when creating access sequences to specific states.

Learning interface (3). The interface is responsible for the communication with the MQTT broker. In our setup, we consider five equal MQTT clients that interact with the broker. In difference to the setup of Tappler et al. [170], we do not assign specific roles to the clients such as publisher or subscriber. The assignment of such roles decreases the state space. For our setup, we define every MQTT command for every client. The mapper assigns an MQTT command to one client at a time that should be forwarded to the MQTT broker. The MQTT broker then responds to this request with a corresponding response. Note that the broker may respond to

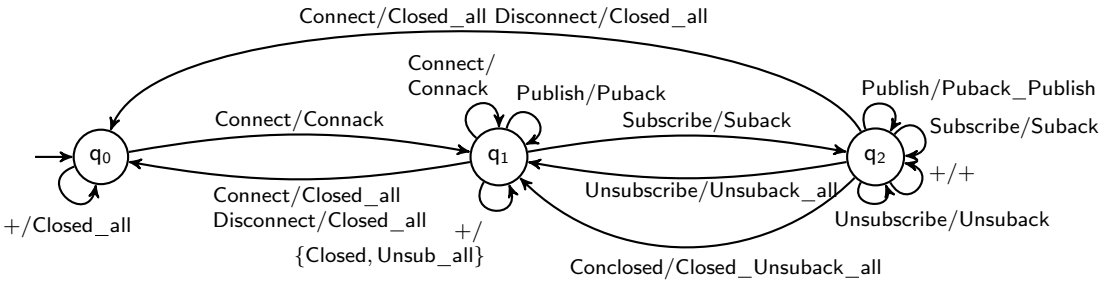


Figure 8.4: Learned abstracted ONFSM that represents the behavior of an MQTT broker that interacts with five clients. Some inputs and outputs have been grouped by the symbol '+'.

Table 8.7: Learning setup and results of our case study on MQTT brokers.

Broker	EJABBERD	EMQ X	HIVEMQ	ECLIPSE MOSQUITTO	VERNEMQ
Timeout	100	50	100	50	50
# Output queries	18 315	18 375	15 950	13 975	14 800
# Equivalence checks	1	1	1	1	1
Runtime (h)	11.28	5.48	9.04	3.98	4.30

more than one client. For example, consider that a client sends a publish message, then apart from the response that the publish message is accepted Puback the broker also forwards to all clients that are subscribed to the topic of the publish message, which is then summarized by a Publish output.

System under learning (4). The MQTT broker implementation represents our SUL. We expect that the broker responds within a certain timeout.

8.5.4 Experimental Setup

We implemented our learning algorithm in Scala 2.12. We used Scala since functional programming was convenient to check the properties of the observation tables and it allows us to use Java libraries such as the MQTT client implementation provided by Tappler et al. [170].

All experiments were executed on an Apple MacBook Pro 2019 with an Intel Quad-Core i5 running at 2.4 GHz and with 8 GB memory.

8.5.5 Results

The following section presents the results of our evaluation on learning an abstracted ONFSM of five different MQTT brokers. We could learn all five MQTT broker implementations. The learned model is shown in Figure 8.4, where the same model was learned for all five implementations.

The learned ONFSM depicts an abstraction of the MQTT broker behavior which interacts with five clients. State q_0 represents the state where no client is connected. After one client is connected, we traverse to state q_1. State q_1 describes a state where clients are connected, but none of the clients is subscribed. Thus, if one connected client subscribes to a topic, state q_2 is entered. In state q_2, at least one client is connected and subscribed. If all clients unsubscribe but remain connected, the first-level abstraction translates the concrete output to Unsuback_all. Furthermore, if only one client is subscribed and that client also disconnects, none of the clients are subscribed, which is indicated by the output Closed_Unsuback_all.

Table 8.7 presents the learning metrics. The metrics include the considered timeout to wait for messages, the number of performed output queries, the number of equivalence queries and the runtime in hours (h). We adjusted the timeouts for EJABBERD and HIVEMQ to learn the

same model for all brokers. With the original timeout of 50, messages may arrive delayed from these brokers. In theory, we could learn models that reflect this non-deterministic behavior, but for this case study our target was to learn the same model for all MQTT broker implementations.

Every MQTT broker could be learned within one learning round, i.e., one equivalence check was performed. Our active learning technique executed between 13 975 and 18 375 queries on the SULs to learn the model depicted in Figure 8.4. The runtime ranged from 3.98 h to 11.28 h, whereby an increased timeout appears to have a direct influence on the runtime.

Learning a three-state model in at least three hours might give the impression that the learning process is a bit excessive. To provide a comparison, we also applied the same learning setup to a state-of-the-art learning library. We used the Java library LearnLib [90], to learn a behavioral model of the five considered MQTT broker implementations interacting with five clients. To enable learning of a deterministic automaton, we have to adapt the abstracted input and output alphabet such that deterministic behavior can be modeled. To do so, we add a representation for each input and output for each client. For this experiment, our learning setup applies L^* for learning Mealy machines with the improvements proposed by Rivest and Schapire [156]. To achieve a fair comparison, we again apply an equivalence oracle approximated by conformance testing, where the test suite is generated via random walks, using a similar setup to our presented learning setup.

With this LearnLib setup, we could only learn a Mealy machine that represents the Eclipse Mosquitto broker, all other devices behaved non-deterministically. Learning with a timeout of 50 took 58.29 h, where the learned Mealy machine has 243 states. Active learning required 151 900 output queries where one equivalence query was performed. Compared to our learned model, we observe how the second-level abstraction helps to keep the model concise. In practice, a model with 243 states might be harder to interpret, especially by humans.

8.6 AALpy Integration

To make our proposed learning algorithm for ONFSMs publicly better accessible, we integrated the algorithm into the Python automata library AALpy. The Python implementation follows the presented learning algorithm. However, it offers other practical approaches to weaken the all-weather assumption. The practical optimizations were presented in the Bachelor's Thesis of Windisch [193].

Similar to our approach, the table can be shrunken during learning to avoid unnecessary queries. The implementation of the table shrinking technique follows the concept presented in Section 8.5.1.

The adaptive repetition of queries is also implemented but slightly differently. The implementation in AALpy considers a tree-based caching structure. The tree-based cache is similar to a prefix-tree acceptor (PTA) which is extended by every observed trace. Each performed query is looked up in the cache. Each input/output pair in the cache is repeated a certain number of times. If the maximum number of samples is reached, the output is directly taken from the cache. Based on the observations in the tree, we then update the entries in the table. The advantage of this technique is that it avoids redundancy in the repetition of queries. To fill an observation table, many queries represent a prefix of other queries. Using the underlying tree structure, we can avoid repeating a trace a finite number of times that represents a prefix of another trace that also needs to be repeated. The results of Windisch [193] show that this data structure together with table-shrinking could speed up learning by a factor of 11.

These optimizations help to improve the runtime of learning ONFSMs, but also improve the number of correctly learned models. If we weaken the all-weather assumption, rows in the table might be incomplete in the sense that they miss observations. This could either lead to more or fewer states than required to model the minimal ground truth automaton of the SUL. By shrinking the table, we reduce the risk of missing observations. The results

of Windisch [193] underline this statement, by showing that the number of correctly learned automata is substantially higher than without these optimizations. For example, for 50 randomly generated ONFSMs with 15 states, 5 inputs and 5 outputs, where every query was repeated 30 times, the algorithm without the optimizations learned correctly only once, whereas with our optimizations we managed to learn the correct automaton for all examples. The trend was similar for the other experimental setups. For further details on these optimizations and their achieved improvements, we refer to the Bachelor's thesis [193].

8.7 Conclusion

In this chapter, we presented a solution to learn non-deterministic systems in practice. To this end, we presented a learning algorithm that learns an abstraction of a non-deterministic system. The presented learning framework follows the L^* algorithm, but extends the MAT framework with an additional level of abstraction. The additional abstraction layer abstracts the observations in the observation table and allows us to learn an abstracted ONFSM. In order to apply the presented learning algorithm for abstracted ONFSMs in practice, we introduced several approaches that allow us to weaken the assumption that all possible behaviors must be directly observed. We demonstrated the feasibility of our learning algorithm by evaluating the learning of behavioral models of MQTT brokers in a multi-client setup. Using our proposed technique, we were able to learn a concise representation. A comparison with a classical learning setup for learning a deterministic system showed that the learned model had 81 times more states than our abstracted model. We have provided examples showing how the abstraction can be performed. However, we did not provide a solution to automate the definition of abstraction mappings and leave this open for future work.

(RQ 1) What are the challenges of learning behavioral models in networked systems?

We investigated how well automata learning performed considering a multi-client setup. The challenge in a multi-client setup was that the state space increased with each additionally considered client. To make learning feasible, we were required to abstract the input and output alphabet. However, the abstraction had to be chosen such that a deterministic model could be learned, which still resulted in large behavioral models.

(RQ 2.2) How to improve automata learning to make it feasible for different challenges in networked environments?

We presented a novel learning algorithm that generates abstracted ONFSMs. Our algorithm extended the classical active learning framework and introduced an additional layer of abstraction to group observations. Using this technique, we could learn behavioral models of MQTT brokers interacting with multiple clients. Furthermore, we discussed adaptations to learning algorithms for non-deterministic systems such that not all observations needed to be observed at once, but also to avoid redundant query executions.

Chapter 9

Learning-based Fuzzing

Declaration of Resources

This chapter presents different methods that are presented in two papers and parts of the Master's Thesis by Benjamin Wunderling. Section 9.2.1 is based on the paper *"Learning-Based Fuzzing of IoT Message Brokers"* [11]. Section 9.2.2 describes the methodology that is presented in the paper *"Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning"* [148]. Sections 9.2.3, 9.2.4, and 9.2.5 are based on the Master's Thesis of Benjamin Wunderling with the title *"Model Learning and Fuzzing of the IPsec-IKEv1 VPN Protocol"* which is co-supervised by the author of this thesis.

This chapter introduces the concept of learning-based fuzzing. Learning-based fuzzing is a black-box fuzzing method and can be defined in a two-step procedure. First, we use automata learning to create a behavioral model. Second, we apply model-based fuzzing using the previously learned model. This chapter opens with an introduction to fuzz testing in Section 9.1. Afterwards, Section 9.2 presents the general methodology of learning-based fuzzing followed by different concrete concepts to implement this technique. We summarize this chapter in Section 9.3. In Chapter 10, we then show the effectiveness of learning-based fuzzing in case studies on different communication protocols.

9.1 Background

9.1.1 Fuzzing

Fuzz-testing or fuzzing is a testing technique that executes a large amount of randomly generated inputs on the SUT to reveal unexpected behavior. The origin of fuzzing dates back to the work of Miller et al. [119]. In their work, they implemented a tool called *fuzz* which generated random input strings for testing UNIX utilities. Their tool was able to reveal many crash scenarios on the tested applications. Today, fuzzing is a popular tool for testing systems due to its ease of use and high success rate.

According to Godefroid [71], today's fuzzing tools often aim to reveal security vulnerabilities. In this work, we also refer to this definition. Thus, we use fuzzing to test security and reliability issues. However, fuzzing techniques can also be used to support other testing techniques such as search-based testing [175, 203].

Figure 9.1 illustrates a standard fuzzing framework that consists of four components: (1) the test-case generator, (2) the test executor, (3) the monitor, and (4) the SUT. The test-case generator (1) is often referred to as the fuzzer. The fuzzer generates a test suite using random input-generation techniques. Depending on the fuzzing approach, the test case generator uses

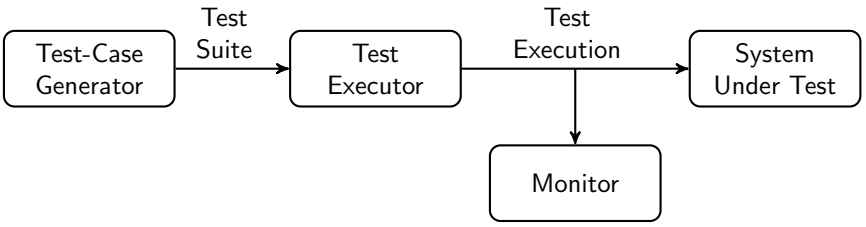


Figure 9.1: A standard fuzzing framework consists of four components.

information about the underlying input structure and/or a given input as a seed for generating the test suite. This test suite is then executed on the SUT by a test executor (2). The executor implements an interface to the SUT and enables also the execution of invalid and unexpected inputs. To observe unexpected behavior, fuzzing requires some kind of monitor (3) that logs the behavior of the SUT, e.g., system crashes. The fourth component is the SUT (4) which is the fuzzing target. Godefroid [71] defines that any system can be fuzzed that parses a given input. To test the effectiveness of fuzzing, we also require to observe the behavior of the SUT of a given input, at least to some extent.

In the literature [106], fuzzing is frequently categorized based on the level of accessibility of the SUT. We define three categories: white box, black box, and gray box.

In white-box fuzzing, we have access to the source code of the SUT. White-box fuzzers such as SAGE [72] use program analysis techniques such as symbolic execution, to automatically generate test cases based on code coverage metrics. Fuzzers such as SAGE use constraint solving to access any constrained path in a program. These constraints might be complex to solve. Furthermore, the requirement of having access to the source code, especially considering environments that rely on third-party components, limits the feasibility of white-box fuzzing to open-source products or in-house development projects.

In black-box fuzzing, we do not have insights into the internal behavior of the system. We can only execute inputs and observe the corresponding outputs. Early results in fuzzing by Miller et al [119] show that the execution of random inputs is already sufficient to detect software bugs. However, black-box fuzzing becomes challenging if the SUT requests complex input structures. An example of a more complex input would be communication protocol packets with many different fields that must be arranged in a certain order. To overcome this problem, black-box fuzzers like SNOOZE [23] or boofuzz [142] use a formal definition of the language to generate inputs. This method is especially well suited for fuzzing network protocols, since the packet structure can be defined by a context-free grammar. However, it remains a problem to determine whether black-box fuzzing is exhaustive enough.

Gray-box fuzzers can be classified between white-box and black-box fuzzers. In gray-box fuzzing, access to the code is not provided, but gray-box fuzzers use code instrumentation to guide the test case generation. Böhme et al. [28] define gray-box fuzzing as a state-of-the-art security testing technique. They argue the success of gray-box fuzzing by the results achieved by tools like AFL [201], and its variations like AFL++ [60] or AFLGo [28]. These fuzzing tools successfully detected bugs in all kinds of systems including Mozilla Firefox, OpenSSL, and the iOS kernel.

Furthermore, fuzz-testing techniques can also be classified based on their input generation technique. The literature distinguishes between mutative and generative input generation techniques. Some work mainly defines these techniques as subclassification of black-box fuzzing [48], whereas others [116] assign these categories independent of the access level of the SUT. Mutative fuzzers randomly modify a given input using mutation operators like bit-flipping. The advantage of mutative fuzzers is that they can be straightforwardly applied without requiring any knowledge about the SUT [48]. However, it can be particularly difficult to explore in-depth behavior by simply changing the existing inputs. Generative fuzzers use given input structures

to generate inputs for fuzzing. In contrast to simple mutative fuzzing, generative fuzzers can test specific aspects of protocols. However, the definition of such an input structure might be laborious.

9.1.2 Protocol State Fuzzing

Protocol state fuzzing is a black-box fuzzing technique that fuzzes communication protocols via automata learning. In general, the technique follows a learning-based testing approach, where active automata learning is used to test the system. In active learning, a large number of inputs are executed on the SUT to learn a behavioral model. The goal is to reveal unexpected behavior by learning a model, where the active learning algorithm executes possible unexpected inputs for the currently investigated state in order to explore the state space. After learning, the learned model is analyzed for any unexpected behavior. For example, if there exist paths to circumvent mandatory steps.

Protocol state fuzzing has been successfully applied to learn security-critical protocols such as TLS [50], parts of SSL/TLS [163], DTLS [63] and OpenVPN [47]. The conducted case studies revealed security issues such as the threat of violating integrity and confidentiality. In addition, some of our learned models show that there exist paths to bypass authentication steps. In addition, Daniel et al. [47] stress that the learned models help to understand the implementation and add additional information to often sparse documentation and specifications.

The protocol state fuzzing approaches [47, 50, 63] follow the same learning setup as presented in Chapter 4 and Chapter 5. In general, a model is learned using an abstracted input alphabet. A mapper component is then used to translate inputs, where the mapper is most of the time stateful to store variables of the current connection. The learned model is then manually analyzed for any security or specification violations.

9.2 Method

We define learning-based fuzzing as a fuzz-testing method that combines automata learning and fuzzing techniques. In contrast to protocol state fuzzing, learning-based fuzzing techniques first learn a behavioral model and then model-based fuzzing techniques are applied to test the SUT. We see in the case studies on BLE and VPN that learning requires a fault-tolerant learning setup, where unusual behavior might occur due to lost, retransmitted or delayed packets. Hence, fuzzing the system during learning with a large number of inputs can be a tedious process. To overcome this issue, we propose a different fuzzing technique, which is also based on automata learning, but in this case, fuzzing is performed after learning.

Learning-based fuzzing can be described in a two-step procedure, where we first learn the behavioral model using automata learning techniques and then use model-based fuzzing techniques to fuzz the SUT. Figure 9.1 provides an overview of the general learning-based fuzzing framework. The framework includes a learning and a fuzzing component. Furthermore, we consider a system interface that is used by the learning and fuzzing component. The system interface provides access to the SUT by an adapter component. The adapter enables communication between the mappers and the SUT. For example, for learning-based fuzzing of a BLE device, the adapter includes another BLE device that sends packets to the SUT and receives the corresponding responses.

The learning-based fuzzing procedure first starts by learning a behavioral model of the SUT. The learning component is similar to the learning setups used in the chapters 4 and 5. The model is learned using an abstracted input alphabet. This makes learning feasible in a reasonable amount of time. Later, we show that the abstraction is useful for further testing purposes. The abstraction for learning is conducted by a mapper component as described in Chapter 3. For learning-based fuzzing, we mainly consider active learning algorithms, since it is useful for

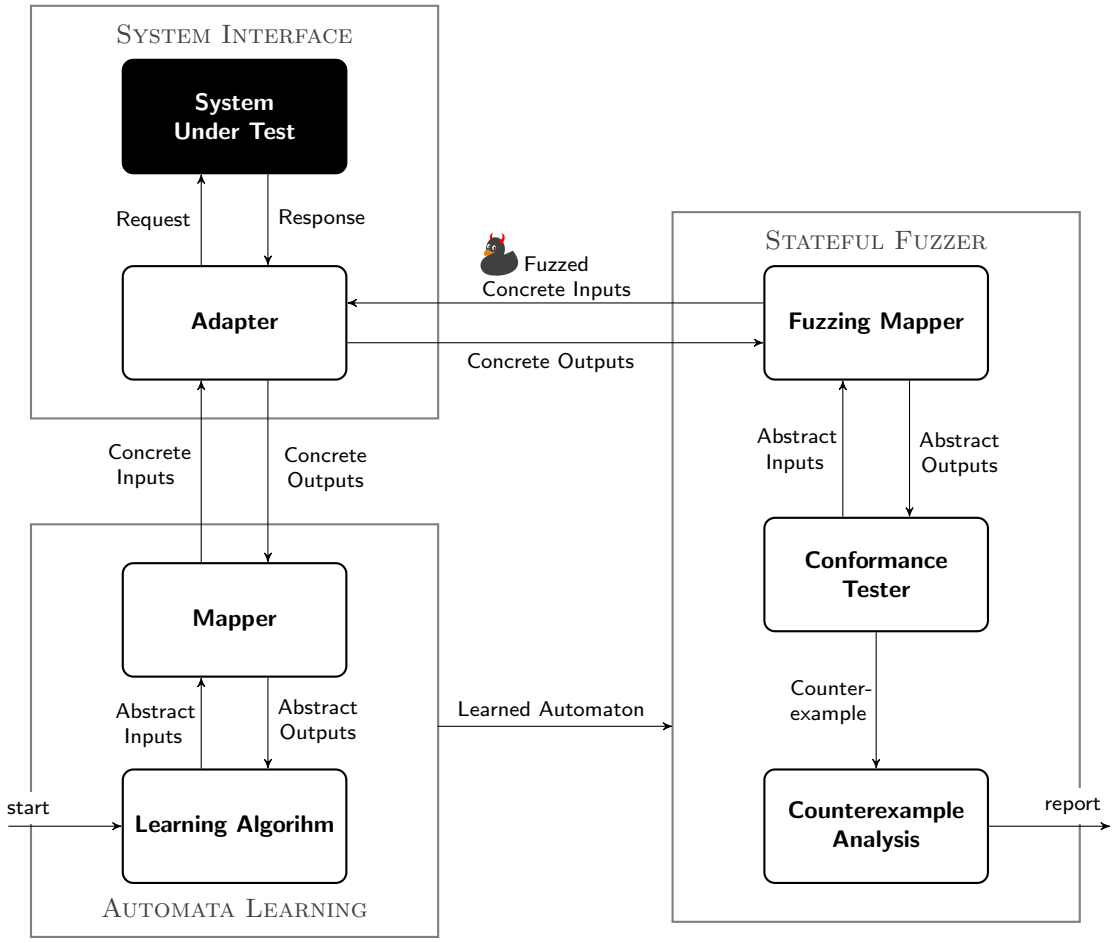


Figure 9.2: General learning-based fuzzing framework, where the system is first learned and then based on the learned model fuzzed.

fuzzing purposes to have a model defined for each input in each state. In practice, passive algorithms that learn from a given sample, such as log files, are likely to not provide input completeness.

The learned abstract model serves as input for the second step in the learning-based fuzzing procedure. The fuzzing component implements a conformance testing technique as introduced in Section 2.2.2. In conformance testing, we test if an implementation $\mathcal{I}$, which is represented by the SUT in our setup, conforms to the given model $\mathcal{H}$. However, the goal of fuzzing is to violate the conformance relation given in Equation 2.1. By applying fuzzing techniques, we create a test suite TC_{fuzz} such that

$$I \textbf{ imp } H \;\Leftrightarrow\; \forall\, tc \in TC_{\text{fuzz}} \colon I \textbf{ passes } tc. \tag{9.1}$$

does **not** hold. In the remainder of this chapter, we discuss different techniques to generate such a test suite.

We use the learned model to generate a fuzzing test suite. Since the learned model is on a more abstract level than the SUT, we again require a mapper component that translates abstract inputs to concrete inputs. To fuzz the SUT, we also include unexpected inputs in the fuzzing test suite. Hence, the fuzzing mapper differs from the mapper used in learning by the applied concretization function. The difference is that the generation of concrete inputs not only includes standard values or a specific range of values but also unexpected values.

In the case that we find a counterexample to the conformance relation of Equation 9.1, the learning-based fuzzing framework reports the found counterexample and performs an additional analysis on the provided counterexample. The counterexample analysis could either be a manual process or automatically performed.

One advantage of this fuzzing framework is that model learning and fuzzing are separated. This is especially useful in real-world scenarios since fuzzing during learning would require advanced mechanisms to deal with unexpected behavior. For example, if the fuzzed input crashes the SUT, mechanisms are required to recover the learning procedure.

Another advantage of this two-step learning-based fuzzing procedure is that it is not required that the SUL and SUT are the same system. For example, the SUL might represent a reference implementation, whereas the SUL can be any other implementation that can be accessed via the same abstract input alphabet. As an additional advantage, learning-based fuzzing is also applicable if only one system can be learned. The model can then be reused for fuzzing other implementations.

9.2.1 Grammar-based Fuzzing

Grammar-based fuzzers generate inputs using an underlying grammar. A grammar includes a set of rules that define a language, where every word that conforms to rules is part of the language. Grammars can be used to determine whether a particular sentence is part of a language. Grammar 9.1 shows an example of a context-free grammar in Backus-Naur form (BNF). The grammar consists of a set of terminal and non-terminal symbols, a non-terminal as starting symbol, and a set of rules. The non-terminal symbols are used to define the grammatical rules in a structured and concise form. In Grammar 9.1, non-terminal symbols are indicated with angle brackets "⟨…⟩". Terminal symbols instead represent letters that are part of the language. Grammar 9.1 illustrates terminal symbols within quotes '…', where the string '`UTF-8 Characters`' represents any allowed UTF-8 character for topic names in the MQTT protocol.

Example 19 (Grammar for MQTT Topics) *Grammar 9.1 describes a language for topic filters as they are used in the MQTT protocol. As described in Section 2.3.1, MQTT is a publish/subscribe protocol, where clients can subscribe and publish messages on topics. Grammar 9.1 defines the language of topic filters a client can subscribe to. The grammar defines different layers for a topic, which are separated by a slash ('/'). Strings with UTF-8 characters can be*

$\langle TopicFilter \rangle$ ::= '$'$\langle Body \rangle$ | $\langle Body \rangle$

$\langle Body \rangle$ ::= '#' | '+'$\langle EmptyLevel \rangle$
| $\langle String \rangle \langle EmptyLevel \rangle$
| $\langle Level \rangle$

$\langle EmptyLevel \rangle$ ::= $\langle Level \rangle$ | ϵ

$\langle Level \rangle$ ::= '/'$\langle EmptyBody \rangle$

$\langle EmptyBody \rangle$ ::= $\langle Body \rangle$ | ϵ

$\langle String \rangle$ ::= 'UTF-8 Characters'

Grammar 9.1: The ruleset of a grammar in BNF that defines the syntax of topic filters used in the MQTT protocol, where the start symbol is $\langle TopicFilter \rangle$.

between the slashes. Some of these characters have a specific semantic. For example, the hash ('#') defines a wildcard, where a client subscribes to all subsequent topic layers starting with the prefix before the hash. The plus ('+') represents also a wildcard, but this time only for one level. Clients can only publish to topic names which means that wildcards are not allowed. Valid topic filters are, e.g., `temp/gf/kitchen`*,* `temp/gf/#`*, or* `///` *and examples for invalid topics are* `temp/#/kitchen` *or* `##`*.*

A grammar not only defines the words that are part of the language, but also enables the creation of words that are part of the language. For fuzzing, the grammar can also be modified to generate invalid words, e.g., including invalid characters. Using the grammar, we test whether parsing the SUT correctly parsed the provided inputs. In addition, based on the grammar, coverage-based metrics could be defined, such as the number of rules covered. The drawback of grammar-based fuzzing is that the manual definition of such a grammar can be a tedious process and requires some expert knowledge about the input format. We show the effectiveness of this technique in a case study on the MQTT protocol which we present in Chapter 10.

9.2.2 Model-based Fuzzing

In model-based fuzzing, we generate a fuzzing-test suite based on a behavioral model. Our model-based fuzzing technique considers a model that is previously learned using automata learning techniques. Similar to the learning step, we use model-based testing techniques to test if the SUT conforms to the learned model. In general, the provision of any coverage metric in black-box fuzzing is difficult. With the underlying model, however, we can define coverage metrics based on state or transition coverage.

Our model-based fuzzing technique derives abstract input sequences by traversing through the provided model. Note that the provided model defines the behavior on an abstract level. Hence, the generated input sequence is a sequence of abstract inputs. We concretize the abstract input sequence in a specific way to create a test suite for fuzzing. A fuzzing sequence consists of three parts: (1) an access sequence to a state, (2) a fuzzed input, and (3) a random input sequence of non-fuzzed inputs. The access sequence (1) is used to generate a fuzzing test suite that provides state coverage. By doing so, our test suite visits at least once each state. The concretization of the inputs of the access sequence is similar to the one used in learning in order to reach the desired state. The access sequence is followed by a single fuzzed input (2). Our fuzzing mapper concretizes the fuzzed input using fuzzing techniques. Hence, the concretization includes unexpected values and invalid inputs. The last part of the fuzzing sequence consists of a sequence of random abstract inputs (3). The random inputs in the suffix are concretized using the same concretization as in learning. The random suffix is used to determine whether the fuzzed input imposed any unexpected state transitions.

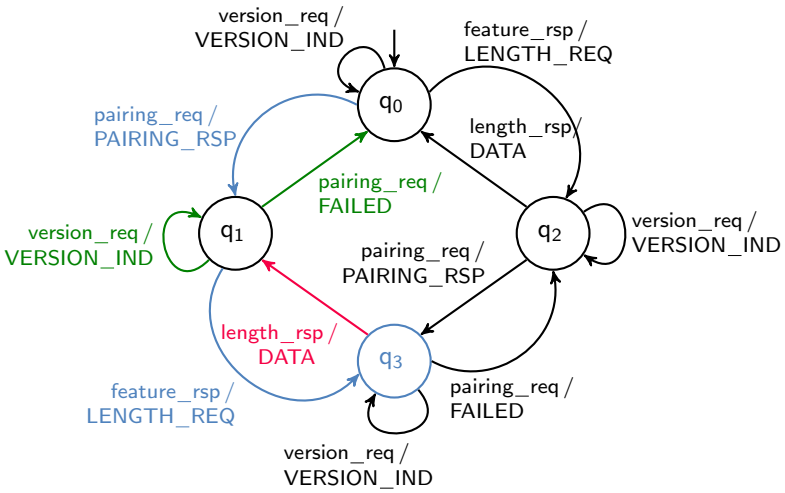


Figure 9.3: Model of a learned BLE device. The highlighted colors indicate the edges traversed during the generation of a fuzzing sequence, where the blue parts indicate the access sequence to a state, the red one the fuzzed input and the green ones the randomly generated suffix of the sequence.

Figure 9.3 introduces the model-based fuzzing technique using a color scheme, where the blue parts show the access sequence to a specific state, red indicates the fuzzed input, and green indicates the suffix of random inputs.

Example 20 (Model-based fuzzing for BLE.) *The colored fuzzing sequence in Figure 9.3 would be the following abstract input sequence*

$$\mathit{pairing\_req} \cdot \mathit{feature\_rsp} \cdot \mathit{length\_rsp} \cdot \mathit{version\_req} \cdot \mathit{pairing\_req}$$

, where the colors identify the corresponding parts. This sequence accesses State q_3*, selects* length_rsp *as input to be fuzzed, and appends as random input sequence the inputs* $\mathit{version\_req} \cdot \mathit{pairing\_req}$*. The fuzzed input is concretized using fuzzing techniques, such as selecting a value that is within the reserved bytes for the field but may be prohibited by the specification. For example, for the fuzzed input* length_rsp *the concrete BLE packet in* Scapy *syntax would be*

$$\mathit{BTLE()/BTLE\_DATA()/BTLE\_CTRL()/LL\_LENGTH\_RSP}(\mathit{max\_tx\_bytes} \in [0, 2^8), \ldots),$$

where we could set the value of the field max_tx_bytes between the range 0 *to* 2^8*. The concrete value can be randomly chosen or selected giving a preference to boundary values. To create an invalid value, our fuzzing mapper chooses, e.g., the value* 0*.*

The last part of the randomly generated input sequence is used to determine whether the fuzzed input introduces any unexpected state change. For this purpose, we check if we still observe the same output sequence after executing the fuzzed input. If the observed output sequence is different, we found new behavior. For further analysis of such unexpected behavior, we can try to identify the reached state by appending the characterization sequences of each state to the input sequence. The characterization sequences correspond to the sequences in the W-set as proposed by Vasilevski [186] and Chow [39]. With this, we can determine which state is accessed or if we explored an unknown state.

The advantage of our proposed model-based fuzzing method is that the causes of unexpected behavior are better traceable since only one input is fuzzed. Furthermore, we can investigate if the fuzzed input only leads to unexpected behavior when executed in a specific state. The disadvantage of our method is that only one input is fuzzed with one concretization at a time.

Hence, a large set of sequences is necessary to fuzz the system sufficiently, especially if the number of possible packets is large. In Chapter 10, we evaluate our proposed model-based fuzzing method on a case study on BLE devices.

9.2.3 Filter-based Fuzzing

We propose another fuzzing method called *filter-based fuzzing*. filter-based fuzzing aims to combine behavioral coverage with thorough fuzzing of a single input. This fuzzing technique reuses the data generated in learning. In active learning, we perform a set of output queries for learning a behavioral model. Consequently, the set of performed output queries covers the whole behavior that is described in the learned model. Therefore, this set provides state and transition coverage.

To overcome the limitations of our model-based fuzzing technique, which only fuzzes one input at a time, we want to fuzz the same abstract input with several different concrete values. Our filter-based fuzzing method iterates through each input sequence of all performed output queries. In this input sequence, we fuzz every input of the sequence, where the input concretization is similar to one for fuzzed inputs in model-based fuzzing. We fuzz the same input multiple times using different values for the concretization of the fuzzed inputs. Hence, an input sequence is executed with the number of inputs times the number of concretizations.

Fuzzing every input of each input sequence multiple times is not very practical, as this can be very time-consuming, especially for larger systems. To make our filter-based fuzzing method feasible, we add an additional filtering step to decrease the considered test suite. In the filtering step, we only select one or two abstract input symbols of each input sequence that we want to fuzz. If fuzzing reveals unexpected behavior, we add the input sequence to our test suite. This test suite is then used for further thorough fuzzing as previously described. In this case, we fuzz now every input of the filtered input sequence. This fuzzing technique fuzzes the system very thoroughly and provides state and transition coverage in the first filtering step. However, this approach takes quite some time, especially, if the filtered set contains a lot of traces.

9.2.4 Search-based Fuzzing

Our fourth developed fuzzing method is called *search-based fuzzing*. Search-based fuzzing aims to minimize the size of the test suite as much as possible. For this purpose, we base the test suite on a single input sequence. Instead of shrinking a given set of traces, our search-based technique generates a single input sequence. For this, we define a fitness function that guides the search. Similar to our filter-based fuzzing technique, the SUT is filtered twice: first during the generation of the input sequence, and then a second time when the input sequence is fuzzed again considering further concretizations of the different input symbols.

The fitness function evaluates a given input sequence according to its effectiveness in fuzzing. The effectiveness is based on the potential to observe unexpected behavior when fuzzing this input sequence on the SUL. In addition, the fitness function also considers the number of visited states. We define the fitness function as follows

$$f_{\text{seq}} = \sum_{j=0}^{n-1} \frac{b_{\text{new}}(i_0 \cdot \ldots \cdot i_j)}{|s_I|} \frac{|Q_{\text{visited}}|}{|Q|}. \tag{9.2}$$

Let s_I be the evaluated input sequence, and b_i^{new} be the number of newly observed behavior during the fuzzing of the i-th input of sequence s_I. New behavior is observed when the execution of the fuzzed input on the SUT leads to a different output than the execution of the non-fuzzed input. Hence, we define b^{new} for an input sequence $i_0 \cdot \ldots \cdot i_j$ as follows

$$b_{\text{new}}(i_0 \cdot \ldots \cdot i_j) = \sum_{k=1}^{m} 1 \qquad \textit{if } \lambda^*_{\mathcal{M}}(q_0, i_0 \cdot \ldots \cdot i_j) \neq \lambda^*_{SUT}(q_0, i_0 \cdot \ldots \cdot i_{j-1} \cdot \mathit{fuzz}(i_j)), \tag{9.3}$$

Algorithm 10 Search-based input sequence generation for fuzzing

Input: black-box access to SUL, input alphabet I, learned model $\mathcal{M}$, maximum iterations $i_{\max}$
Output: input sequence s_I

```
1:  s_I ← init_sequence(I)
2:  f_seq ← calc_fitness(s_I, SUL, M)
3:  i ← 0
4:  while i < i_max do
5:      s'_I ← mutate(s_I, I)
6:      f'_seq ← calc_fitness(s'_I, SUL, M)
7:      if f'_seq > f_seq then
8:          s_I ← s'_I
9:          f_seq ← f'_seq
10:     end if
11:     i ← i + 1
12: end while
```

where $m \in \mathbb{N}$ is the number of repetitions for fuzzing the input i_j, $\lambda^*_{\mathcal{M}}$ and λ^*_{SUT} being the output functions for the learned model $\mathcal{M}$ and the SUT respectively, and $fuzz(i_j)$ be a function that concretizes the input i_j according to fuzzing methods, all other inputs are concretized similar to the concretization method applied during learning. Note that Equation 9.2 also considers a factor of the number of visited states $|Q_{\text{visited}}|$ in relation to the total number of states $|Q|$, where $Q_{\text{visited}} \subseteq Q$ is the set of accessed states in the learned model $\mathcal{M}$ when executing s_I.

Algorithm 10 describes the procedure for generating the input sequence for fuzzing. The algorithm starts by initializing the input sequence s^I in Line 1. The input sequence s_I can either be initialized with the empty sequence or a random input sequence generated from the given input alphabet I. As a next step in Line 2, we calculate the fitness value of the initial sequence based on the fitness function defined in Equation 9.2. The generation of the input sequence is fixed to a maximum number of iterations $i_{\max} \in \mathbb{N}$. We modify the input sequence from Line 5 to Line 11. Each iteration starts with the mutation of s_I in Line 5.

The mutation of the input sequence in Line 5 considers two different operations for mutation. First, we can add a new input from the input alphabet to the sequence. Second, we can exchange an existing input with another input. However, to explore more new behavior, we append new inputs at the end of the sequence with a higher probability. In Line 6, we then calculate the fitness value of the mutated input sequence based on Equation 9.2 and compare it with the fitness value of the sequence before mutation in Line 7. If the new fitness value is higher, we then consider the mutated input sequence as a new sequence including an update of the fitness value.

With this technique, we can generate a single input sequence that can later be used for a more exhaustive fuzzing of the SUT. Note that the calculation of the fitness value takes some time since every input in the sequence is fuzzed several times. However, executing a single sequence afterwards is much faster than fuzzing a set of sequences. The disadvantage of this method is that the fitness function may optimize to a local maximum of the fitness value. Furthermore, depending on the learned model, a single sequence might not be sufficient to cover all states.

9.2.5 Genetic-based Fuzzing

Our genetic-based fuzzing approach is based on our search-based fuzzing technique presented in the previous section. Genetic-based fuzzing aims to overcome the disadvantages of search-based fuzzing by applying a genetic algorithm. In general, a genetic algorithm searches for a solution by evolving a population of solutions within a certain number of generations. For the generation of a new population, the current population is mutated using mutation operators. For

mutation, individuals are changed as in the search-based approach, crossover between individuals is applied, or newly generated individuals are added. The selection of the individuals for the next generation is then based on the fitness value of each individual. For the calculation of the fitness value, we consider again the fitness function presented in Equation 9.2.

The population consists of a finite set of input sequences. We evolve this set of input sequences iteratively. In each generation, the set of input sequences is modified according to the search-based fuzzing technique, but only for a smaller number of iterations. Based on the fitness value of each input sequence, we select a certain number of the fittest sequences. The fittest sequences are sequences with the highest fitness value. To generate a new population, the fittest sequences are added to the next population, as well as sequences that are generated using the crossover of two sequences. For the crossover operation, we take two of the fittest sequences and cut them into two parts, where the sizes of the shares are randomly chosen. The first part of the first sequence is then appended by the second part of the second sequence. The same applies in reverse for the other parts. In addition, the new population is extended by a sequence of randomly generated input sequences. In the next evolution step, the new population is again modified according to the search-based fuzzing technique followed by the input selection process. The genetic algorithm terminates after a certain amount of generations. For fuzzing, we either take the fittest sequence or a (sub)set of the population of the last generation.

The advantage of this technique is that the search space is larger since several traces are considered at the time. Furthermore, it is more likely to find traces with a higher fitness score. The disadvantage is that fuzzing takes more time since the fitness calculation has to be applied to each input sequence in the population in every generation.

9.3 Conclusion

This chapter introduced different black-box fuzzing concepts. One challenge in black-box fuzzing is to determine if the SUT has been tested in-depth. To overcome this issue, we presented model-based fuzzing techniques. The model builds the basis to provide coverage metrics but also supports the analysis of found issues. Our learning-based fuzzing framework showed that the required models can be automatically generated using automata learning techniques. In our framework, we use active learning for mining a behavioral model. Parts of the framework required for active learning can be reused for fuzzing the SUT. Especially, the structure of the mapper component can be modified for fuzzing, where abstract inputs are not only translated to valid concrete inputs but also to invalid values. Learning-based fuzzers implement conformance testing techniques to reveal behavioral differences between the learned model and the SUT.

We then presented different strategies for concretizing inputs and generating input sequences. For the concretization of inputs, we introduced a grammar-based fuzzing technique, where inputs are generated considering an underlying set of rules that define the language of the input format. Furthermore, we present different techniques to build a fuzzing test suite based on the learned model, the set of queries used for learning, and search-based techniques. In the next chapter, we evaluate the presented fuzzing techniques in different case studies on communication protocols.

(RQ 3.1) How can black-box fuzzing techniques be extended with automata learning?

We presented learning-based fuzzing as a novel fuzzing technique that combines automata learning and fuzz testing to develop a stateful black-box testing technique. We defined learning-based fuzzing as a two-step procedure in which we first learn a behavioral model and then use the model to fuzz a system. To create input sequences that reveal unexpected behavior, we introduced different approaches, including grammar-based, coverage-based, or search-based techniques.

Chapter 10

Case Studies on Learning-based Fuzzing

Declaration of Resources

This chapter provides case studies for the methods presented in Chapter 9. Hence, the resources stay the same. Section 10.1 is based on the paper *"Learning-Based Fuzzing of IoT Message Brokers"* [11]. Section 10.2 describes the case study that is presented in the paper *"Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning"* [148]. Sections 10.3 is based on the Master's Thesis of Benjamin Wunderling with the title *"Model Learning and Fuzzing of the IPsec-IKEv1 VPN Protocol"* which is co-supervised by the author of this thesis.

The following chapter presents different case studies on the learning-based fuzzing approach introduced in the previous chapter. First, we apply grammar-based fuzzing to test different implementations of MQTT brokers. Section 10.1 presents the results on fuzzing MQTT. In Section 10.2, we evaluate model-based fuzzing on BLE devices. As a last case study, Section 10.3 compares filtering-based, search-based, and genetic-based techniques for fuzzing VPN servers.

10.1 Grammar-based Fuzzing of MQTT

In this section, we introduce a learning-based fuzzing technique for MQTT brokers. The goal is to test different MQTT brokers for unexpected behavior using grammar-based fuzzing. Testing MQTT brokers is critical since they present a single point of failure in an MQTT network and are responsible for reliable communication. Hence, it is essential to test whether an MQTT broker does not introduce any reliability issues or security vulnerabilities.

Tappler et al. [170] showcased that learning deterministic models of different MQTT broker implementations is indeed possible. In their paper, they present that their learned models already reveal violations of the MQTT specification. Our fuzzing technique focuses on testing for security issues on the MQTT broker. For this purpose, we test if any unexpected concrete input values reveal such issues.

Figure 10.1 shows the general learning-based fuzzing setup for fuzzing MQTT brokers. Our fuzzing framework follows the two-step procedure of learning-based fuzzing. In this two-step procedure, we first learn the behavioral model of one SUL which represents an implementation of one MQTT broker. As discussed in the previous chapter, we learn the behavioral model on a more abstract level. The abstracted model is then used as the basis for fuzzing different SUTs, which represent different MQTT broker implementations.

Considering only one behavioral model for fuzzing comes with several advantages. First, the learning setup needs only be created for one SUL. Tappler et al. [170] outline that different

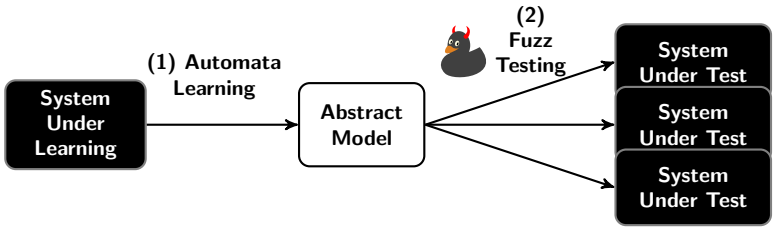


Figure 10.1: Learning-based fuzzing setup for fuzzing MQTT brokers. We first learn a behavioral model of one system. The learned model builds the basis for fuzzing different implementations.

configurations were required to learn deterministic models of the different MQTT broker implementations. We also presented in Chapter 8 that the learning runtime quite differs between the considered MQTT broker implementations, since the waiting time for responses is quite high for some brokers. In addition, results of previous work [122, 170] show that one broker implementation exists that best conforms to the MQTT specification [22]. Note that this does not imply that the SUL can be part of the set of tested SUTs, as the fuzzing is performed on the concrete packet level. Since a different concretization is used, we may observe different behavior.

All experiments for this case study have been performed on a MacBook Pro 2018 with an Intel Quad-Core i5 running at 2.3 GHz using 16 GB memory. The source code of the learning and fuzzing framework is also publicly available **online** [126].

10.1.1 Learning Setup for MQTT

For learning the behavioral model on an MQTT broker, we follow the learning setup provided by Tappler et al. [170]. Figure 10.2 illustrates the learning framework that consists of four components related to the automata learning framework presented in Chapter 4 and Chapter 5. The four components include a (1) learning algorithm, (2) a mapper, (3) an MQTT client and (4) an MQTT broker. The goal of this framework is to learn a behavioral model representing the MQTT broker.

Learning algorithm (1). Our learning-based fuzzing technique, considers always an active learning technique. Similar to Chapter 8, we reuse for fuzzing an MQTT client written in Java. Therefore, we prefer to use the same programming language throughout the learning framework. We use the automata learning library LearnLib [90] version 8, to learn the behavioral model. Considering the benchmarks for LearnLib performed by Aichernig et al. [15], we use the L^* algorithm [17] with the improvements proposed by Rivest and Schapire [156]. For equivalence testing during active learning, we used an equivalence oracle that is based on a finite number of random walks. During this random walk, we perform 3 000 steps on the SUL with a probability of 0.09 to reset the SUL to the initial state.

Mapper (2). To make learning feasible and to enable learning-based fuzzing, we require the model to represent an abstraction of an MQTT broker. Hence, we implemented a mapper component that translates the abstract inputs into concrete inputs and concrete outputs into ab-

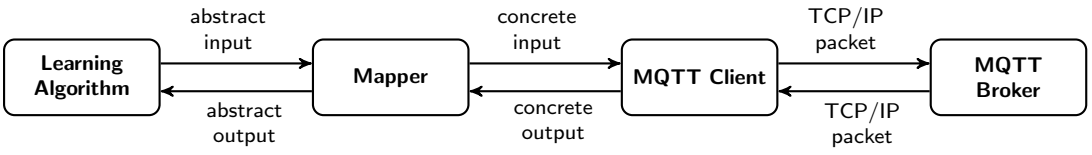


Figure 10.2: Learning framework for learning MQTT brokers.

stract outputs. The abstract input alphabet for learning is $I^{\mathcal{A}} = \{\mathsf{connect}, \mathsf{disconnect}, \mathsf{subscribe}, \mathsf{unsubscribe}, \mathsf{publish}, \mathsf{invalid}\}$, which is concretized by the mapper to the concrete input alphabet $I = \{\mathsf{connect}, \mathsf{disconnect}, \mathsf{subscribe}(\mathit{topicFilter}), \mathsf{unsubscribe}(\mathit{topicFilter}), \mathsf{publish}(\mathit{topicName}, \mathit{message})\}$. The fields *topicFilter*, *topicName*, and *message* are concretized by the mapper to a sequence of UTF-8 characters that are valid according to the MQTT specification. In contrast to the learning setup of Tappler et al. [170], we also consider an input invalid which represents an input, where the mapper concretizes the fields *topicFilter*, *topicName* with the UTF-8 character `U+0000`. We refer to the character `U+0000` as `NULL` character. The `NULL` character is according to the MQTT specification forbidden and should be rejected by an MQTT broker. For the output abstraction, we follow the translation provided by Tappler et al. [170], where the abstract outputs are $O^{\mathcal{A}} = \{\mathsf{connack}, \mathsf{conclosed}, \mathsf{puback}, \mathsf{suback}, \mathsf{unsuback}\}$ with the concrete inputs $O = \{\mathsf{connack}, \mathsf{conclosed}, \mathsf{puback\_publish}(\mathit{topicName}, \mathit{message}), \mathsf{suback}, \mathsf{unsuback}\}$.

The mapper is stateful in the sense that it stores the topics to which the client subscribes, to create deterministic behavior on publishing messages. For example, if a client subscribes to the topic filter `temperature/gf/+`, then the client publishes to the corresponding topic names, e.g., `temperature/gf/kitchen`.

MQTT client (3) The used MQTT client is equal to the one used in Chapter 8 which is based on the implementation of Tappler et al. [170]. The client enables communication with the MQTT broker. In our case study, we consider only MQTT brokers that support the MQTT standard v5.0 [22]. Hence, the implemented client also supports the MQTT v5.0 standard and is implemented in Java. For fuzzing and testing purposes the client does not perform any packet validation or sanitization. This enables to send and receive packets that do not conform to the MQTT standard. In contrast to the learning setup for MQTT presented in Chapter 8, we only consider one client for learning. Since the focus in fuzzing is more on providing unexpected inputs, the number of clients is assumed to be of less importance.

MQTT broker (4) The last component in the learning setup is the SUL, which is an MQTT broker implementation. For our learning-based fuzzing technique, we only learn one MQTT broker implementation. Tappler et al. [170] and Mladenov [122] found that the Eclipse Mosquitto MQTT conforms best to the MQTT specification compared to other investigated MQTT broker implementations. Thus, we only learn the behavioral model of the Eclipse Mosquitto broker.

Note that we decided not to reuse the learned models presented in the work of Tappler et al. [170] due to several reasons. First, at the time of the experiment execution, the broker implementations were updated to newer versions. In addition, also the MQTT standard changed from v3.1.1 to v5.0. Second, we consider a slightly different input alphabet that also models the behavior in case invalid inputs are provided.

Figure 10.3 depicts the learned model of the Eclipse Mosquitto MQTT broker interacting with one client. The model shows three different states: q_0, q_1, and q_2. In the initial state q_0 the client is not connected. If the client connects to the broker, we enter q_1. In this state, the client can publish messages which are acknowledged by the broker with a PUBACK message. If the client subscribes to a topic, we traverse to state q_2, where the client receives in addition to the acknowledgment of the published message, the published message itself, which is indicated by the output PUBACK_PUBLISH. We learned with the previously discussed learning setup. The used L^* variant required 114 output queries and one equivalence query to learn the model. Learning took approximately 260 seconds.

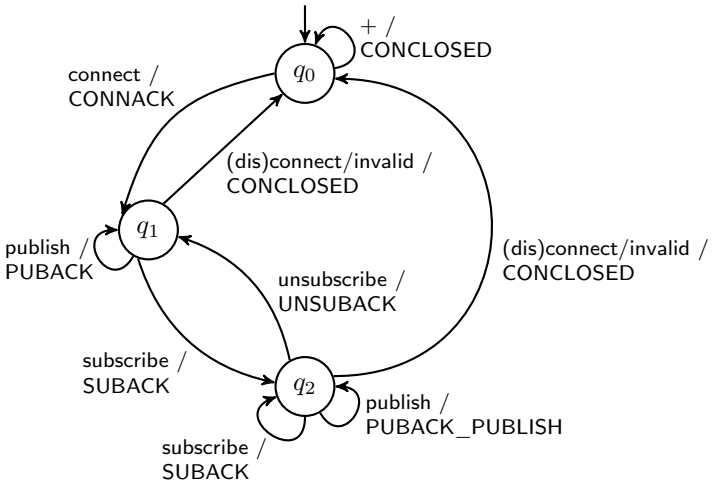


Figure 10.3: Learned behavioral model of the Eclipse Mosquitto MQTT broker. This model serves as a basis for our learning-based fuzzing different MQTT broker implementations. Note that some input actions are grouped by the '+' symbol.

10.1.2 Fuzzing setup

Based on the learned model of the Eclipse Mosquitto MQTT, we use learning-based fuzzing to fuzz the different MQTT broker implementations. The goal is to test, whether the MQTT parses the received fuzzed inputs correctly. Moreover, we want to check if the broker introduces any security vulnerabilities in an MQTT framework.

Figure 10.4 illustrates a possible attack of a malicious client that publishes to a topic that contains the invalid `NULL` character. An MQTT broker that conforms to the specification does not forward such a message to any subscribed client. The `NULL` is forbidden since some programming languages such as C use this character to delimit strings. If the client parses the received packet from the broker, the given packet length does not correspond to the length of the provided string, which can lead to possible attack scenarios.

For fuzzing, we simulate such a malicious client, where the fuzzing mapper is responsible for generating concrete inputs that contain unexpected and invalid characters. To fuzz such a scenario, we apply fuzzing on the application level, where we test the parsing of topic filters.

In Section 9.2.1, we explained grammar-based fuzzing on the example of a grammar that defines the language of topic filters as they are used in MQTT. We use Grammar 9.1 to generate invalid and unexpected topic filters and names, where topic names exclude the wildcard characters '`+`' and '`#`'.

The MQTT specification [22] defines several special cases for topics that should be rejected by an MQTT broker implementation. For example, Grammar 9.1 includes topics starting with `$SYS`. `$SYS`-topics should be considered invalid since they are used for internal broker communication. Hence, a client should not be able to mimic such internal communication. In learning, we already used topics that include the `NULL` character, which is prohibited by the MQTT specification.

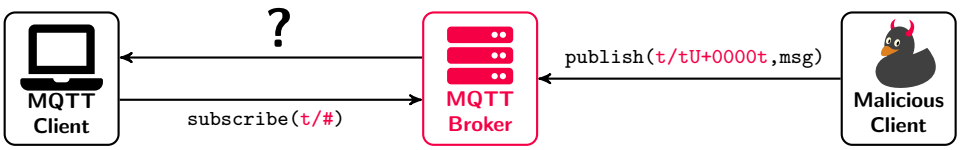


Figure 10.4: A malicious client publishing a message to an invalid topic filter. A correctly implemented MQTT broker, should not forward such an input to any subscribed clients.

Furthermore, the MQTT specification defines the UTF-8 characters from `U+0001` to `U+001F` as characters that should not be accepted in topic filters. For generating other unexpected inputs, we can simply adapt the set of UTF-8 characters to consider characters that are well-known in fuzzing to enforce unexpected behavior, such as characters for string format attacks.

We create our fuzzing test suite based on the same conformance testing technique used during learning. Hence, we generate the fuzzing test suite by randomly walking through the learned model. The learning interface can be adapted for fuzzing, where the client implementation can be reused and the mapper is replaced by the grammar-based fuzzing mapper. Hence, our learning-based fuzzing framework follows the framework presented in Figure 9.2. For the concretization of abstract inputs, the fuzzing mapper considers invalid and unexpected topic filters and names that are generated based on the provided grammar. As in conformance testing, we then compare if the received output conforms to the output defined in the model. If the outputs are different, we possibly found an issue that should be further investigated. To generate the random input sequences, we decided not to reuse the conformance testing technique provided by LearnLib. Instead, we implemented our own model-based testing framework that generates random input sequences. The custom implementation was required, to better log the executed fuzzing inputs and to deal with the case that the SUT crashes due to an unexpected input. Our conformance testing test suite for fuzzing comprises 1 000 conformance tests, where each conformance test includes 50 inputs. Each input is concretized with the fuzzing mapper, where only inputs with topic filters and names are fuzzed with unexpected characters.

10.1.3 Case Study Subjects

In our performed case study, we evaluate our grammar-based fuzzing technique by testing five different MQTT broker implementations. The following MQTT brokers with their corresponding versions have been considered:

- Eclipse Mosquitto 1.6.8[1],
- ejabberd 20.7.0,[2]
- EMQ X v4.0.0[3],
- HiveMQ 2020.2[4], and
- VerneMQ 1.11.0[5],

where all brokers support the MQTT v5.0 standard [22]. Note that the Eclipse Mosquitto broker is used as SUL as well as SUT. This is useful since the fuzzing mapper considers a different concretization of inputs than the mapper used in learning. All brokers run in a local network to avoid outside interference. Note that our setup allows us to run several fuzzing instances in parallel if the MQTT brokers are configured to listen to different ports in the local network. In Chapter 8, we mentioned that brokers take different amounts of time to respond. For fuzzing, we set the waiting time for a response to the time it takes the slowest broker to respond. For this, the waiting time is set to 200 milliseconds.

[1] https://mosquitto.org/
[2] https://www.ejabberd.im/
[3] https://github.com/emqx/emqx
[4] https://github.com/hivemq/hivemq-community-edition
[5] https://github.com/vernemq/vernemq

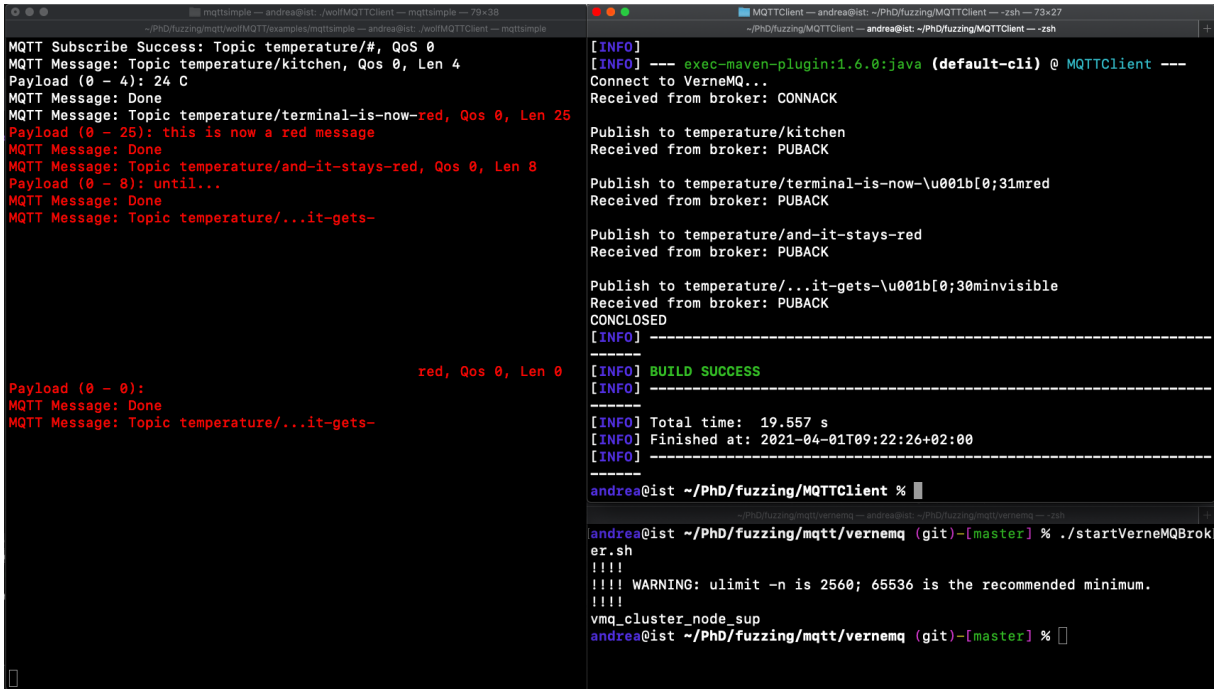


Figure 10.5: The screenshot shows several terminal windows. We see that a malicious client (upper right corner) can send UTF-8 control characters to the VerneMQ broker (lower right corner) and that the broker forwards these characters to a subscribed client (left side). We see that the text first becomes red and then becomes invisible due to the changed font color.

10.1.4 Results

The following section presents the results of our learning-based fuzzing technique. The results of our case study revealed that all tested MQTT brokers show non-conforming behavior to the learned model. The found behavioral differences hint at possible security vulnerabilities or show violations of the MQTT specification.

We divide the found issues and inconsistencies into three categories: (1) acceptance of invalid characters, (2) acceptance of topics beginning with the dollar symbol '$', and (3) inconsistent number of received publications. In the following, we describe the three categories in more detail.

Acceptance of invalid characters (1). In this category, we evaluate if the MQTT brokers accept characters that should not be accepted or must not be accepted. We found that three out of five tested MQTT brokers accept characters that should not be accepted according to the specification. The affected brokers are ejabberd, EMQ X, and VerneMQ. For example, these brokers accept publications and subscriptions to the topic `/te U+000A st`, where the UTF-8 character `U+000A` is in the range of non-recommended characters ranging from `U+0001` to `U+001F`.

That these characters might be harmful is shown in Figure 10.5. The screenshot depicts three terminal windows. In the left window, we run a publicly available MQTT client called wolfMQTT[6]. The terminal window in the upper right corner runs our custom MQTT client implementation which represents a malicious client. On the lower right, we run the VerneMQ MQTT broker, to which both clients are connected. The wolfMQTT client subscribes to the topic filter `temperature/#` and our malicious client publishes messages to the following topics:

1. `temperature/kitchen`

[6] https://github.com/wolfSSL/wolfMQTT

2. `temperature/terminal-is-now-` `U+001b` `[0;31m` `red`

3. `temperature/and-it-stays-red`

4. `temperature/and-it-gets-` `U+001b` `[0;30m` `invisible`

Figure 10.5 shows that the first message is normally displayed on the wolfMQTT windows. However, the second received topic name is different from the topic name used by our malicious client. In addition, the font color changed to red. The control sequence `U+001b` `[0;31m` can be used to change the font color in the terminal permanently. And the fourth input shows, that text can also be made invisible in case it is set to the background color of the terminal. This reveals that the affected broker forwards UTF-8 control characters to other clients.

More seriously, VerneMQ also distributes the prohibited `NULL` character. This issue can be exploited on clients in the MQTT network that are written in C. We can again showcase a possible attack scenario using the wolfMQTT client. A possible attack might include the following steps:

1. wolfMQTT & malicious client connect to VerneMQ

2. wolfMQTT subscribes to `test/#`

3. malicious client publishes to `test/te` `U+0000` `st`

4. wolfMQTT processes truncated topic name `test/te`

This can be dangerous because clients written in C are probably allocating memory according to the specified packet length. The displayed topic filter only shows a subset and an attacker can then load malicious code into memory unnoticed by the client. Thus, it appears that VerneMQ does not perform any topic validation or sanitization, which means that a malicious client can distribute arbitrary topic filters and names within an MQTT network.

We also found one inconsistency between the behavior of the brokers HiveMQ and Eclipse Mosquitto. In contrast to Eclipse Mosquitto, HiveMQ sends an additional disconnect message, which according to the MQTT specification is also valid. Hence, both brokers act according to the specification. This shows that still a manual investigation of found inconsistencies is required to assess whether the inconsistency is a real issue.

Acceptance of topics beginning with the dollar symbol '$' (2). The MQTT specification defines that clients should not be able to communicate over topics that begin with the '`$`' symbol. Additionally, the MQTT specification defines that topics starting with '`$SYS`' should be read-only for clients and publications on these topics are reserved for broker communication.

The results of our grammar-based fuzzing technique show that only HiveMQ conforms to the learned model. However, HiveMQ and VerneMQ conform to the MQTT specification since they allow subscriptions to topics beginning with '`$`', but neglect received publish message. However, their behavior in handling publish messages on such topics is different. HiveMQ disconnects the client, whereas VerneMQ keeps the connection alive, but does not distribute the message to other clients. The broker ejabberd does not strictly follow the MQTT specification by responding to such messages with an acknowledge message but the broker does not distribute them to other clients. We found that EMQ X and even the Eclipse Mosquitto broker, which was shown in other case studies on MQTT as the most confirming MQTT broker, violate the MQTT specification in this regard.

Figure 10.6 shows the extended model of the Eclipse Mosquitto broker including the behavior on topics beginning with '`$`'. We see that the broker does not treat publish messages on topics beginning with '`$`' differently than those on valid topics. Hence, a client can also publish messages on topics beginning with '`$`'. The same is possible for the EMQ X broker. However,

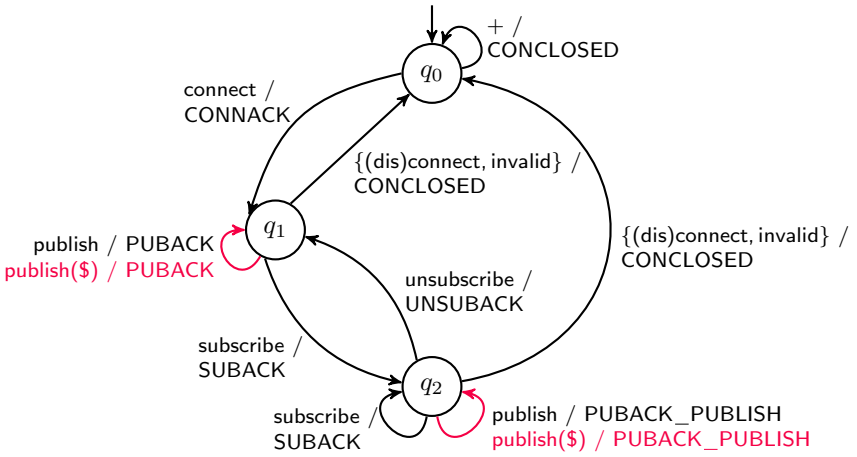


Figure 10.6: Behavioral model of the Eclipse Mosquitto MQTT broker that shows that clients can communicate via topics beginning with '$'. Note that some input actions are grouped by the '+' symbol.

there the behavior is even more alarming since this broker also allows clients to communicate via topics with the prefix '$SYS'. Hence, clients can simulate broker communication which may be security-critical in an MQTT network.

Inconsistent number of received publications (3). Our conformance testing technique that we applied for fuzzing found differences between the learned model and the broker HiveMQ regarding the number of received publish messages. If a client connects to a HiveMQ broker and subscribes to overlapping topic filters, e.g., '`temperature/#`' and '`temperature/gf/kitchen`', the client receives two publish messages if another client publishes to '`temperature/gf/kitchen`'. All other investigated MQTT brokers only send one message in this case.

Sending multiple publish messages does not violate the MQTT specification. However, since only one broker implements this behavior it enables the fingerprinting of this broker. This again shows that found differences to the learned model require a manual analysis against the specification to evaluate if they present an actual violation of the specification.

In summary, our conducted case study found violations to the MQTT specification in four out of five investigated MQTT brokers. The only broker that conforms to the MQTT specification was HiveMQ. We saw that also the broker that served as our SUL does not correspond to the MQTT specification. This shows that the chosen level of abstraction was sufficient to test the SUL. Furthermore, our fuzzing technique added value as the MQTT broker, which was classified to be correct in the previous work of Tappler et al. [170], was shown to violate the MQTT specification.

10.2 Model-based Fuzzing of BLE

The following section presents our case study on learning-based fuzzing of BLE devices. For learning-based fuzzing, we apply the model-based fuzzing as introduced in Section 9.2.2. In contrast to the case study on MQTT, the BLE devices heavily differ in their behavior as shown in Chapter 4. Consequently, model-based fuzzing tests for behavioral differences considering the individual models of each device. Following our case study on learning BLE devices presented in Chapter 4, we apply our learning-based fuzzing technique on peripheral devices. Furthermore, since many different BLE packets have been considered at different layers of the BLE stack, we do not focus only on fuzzing a single field with a particular structure. In contrast, we present a

more general learning-based fuzzing framework for testing BLE devices.

10.2.1 Learning Setup

The learning setup for learning BLE devices is described in Chapter 4. As learning algorithm, we consider the L^* algorithm for learning Mealy machines with the improvements proposed by Rivest and Schapire. This time in fuzzing, we consider the individually learned model of each device, since the models show a lot of difference in their implementation and supported features. Hence, for our proposed approach it is necessary to learn a behavioral model of each fuzzed device. Note that the models represent an abstraction, where the abstract input alphabet for learning is $I^{\mathcal{A}} = \mathsf{scan\_req}, \mathsf{connection\_req}, \mathsf{length\_req}, \mathsf{length\_rsp}, \mathsf{feature\_req}, \mathsf{feature\_rsp}, \mathsf{mtu\_req}, \mathsf{version\_req}, \mathsf{pairing\_req}$, where the $\mathsf{pairing\_req}$ implements the legacy-pairing request. As shown in the learning results of Chapter 4, some of the case study subjects could only be learned with a subset of $I^{\mathcal{A}}$. For learning the same mapper was used for learning the connection procedure presented in Section 4.2.3. In contrast to the learning setup of MQTT, we do not consider invalid inputs for learning.

10.2.2 Fuzzing Setup

For our fuzzing setup, we apply model-based fuzzing. Each SUT is tested for conformance with its learned model. The generation of the test suite follows the approach described in Section 9.2.2. The input sequences generated with this technique contain non-fuzzed inputs and one fuzzed input. For the concretization of non-fuzzed inputs, we simply use the concretization as it was implemented by the mapper component that we used for learning. For fuzzing inputs, we extended this mapper to also generate unexpected and invalid values.

The generation of fuzzed inputs is based on randomness. An abstract BLE input is translated to a concrete BLE packet that contains many different fields that need to be concretized. For example, the input $\mathsf{connection\_req}$ is translated to the BLE packet $\mathsf{BTLE}()/\mathsf{BTLE\_ADV}(\ldots)/\mathsf{BTLE\_CONNECT\_REQ}(\mathit{interval}, \mathit{timeout}, \ldots)$. The concrete BLE packet $\mathsf{BTLE\_CONNECT\_REQ}(\mathit{interval}, \mathit{timeout}, \ldots)$ has, e.g., the fields *interval* and *timeout*. The fuzzing mapper would then randomly choose one of the fields to concretize using fuzzing techniques. All other fields of the BLE packet are translated as it has been done during learning. For example, if the fuzzing mapper selects the field *timeout* for fuzzing, the mapper randomly selects a value within a range. The packet $\mathsf{BTLE\_CONNECT\_REQ}$ considers two bytes for the *timeout* field, hence the fuzzer concretizes the value by selecting a value between 0 and $2^{16} - 1$. The fuzzing mapper selects these values randomly, but a higher probability is given to the selection of boundary values, i.e., 0 or $2^{16} - 1$. According to the BLE specification, the timeout is used to assess whether a connection is lost. The BLE specification defines that the value must be set between 0.1 seconds and 32 seconds, which corresponds to the integer values between 10 and 3 200. Hence, the probability of choosing a value that is out of the specified range is high.

The set of considered abstract inputs for fuzzing is equal to one used for learning. The fuzzing mapper implements a fuzzing-based concretization for each field in each input that is based on the byte size of the corresponding field. For some fields, it makes sense to consider a predefined set of possible concretizations such as for the features provided in feature request or response. However, the set of possible values for all the fields could be easily derived from the packet manipulation library Scapy [158].

For fuzzing, we generate a test suite generating n_{fuzz} tests, which contain a random suffix of inputs with length n_{suffix}. Since our used conformance testing technique provides state coverage, we distribute the number of conformance tests equally in each state. In general, we want to perform at least 1 000 conformance tests. Hence, we execute for each state $\lceil \frac{1000}{|Q|} \rceil$ tests, which means that the actual number of executed conformance tests is $n_{\text{fuzz}} = \lceil \frac{1000}{|Q|} \rceil \cdot |Q|$. Furthermore,

Table 10.1: Results of the model-based fuzzing techniques for testing six BLE devices.

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

n_{suffix} is set to the number of states $|Q|$ of the currently considered model, but in case the model has less than five states n_{suffix} is at least set to five.

We calculate for each learned automaton the characterization set using the W-method [39, 186]. Since all models could be learned with L^* in one learning round, the characterization set is a subset or equal to the considered input alphabet. Hence, if a fuzzed sequence reveals an unexpected observation, it requires executing at most $|I^{\mathcal{A}}|$ additional input queries.

As discussed in Chapter 4, BLE communication might suffer from delayed or lost packets. This can lead to non-deterministic observations or failed connection attempts. Hence, we repeat the queries when we observe non-deterministic behavior or could not reset the system. The maximum number of non-deterministic errors as well as the maximum number of connection errors is set to 20. If we found unknown behavior on a fuzzed input, we repeat the query five times to be sure that the new observation did not occur due to non-deterministic behavior. In case the fuzzed input crashes the device, we terminate after observing 20 connection errors.

10.2.3 Evaluation

Framework. The model-based fuzzing framework was implemented in Python 3.9 in order to provide a needless integration with the learning framework and the driver software for the BLE central device. For conformance testing during fuzzing, we utilized the conformance testing techniques of the learning library AALpy. For our case study, we used a customized version of AALpy version 1.1.5, which was extended by a method to calculate the characterization set. This method was included starting from version v1.1.7. For parsing and creation of BLE packets, we use an adapted version of the Python library Scapy version 2.4.4., the adaptions are available starting from version 2.4.5. The model-based fuzzing framework and the learning framework are available **online** [145]. The repository also includes scripts to test and execute the found issues.

Case Study Subjects. For our case study on model-based fuzzing of BLE devices, we considered the same six BLE devices as in the evaluation presented in Section 4.3.

Environment Setup. All experiments have been conducted on an Apple MacBook Pro 2019 operating at 2.4 GHz on an Intel Core i5 and 8 GB RAM. As central device, we use the Nordic nRF52841 Dongle and the Nordic nRF52840 Development Kit, flashed with the firmware available on the SweynTooth [67] repository. This firmware allows to send custom BLE packets, which are required for fuzzing the peripheral device.

10.2.4 Results

Table 10.1 shows the results of our model-based fuzzing approach for BLE devices. The table lists the results for every investigated SUT. In case a special setup was used for learning and fuzzing it is also indicated. For example, since it was not possible to learn a deterministic

model of the CC2640R2 considering the whole input alphabet, we learned three different models with an input alphabet reduced by one input. The three models are then individually fuzzed with the corresponding reduced input alphabet. The devices indicated suffixed by '*' mark setups, where learning starts after an established connection since for these devices it was not possible to reliably establish several consecutive connections. We will discuss this problem for the CC2652R1 in more detail later. For the devices marked by '*', we did not consider the scan and connection request for fuzzing. For orientation, Table 10.1 also provides the number of states of the learned models, which serve as a base for model-based fuzzing.

Table 10.1 presents the number of performed fuzzing rounds. A fuzzing round indicates one attempt to execute n_{fuzz} conformance tests. If the device crashes while executing the conformance tests, the current fuzzing round is aborted. We declare a crash as a state of the tested device, where the device becomes unreachable. In case of a crash, the device stops sending advertisements and no BLE requests can be sent to the device to reset it to the advertising state. Thus, the device needs to be hard reset. In practice, this vulnerability can be exploited by a malicious device to make a device unreachable.

In case of a crash within the execution of our experiments, the cause for the crash is identified and if it refers to a specific fuzzed field, the field is excluded in the next fuzzing round. If no cause for the crash can be found, the conformance testing is simply restarted without any changes. In case the device crashes two consecutive times and the reason cannot be determined, no further fuzzing attempts are started.

The fuzzing results show that our fuzzing technique crashes four out of six investigated devices. More seriously, fuzzing crashes in every attempt the CC2652R1, and the CC2640R2 in the setup where it includes the pairing request. Note that the CC2652R1 states more crashes than fuzzing attempts. This is possible since we already discovered a scenario that the device becomes unreachable in learning the behavioral device. Initially, we tried to learn the model considering the whole input alphabet. This was not possible since the device becomes unreachable after performing two consecutive connection requests. Hence, the following sequence already caused a crash on the CC2652R1:

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

Table 10.1 also presents the sum of executed queries during all conformance testing attempts. The number of performed queries includes also queries that were repeated due to non-deterministic observations or connection errors. Furthermore, queries were also performed to validate found counterexamples and to determine the target state of an unknown transition. The table also provides the number of counterexamples found during conformance testing. Note that a high number of counterexamples does not necessarily imply that the SUT does not conform to the BLE specification. It rather states that the SUT implements some error-recovery mechanisms when receiving unexpected or invalid input.

The median runtime of all fuzzing attempts without crashing was 6.3 hours, where the minimum was 3.7 hours for the CC2640R2 (no pairing_req) and the maximum 42.2 hours for the nRF52832. This long runtime of nRF52832 corresponds with the also disproportionately long runtime for learning and the rather large number of counterexamples found, leading to a large number of queries performed. The determination that a counterexample is valid also includes several repetitions of the same query. Smart approaches to provide fault-tolerant fuzzing might improve the runtime of the model-based fuzzing technique.

Table 10.2 presents the issues found during fuzzing. The issues and behavioral anomalies were revealed by a manual analysis of the fuzzing logs in case the device crashes or a counterexample to the conformance between the learned model and the output for fuzzing is provided. This analysis filters out error recovery behavior which is assumed to be normal in case an invalid input is provided. Table 10.2 enumerates the found issues and anomalies in three categories and also lists which SoCs are affected. The three categories differentiate between crash scenarios

Table 10.2: The found issues that are revealed by our model-based fuzzing technique for BLE devices.

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

(C), behavioral anomalies (A), and security vulnerabilities (V). Every entry is indexed by the letter representing the category, e.g., 'C' for crash scenarios followed by an increasing numerical index.

Crash scenarios (C). All the detected crash scenarios are detected by fuzzing the connection request.

(C1) As explained above, the crashing scenario C1 was already detected during learning of the CC2652R1 and is due to the execution of an additional connection request in case a connection is already established. We contacted Texas Instruments Inc. about this issue and they explained to us that the preinstalled application stops sending advertisements if two consecutive connection requests are performed. However, the issue is that it does not return to the advertising state even if both clients disconnect. This issue can be fixed in the code of the running application.

(C2-C4) We found out that three devices crash on invalid values if we fuzz the fields *interval*, *timeout*, or *latency*. Where CC2640R2 and CC2650 crashed on fuzzed values for all these fields, CYBLE-416045-02 crashed for fuzzed values of the field *interval*. This issue has already been published for the CC2640R2 in CVE-2019-19193 [1], and is based on the BLE fuzzing results of Garbelini et al. [66]. Our results show that this vulnerability also applies to other SoCs as the CYBLE-416045-02 manufactured by Cypress Semiconductor Corporation is also affected by this issue. All manufacturers have been contacted about the found issues and we assumed the issues to be fixed in newer firmware versions.

Behavioral anomalies (A). Behavioral anomalies describe found behavior where our fuzzing approach or a manual analysis shows that a particular behavior is different from all other considered devices. Table 10.2 shows that the found anomalies only apply to one of the devices, which shows that these anomalies can also be used to fingerprint these devices. A1 and A2 refer to anomalies that are already detected by analyzing the learned model. A3 and A4 are found via our fuzzing method.

(A1) As already reported in Chapter 4, we found that CC2652R1 always responds to version requests, even if the response is only allowed once according to the BLE specification.

(A2) The nRF52832 shows a very restrictive behavior when it receives an unexpected length response. In this case, the nRF52832 terminates the currently established connection. Other devices ignore an unrequested response and keep the connection alive.

(A3) Our fuzzing technique also revealed another behavioral anomaly for the CC2652R1. We explored an unknown state in the CC2652R1 when fuzzing either the field *max_tx_bytes* or *max_rx_bytes* in the inputs length_req and length_rsp. If the fields are set to an invalid

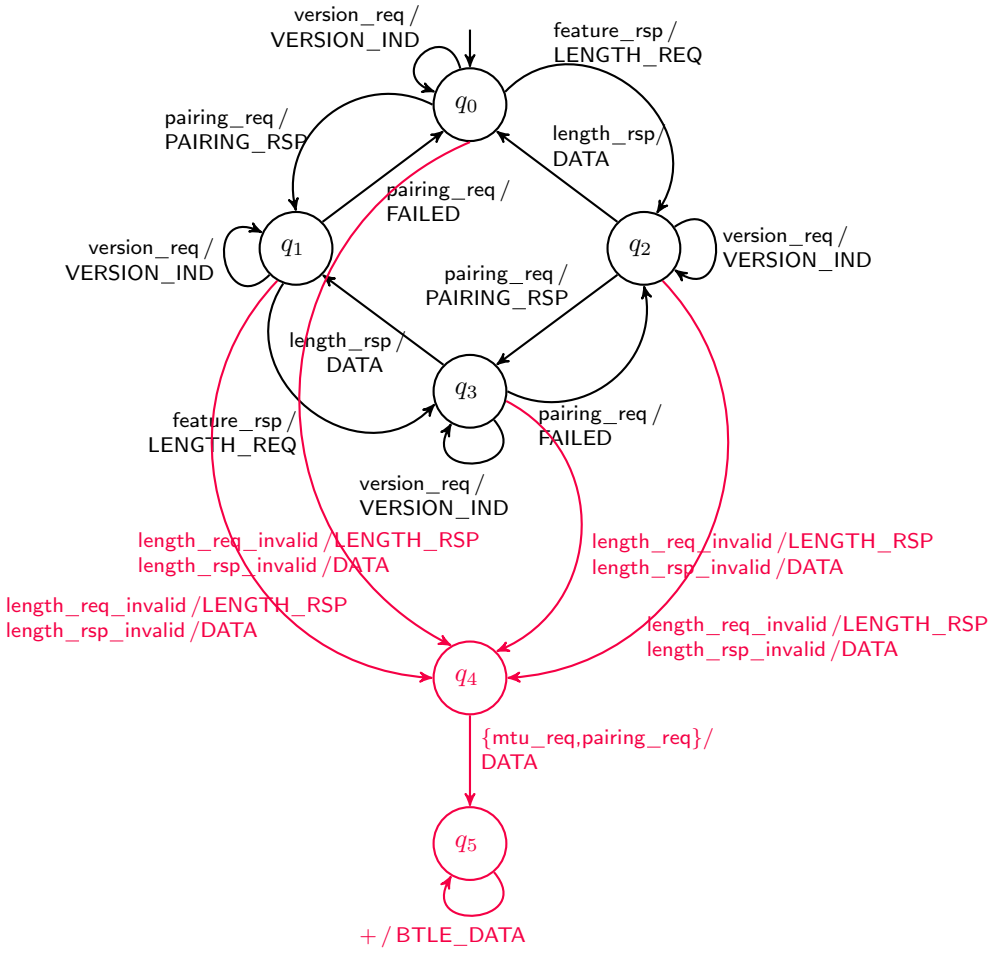


Figure 10.7: Partially extended model of the CC2652R1 that shows that an unexpected length_req or length_rsp immediately followed by a valid mtu_req and pairing_req reveals an unknown state.

value, e.g., 0, and then immediately afterwards a valid mtu_req and pairing_req is performed the systems enters an unknown deadlock state, where only BTLE_DATA packets for any further requests are received. Figure 10.7 depicts the partially extended model of the CC2652R1 that shows how the previously unknown states can be reached. The model includes the state q_5 which cannot be left by any other request.

(A4) We also found an anomaly in the CYW43455 which accepts pairing requests where the maximum key length is set bigger than 16. According to the BLE specification, the maximum key length should be defined between 7 and 16.

Security vulnerabilities (V). We also found one possible security issue in all devices that accept pairing requests, i.e., all devices except the CYBLE-416045-02. The devices accepted values between 7 and 16 as the key length in the pairing request. Note that these values conform to the BLE specification, but they still provide a security issue since shorter keys, especially if they only have 7 bytes, can be easier brute-forced. The National Institute of Standards (NIST) [24] also considers that key lengths of less or equal than 10 bytes are not secure. Antonioli et al. [19] showed that this vulnerability can be exploited by key downgrade attacks.

10.3 Fuzzing of VPN

We have successfully fuzzed MQTT brokers with grammar-based fuzzing and BLE devices with model-based fuzzing. As a last step, we want to compare the filter-based, search-based and genetic-based fuzzing techniques in a case study on testing VPN servers. More specifically, we fuzz IPsec IKEv1 implementations, since the key exchange for encryption must not introduce any security vulnerabilities.

In the performed comparison, we compare our three learning-based fuzzing approaches against each other and a baseline. The baseline represents fuzzing with randomly generated sequences. We also show in this case study that our learning-based fuzzing techniques can be extended by using already existing fuzzing libraries to create concrete values. For mining the models, we extend the learning setup that we presented in Chapter 5 of the two learned IPsec-IKEv1 server implementations.

10.3.1 Learning Setup

For this learning-based fuzzing approach on IKEv1 servers, we reuse the learning setup as presented in Chapter 5. The considered SULs remain the same including their configuration. Thus, we learn and then fuzz the IKEv1 implementation of the strongSwan and libreswan server.

We learn the behavioral models of both IPsec servers, since the brokers behave differently. We learned the models by filtering out retransmitted packets. Filtering-out retransmission was enabled due to two reasons. First, we can reliably learn a model that serves as a basis for fuzzing. Second, we aim to avoid false positive counterexamples during testing the conformance between the learned model and the SUT.

For fuzzing, we learn new behavioral models using an extended input alphabet compared to the one used in Chapter 5. For each input in the previously considered input alphabet, we add an erroneous counterpart. Erroneous inputs are inputs where the mapper concretizes the input by using invalid values. This is similar to the approach used for MQTT, but this time each input also has an erroneous version. The following abstract input alphabet is used to learn models that serve as a basis for learning-based fuzzing: $I^{\mathcal{A}} = \{\mathsf{main\_sa}, \mathsf{main\_key\_ex}, \mathsf{authenticate}, \mathsf{quick\_sa}, \mathsf{quick\_ack}, \mathsf{main\_sa\_err}, \mathsf{main\_key\_ex\_err}, \mathsf{authenticate\_err}, \mathsf{quick\_sa\_err}, \mathsf{quick\_ack\_err}\}$. The increased size of the input alphabet implies that more queries and input steps are required for

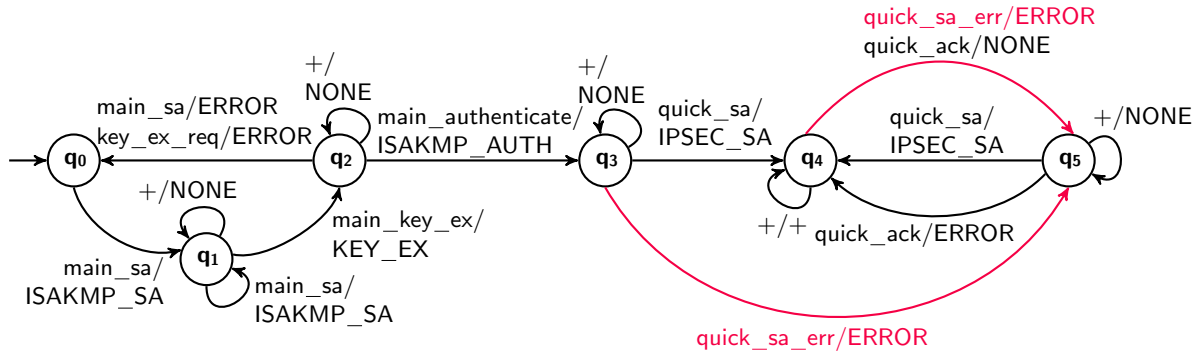


Figure 10.8: Learned model for fuzzing the strongSwan server. Note that some inputs and outputs are grouped by the '+' symbol.

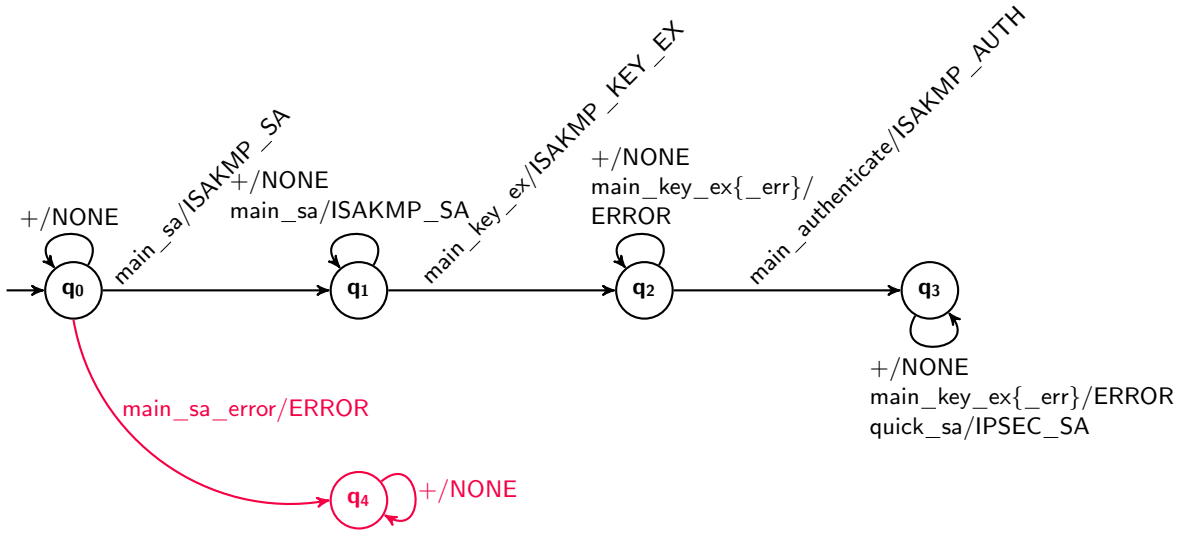


Figure 10.9: Learned model for fuzzing the libreswan implementation. Note that some inputs and outputs are grouped by the '+' symbol.

learning. For learning, we used the KV learning algorithm with the Rivest and Schapire [156] counterexample processing and all caching mechanisms enabled as described in Chapter 3.

Figure 10.8 shows the learned behavioral model of the strongSwan IKEv1 implementation. We see that the model has the same number of states and most parts are similar to the base model shown in Figure 5.4. The erroneous inputs usually lead to self-transitions that do not influence the current state of the server. However, there exists one exception on performing a quick_sa_error message in State q_3. According to the protocol specification, the client sends a quick_sa message, where the server responds with an IPSEC_SA message. The client then acknowledges this response by a quick_ack message. However, if the client sends a quick_sa_error message, the server responds with an error. To complete the quick mode, the client needs to send another valid quick_sa message.

We repeated learning of the strongSwan model eleven times and it took on average 24.1 minutes to learn the model within four to five learning rounds, where 174 output queries and 60 conformance tests were performed.

Figure 10.9 shows the learned model of the libreswan server. The learned model of the libreswan IKEv1 implementation defines questionable behavior. In contrast, to the model shown in Chapter 5 (Figure 5.5) the model learned with erroneous inputs defines an additional state (q_4). This additional state is immediately reached after performing a main_sa_err message.

This state cannot be left by any input, since this state has no outgoing transitions. Thus, it represents a deadlock state. Further manual analysis revealed the causes of this behavior. In the configuration of the LIBRESWAN server, we defined that clients connecting with the same identification number do not trigger a new connection. Instead, the connection is assigned to the already connected client with the same identification number. For LIBRESWAN, using this setup implies that it is impossible to recover from an initially send main_sa_err message. Hence, resending a main_sa message does not result in a valid connection.

10.3.2 Fuzzing Setup

For fuzzing IPSec servers, we follow a conformance testing approach where we test for behavioral differences between the SUT and the learned model. For fuzzing STRONGSWAN and LIBRESWAN, learning-based fuzzing is based on the corresponding learned models presented in the previous subsection.

For fuzzing an input sequence, we select one input at a time that should be fuzzed. The concretization of the input is then performed by a fuzzing mapper, similar to the fuzzing techniques of MQTT and BLE. IKEv1 packets consider different fields that must be concretized, where we fuzz one field at a time. An input is fuzzed several times by choosing different fields and concrete values for the selected fuzzed input. Hence, in contrast to BLE, fuzzing an abstract input for IPsec yields several concrete input sequences that should be executed. Furthermore, we repeat fuzzing of the same input sequence by selecting different inputs in the sequence.

In our fuzzing evaluation, we consider four different fuzzing setups: (1) filter-based fuzzing, (2) search-based fuzzing, (3) genetic-based fuzzing and (4) random fuzzing. We introduced the approaches (1)-(3) in Chapter 9. We also consider random fuzzing (4) to evaluate if generating random sequences is sufficient to find the same issues as our more elaborate methods. In the following, we present the results that compare the different fuzzing techniques on the STRONGSWAN server. Due to the more complicated resetting procedure, we then only fuzzed the LIBRESWAN server with the most successful technique from our evaluation of the STRONGSWAN server.

Filter-based fuzzing (1). Our filter-based method uses the input sequences performed during the learning of the model. This set provides state and transition coverage, which we assume to be beneficial in a black-box scenario. However, since the set of queries can be large, we filter the set only for interesting input sequences.

We took the set that was generated during our application of the KV learning algorithm. The considered set of queries from learning with KV includes 220 queries. Hence, we consider 220 queries as a basis for our filter-based technique. By applying filtering as explained in Section 9.2.3, we could reduce the set to 55 queries. For each input sequence, we pick two to three inputs that should be fuzzed. For each input, we concretize up to three fields of the IKEv1 packet. For one selected field, we consider three different concretizations to enable fast filtering. After this filtering step, we fuzz the SUT again with the filtered test suite. In this second round of fuzzing, we fuzz the same inputs of the sequences more thoroughly. Therefore, we generate about 250 concretizations for fuzzing one input.

Search-based fuzzing (2). For search-based fuzzing, we started with an empty sequence and mutate this sequence according to Algorithm 10 for 50 iterations. To evaluate the fitness score, fuzzing considers only a few concretizations. The concretization technique is similar to the one used in the filter-based approach. For STRONGSWAN, our search-based fuzzer generated after 50 iterations the following sequence with eight inputs

$$\mathsf{sa\_main} \cdot \mathsf{key\_ex\_main} \cdot \mathsf{sa\_main\_err} \cdot \mathsf{sa\_main} \cdot \\ \mathsf{sa\_main\_err} \cdot \mathsf{sa\_main\_err} \cdot \mathsf{sa\_main\_err} \cdot \mathsf{authenticate\_err}.$$

In a second fuzzing step, this sequence is then again executed on the SUT. This time, however, with a more thorough fuzzing, where each input of the sequence, one after another, with several concretizations is fuzzed.

Genetic-based fuzzing (3). For our genetic-based algorithm, we set the parameters to ensure that the approach remains comparable to the search-based technique. Hence, we consider only five generations and each population consists of ten input sequences. The initial population is generated by random input sequences of length three. For mutating the sequence, we apply to each sequence two mutations at a time. The next population of the next generations consists of the three fittest individuals of the current population. We choose randomly two of the three fittest sequences to generate two further sequences using crossover as described in Section 9.2.4. The remaining five traces are generated by simply random input generation. The length of the random sequence is randomly selected with an upper bound $|s_I^{\text{long}}|+1$, where s_I^{long} is the longest input sequence in the population. To score the population, we again use fewer concretizations to allow fast fuzzing.

For fuzzing strongSwan, the fittest sequence in the last generation has 18 inputs. Similar to search-based fuzzing, we use the input sequence with the highest fitness score to fuzz the system more thoroughly a second time.

Random fuzzing (4). The random input sequences serve as a baseline to evaluate if a simple random sequence could find the same issues as the techniques (1)-(3). For this purpose, we generate random input sequences considering the given input alphabet with a length similar to the sequences generated by the search-based and genetic-based techniques. After the generation of a random sequence, we then fuzz each input of the sequence.

For all four discussed techniques, we applied the same technique to concretize abstract inputs. For this purpose, we take advantage of already existing fuzzing tools, as they proved their success in revealing issues in other systems. We used the network protocol fuzzing library boofuzz [142], which comprises techniques for generative and mutative fuzzing. For generative fuzzing predefined packet structures are required. However, since our mapper component already considers the underlying packet structures, we only use the mutative component. The mutative fuzzer of boofuzz generates besides valid and boundary values also concrete values for a given datatype that are known to be harmful to systems.

10.3.3 Environmental Setup

The environmental setup stays the same as for learning VPN servers as presented in Chapter 5. We use two VMs, where the one that simulates the client also runs the learning-based fuzzing framework. The second VM runs the corresponding IKEv1 servers. Both VMs are simulated in VirtualBox 6.1 running with Ubuntu 22.04 LTS and use 4 GB RAM and one CPU core. Our SUTs are the strongSwan U5.9.5/K5.15.0-25-generic and libreswan U3.32/K5.15.0-41-generic. For fuzzing, we use boofuzz v0.4.1.

10.3.4 Results

The following section presents the results of comparing different fuzzing techniques for the strongSwan server. Afterwards, the section presents the found violations of the IKEv1 specification for strongSwan and libreswan.

Table 10.3: The different fitness scores of the input sequences for fuzzing the strongSwan server. The fitness scores are calculated according to Equation 9.2. The higher the fitness value, the better it is for fuzzing.

	Fitness Score
Random Sequence (length 8)	0.054
Random Sequence (length 18)	0.392
Filtering (avg)	1.471
Search	2.813
Genetic	4.700

Table 10.4: Runtime statistics of the different fuzzing techniques for the VPN case study.

	Filtering	Search	Genetic
# Input sequences	55	1	1
Input sequence length	14	8	18
Seconds/input sequence	14	8	18
# Executed tests	192 500	2800	4500
Runtime (h)	60	26.5	31.6

Comparison of Fuzzing Techniques

Table 10.3 shows the fitness scores of the four different methods. For the filtering-based approach, we selected randomly 20 input sequences from the filtered test suite and calculated the average fitness value for the selected subset. Our results show, that the genetic technique achieved the highest fitness score, where the presented score is taken from the fittest input sequence of the last generation.

The second-highest fitness score was achieved by the search-based technique. Compared to the genetic-based approach, we see that genetic-based fuzzing achieved a score that is 1.67 times higher than the search-based technique. This shows that it is worth mutating multiple sequences in parallel to observe more new behaviors and achieve higher state coverage.

The filtering-based technique has the third-highest fitness score. All our proposed fuzzing techniques achieve a significantly higher fitness score than simple random generation. This shows that our methods are more effective in revealing unexpected behavior while better traversing the state space of the SUT.

Regarding the time required to generate the fuzzing sequence, the search-based approach took 19.5 hours followed by the genetic-based approach that took 15.6 hours and the filtering-based approach took 7 hours. Note that even searching one sequence took quite some time, the parameters of the other two approaches were set up in such a way that they can compete with the search-based approach. For example, to filter the set of queries, we do not fuzz all inputs of a sequence and apply fewer concretizations for one input. Furthermore, our genetic approach evolves the set of sequences only over five generations. Moreover, the fittest sequence found by our genetic-based approach was more than twice as long as the sequence for search-based fuzzing. Longer sequences have the disadvantage that they require significantly more time for fuzzing.

Table 10.4 shows the fuzzing statistic for the filter-based, search-based, and genetic-based fuzzing approaches. For our filter-based approach, we see that even if we only fuzz two or three inputs for each input sequence in the filtered set, we still execute a huge number of concrete input sequences. On average, we fuzzed for each input 283.80 different values. Since executing one concrete input sequence takes approximately one second per input, longer sequences also increase the runtime accordingly. For the filter-based approach, this sums up to a total runtime of 60 hours, which means that fuzzing took 2.5 days. We see that fuzzing only one sequence takes quite less time. Fuzzing the longer sequence of the genetic approach took 31.6 hours. At 26.5 hours the search-based approach took the least amount of time.

Bug Hunt

Our investigated fuzzing techniques found specification violations in both tested IKEv1 implementations. We found two issues: (1) a missing ISAKMP header length check and (2) a delayed isakmp authentication validation. Issue (1) was found in both SUTs, whereas (2) was only observable for strongSwan.

ISAKMP header length check (1). All inputs from the main mode translate into ISAKMP packets. The header of the ISAKMP packet includes a field for defining the length of the packet. According to RFC 2408 [182] that defines the ISAKMP, the provided length should correspond to the actual length of the packet (header plus payload) and the message must be rejected if the provided length does not correspond. However, we see for strongSwan and for libreswan that they both ignore this field. Hence, any value between 0 and $2^{32} - 1$ can be set as payload length in the ISAKMP header, where, e.g., 0 is obviously invalid. Both IKEv1 implementations accept such messages and do not respond with an error indication. In practice, this means that a malicious client can set any value for this field. This could be beneficial for a malicious when designing an attack.

Delayed ISAKMP authentication validation (2). The second issue was found when fuzzing the sa_main packet. The main_sa packet concretizes to a ISAKMP packet containing SAs. A single SA contains a field *authentication*, which specifies the used type of authentication, e.g., pre-shared keys (PSKs). For the strongSwan server, we observed that also invalid values can be provided and the server will still accept the SA proposal. The server responds later on a main_key_ex input with an error identification due to the invalid *authentication* value in the previous message. This behavior is especially suspicious since fuzzing any other field of an SA proposal with an invalid value lead to immediate error notifications. Furthermore, the RFC 2409 [33] defines that an invalid proposal should be immediately rejected. This issue shows that in fuzzing faults are not always immediately observable. However, our model-based approach supports the localization of faults, since the observed behavior can be compared to the behavior defined in the model.

Note that both issues for strongSwan were found with the filter-based, search-based, and genetic-based fuzzing techniques. However, the randomly generated input sequences could not reveal the found issues. For libreswan, we only tested search-based fuzzing, which was successful to find the issue regarding the length field in the ISAKMP header.

10.4 Conclusion

We applied our developed learning-based fuzzing techniques that we presented in Chapter 9 to case studies on communication protocols. Our achieved results show that our techniques are indeed successful in finding violations of the specification and possible security issues. We found with our grammar-based fuzzing technique that four out of five investigated MQTT brokers violate the MQTT specification. The found issues reveal possible attack scenarios, where malicious clients can forward possible harmful characters or mimic internal broker communication. For learning-based fuzzing of BLE devices, we applied our model-based fuzzing technique. This technique found that four out of six devices crash on fuzzed inputs. Furthermore, we found with fuzzing previously unknown behavior and show that is possible to decrease the level of security in the pairing procedure. Finally, this chapter compares different learning-based fuzzing techniques in a case study on VPN servers. We showed that our filter-based, search-based and genetic-based fuzzing techniques were successful in revealing two violations of the IKEv1 specification. The comparison indicates that search-based approaches are useful to generate input sequences for

fuzzing that can be efficiently executed but also sufficiently reveal unknown behavior. The case study on VPN also shows that already existing fuzzing tools, such as BOOFUZZ [142], can be used to extend our learning-based fuzzing framework in order to generate interesting inputs. Thus, learning-based fuzzing represents a powerful tool for stateful black-box fuzzing.

(RQ 3.2) Is learning-based fuzzing effective at revealing security issues?

In the presented case study, we evaluated the proposed learning-based fuzzing technique of Chapter 9. Our evaluation considered three popular communication protocols: BLE, MQTT, and IPsec-IKEv1. In all investigated protocols, we found violations of the corresponding protocol specifications. Our fuzzing technique uncovered that malicious messages could be forwarded to other components in the network and that we could mimic server-internal communication. Furthermore, we showed that the key length of the encryption key could be reduced, lowering the level of security. Our fuzzing technique also revealed reliability issues in the tested BLE devices.

Chapter 11

Related Work

The following chapter discusses related work and compares it to the approaches presented in this thesis. The chapter divides the related work into five categories, with each category represented by a distinct section. First, Section 11.1 presents automata learning applications for learning different communication protocols, followed by a discussion on improvements of automata learning frameworks in practice in Section 11.2. In Section 11.3, we provide an overview of RNN-based automata learning techniques. Then, Section 11.4 lists related work in the field of black-box fuzzing, focusing on the communication protocols covered in this thesis. Finally, Section 11.5 provides some other related work in the context of BLE protocol analysis.

11.1 Learning Communication Protocols

Automata learning has been successfully applied to learn behavioral models of communication protocols. In general, protocol implementations represent reactive systems that can be modeled by, e.g., Mealy machines. With the introduction of learning algorithms for Mealy machines by Margaria et al. [113], and Shahbaz and Groz [162], automata learning became a popular tool for reverse engineering behavioral models of communication protocols. Furthermore, Damasceno and Strüber [46] motivate the idea to use automata learning to fingerprint reactive black-box systems such as communication protocols. The literature provides several case studies on learning different protocols. In the following, we list a selection of case studies that are closely related to the work presented in this thesis. For this purpose, a particular focus is on applications for fingerprinting and security testing.

11.1.1 Learning-based Testing

Learning-based testing is a method to test systems by learning a behavioral model using active automata learning techniques. Aichernig et al. [8] provides a survey that categorizes the work on learning-based testing based on its purposes such as conformance testing or security testing. For conformance testing purposes, the learned models are used to reveal behavioral differences to a specification. Their survey points out that there also exist techniques that utilize conformance testing to compare learned models of different SULs for equivalence, i.e., automata learning is used for differential testing [115]. Such an approach has been formalized and evaluated by Aarts et al. [6]. We also find applications of this learning-based conformance testing technique for different communication protocols. Argyros et al. [20] presents a framework for actively learning symbolic finite automata of Transmission Control Protocol (TCP) implementations, web application firewalls, and HTML/JavaScript parsers. Their presented framework includes the analysis of behavioral differences between the learned models. Similar to our method, they stress that differences between the models can be used for fingerprinting implementations. Another case study is presented by Tappler et al. [170], where they learned behavioral models

of MQTT brokers. For learning, they followed an active approach using the learning library LearnLib [90]. They compared the learned model with each other, where behavioral differences possibly witness violations of the MQTT specification. In their work, they investigated five MQTT brokers and found four brokers that violate the MQTT specification. Only, Eclipse Mosquitto does not reveal any inconsistencies to the MQTT specification. Similar to the case study of Tappler et al. [170], our case study on learning-based fuzzing includes the Eclipse Mosquitto and the VerneMQ broker. In contrast to their results, our fuzzing technique also revealed inconsistencies to the specification in the Eclipse Mosquitto implementation.

Aarts et al. [3, 5] showed that learning-based testing is also applicable for physical devices. In their work, they learned implementations of communication protocols on bank cards [3] and on biometric passports [5]. For both techniques, they use a card reader to execute the queries from the learner on the SUT. All models were learned on a more abstract level, using a mapper component similar to the one introduced in Section 2.2.2. For learning, they applied the L^* algorithm for Mealy machines implemented in the learning library LearnLib. After learning, they performed a manual analysis comparing the learned models with each other and with a specification model, if available. To validate the learned models of biometric passports [5], they used JTorX [25] to test whether the **ioco** conformance relation between the learned model and a provided reference model holds. The **ioco** conformance relation can be used to test for output inclusion between an implementation and a specification, and is formally defined by Tretmans [179, 180].

11.1.2 Protocol State Fuzzing

Learning-based testing proved to be a useful tool for reverse-engineering and testing communication protocol implementations of black-box systems. Testing for specification violations is especially important when the protocol implements security-critical behavior. Hossen et al. [83] outlines also the possibility of model learning to investigate security flaws in protocols. Thus, it is not surprising that the literature provides several case studies on learning-based testing security-critical protocols. These techniques are often referred to as *protocol state fuzzing*.

Closely related to our work on learning VPN servers, Daniel et al. [47] learned behavioral models of OpenVPN servers. In contrast to our approach, they consider several layers of the protocol, where the key exchange is abstracted to a single input. Similar to our approach they experienced multiple challenges in the generation of a learning setup that are discussed in more detail in the Master's thesis of Novickis [134].

De Ruiter and Poll [50] used protocol state fuzzing to analyze implementations of the TLS protocol for security violations. Similar to other learning-based testing case studies, they applied the L^* implementation in LearnLib to learn behavioral models. The authors then manually analyzed for suspicious transitions in the model that could cause security issues. They revealed security issues in three out of nine tested implementations. In addition to the work on BLE and VPN presented in Chapter 4 and Chapter 5, a case study on DTLS [63] demonstrates the success of protocol state fuzzing. The case study on DTLS investigates thirteen implementations and found several security flaws. For example, one of the learned models shows the possibility to bypass an authentication procedure. Sivakorn et al. [163] propose also a framework that uses automata learning to analyze and test the hostname verification of different SSL/TLS implementations. Their framework found several violations in the tested implementations. Furthermore, their evaluation shows that their automata learning technique achieves better code coverage than black-box and coverage-guided gray-box fuzzing techniques. Further case studies present that other protocols such as QUIC [154] can also be learned by active automata learning techniques.

In our case study on BLE, we showed that protocol state fuzzing is also feasible for the learning of physical devices. However, applying automata learning on physical devices comes with various challenges and requires a robust learning interface to deal with these kinds of challenges such as packet loss or delayed packets. Similar challenges were also reported in the

case study presented by McMahon Stone et al. [168] on learning the 802.11 4-Way Handshake implementations on Wi-Fi routers. However, the disclosure of three novel security vulnerabilities by McMahon Stone et al. shows that a possibly tedious learning setup pays off.

Learning physical devices becomes challenging if buttons or keys must be pressed to trigger an input. Chalupar et al. [37] showed how a learning interface can be built using Lego® to query a smartcard reader. Based on this interface, active learning can be used to learn a behavioral model of a smartcard reader. In Chapter 4, we also presented a learning setup that uses Lego® that keeps the SUL in motion to prevent the device from entering a sleep mode.

Learning-based testing has also been proposed as a toolkit component for automotive security testing by Ebrahimi et al. [55]. Their work includes case studies on learning wireless protocols like Bluetooth Classic and BLE, with the BLE approach following our technique presented in Chapter 4. With the support of the author of this thesis, they also learned a behavior model of the Unified Diagnostic Services (UDS) protocol implemented on an electronic control unit. The jointly learned model discloses a security vulnerability.

Learning models of communication protocols can also be useful for applying further analysis techniques. Fiterău-Broştean et al. [61] used the actively learned models of TCP protocol implementations as the basis for model checking. For model checking, they manually derived properties from the RFC and checked them on the learned model using NuSMV [40]. To do so, the learned model must be translated into a model that enables model checking via NuSMV. Fiterău-Broştean et al. [62] applied a similar approach for learning and model checking SSH implementations. Closest to our work in this regard is Guo et al. [80]. They learned behavioral models of IPsec-IKEv2 implementations. Subsequently, they use the learned models to verify properties using NuSMV.

11.1.3 Passive Automata Learning

All techniques discussed so far for learning behavioral models of communication protocols consider active automata learning approaches. The advantage of active techniques is that the state exploration simultaneously tests the SUL. However, this also implies the requirement that an active interface can be established. In contrast, passive automata learning does not require such an interface but still allows the reverse engineering of behavioral models from a given sample, e.g., log files. Therefore, passive learning might be a promising technique when traces can be monitored or are available, but establishing an active interface is not feasible. In the following, we provide examples of passive learning for network protocols.

Cui et al. [45] cluster similar messages in a given sample of network traffic to learn finite state models of network protocols. Their case study includes the following protocols: Hypertext Transfer Protocol (HTTP), Remote Procedure Call (RPC), and Server Message Block (SMB). In their work, they find that their generated models are larger than necessary and need to be minimized. They also point out the limited behavioral coverage of their sample. Therefore, their learned models are unlikely to sufficiently generalize the behavior of the SUL.

Hsu et al. [86] passively learn the finite state machine of the Microsoft MSN instant messaging (MSNIM) protocol by applying a state merging algorithm that is akin to the RPNI algorithm. Similar to our learning-based fuzzing technique discussed in Chapter 9, they then use the learned model as the baseline for model-based fuzzing. Their model-based fuzzing technique generates traces that target transition coverage. However, since their learned model does not define every input in every state, it is more difficult to detect any unexpected behavior on undefined inputs than is the case with our actively learned models. Moreover, the approach misses a statement about the quality of the passively learned model in terms of behavioral coverage of the underlying black-box system. Therefore, the coverage achieved later can only be as good as the coverage provided by the given sample used for learning. Nevertheless, their approach was successful to reveal crash scenarios in the MSNIM implementations.

To filter data for passive learning of communication protocols, Comparetti et al. [43] proposed

a framework that selects the sample based on its impact on the system behavior. They then group similar messages to create an appropriate abstraction. The filtered sample is then used to passively learn a behavioral model applying existing state merging techniques. With this technique, they learn behavioral models of a chatbot protocol, Simple Mail Transfer Protocol (SMTP), SMB, and Session Initiation Protocol (SIP). Even though their proposed tool filters messages, the quality of the sample in terms of behavioral coverage of the SUL is not discussed. As our results in Chapter 6 show, passive learning requires a large sample to achieve the same state coverage as active learning.

11.2 Improvements for Automata Learning in Practice

The case studies on learning communication protocols show that the reverse engineering of behavioral models is useful for the analysis of black-box systems. However, automata learning has practical applications not only in learning communication protocols. Aichernig et al. [9] showed that learning-based testing can also be applied in industry. In their case study, they present a learning framework for learning a measurement device that is used in the automotive industry. However, their case study also showed that the learning setup was not straightforward and required an enhanced adaption of the learning interface. To this end, they extend the model to include sink states that are reached upon detecting non-det behavior.

In particular, the application of active automata learning for learning physical systems usually requires a sophisticated setup to enable robust learning. Chapter 4 presents a framework for learning wireless protocols. As discussed earlier, such a setup presents several challenges that require different countermeasures to make learning feasible. In the literature, we find similar approaches to enable learning from physical systems.

11.2.1 Alphabet Abstraction

The first challenge in learning real systems is the size of SUL. For example, when learning communication protocols, the protocols use messages that contain arbitrary byte streams. Considering all possible messages would make learning infeasible in an appropriate amount of time. For this purpose, we abstract the input and output alphabet that is considered in learning. Section 2.2.2 introduces the mapper concept originally proposed for automata learning by Cho et al. [38]. In their work, they learned behavioral models of botnet protocols. To concretize and abstract the alphabet they deployed existing tools to generate valid messages, and then manually created the mapping. In general, mappers are not new in testing, e.g. in model-based testing they bridge the level of abstraction between a given test model and the SUT [151]. Aarts et al. [7] formally define the mapper as a transducer following the definition of Mohri [123]. Aarts et al. demonstrate the practicability of their presented mapper component with a case study of learning an abstract model of SIP and TCP implementations. The use of such an abstraction is common for learning communication protocols, as demonstrated in the case studies on TCP [61], MQTT [170], or DTLS [63]. We also use such an abstraction component for learning MQTT brokers, BLE devices, and VPN servers. The concept of the mapper component is also used in the first level abstraction for learning abstracted non-deterministic systems as explained in Chapter 8. In addition, the learning-based fuzzing methods, introduced in Chapter 9, reuse the structure of the mapper to generate inputs for fuzzing.

11.2.2 Algorithmic Improvements

Abstraction techniques reduce the size of the observable state space of the SUL. However, for the application of automata learning, the goal is to keep the number of interactions with the SUL as low as possible without compromising the expressiveness of the learned model. We presented several techniques to reduce the number of executed queries on the SUL in Chapter 3.

In addition, Chapter 6 compares the efficiency of systematic querying with random sampling. The problem of reducing the number of queries formed the baseline for different competitions on active automata learning. The Zulu competition [42] asks for learning approaches that can learn systems solely by asking a limited amount of membership queries. Thus, equivalence queries are not provided and must be substituted by further membership queries. Howar et al. [85] present the winning solution and outline goals for future challenges, e.g., focus on learning real systems. The successor to the Zulu challenge is the Rigorous Examination of Reactive Systems (RERS) challenge [91], which asks for testing approaches for analyzing reactive systems.

Berg et al. [26] provide insights into Angluin's [17] L^* algorithm based on different metrics. For this purpose, they investigate the number of required queries, the runtime, and the memory consumption required by L^*. Furthermore, they propose an adapted version of L^* that includes improvements to learn prefix-closed languages. In particular, their evaluation on real systems shows that the prefix-closed assumption is sufficient for modeling reactive systems and helps to reduce the number of membership queries. In their work, they assume to have access to an equivalence oracle. However, this assumption does not hold in practice. Considering the aspect of the absence of an equivalence oracle, Aichernig et al. [15] evaluated different conformance testing techniques that substitute the equivalence oracle. In their evaluation, they combined the testing techniques with different learning algorithms to assess the influence of the chosen conformance testing technique on the learning algorithm. They also extended a passive learning algorithm for active learning as proposed by Walkinshaw et al. [189]. Similar to the results presented in Chapter 6, their results show that random querying requires large samples to learn a model that sufficiently defines the SUL. This applies especially when only random samples are provided to the active version of the passive learning algorithm. Another result of their conducted case study is that counterexample processing is beneficial for the number of queries required in active learning. Their observation correlates with our results for our improved version of the KV algorithm that implements the counterexamples processing of Rivest and Schapire [156]. Finally, they recommend using a randomized version of the partial W-method [65] for conformance testing or at least some coverage-based methods. As learning algorithm, they recommmend to use either ADT [64], TTT [89], or L^* [17] with the improvements proposed by Rivest and Schapire [156]. Our case studies followed this recommendation by using the improved version of L^* and a coverage-based conformance testing technique as a substitution for the equivalence oracle. However, our improved version of KV also shows promising results and should be considered as part of the evaluation performed by Aichernig et al. [15].

11.2.3 Choice of Modeling Formalism

Our presented case studies on learning real systems show that not only an efficient learning algorithm is required to make learning applicable in practice. Other countermeasures are needed to deal with environmental conditions such as packet loss or delayed messages. One possibility to overcome this issue is to develop an advanced learning interface. This is consistent with the common procedure for learning real systems as described in the literature [9, 50, 61, 62, 168, 170] and by us in Chapter 4 and Chapter 5. Another approach is to consider a different modeling formalism that can model behavioral aspects such as non-deterministic, timed, or stochastic behavior.

In Chapter 8, we used a non-deterministic learning algorithm to learn MQTT brokers. El-Fakih et al. [56] propose another L^*-based learning algorithm for learning ONFSMs. However, they assume that the teacher provides all possible outputs at once. To make the approach feasible in practice, they assume that all outputs can be observed after a finite number of repetitions of an output query. Similar approaches for learning ONFSMs are proposed by Pacharoen et al. [140], and Khalili and Tacchella [97]. Pacharoen et al. [140], however, take the number of repetitions that are required to observe all outputs into account when analyzing the runtime of their proposed learning algorithm. Khalili and Tacchella [97] evaluated their approach on

a practical case study on learning a non-deterministic model of a Trivial File Transfer Protocol (TFTP) server. In our case study on learning non-deterministic models of MQTT brokers, we found that learning is possible by repeating queries for a certain number of times. However, we weaken the assumption that all observations must be observable after a finite number of repetitions. To do this, we adapt the learning algorithm so that it can handle additional observations that are observed at a later time. Learning non-deterministic systems has still the disadvantage of requiring a large number of queries to be executed on the SUL. To overcome this drawback, we added an additional layer of abstraction to keep the model small. Bolling et al. [29] used a similar concept for their learning algorithm, where they model deterministic systems with a non-deterministic modeling formalism.

The literature also provides learning algorithms for learning timed or stochastic behavior. The learning algorithms for timed and stochastic behavior can be divided into passive [32, 110, 111, 171, 174, 188] and active [10, 76, 77, 172, 173, 184] techniques. Considering these additional behavioral characteristics increases the complexity of learning and limits its feasibility. For example, due to the infinite state space of timed systems, it is especially for active approaches important that the number of required queries is kept at a feasible level. For this purpose, Aichernig et al. [10] propose a search-based technique to learn timed automata. For learning stochastic systems, the literature also provides active [172, 173] and passive [32, 110, 111] techniques.

11.2.4 Alternative Assumptions for Learning

Our presented learning frameworks require that we can reset the SUL and that unexpected behavior due to environmental conditions can be identified and filtered out by our learning interface. We also discussed the costs of the countermeasures in terms of the additional interaction required to meet these assumptions. In the literature, we find learning algorithms that implement different solutions to overcome these assumptions. Rivest and Schapire [156] propose a learning algorithm that does not require resetting the SUL. Groz et al. [79] present another reset-free learning algorithm for learning Mealy machines. The issue of these learning reset-free learning algorithms is that they require to execute a *homing sequence* that enables the identification of individual states. Similar to executing a reliable reset, the execution of homing sequences must be reliable. This can be problematic for systems that occasionally exhibit non-deterministic behavior, since the algorithm assumes to have reached a state in the SUL that does not correspond to the actual state of the SUL. Therefore, the execution of homing sequences requires the same countermeasures as a reliable reset.

To deal with occasional non-deterministic behavior due to environmental conditions, there also exist learning algorithms [109, 159, 183] that can handle this type of behavior. In the literature, samples are called *noisy* if the sample contains data that does not reflect the actual or common behavior of the system. All learning algorithms for noisy data operate passively, applying different techniques like state-merging [159], evolutionary algorithms [109], or encoding as SAT problem [183]. From a theoretical point of view, Angluin and Laird [18] investigate the amount of noise that a sample might contain in order to learn a probably approximately correct (PAC) [185] model. For active learning, Khmelnitsky et al. [99] investigate the influence of different kinds of noisy data on the L^* algorithm, where the equivalence oracle is approximated with random samples that are large enough to provide PAC guarantees. Using learning algorithms that can handle noisy data is indeed an interesting future direction in learning real systems.

11.3 RNN-based Learning Approaches

Passive learning techniques are required when a reliable interface that allows active interaction cannot be established or causes too much overhead in learning. However, in our comparison of active and passive learning algorithms presented in Chapter 6, we found that passive learning requires a large random sample to compete with active approaches. In practice, this may justify the overhead active learning algorithms require to handle unexpected behavior due to environmental conditions.

Another angle to address this problem is to develop learning algorithms that better generalize on a sparse sample. For this purpose, we presented an RNN-based learning algorithm in Chapter 7. In the literature, we find several approaches that utilize RNNs in the context of automata learning. The approaches can be divided into three categories: (1) training of RNNs to simulate finite automata, (2) inferring finite automata from trained RNNs, (3) training RNNs to predict the structure and behavior of a finite automaton.

First RNN-based techniques can be assigned to Category (1). Already in 1956, Kleene [101] investigated the suitability of neural networks to simulate finite automata that accept regular languages. Later in 1967, Minsky [121] followed by presenting a general methodology to construct a neural network to simulate finite automata.

It took some decades to bring this topic back to life by approaches that can be assigned to Category (2). Omlin and Giles [137] infer DFAs from RNNs that accepts regular languages. Their approach is based on clustering hidden states of the RNN, where each cluster represents a state in the DFA. Dong et al. [53] uses the clustering approach of Omlin and Giles [137] to learn Markov chains. Tiňo and Šajda [177] proposed a different clustering technique for hidden states based on self-organizing maps to infer Mealy machines. An analysis by Michalenko et al. [118] shows that hidden state clusters do not necessarily allow an exact mapping to states in a deterministic automaton. However, they showed that sufficient encoding could be found for an abstraction of the automaton.

There also exist several techniques to generate finite automata from RNNs by applying active automata learning techniques. Weiss et al. [190] apply the L^* algorithm to query the RNN to extract a DFA. For the implementation of the equivalence oracle, they followed the clustering idea proposed by Omlin and Giles [137] to compare the learned hypothesis with the RNN. In subsequent work, this L^*-based approach has been extended for weighted automata [191] and context-free languages [200]. Mayr and Yovine [114] learn DFAs from trained RNNs by applying L^*, where the equivalence oracle is based on PAC sampling. Muškardin et al. [130] compared different conformance testing techniques for the equivalence oracle in learning DFAs from trained RNNs. They recommend model-based testing techniques that take the underlying structure of the model into account. For active learning, they used the learning library AALpy [129]. The initiation of the TAYSIR competition [58] also indicates the increasing interest in finding solutions for extracting finite automata from trained RNNs. The winner [125] of the first edition of this competition is based on our active automaton learning library AALpy [129]. Khmelnitsky et al. [98] showed that these automata learning approaches can be used to verify RNNs. For this purpose, they first learn a behavioral model from the RNN using active automata learning and then they apply model checking to very properties based on the learned models.

Category (3) considers techniques that train RNNs, given specific regularization constraints, to infer structural information about the underlying finite automaton. The RNN-based learning framework presented in Chapter 7 also falls into this category. Oliva and Lago-Fernández [136] propose an approach that is closely related to our technique. To regularize the RNN, they consider noise in the activation function of the hidden layer to enforce binary predictions. Similar to our approach their network simulates a regular language. Unlike our technique, they do not predict the next state, but perform training that enforces dense clusters of the hidden states, where the clusters correspond to states in the model.

11.4 Black-box Fuzzing

Fuzzing has its origin in the testing of UNIX utilities [119]. From then on, the success story of fuzzing began with the disclosure of numerous bugs in different software applications. The success of fuzzing is due to its simple and fast applicability. Nowadays, there exist several tools [69, 72, 142, 201] that allow immediate fuzzing of a system. In the following, we will only discuss related black-box fuzzing techniques of the investigated protocols in this thesis. For a general classification and explanation of fuzzing, we refer to the "*The Fuzzing Book*" by Zeller et al. [202] and to the survey by Godefroid [71].

To guide black-box fuzzing, we provided a behavioral model of the SUT. The literature lists several fuzzing tools for communication protocols that are model-based [23, 69, 92]. The fuzzers "*GitLab Protocol Fuzzer Community Edition*" [69] and SNOOZE [23] apply black-box fuzzing based on a given model. The tool T-Fuzz [92] extracts the model during compile time of the SUT. Therefore, the framework is not applicable in a black-box setting.

In the literature, we also find model-based fuzzing techniques targeted to a specific communication protocol. Garbelini et al. [66] apply model-based fuzzing to BLE devices. By applying this technique, they found several BLE issues, where they call the presented collection Sweyntooth. As a successor, Garbelini et al. [68] create a similar framework for Bluetooth Classic. Their model-based fuzzing technique was again successful in creating a collection of found Bluetooth flaws assigned to BRAKTOOTH collection. Their framework forms the base for our learning-based fuzzing approach for BLE devices that we presented in Section 10.2. In contrast to their work, we use automata learning to automatically generate a behavioral model of each BLE device, whereas Garbelini et al. manually created one general model. The disadvantage of generating a general model for BLE is that the BLE specification [87] is underspecified in some parts. Hence, a general model also needs to cover all possibilities for underspecified behavior. The usage of a general model could hamper the identification of behavioral differences due to unexpected inputs. Furthermore, individual models allow a more precise coverage measurement for model-based fuzzing.

In general, manually created models need to be continuously updated, and the manual modeling process can be error-prone. Therefore, learning is preferable when it is feasible. Doupé et al. [54] fuzz web applications based on inferred models. For this purpose, they first crawl the web application under test. Based on the crawled traces, which are a set of HTTP command sequences, they then generate a model using state-merging techniques. In the last step, they use the model to generate sequences that are fuzzed with existing tools, similar to our fuzzing technique for VPN servers. As discussed in the previous section, Comparetti et al. [43] proposed a general framework that passively learns behavioral models from different communication protocols. They then show that the learned models can serve as an input for the model-based black-box fuzzer Peach, which is the predecessor of "*GitLab Protocol Fuzzer Community Edition*" [69].

Black-box fuzzers like boofuzz [142] represent frameworks that are specialized to fuzz communication protocols. They require as input a syntactic definition of the underlying protocol packets that should be fuzzed. These syntactic definitions are often called templates. In addition, these fuzzers frequently apply mutative fuzzing techniques to insert invalid or unexpected inputs. For fuzzing VPN servers, we used boofuzz [142], but only to mutate inputs. We did not provide templates since our fuzzing mapper already considers the structure of the individual packets to generate fuzzed inputs. Another mutative black-box fuzzer for VPN, more specifically for IKEv1, was proposed by Yang et al. [197]. They provide structural templates for the packets, but also include a database of known vulnerabilities to create interesting fuzzing test cases. Similar to our approach, they found in the tested implementations that the ISAKMP header length field can be set to zero without rejecting the packet. In contrast to our results, they do not report the missing validation of the header length field for the Strongswan server, which was also part of their considered implementations under test. Tsankov et al. [181] propose

another black-box fuzzer for IKEv1 implementations. For their fuzzer, they manually derive a set of constraints from the IKEv1 specification, which built the baseline to generate invalid and valid inputs. With their approach, they found a vulnerability in the OpenSwan implementation which was the successor of libreSwan, which we investigated in this thesis.

Black-box fuzzing is also a popular tool to test MQTT protocol implementations. For doing so, there exist publicly available tools like mqtt_fuzz [44] or "*Eclipse MQTT test suite*" [94]. In the literature, we also find work on fuzzing the MQTT protocol. Ramos et al. [153] present a black-box approach that requires templates similar to the fuzzer boofuzz [142]. Mutation-based fuzzing is then used to generate test cases based on the provided templates for MQTT brokers and clients. Casteur et al. [34] also present a black-box fuzzing technique for MQTT brokers, they add a scoring technique that serves as an indicator for observed unusual behavior. Palmieri et al. [141] propose a general assessment tool for MQTT brokers. The tool generates a PDF-file that reports the results of the various performed security analysis techniques. One of the applied techniques also includes fuzzing, where akin to our method malicious/forbidden topic filters are tested, e.g., topic filters starting with `$SYS/`. In difference to our technique, the existing techniques rather test for possible denial of service attacks. In contrast to our technique for model-based fuzzing of MQTT brokers, the provided techniques do not provide any behavioral coverage metrics. Furthermore, unexpected state transitions are more difficult to determine without any notion of states. Sochor et al. [165] also present a black-box testing framework for MQTT brokers. Based on attack patterns, they generate a test suite that is executed on different MQTT broker implementations. Similar to our learning-based fuzzing technique, they used the Eclipse Mosquitto broker as a reference implementation. In difference to our fuzzer, their testing technique assumes that the Eclipse Mosquitto broker conforms to the MQTT specification, where our results show that this is not the case. There also exists a gray-box fuzzer for MQTT: Zeng et al. [204] propose a coverage-based fuzzer that uses a custom socket implementation to interact directly with the SUT, which enables the simultaneous establishment of multiple connections and faster fuzzing. However, such an approach is only applicable if the MQTT broker socket could be instrumented to directly execute commands.

One challenge in fuzzing is that a large amount of inputs are usually executed on the SUT. Especially for real devices, executing a fuzzing test suite takes some time, often several days. To speed up fuzzing, Ruge et al. [157] present a framework called Frankenstein that enables BLE fuzzing of a locally emulated version of the BLE firmware. However, this technique requires the elaborative design and development of a framework that enables emulation.

11.5 Other Related Work

There also exists related work on the analysis and testing of the BLE protocol that can neither be assigned to automata learning nor fuzzing. Celosia and Cunche [36] proposed an approach for fingerprinting BLE devices. For fingerprinting, they collected data on the Generic Attribute (GATT) profile layer of the BLE stack. For their performed case study, they collect over five months over 13 000 distinct profiles. Their results show that many devices leak privacy-critical information. Compared to our approach for fingerprinting BLE devices, they only investigate the data of one layer of the BLE stack. The GATT profile provides information about the offered service and other characteristics, hence fingerprinting is solely based on these data and not on the actual BLE stack implementation.

That the Bluetooth protocol is an interesting target for security analysis is underlined by the number of published collections of attacks and vulnerabilities. We have already discussed the model-based fuzzers for BLE and Bluetooth Classic that created the vulnerability collections SweynTooth [66] and BRAKTOOTH [68] respectively. Furthermore, the BLE fuzzing tool Frankenstein [157] found memory corruptions that enable, e.g., write access to the memory. Published attacks like BlueBorne [160] and BLEEDINGBIT [161] demonstrate that Bluetooth

attacks can be distributed unnoticed over the air and enable the remote control of vulnerable devices. Our fuzzing results also show that some devices are vulnerable to downgrades of the encryption key length as it is shown by the KNOB attack [19]. Wu et al. [195] demonstrate that actual security vulnerabilities can be found by analyzing the Bluetooth specification. Their work shows that an attacker can mimic a BLE device in such a way that the other party thinks it is a trusted device. All these results on BLE motivate (automated) techniques to verify protocol implementations.

Chapter 12

Conclusion

This thesis presented a holistic evaluation of the suitability of automata learning for the testing and analysis of communication protocols in networked systems. A special focus of this evaluation was to assess whether automata learning can support the security analysis of the investigated black-box systems. For this purpose, we applied automata learning to learn behavioral models of communication protocol implementations. We outlined observed challenges in doing so and evaluated alternative learning techniques. As a last step, we showed how automata learning can successfully support security testing techniques to reveal security issues.

The following chapter summarizes our results and concludes the thesis with a final discussion of the proposed research questions. Finally, we provide an outlook on future work.

12.1 Summary

This thesis investigated whether automata learning can support the security analysis of networked environments. This evaluation focused on the feasibility of the proposed methods. The goal was to outline security analysis techniques that successfully can be translated to other network components following the proposed methods. To achieve this goal, we (1) learned protocols, (2) discussed alternatives to overcome challenges, and finally (3) proposed learning-based security testing techniques. We structure the summary respectively.

12.1.1 Learning Communication Protocols

As a first step, we investigated if automata learning can be used to learn actual protocol implementations on physical devices. For this purpose, we investigated the Bluetooth Low Energy (BLE) protocol, which is a popular communication protocol in the Internet of Things (IoT) for short-distance communication. For our case study on learning BLE devices, we followed a protocol state fuzzing approach. To actively query the BLE devices, we proposed a learning framework that considered additional hardware that allowed us to send custom BLE packets to the system under learning (SUL). Learning a wireless communication protocol introduced several challenges such as lost or delayed packets. Furthermore, we had to deal with non-deterministic observations. We created a learning interface that allowed us to react to such unexpected behavior during learning.

We learned the behavioral models of eight different BLE devices. The considered devices included system on the chips (SoCs) from manufacturers such as Texas Instruments, Cypress, or Nordic Semiconductor. The investigated devices also included BLE chips on popular hardware such as the Raspberry Pi. Additionally, we showed that our learning framework could be applied to learn behavioral models of BLE devices that are installed in a Tesla Model 3 and the key fob of the Tesla Model 3 and Y.

The BLE case study provided several insights into the specifics of the BLE protocol implementations. We observed that some input sequences led to reliability issues in the investigated devices. Moreover, one learned model showed a violation of the BLE specification. For eight investigated BLE devices, we learned seven different models for the connection procedure. Only the two models in the Tesla components were equal to each other. This showed that the learned models could be used to fingerprint BLE devices.

In another case study, we investigated the IPsec Internet Key Exchange (IPsec-IKEv1) protocol, which is used in VPN protocol implementations to establish an encrypted communication. Thus, these protocol implementations must not introduce any security issues. Our case study on learning two different VPN server implementations showed again that unexpected input sequences led to unexpected observations. For this case study, we compared the performance of two learning algorithms: an improved version of L^* and an improved version of the KV algorithm. We observed that KV can help to reduce the number of queries especially if L^* requires several learning rounds. The exhaustive querying of active learning also revealed a security issue in the used Python library that implements the Diffie-Hellman key exchange.

12.1.2 Alternative Techniques for Automata Learning

Network environments in practice often consider a multi-client setup. We also approached this problem in this thesis. For this purpose, we tried to learn the Message Queuing Telemetry Transport (MQTT) protocol in a multi-client setup following a similar technique as proposed in the other learning setups, but failed for most of the implementation due to non-deterministic observations. To overcome this problem, we proposed an active learning algorithm for learning abstracted non-deterministic systems. Our developed learning algorithm extends the classic L^* algorithm by further levels of abstraction for the considered input and output alphabet. Using this approach, we managed to learn underspecifications of MQTT broker implementations that interact with multiple clients.

In practice, creating an active learning setup might not always be feasible or a tedious process to establish. Therefore, we also investigated passive learning techniques. First, we evaluated classic state-merging-based techniques based on random samples. The passively learned models achieve high accuracy in our performed conformance test if we align the random sample size to the number of queries that active learning requires. However, in most cases, passive learning did not manage to learn the correct minimal automaton.

The problem with passive learning techniques that are solely based on state merging is that they might not generalize well enough if the provided data misses behavior. We investigated if machine learning could overcome this challenge by generalizing better on sparse data. We proposed an recurrent neural network (RNN) architecture that predicts the behavior and the structure of a Mealy machine. For the training of the RNN model, we introduced a specific regularization term that enforces deterministic predictions of output and state behavior. To learn minimal automata, we iteratively reduce the considered upper bound for states. The evaluation of our RNN-based learning technique showed that this technique worked and achieved promising results also on random data especially if the considered input and output alphabet is rather small. For larger systems, the algorithm did not always terminate with the given budget of resources or learned a wrong generalization. However, we found a setup for all of the considered examples that enabled us to learn the correct model.

12.1.3 Learning-based Security Testing Techniques

We showed that different automata learning techniques managed to successfully learn behavioral models of communication protocol implementations. Our case studies on learning communication protocols already revealed violations of the corresponding protocol specification.

For testing further security issues, we combined automata learning and fuzzing techniques. We created a stateful black-box fuzzing technique which we denoted as learning-based fuzzing. We defined learning-based fuzzing in a two-step procedure: in the first step, we learn the behavioral model using automata learning techniques, and in the second step, we apply a model-based fuzzing technique. We proposed a fuzzing framework that took advantage of the performed abstraction that was required for learning a behavioral model. Instead of concretizing abstract input into valid input, we considered also invalid and unexpected concretizations during fuzzing.

Our model-based fuzzing technique followed a conformance-testing approach, where we used fuzzing techniques to generate the corresponding test suite. To generate this fuzzing test suite, we proposed different techniques in order to reveal unexpected behavior. For the fuzzing of the MQTT protocol, we proposed a grammar-based fuzzing technique that generates inputs based on a provided set of rules that describe the language of the concrete inputs. We extended the grammar by invalid characters to test for unexpected behavior. Our grammar-based fuzzing technique revealed various violations of the MQTT specification. For example, tested broker implementations forwarded prohibited characters to clients. We also demonstrated the possibility that a malicious client could mimic internal broker communication.

For fuzzing BLE devices, we proposed a model-based fuzzing technique that generates the fuzzing test suite using the structure of the provided model. This allowed us to provide state coverage in black-box fuzzing. Furthermore, we could assign found issues to specific states in the learned model. This technique found several reliability issues in the tested BLE devices. We could crash four out of the six investigated devices. Furthermore, fuzzing revealed unexplored states and we found that for all devices that enable encrypted communication the length of the encryption key could be reduced to a security-critical size.

We compared different fuzzing techniques for testing VPN server implementations. When fuzzing communication protocols, there exist a lot of possibilities for different concrete inputs. Thus, we needed to select interesting behavioral aspects that should be fuzzed. For this purpose, we evaluated different techniques. For example, we evaluated fuzzing based on mutating the set of queries that are generated during active learning. Furthermore, we proposed search-based techniques that use the underlying model to assess if an input sequence generates interesting behavior. Based on these techniques, we found that the considered VPN server implementations violate the IPsec IKEv1 protocol specification. We also compared our techniques to simple random fuzzing and observe that random fuzzing was not successful to uncover the same issues.

12.2 Discussion

We discuss our gained research results based on the initially proposed research questions. For completeness, we list the research questions once more.

- **(RQ 1)** What are the challenges of learning behavioral models in networked systems?
  - **(RQ 1.1)** Does active automata learning perform well for learning communication protocol implementations on physical devices?
  - **(RQ 1.2)** Is automata learning useful to learn security-critical behavior?
- **(RQ 2)** How can automata learning be improved for practical applications?
  - **(RQ 2.1)** Does passive learning represent an alternative to active learning?
  - **(RQ 2.2)** How to improve automata learning to make it feasible for different challenges in networked environments?
- **(RQ 3)** Can automata learning support security testing techniques?
  - **(RQ 3.1)** How can black-box fuzzing techniques be extended with automata learning?

– **(RQ 3.2)** Is learning-based fuzzing effective at revealing security issues?
– **(RQ 3.3)** Can automata learning be used to fingerprint black-box devices?

(RQ 1) *What are the challenges of learning behavioral models in networked systems?* This thesis investigates the communication protocols BLE, MQTT, and IPsec IKEv1. For all these communication protocols, we considered real-world implementations as they can be used by anybody who works with this communication protocol. We managed to find a learning setup that enabled us to learn behavioral models of the different implementations. During learning, however, we faced different challenges.

First, we had to find an abstraction of the input and output alphabet to make learning feasible in a considerable amount of time. Considering all possible messages would be impossible. Hence, we need to come up with an appropriate abstraction. For BLE and IKEv1 it was useful to consider the packet structure that was already provided by packet manipulation libraries such as Scapy [158]. However, for both libraries, we observed that not all required packets have been defined, especially if we receive error messages. Therefore, we had to manually extend the applied libraries with definitions for these unknown packets.

For testing communication protocols in a multi-client setup as presented in Chapter 8, we additionally observed the problem that a too-coarse abstraction could lead to non-deterministic behavior. To overcome this problem, we introduced a learning algorithm for learning an abstracted non-deterministic finite state machine.

Another challenge in learning communication protocols was the creation of an interface that allows us to communicate with the SUL. The problem is that automata learning requires sending any input in any state which might be prohibited by existing interface implementations. For all our learning setups, we required custom interface implementations. To set up such interfaces, at least some domain knowledge about the protocol is useful and required.

The last challenge that we outlined is the observation of non-deterministic behavior. When learning real implementations, packets might arrive delayed or get lost. We approached this challenge by creating learning setups that allowed us to adapt the timeout for responses based on the performance of the SUL. However, increasing only the timeout was not sufficient. In addition, we needed the possibility to repeat queries during learning. We observed that sending inputs in an unexpected order can lead to non-deterministic observations, where the SUL retransmits previous messages.

(RQ 1.1) *Does active automata learning perform well for learning communication protocol implementations on physical devices?* A special challenge when learning real systems is to learn implementations on physical devices. Besides the challenges listed above, we also faced physical challenges such as interference with any other communication protocols or the distance to the SUL. Furthermore, the devices might become unreachable after a while, when no proper interactions were performed.

In this thesis, we successfully learned BLE stack implementations on physical devices. To interact with the BLE devices, we required additional BLE hardware that runs a custom firmware that allowed us to send custom BLE packets. However, this setup needed to be created only once and could then be reused for all devices. This setup enabled us to interact with any BLE device that sends advertisements. We showed that this approach is also applicable for learning behavioral models of the BLE device that is built into a Tesla Model 3. For learning the corresponding key fob, we faced another challenge, where the device stopped sending advertisements when it was not moved for some time. For this an additional hardware setup was required that keeps the device in motion. This showed that learning behavioral models of implementations on physical devices might require some additional engineering efforts, but it was feasible and provided insights into the difference between the protocol implementations of different BLE devices.

This case study complements the other case studies [37, 55, 168] that showed that automata learning is applicable to learn behavioral models of implementations of physical devices.

(RQ 1.2) *Is automata learning useful to learn security-critical behavior?* We used automata learning to learn behavioral models that formalize the key exchange procedure of the BLE protocol as well as the key exchange procedure that is performed by VPN implementations in order to establish an encrypted communication. We showed that these security-critical protocols could be learned. However, the setup of the learning framework again required some knowledge about the protocol. The concretization and abstraction of inputs and outputs had to consider the current state of the protocol, e.g., if a message must be encrypted or decrypted.

Learning key-exchange protocols was useful since the model showed all possible paths that allow a successful key exchange. Active automata learning algorithms exhaustively query the SUL to explore the state space. This might reveal unexpected paths, which could introduce a security issue. In our case study, we especially observed behavioral differences for the different implementations in the treatment of unexpected messages. Some implementations simply ignored unexpected messages, whereas others reset to a specific point in the protocol.

Furthermore, the exhaustive querying of active automata learning revealed that unusual input sequences lead to reliability issues, where the device stopped being reachable. An example of this was the sudden termination of the key exchange procedure from one side. Thus, especially active learning techniques were beneficial for revealing unexpected behavior.

(RQ 2) *How can automata learning be improved for practical applications?* The challenges discussed in **RQ 1** show that the application of automata learning in practice is not always straightforward. In the following two subsequent questions, we discuss if passive learning algorithms represent an alternative and propose methods to improve automata learning for learning communication protocol implementations.

(RQ 2.1) *Does passive learning represent an alternative to active learning?*

Passive learning techniques do not require an interface that enables the active interaction with the SUL during learning. Furthermore, the given set of samples can be preprocessed to filter out invalid observations that do not correctly reflect the behavior of the SUL.

In Chapter 6, we evaluated a passive learning algorithm that is based on state merging for learning models of communication protocols. Given that we align the sample size to the required queries by an active learning algorithm, the passively learned models achieved high behavioral accuracy with the SUL. Nevertheless, in most cases, we did not manage to learn the correct minimal automaton. To learn the correct model, large random samples would have been required. The problem with traditional state-merging algorithms is that they can only merge states which have a similar future. Thus, if the provided sample is incomplete then states might not be merged.

To overcome this problem, we proposed an RNN-based learning algorithm in Chapter 7. Our proposed technique achieved promising results, especially if the maximum number of states is known. If the number of states is unknown, which is usually the assumption for black-box systems in networked environments, we had to adapt for some examples the given budget, and had to repeat the experiment several times in order to verify that the model is learned correctly. Thus, we state that there is room for improvement, but it provides an alternative if active learning is not feasible.

(RQ 2.2) *How to improve automata learning to make it feasible for different challenges in networked environments?*

To overcome the challenges of actively learning behavioral models of communication protocols, we had to develop a fault-tolerant learning setup. For example, such a setup included the

repetition of queries in case we observed non-deterministic behavior. In general, the goal is to learn a correct model with the lowest possible number of queries. This is especially critical when learning networked systems since every query is costly to execute and comes with the risk that it must be repeated. In Chapter 3, we discussed some general improvements for automata learning algorithms such as an improved counterexample processing or the usage of caching structures to avoid queries on the SUL.

In Chapter 6, we compared the sample required by the improved version of the L^* algorithm that we used for most of the case studies with an optimized sample. Our results showed that there is room for improvement. For this purpose, we developed an improved version of the active learning algorithm of Kearns and Vazirani [96], which we referred to as KV. Our evaluation in Chapter 5 showed that our improved version of KV was beneficial to decrease the number of required queries, especially if L^* required more than one learning round.

In our performed evaluation, we observed that some communication protocol implementations could not be represented by a deterministic modeling formalism. To overcome this problem, we proposed that these implementations can be learned with non-deterministic learning algorithms. We note that learning non-deterministic systems could come with additional costs since queries must be repeated in order to observe all possible outputs. Chapter 8 proposed techniques that make learning in practice feasible without the requirement that everything must be observed at once.

In order to learn communication protocol implementations in a multi-client setup, we presented in Chapter 8 an abstraction technique that learns abstracted non-deterministic finite state machines. To find a proper generalization, we extended the minimally adequate teacher framework by an additional output abstraction level. Using this technique made learning multi-client communication protocols in a feasible amount of time possible.

(RQ 3) *Can automata learning support security testing techniques?*

This thesis proposed security testing techniques that use automata learning to generate behavioral models of black-box systems. The learned models then built the basis for a stateful black-box testing technique. In this thesis, we extended fuzz testing techniques by automata learning. The following questions discuss the approaches and their results. Furthermore, we discuss why the learned models are in general useful for the analysis in a networked environment.

(RQ 3.1) *How can black-box fuzzing techniques be extended with automata learning?*

The drawback of black-box fuzzing in practice is that it is hard to determine which parts of the system have been tested. Furthermore, if a black-box fuzzer uncovers issues it might not be straightforward to find the cause for the revealed issue. We approached both problems by extending black-box fuzzing with automata learning as a preliminary step.

In Chapter 9, we introduced the concept of learning-based fuzzing which we defined in a two-step procedure. In the first step, we applied automata learning to learn a behavioral model. In the second step, we built a model-based fuzzing technique that tests the conformance between a black-box system and the behavioral model. This created a stateful black-box fuzzing technique.

For testing the conformance between the learned model and the SUT, we proposed different fuzzing techniques. We created fuzzed input by considering grammatical structures, boundary values, or utilizing input generators of existing fuzzing tools. At the same time, we could generate test suites that fulfill certain coverage criteria or were generated based on search-based techniques.

Another big advantage of learning-based fuzzing was that many components from an active learning setup could be reused. For example, we could reuse the developed learning interfaces in order to provide fuzzed inputs to the SUT. Furthermore, the applied abstraction and concretizing technique could be used to create invalid or unusual concrete inputs.

(RQ 3.2) *Is learning-based fuzzing effective at revealing security issues?*

We evaluated the effectiveness of our learning-based fuzzing technique in Chapter 10. For this purpose, we fuzzed MQTT broker implementations, BLE devices, and VPN servers.

We used grammar-based fuzzing for testing MQTT broker implementations. We found violations of the MQTT specification in four out of five investigated MQTT brokers. One broker not only accepted prohibited characters, it also forwards them to connected clients, where these characters could be vulnerable to an innocent client.

Our case study on BLE devices revealed that four out of six of the investigated devices crashed on unexpected inputs. Furthermore, we showed that for all devices that allowed the establishment of encrypted communication the key size could be reduced such that the key could be theoretically generated by brute-force.

For the VPN server implementations, we found that both investigated implementations violate the specification since they missed evaluating field values or only responded to invalid inputs a few steps later in the protocol.

This showed that our learning-based fuzzing technique was successful in revealing issues in the investigated communication protocol implementations.

(RQ 3.3) *Can automata learning be used to fingerprint black-box devices?* We highlighted another aspect of automata learning in the context of the security analysis of networked environments. The learned models could be used to fingerprint black-box devices. We saw in the BLE and the VPN case study that the learned models of almost all investigated implementations were different. Only the two BLE models from the Tesla case study were equivalent, which might hint at the fact that the same BLE stack implementation is used. Based on the different models, an attacker could generate a fingerprinting sequence that allows one to determine which implementation is used. This could be dangerous in case the implementation has known vulnerabilities that can be exploited by the attacker.

12.3 Future Work

Our results of the presented techniques were promising. We showed that automata learning could successfully be used to analyze the security-critical behavior of black-box systems. In the following, we outline future directions for our proposed techniques.

General framework for learning communication protocols. We showed that automata learning, especially active learning techniques, could learn behavioral models of communication protocol implementations. These models were useful for the further analysis of the protocol. However, the creation of a learning interface was a tedious process. In the future, it would be useful to have a general tool that enables the automatic setup of such an interface to some extent. For example, Aarts et al. [4] outline a technique to automize the alphabet abstraction. Furthermore, presented components of our learning frameworks such as fault-tolerant caching structures can be reused for learning other communication protocols.

Automatic analysis of learned models. Most of the performed analysis of the learned models was done manually. This was sufficient to indicate that model learning is useful to reveal violations of the specification. However, if someone considers a larger set of investigated implementations or wants to integrate automata learning in a continuous development process, it would be useful to automize the analysis of the learned models. The literature provides different directions for automated analysis or behavioral models. Tappler et al. [170] proposed a method to check for behavioral differences between two models. The differences are then an indicator of specification violations. Another approach would be to define formal properties and verify them using model-checking techniques.

Extending RNN-based techniques for other formalisms. Our RNN-based learning technique achieved promising first results on learning behavioral models. In the literature, recurrent neural network (RNN) are commonly used to predict sequential data. Thus, it would be interesting to investigate if RNN-based learning techniques are sufficient to learn other behavioral aspects such as timed or stochastic behavior. The idea is that recurrent neural network (RNN) might generalize better on incomplete data sets which is often the case when considering continuous values such as time.

Learning-based fuzzing for other formalisms. This thesis discusses learning-based fuzzing only for deterministic systems. For future work, it would interesting to consider other behavioral aspects. Our presented results show that several systems behaved non-deterministically. Moreover, several security-critical aspects of systems depend on other advanced system properties such as timed or stochastic behavior. For example, key-exchange protocols frequently rely on random value generation. To ensure that the key generation is secure, the random value generation must not allow any conclusions to be drawn about the other values. Another approach would be to consider side channel information such as timed behavior in the learned model and then check if fuzz testing could reveal any unexpected timed dependencies.

The work presented in this thesis shows promising techniques for applying automata learning to the testing and analysis of networked environments for security issues. We show that there are still open questions and hope that the techniques presented in this thesis will be useful in providing a foundation for future research in this area.

Bibliography

[1] CVE-2019-19193. Available from MITRE, CVE-ID CVE-2019-19193., February 2020. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[2] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In Alexandre Petrenko, Adenilso da Silva Simão, and José Carlos Maldonado, editors, *Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010. ISBN 978-3-642-16572-6. doi: 10.1007/978-3-642-16573-3_14. URL https://doi.org/10.1007/978-3-642-16573-3_14.

[3] Fides Aarts, Julien Schmaltz, and Frits W. Vaandrager. Inference and abstraction of the biometric passport. In Margaria and Steffen [112], pages 673–686. ISBN 978-3-642-16557-3. doi: 10.1007/978-3-642-16558-0_54. URL https://doi.org/10.1007/978-3-642-16558-0_54.

[4] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits W. Vaandrager. Automata learning through counterexample guided abstraction refinement. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 10–27. Springer, 2012. ISBN 978-3-642-32758-2. doi: 10.1007/978-3-642-32759-9_4. URL https://doi.org/10.1007/978-3-642-32759-9_4.

[5] Fides Aarts, Joeri de Ruiter, and Erik Poll. Formal models of bank cards for free. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*, pages 461–468. IEEE Computer Society, 2013. ISBN 978-1-4799-1324-4. doi: 10.1109/ICSTW.2013.60. URL https://doi.org/10.1109/ICSTW.2013.60.

[6] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits W. Vaandrager, and Sicco Verwer. Improving active Mealy machine learning for protocol conformance testing. *Machine Learning*, 96(1-2):189–224, 2014. doi: 10.1007/s10994-013-5405-0. URL https://doi.org/10.1007/s10994-013-5405-0.

[7] Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015. doi: 10.1007/s10703-014-0216-x. URL https://doi.org/10.1007/s10703-014-0216-x.

[8] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. Model learning and model-based testing. In Amel Bennaceur, Reiner Hähnle, and Karl Meinke, editors, *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, volume 11026 of *Lecture Notes*

in Computer Science, pages 74–100. Springer, 2018. ISBN 978-3-319-96561-1. doi: 10.1007/978-3-319-96562-8_3. URL https://doi.org/10.1007/978-3-319-96562-8_3.

[9] Bernhard K. Aichernig, Christian Burghard, and Robert Korosec. Learning-based testing of an industrial measurement device. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2019. ISBN 978-3-030-20651-2. doi: 10.1007/978-3-030-20652-9_1. URL https://doi.org/10.1007/978-3-030-20652-9_1.

[10] Bernhard K. Aichernig, Andrea Pferscher, and Martin Tappler. From passive to active: Learning timed automata efficiently. In Ritchie Lee, Susmit Jha, and Anastasia Mavridou, editors, *NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings*, volume 12229 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2020. ISBN 978-3-030-55753-9. doi: 10.1007/978-3-030-55754-6_1. URL https://doi.org/10.1007/978-3-030-55754-6_1.

[11] Bernhard K. Aichernig, Edi Muškardin, and Andrea Pferscher. Learning-based fuzzing of IoT message brokers. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*, pages 47–58. IEEE, 2021. ISBN 978-1-7281-6836-4. doi: 10.1109/ICST49551.2021.00017. URL https://doi.org/10.1109/ICST49551.2021.00017.

[12] Bernhard K. Aichernig, Sandra König, Cristinel Mateis, Andrea Pferscher, Dominik Schmidt, and Martin Tappler. Constrained training of recurrent neural networks for automata learning. In Bernd-Holger Schlingloff and Ming Chai, editors, *Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings*, volume 13550 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2022. ISBN 978-3-031-17107-9. doi: 10.1007/978-3-031-17108-6_10. URL https://doi.org/10.1007/978-3-031-17108-6_10.

[13] Bernhard K. Aichernig, Edi Muškardin, and Andrea Pferscher. Active vs. passive: A comparison of automata learning paradigms for network protocols. In Matt Luckcuck and Marie Farrell, editors, *Proceedings Fourth International Workshop on Formal Methods for Autonomous Systems (FMAS) and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE), FMAS/ASYDE@SEFM 2022, and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE) Berlin, Germany, 26th and 27th of September 2022*, volume 371 of *EPTCS*, pages 1–19, 2022. doi: 10.4204/EPTCS.371.1. URL https://doi.org/10.4204/EPTCS.371.1.

[14] Bernhard K. Aichernig, Sandra König, Cristinel Mateis, Andrea Pferscher, and Martin Tappler. Learning minimal automata with recurrent neural networks. *Software and Systems Modeling*, 2023. under submission, submitted in February 2023.

[15] Bernhard K. Aichernig, Martin Tappler, and Felix Wallner. Benchmarking combinations of learning and testing algorithms for automata learning. *Formal Aspects of Computing*, June 2023. ISSN 0934-5043. doi: 10.1145/3605360. URL https://doi.org/10.1145/3605360. Just Accepted.

[16] Dana Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76–87, 1981. doi: 10.1016/S0019-9958(81)90090-5. URL https://doi.org/10.1016/S0019-9958(81)90090-5.

[17] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. doi: 10.1016/0890-5401(87)90052-6. URL https://doi.org/10.1016/0890-5401(87)90052-6.

[18] Dana Angluin and Philip D. Laird. Learning from noisy examples. *Machine Learning*, 2(4): 343–370, 1987. doi: 10.1007/BF00116829. URL https://doi.org/10.1007/BF00116829.

[19] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Key negotiation downgrade attacks on Bluetooth and Bluetooth Low Energy. *ACM Transactions on Privacy and Security*, 23(3):14:1–14:28, 2020. doi: 10.1145/3394497. URL https://doi.org/10.1145/3394497.

[20] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1690–1701. ACM, 2016. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978383. URL https://doi.org/10.1145/2976749.2978383.

[21] Nordic Semiconductor ASA. nRF connect for mobile, 2023. URL https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp. Accessed: 2023-06-13.

[22] Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. MQTT version 5.0, March 2019. URL https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. Accessed: 2023-05-15.

[23] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin C. Almeroth, Richard A. Kemmerer, and Giovanni Vigna. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, volume 4176 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2006. ISBN 3-540-38341-7. doi: 10.1007/11836810_25. URL https://doi.org/10.1007/11836810_25.

[24] Elaine Barke. Recommendation for key management: Part 1, general. Technical report, National Institute of Standards and Technology, 2020. URL https://doi.org/10.6028/NIST.SP.800-57pt1r5.

[25] Axel Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010. ISBN 978-3-642-12001-5. doi: 10.1007/978-3-642-12002-2_21. URL https://doi.org/10.1007/978-3-642-12002-2_21.

[26] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to Angluin's learning. In Sandro Etalle, Supratik Mukhopadhyay, and Abhik Roychoudhury, editors, *Proceedings of the International Workshop on Software Verification and Validation, SVV@ICLP 2003, Mumbai, India, December 14, 2003*, volume 118 of *Electronic Notes in Theoretical Computer Science*, pages 3–18. Elsevier, 2003. doi: 10.1016/j.entcs.2004.12.015. URL https://doi.org/10.1016/j.entcs.2004.12.015.

[27] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the correspondence between conformance testing and regular inference. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005. ISBN 3-540-25420-X. doi: 10.1007/978-3-540-31984-9_14. URL https://doi.org/10.1007/978-3-540-31984-9_14.

[28] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2329–2344. ACM, 2017. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134020. URL https://doi.org/10.1145/3133956.3134020.

[29] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of NFA. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1004–1009, 2009. URL http://ijcai.org/Proceedings/09/Papers/170.pdf.

[30] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-26278-4. doi: 10.1007/b137241. URL https://doi.org/10.1007/b137241.

[31] Srdjan Capkun and Franziska Roesner, editors. *29th USENIX Security Symposium, USENIX Security 2020*, Virtual Event, 2020. USENIX Association. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20.

[32] Rafael C. Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In Rafael C. Carrasco and José Oncina, editors, *Grammatical Inference and Applications, Second International Colloquium, ICGI-94, Alicante, Spain, September 21-23, 1994, Proceedings*, volume 862 of *Lecture Notes in Computer Science*, pages 139–152. Springer, 1994. ISBN 3-540-58473-0. doi: 10.1007/3-540-58473-0_144. URL https://doi.org/10.1007/3-540-58473-0_144.

[33] David Carrel and Dan Harkins. The Internet Key Exchange (IKE). RFC 2409, November 1998. URL https://www.rfc-editor.org/info/rfc2409.

[34] G. Casteur, A. Aubaret, B. Blondeau, V. Clouet, A. Quemat, V. Pical, and Rafik Zitouni. Fuzzing attacks for vulnerability discovery within MQTT protocol. In *16th International Wireless Communications and Mobile Computing Conference, IWCMC 2020, Limassol, Cyprus, June 15-19, 2020*, pages 420–425. IEEE, 2020. ISBN 978-1-7281-3129-0. doi: 10.1109/IWCMC48107.2020.9148320. URL https://doi.org/10.1109/IWCMC48107.2020.9148320.

[35] Ana Cavalcanti and Dennis Dams, editors. *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, 2009. Springer. ISBN 978-3-642-05088-6. doi: 10.1007/978-3-642-05089-3. URL https://doi.org/10.1007/978-3-642-05089-3.

[36] Guillaume Celosia and Mathieu Cunche. Fingerprinting Bluetooth-Low-Energy devices based on the generic attribute profile. In Peng Liu and Yuqing Zhang, editors, *Proceedings*

of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, IoT S&P@CCS 2019, London, UK, November 15, 2019, pages 24–31. ACM, 2019. ISBN 978-1-4503-6838-4. doi: 10.1145/3338507.3358617. URL https://doi.org/10.1145/3338507.3358617.

[37] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri de Ruiter. Automated reverse engineering using Lego®. In Sergey Bratus and Felix "FX" Lindner, editors, *8th USENIX Workshop on Offensive Technologies, WOOT '14, San Diego, CA, USA, August 19, 2014*. USENIX Association, 2014. URL https://www.usenix.org/conference/woot14/workshop-program/presentation/chalupar.

[38] Chia Yuan Cho, Domagoj Babic, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 426–439. ACM, 2010. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866355. URL https://doi.org/10.1145/1866307.1866355.

[39] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978. doi: 10.1109/TSE.1978.231496. URL https://doi.org/10.1109/TSE.1978.231496.

[40] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An open-source tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002. ISBN 3-540-43997-8. doi: 10.1007/3-540-45657-0_29. URL https://doi.org/10.1007/3-540-45657-0_29.

[41] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. ISBN 978-3-319-10574-1. doi: 10.1007/978-3-319-10575-8. URL https://doi.org/10.1007/978-3-319-10575-8.

[42] David Combe, Colin de la Higuera, and Jean-Christophe Janodet. Zulu: An interactive learning competition. In Anssi Yli-Jyrä, András Kornai, Jacques Sakarovitch, and Bruce W. Watson, editors, *Finite-State Methods and Natural Language Processing, 8th International Workshop, FSMNLP 2009, Pretoria, South Africa, July 21-24, 2009, Revised Selected Papers*, volume 6062 of *Lecture Notes in Computer Science*, pages 139–146. Springer, 2009. ISBN 978-3-642-14683-1. doi: 10.1007/978-3-642-14684-8_15. URL https://doi.org/10.1007/978-3-642-14684-8_15.

[43] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Krügel, and Engin Kirda. Prospex: Protocol specification extraction. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 110–125. IEEE Computer Society, 2009. ISBN 978-0-7695-3633-0. doi: 10.1109/SP.2009.14. URL https://doi.org/10.1109/SP.2009.14.

[44] F-Secure Corporation. mqtt_fuzz, 2015. URL https://github.com/F-Secure/mqtt_fuzz. Accessed: 2023-06-29.

[45] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In Niels Provos, editor, *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*. USENIX Association, 2007. URL https://www.usenix.org/conference/16th-usenix-security-symposium/discoverer-automatic-protocol-reverse-engineering-network.

[46] Carlos Diego Nascimento Damasceno and Daniel Strüber. Family-based fingerprint analysis: A position paper. In Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, editors, *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, volume 13560 of *Lecture Notes in Computer Science*, pages 137–150. Springer, 2022. ISBN 978-3-031-15628-1. doi: 10.1007/978-3-031-15629-8_8. URL https://doi.org/10.1007/978-3-031-15629-8_8.

[47] Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. Inferring OpenVPN state machines using protocol state fuzzing. In *2018 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2018, London, United Kingdom, April 23-27, 2018*, pages 11–19. IEEE, 2018. ISBN 978-1-5386-5445-3. doi: 10.1109/EuroSPW.2018.00009. URL https://doi.org/10.1109/EuroSPW.2018.00009.

[48] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. Fuzzers for stateful systems: Survey and research directions. *CoRR*, abs/2301.02490, 2023. doi: 10.48550/arXiv.2301.02490. URL https://doi.org/10.48550/arXiv.2301.02490.

[49] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010. ISBN 9780521763165. doi: 10.1017/CBO9781139194655.

[50] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15*, pages 193–206, Washington, D.C., USA, 2015. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[51] Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors. *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, 2022. Springer. ISBN 978-3-031-06772-3. doi: 10.1007/978-3-031-06773-0. URL https://doi.org/10.1007/978-3-031-06773-0.

[52] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. doi: 10.1109/TIT.1976.1055638. URL https://doi.org/10.1109/TIT.1976.1055638.

[53] Guoliang Dong, Jingyi Wang, Jun Sun, Yang Zhang, Xinyu Wang, Ting Dai, Jin Song Dong, and Xingen Wang. Towards interpreting recurrent neural networks through probabilistic abstraction. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 499–510. IEEE, 2020. ISBN 978-1-4503-6768-4. doi: 10.1145/3324884.3416592. URL https://doi.org/10.1145/3324884.3416592.

[54] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 523–538. USENIX Association, 2012. URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe.

[55] Masoud Ebrahimi, Stefan Marksteiner, Dejan Nickovic, Roderick Bloem, David Schögler, Philipp Eisner, Samuel Sprung, Thomas Schober, Sebastian Chlup, Christoph Schmittner, and Sandra König. A systematic approach to automotive security. *CoRR*, abs/2303.02894, 2023. doi: 10.48550/arXiv.2303.02894. URL https://doi.org/10.48550/arXiv.2303.02894.

[56] Khaled El-Fakih, Roland Groz, Muhammad Naeem Irfan, and Muzammil Shahbaz. Learning finite state models of observable nondeterministic systems in a testing context. In *22nd IFIP International Conference on Testing Software and Systems, Natal, Brazil*, pages 97–102, 2010. URL https://hal.inria.fr/hal-00953395.

[57] Hakan Erdogmus and Klaus Havelund, editors. *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, Santa Barbara, CA, USA, 2017. ACM. ISBN 978-1-4503-5077-8.

[58] Rémi Eyraud, Dakotah Lambert, Badr Tahri Joutei, Aidar Gaffarov, Mathias Cabanne, Jeffrey Heinz, and Chihiro Shibata. TAYSIR competition: Transformer+RNN: Algorithms to yield simple and interpretable representations. In *Proceedings of the 16th International Conference on Grammatical Inference*, 2023. URL https://remieyraud.github.io/TAYSIR/TAYSIR_peer_reviewed_paper-ICGI23.pdf. online preprint, accessed: 2023-07-09.

[59] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. A year in lockdown: How the waves of COVID-19 impact internet traffic. *Communications of the ACM*, 64(7):101–108, 2021. doi: 10.1145/3465212. URL https://doi.org/10.1145/3465212.

[60] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In Yarom and Zennou [198]. URL https://www.usenix.org/conference/woot20/presentation/fioraldi.

[61] Paul Fiterău-Broştean, Ramon Janssen, and Frits W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 454–471, Toronto, ON, Canada, 2016. Springer. ISBN 978-3-319-41539-0. doi: 10.1007/978-3-319-41540-6_25.

[62] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In Erdogmus and Havelund [57], pages 142–151. ISBN 978-1-4503-5077-8. doi: 10.1145/3092282.3092289.

[63] Paul Fiterău-Broştean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In Capkun and Roesner [31], pages 2523–2540. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean.

[64] Markus Theo Frohme. Active automata learning with adaptive distinguishing sequences. *CoRR*, abs/1902.01139, 2019. URL http://arxiv.org/abs/1902.01139.

[65] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991. doi: 10.1109/32.87284. URL https://doi.org/10.1109/32.87284.

[66] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. SweynTooth: Unleashing mayhem over Bluetooth Low Energy. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference,*

USENIX ATC 2020, July 15-17, 2020, pages 911–925. USENIX Association, 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/garbelini.

[67] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. SweynTooth - Unleashing mayhem over Bluetooth Low Energy, 2020. URL https://github.com/Matheus-Garbelini/sweyntooth_bluetooth_low_energy_attacks. Accessed: 2023-05-26.

[68] Matheus E. Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. Braktooth: Causing havoc on Bluetooth link manager via directed fuzzing. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 1025–1042. USENIX Association, 2022. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/garbelini.

[69] Gitlab.org. Gitlab protocol fuzzer community edition. URL https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce. Accessed: 2023-07-03.

[70] AVM GmbH. Connecting the FRITZ!Box with a company's VPN (IPSec), 2023. URL https://en.avm.de/service/vpn/connecting-the-fritzbox-with-a-companys-vpn-ipsec/. Access on: 2023-05-22.

[71] Patrice Godefroid. Fuzzing: hack, art, and science. *Communications of the ACM*, 63(2): 70–76, 2020. doi: 10.1145/3363824. URL https://doi.org/10.1145/3363824.

[72] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012. doi: 10.1145/2093548.2093564. URL https://doi.org/10.1145/2093548.2093564.

[73] E. Mark Gold. System identification via state characterization. *Automatica*, 8(5):621–636, 1972.

[74] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978. doi: 10.1016/S0019-9958(78)90562-4. URL https://doi.org/10.1016/S0019-9958(78)90562-4.

[75] Martijn A. Goorden, Kim G. Larsen, Axel Legay, Florian Lorber, Ulrik Nyman, and Andrzej Wasowski. Timed I/O automata: It is never too late to complete your timed specification theory. *CoRR*, abs/2302.04529, 2023. doi: 10.48550/arXiv.2302.04529. URL https://doi.org/10.48550/arXiv.2302.04529.

[76] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of event-recording automata using timed decision trees. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 435–449. Springer, 2006. ISBN 3-540-37376-4. doi: 10.1007/11817949_29. URL https://doi.org/10.1007/11817949_29.

[77] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theoretical Computer Science*, 411(47):4029–4054, 2010. doi: 10.1016/j.tcs.2010.07.008. URL https://doi.org/10.1016/j.tcs.2010.07.008.

[78] NCC Group. NCC Group uncovers Bluetooth Low Energy (BLE) vulnerability that puts millions of cars, mobile devices and locking systems at risk, 2022. URL https://newsroom.nccgroup.com/news/ncc-group-uncovers-bluetooth-low-

energy-ble-vulnerability-that-puts-millions-of-cars-mobile-devices-and-locking-systems-at-risk-447952. Accessed: 2023-06-13.

[79] Roland Groz, Nicolas Brémond, Adenilso da Silva Simão, and Catherine Oriat. *hW*-inference: A heuristic approach to retrieve models through black box testing. *Journal of Systems and Software*, 159, 2020. doi: 10.1016/j.jss.2019.110426. URL https://doi.org/10.1016/j.jss.2019.110426.

[80] Jiaxing Guo, Chunxiang Gu, Xi Chen, and Fushan Wei. Model learning and model checking of IPSec implementations for internet of things. *IEEE Access*, 7:171322–171332, 2019. doi: 10.1109/ACCESS.2019.2956062. URL https://doi.org/10.1109/ACCESS.2019.2956062.

[81] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, , 1971. ISBN 978-0-12-417750-5. doi: 10.1016/B978-0-12-417750-5.50022-1.

[82] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007. ISBN 978-0-321-47617-3.

[83] Karim Hossen, Roland Groz, and Jean-Luc Richier. Security vulnerabilities detection using model inference for applications and security protocols. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, pages 534–536. IEEE Computer Society, 2011. ISBN 978-0-7695-4345-1. doi: 10.1109/ICSTW.2011.83. URL https://doi.org/10.1109/ICSTW.2011.83.

[84] Zhe Hou and Vijay Ganesh, editors. *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings*, volume 12971 of *Lecture Notes in Computer Science*, 2021. Springer. ISBN 978-3-030-88884-8. doi: 10.1007/978-3-030-88885-5. URL https://doi.org/10.1007/978-3-030-88885-5.

[85] Falk Howar, Bernhard Steffen, and Maik Merten. From ZULU to RERS - Lessons learned in the ZULU challenge. In Margaria and Steffen [112], pages 687–704. ISBN 978-3-642-16557-3. doi: 10.1007/978-3-642-16558-0_55. URL https://doi.org/10.1007/978-3-642-16558-0_55.

[86] Yating Hsu, Guoqiang Shu, and David Lee. A model-based approach to security flaw detection of network protocol implementations. In *Proceedings of the 16th annual IEEE International Conference on Network Protocols, 2008. ICNP 2008, Orlando, Florida, USA, 19-22 October 2008*, pages 114–123. IEEE Computer Society, 2008. ISBN 978-1-4244-2506-8. doi: 10.1109/ICNP.2008.4697030. URL https://doi.org/10.1109/ICNP.2008.4697030.

[87] Bluetooth SIG Inc. Bluetooth core specification version 5.3, July 2021. URL https://www.bluetooth.com/specifications/specs/core-specification-5-3/. Accessed: 2023-05-16.

[88] Bluetooth SIG Inc. 2023 Bluetooth market update, April 2023. URL https://www.bluetooth.com/2023-market-update/. Accessed: 2023-05-16.

[89] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka,

editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2014. ISBN 978-3-319-11163-6. doi: 10.1007/978-3-319-11164-3_26. URL https://doi.org/10.1007/978-3-319-11164-3_26.

[90] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source LearnLib - A framework for active automata learning. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 487–495. Springer, 2015. ISBN 978-3-319-21689-8. doi: 10.1007/978-3-319-21690-4_32. URL https://doi.org/10.1007/978-3-319-21690-4_32.

[91] Marc Jasper, Maximilian Fecke, Bernhard Steffen, Markus Schordan, Jeroen Meijer, Jaco van de Pol, Falk Howar, and Stephen F. Siegel. The RERS 2017 challenge and workshop (invited paper). In Erdogmus and Havelund [57], pages 11–20. ISBN 978-1-4503-5077-8. doi: 10.1145/3092282.3098206. URL https://doi.org/10.1145/3092282.3098206.

[92] William Johansson, Martin Svensson, Ulf E. Larson, Magnus Almgren, and Vincenzo Gulisano. T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 323–332. IEEE Computer Society, 2014. ISBN 978-0-7695-5185-2. doi: 10.1109/ICST.2014.45. URL https://doi.org/10.1109/ICST.2014.45.

[93] Colin G. Johnson. Genetic programming with fitness based on model checking. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Esparcia-Alcázar, editors, *Genetic Programming, 10th European Conference, EuroGP 2007, Valencia, Spain, April 11-13, 2007, Proceedings*, volume 4445 of *Lecture Notes in Computer Science*, pages 114–124. Springer, 2007. ISBN 978-3-540-71602-0. doi: 10.1007/978-3-540-71605-1_11. URL https://doi.org/10.1007/978-3-540-71605-1_11.

[94] Alexander Kaiser. Eclipse MQTT test suite, 2019. URL https://github.com/eclipse/iottestware.mqtt. Accessed: 2023-06-29.

[95] Charlie Kaufman, Paul E. Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296, October 2014. URL https://www.rfc-editor.org/info/rfc7296.

[96] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994. ISBN 978-0-262-11193-5. URL https://mitpress.mit.edu/books/introduction-computational-learning-theory.

[97] Ali Khalili and Armando Tacchella. Learning nondeterministic Mealy machines. In Alexander Clark, Makoto Kanazawa, and Ryo Yoshinaka, editors, *Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014, Kyoto, Japan, September 17-19, 2014*, volume 34 of *JMLR Workshop and Conference Proceedings*, pages 109–123. JMLR.org, 2014. URL http://proceedings.mlr.press/v34/khalili14a.html.

[98] Igor Khmelnitsky, Daniel Neider, Rajarshi Roy, Xuan Xie, Benoît Barbot, Benedikt Bollig, Alain Finkel, Serge Haddad, Martin Leucker, and Lina Ye. Property-directed verification and robustness certification of recurrent neural networks. In Hou and Ganesh [84], pages 364–380. ISBN 978-3-030-88884-8. doi: 10.1007/978-3-030-88885-5_24. URL https://doi.org/10.1007/978-3-030-88885-5_24.

[99] Igor Khmelnitsky, Serge Haddad, Lina Ye, Benoît Barbot, Benedikt Bollig, Martin Leucker, Daniel Neider, and Rajarshi Roy. Analyzing robustness of Angluin's L* algorithm in presence of noise. In Pierre Ganty and Dario Della Monica, editors, *Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022*, volume 370 of *EPTCS*, pages 81–96, 2022. doi: 10.4204/EPTCS.370.6. URL https://doi.org/10.4204/EPTCS.370.6.

[100] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[101] Stephen C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies: Annals of Mathematics Studies*, 34, 1956.

[102] Zhifeng Lai, S. C. Cheung, and Yunfei Jiang. Dynamic model learning using genetic algorithm under adaptive model checking framework. In *Sixth International Conference on Quality Software (QSIC 2006), 26-28 October 2006, Beijing, China*, pages 410–417. IEEE Computer Society, 2006. ISBN 0-7695-2718-3. doi: 10.1109/QSIC.2006.25. URL https://doi.org/10.1109/QSIC.2006.25.

[103] Khanh Tuan Le. Bluetooth Low Energy and the automotive transformation, 2017. URL https://www.ti.com/lit/wp/sway008/sway008.pdf. Accessed: 2023-05-25.

[104] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996. doi: 10.1109/5.533956. URL https://doi.org/10.1109/5.533956.

[105] Raluca Lefticaru, Florentin Ipate, and Cristina Tudose. Automated model design using genetic algorithms and model checking. In Petros Kefalas, Demosthenes Stamatis, and Christos Douligeris, editors, *2009 Fourth Balkan Conference in Informatics, BCI 2009, Thessaloniki, Greece, 17-19 September 2009*, pages 79–84. IEEE Computer Society, 2009. ISBN 978-0-7695-3783-2. doi: 10.1109/BCI.2009.15. URL https://doi.org/10.1109/BCI.2009.15.

[106] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018. doi: 10.1109/TR.2018.2834476. URL https://doi.org/10.1109/TR.2018.2834476.

[107] libreswan. libreswan, 2023. URL https://libreswan.org/. Accessed: 2023-06-16.

[108] Simon M. Lucas and T. Jeff Reynolds. Learning DFA: evolution versus evidence driven state merging. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2003, Canberra, Australia, December 8-12, 2003*, pages 351–358. IEEE, 2003. ISBN 0-7803-7804-0. doi: 10.1109/CEC.2003.1299597. URL https://doi.org/10.1109/CEC.2003.1299597.

[109] Simon M. Lucas and T. Jeff Reynolds. Learning deterministic finite automata with a smart state labeling evolutionary algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7):1063–1074, 2005. doi: 10.1109/TPAMI.2005.143. URL https://doi.org/10.1109/TPAMI.2005.143.

[110] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. Learning Markov decision processes for model checking. In Uli Fahrenberg, Axel Legay, and Claus R. Thrane, editors, *Proceedings Quantities in Formal Methods, QFM*

2012, Paris, France, 28 August 2012, volume 103 of *EPTCS*, pages 49–63, 2012. doi: 10.4204/EPTCS.103.6. URL https://doi.org/10.4204/EPTCS.103.6.

[111] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. Learning deterministic probabilistic automata from a model checking perspective. *Machine Learning*, 105(2):255–299, 2016. doi: 10.1007/s10994-016-5565-9. URL https://doi.org/10.1007/s10994-016-5565-9.

[112] Tiziana Margaria and Bernhard Steffen, editors. *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, volume 6415 of *Lecture Notes in Computer Science*, 2010. Springer. ISBN 978-3-642-16557-3. doi: 10.1007/978-3-642-16558-0. URL https://doi.org/10.1007/978-3-642-16558-0.

[113] Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard Steffen. Efficient test-based model generation for legacy reactive systems. In *Ninth IEEE International High-Level Design Validation and Test Workshop 2004, Sonoma Valley, CA, USA, November 10-12, 2004*, pages 95–100. IEEE Computer Society, 2004. ISBN 0-7803-8714-7. doi: 10.1109/HLDVT.2004.1431246. URL https://doi.org/10.1109/HLDVT.2004.1431246.

[114] Franz Mayr and Sergio Yovine. Regular inference on artificial neural networks. In Andreas Holzinger, Peter Kieseberg, A Min Tjoa, and Edgar R. Weippl, editors, *Machine Learning and Knowledge Extraction - Second IFIP TC 5, TC 8/WG 8.4, 8.9, TC 12/WG 12.9 International Cross-Domain Conference, CD-MAKE 2018, Hamburg, Germany, August 27-30, 2018, Proceedings*, volume 11015 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2018. ISBN 978-3-319-99739-1. doi: 10.1007/978-3-319-99740-7_25. URL https://doi.org/10.1007/978-3-319-99740-7_25.

[115] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998. URL https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf.

[116] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. *Technical Report*, 2012.

[117] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955. doi: 10.1002/j.1538-7305.1955.tb03788.x.

[118] Joshua J. Michalenko, Ameesh Shah, Abhinav Verma, Richard G. Baraniuk, Swarat Chaudhuri, and Ankit B. Patel. Representing formal languages: A comparison between finite automata and recurrent neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=H1zeHnA9KX.

[119] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990. doi: 10.1145/96267.96279. URL https://doi.org/10.1145/96267.96279.

[120] Robin Milner. *A Calculus of Communication Systems*, volume 92. Springer Berlin, Heidelberg, 1980. ISBN 978-3-540-10235-9. doi: https://doi.org/10.1007/3-540-10235-3.

[121] Marvin L. Minsky. *Computation: Finite and infinite machines*. Prentice-Hall, Inc., USA, 1967. ISBN 0131655639.

[122] Kristiyan Mladenov. Formal verification of the implementation of the MQTT protocol in IoT devices. Technical report, University of Amsterdam, Netherlands, 2017.

[123] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[124] Sean Murphy. The advanced encryption standard (AES). *Information Security Technical Report*, 4(4):12–17, 1999. doi: 10.1016/S1363-4127(99)80083-1. URL https://doi.org/10.1016/S1363-4127(99)80083-1.

[125] Edi Muškardin, Martin Tappler, and Bernhard K. Aichernig. Testing-based black-box extraction of simple models from RNNs and transformers. In François Coste, Faissal Ouardi, and Guillaume Rabusseau, editors, *Proceedings of 16th edition of the International Conference on Grammatical Inference*, volume 217 of *Proceedings of Machine Learning Research*, pages 291–294. PMLR, 10–13 Jul 2023. URL https://proceedings.mlr.press/v217/muskardin23a.html.

[126] Edi Muškardin and Andrea Pferscher. Supplemental material for "Learning-based fuzzing of IoT message brokers", Jan 2021. URL https://github.com/DES-Lab/Learning-Based-Fuzzing.

[127] Edi Muškardin and Andrea Pferscher. Supplemental material for "Active vs. passive: A comparison of automata learning paradigms for network protocols", 2022. URL https://github.com/apferscher/ble-learning-passive.

[128] Edi Muškardin, Bernhard K. Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. Aalpy: An active automata learning library. In Hou and Ganesh [84], pages 67–73. ISBN 978-3-030-88884-8. doi: 10.1007/978-3-030-88885-5_5. URL https://doi.org/10.1007/978-3-030-88885-5_5.

[129] Edi Muškardin, Bernhard K. Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. AALpy: an active automata learning library. *Innovations in Systems and Software Engineering*, 18(3):417–426, 2022. doi: 10.1007/s11334-022-00449-3. URL https://doi.org/10.1007/s11334-022-00449-3.

[130] Edi Muškardin, Bernhard K. Aichernig, Ingo Pill, and Martin Tappler. Learning finite state models from recurrent neural networks. In Maurice H. ter Beek and Rosemary Monahan, editors, *Integrated Formal Methods - 17th International Conference, IFM 2022, Lugano, Switzerland, June 7-10, 2022, Proceedings*, volume 13274 of *Lecture Notes in Computer Science*, pages 229–248. Springer, 2022. ISBN 978-3-031-07726-5. doi: 10.1007/978-3-031-07727-2_13. URL https://doi.org/10.1007/978-3-031-07727-2_13.

[131] Daniel Neider, Rick Smetsers, Frits W. Vaandrager, and Harco Kuppens. Benchmarks for automata learning and conformance testing. In Tiziana Margaria, Susanne Graf, and Kim G. Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, volume 11200 of *Lecture Notes in Computer Science*, pages 390–416. Springer, 2018. ISBN 978-3-030-22347-2. doi: 10.1007/978-3-030-22348-9_23. URL https://doi.org/10.1007/978-3-030-22348-9_23.

[132] Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. ISSN 0002-9939, 1088-6826/e. doi: 10.2307/2033204. URL https://www.jstor.org/stable/2033204.

[133] Oliver Niese. *An integrated approach to testing complex systems*. PhD thesis, Technical University of Dortmund, Germany, 2003. URL https://d-nb.info/969717474/34.

[134] Tomas Novickis. Protocol state fuzzing of an OpenVPN, 2016. URL https://www.ru.nl/publish/pages/769526/tomas_novickis.pdf. Master's thesis, Radboud University.

[135] National Institute of Standards and Technology. Guide to SSL VPNs. Technical Report NIST Special Publication (SP) 800-113, U.S. Department of Commerce, Washington, D.C., July 2008. URL https://doi.org/10.6028/NIST.SP.800-113.

[136] Christian Oliva and Luis Fernando Lago-Fernández. Stability of internal states in recurrent neural networks trained on regular languages. *Neurocomputing*, 452:212–223, 2021. doi: 10.1016/j.neucom.2021.04.058. URL https://doi.org/10.1016/j.neucom.2021.04.058.

[137] Christian W. Omlin and C. Lee Giles. Extraction of rules from discrete-time recurrent neural networks. *Neural Networks*, 9(1):41–52, 1996. doi: 10.1016/0893-6080(95)00086-0. URL https://doi.org/10.1016/0893-6080(95)00086-0.

[138] José Oncina and Pedro García. Identifying regular languages in polynomial time. *Advances in Structural and Syntactic Pattern Recognition*, 5:99–108, 1993. doi: 10.1142/9789812797919_0007.

[139] José Oncina and Pedro García. Identifying regular languages in polynomial time. *Advances in Structural and Syntactic Pattern Recognition*, 5:99–108, 1993. doi: 10.1142/9789812797919_0007. URL https://www.worldscientific.com/doi/abs/10.1142/9789812797919_0007.

[140] Warawoot Pacharoen, Toshiaki Aoki, Pattarasinee Bhattarakosol, and Athasit Surarerks. Active learning of nondeterministic finite state machines. *Mathematical Problems in Engineering*, 2013:1–11, 2013. doi: 10.1155/2013/373265.

[141] Andrea Palmieri, Paolo Prem, Silvio Ranise, Umberto Morelli, and Tahir Ahmad. MQTTSA: A tool for automatically assisting the secure deployments of MQTT brokers. In Carl K. Chang, Peter Chen, Michael Goul, Katsunori Oyama, Stephan Reiff-Marganiec, Yanchun Sun, Shangguang Wang, and Zhongjie Wang, editors, *2019 IEEE World Congress on Services, SERVICES 2019, Milan, Italy, July 8-13, 2019*, pages 47–53. IEEE, 2019. ISBN 978-1-7281-3851-0. doi: 10.1109/SERVICES.2019.00023. URL https://doi.org/10.1109/SERVICES.2019.00023.

[142] Joshua Pereyda. boofuzz, 2023. URL https://github.com/jtpereyda/boofuzz. Accessed: 2023-04-18.

[143] Andrea Pferscher. Active model learning of timed automata via genetic programming, 2019. URL https://apferscher.github.io/docs/masters-thesis.pdf. Master's thesis, Graz University of Technology.

[144] Andrea Pferscher. Supplemental material for: "Fingerprinting Bluetooth Low Energy via active automata learning", May 2021. URL https://github.com/apferscher/ble-learning.

[145] Andrea Pferscher. Supplemental material for: "Stateful black-box fuzzing of BLE devices using automata learning", January 2022. URL https://git.ist.tugraz.at/apferscher/ble-fuzzing.

[146] Andrea Pferscher and Bernhard K. Aichernig. Learning abstracted non-deterministic finite state machines. In Valentina Casola, Alessandra De Benedictis, and Massimiliano Rak, editors, *Testing Software and Systems - 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9-11, 2020, Proceedings*, volume 12543 of *Lecture Notes in Computer Science*, pages 52–69. Springer, 2020. ISBN 978-3-030-64880-0. doi: 10.1007/978-3-030-64881-7_4. URL https://doi.org/10.1007/978-3-030-64881-7_4.

[147] Andrea Pferscher and Bernhard K. Aichernig. Fingerprinting Bluetooth Low Energy devices via active automata learning. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 524–542. Springer, 2021. ISBN 978-3-030-90869-0. doi: 10.1007/978-3-030-90870-6_28. URL https://doi.org/10.1007/978-3-030-90870-6_28.

[148] Andrea Pferscher and Bernhard K. Aichernig. Stateful black-box fuzzing of Bluetooth devices using automata learning. In Deshmukh et al. [51], pages 373–392. ISBN 978-3-031-06772-3. doi: 10.1007/978-3-031-06773-0_20. URL https://doi.org/10.1007/978-3-031-06773-0_20.

[149] Andrea Pferscher and Bernhard K. Aichernig. Fingerprinting and analysis of Bluetooth devices with automata learning. *Formal Methods in System Design*, 2023. doi: 10.1007/s10703-023-00425-y. URL https://doi.org/10.1007/s10703-023-00425-y.

[150] Andrea Pferscher, Benjamin Wunderling, Bernhard K. Aichernig, and Edi Muškardin. Mining digital twins of a VPN server. In Eduard Kamburjan and Stefan Hallerstede, editors, *Preproceedings of the Workshop on Applications of Formal Methods and Digital Twins*, number 505 in Technical Report. Department of Informatics, University of Oslo, 2023. ISBN 978-82-7368-605-3. URL https://fm2023.isp.uni-luebeck.de/index.php/workshop-applications-of-formal-methods-and-digital-twins/.

[151] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. In Mauro Pezzè, editor, *Proceedings of the International Workshop on Test and Analysis of Component Based Systems, TACoS 2004, Barcelona, Spain, March 27-28, 2004*, volume 116 of *Electronic Notes in Theoretical Computer Science*, pages 59–71. Elsevier, 2004. doi: 10.1016/j.entcs.2004.02.086. URL https://doi.org/10.1016/j.entcs.2004.02.086.

[152] BlueZ Project. BlueZ - Official linux Bluetooth protocol stack. URL http://www.bluez.org/. Accessed: 2023-05-29.

[153] Santiago Hernández Ramos, M. Teresa Villalba, and Raquel Lacuesta. MQTT security: A novel fuzzing approach. *Wireless Communications and Mobile Computing*, 2018, 2018. doi: 10.1155/2018/8261746. URL https://doi.org/10.1155/2018/8261746.

[154] Abdullah Rasool, Greg Alpár, and Joeri de Ruiter. State machine inference of QUIC. *CoRR*, abs/1903.04384, 2019. URL http://arxiv.org/abs/1903.04384.

[155] Maximilian Rindler. Implementing the Kearns-Vazirani algorithm for learning deterministic finite automata in AALpy, 2023. Bachelor's thesis, Graz University of Technology.

[156] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Control*, 103(2):299–347, 1993. doi: 10.1006/inco.1993.1021. URL https://doi.org/10.1006/inco.1993.1021.

[157] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new Bluetooth escalation targets. In Capkun and Roesner [31], pages 19–36. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/ruge.

[158] Rohith Raj S, Rohith R, Minal Moharir, and Shobha G. SCAPY - A powerful interactive packet manipulation program. In *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*, pages 1–5, 2018. doi: 10.1109/ICNEWS.2018.8903954.

[159] Marc Sebban and Jean-Christophe Janodet. On state merging in grammatical inference: A statistical approach for dealing with noisy data. In Tom Fawcett and Nina Mishra, editors, *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*, pages 688–695. AAAI Press, 2003. ISBN 1-57735-189-4. URL http://www.aaai.org/Library/ICML/2003/icml03-090.php.

[160] Ben Seri and Alon Livne. Exploiting BlueBorne in Linux-based IoT devices. Armis, Inc, 2019. URL https://www.armis.com/research/blueborne/. Accessed: 2023-07-03.

[161] Ben Seri, Gregory Vishnepolsky, and Dor Zusman. BLEEDINGBIT: The hidden attack surface within BLE chips. Armis, Inc, 2019. URL https://www.armis.com/research/bleedingbit/. Accessed: 2023-07-03.

[162] Muzammil Shahbaz and Roland Groz. Inferring Mealy machines. In Cavalcanti and Dams [35], pages 207–222. ISBN 978-3-642-05088-6. doi: 10.1007/978-3-642-05089-3_14. URL https://doi.org/10.1007/978-3-642-05089-3_14.

[163] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 521–538. IEEE Computer Society, 2017. ISBN 978-1-5090-5533-3. doi: 10.1109/SP.2017.46. URL https://doi.org/10.1109/SP.2017.46.

[164] Wouter Smeenk, Joshua Moerman, Frits W. Vaandrager, and David N. Jansen. Applying automata learning to embedded control software. In Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, volume 9407 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2015. ISBN 978-3-319-25422-7. doi: 10.1007/978-3-319-25423-4_5.

[165] Hannes Sochor, Flavio Ferrarotti, and Rudolf Ramler. Automated security test generation for MQTT using attack patterns. In Melanie Volkamer and Christian Wressnegger, editors, *ARES 2020: The 15th International Conference on Availability, Reliability and Security, Virtual Event, Ireland, August 25-28, 2020*, pages 97:1–97:9. ACM, 2020. ISBN 978-1-4503-8833-7. doi: 10.1145/3407023.3407078. URL https://doi.org/10.1145/3407023.3407078.

[166] Andreas Steffen, Martin Willi, Tobias Brunner, and Daniel Röthlisberger. strongSwan, 2023. URL https://www.strongswan.org/. Accessed: 2023-06-16.

[167] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011. ISBN 978-3-642-21454-7. doi: 10.1007/978-3-642-21455-4_8. URL https://doi.org/10.1007/978-3-642-21455-4_8.

[168] Chris McMahon Stone, Tom Chothia, and Joeri de Ruiter. Extending automated protocol state learning for the 802.11 4-way handshake. In Javier López, Jianying Zhou, and Miguel Soriano, editors, *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Proceedings, Part I*, volume 11098 of *Lecture Notes in Computer Science*, pages 325–345, Barcelona, Spain, 2018. Springer. ISBN 978-3-319-99072-9. doi: 10.1007/978-3-319-99073-6_16.

[169] Martin Tappler. *Learning-Based Testing in Networked Environments in the Presence of Timed and Stochastic Behaviour*. PhD thesis, Graz University of Technology, 2019. URL https://mtappler.files.wordpress.com/2019/12/thesis.pdf.

[170] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. Model-based testing IoT communication via active automata learning. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 276–287, Tokyo, Japan, 2017. IEEE Computer Society. ISBN 978-1-5090-6031-3. doi: 10.1109/ICST.2017.32.

[171] Martin Tappler, Bernhard K. Aichernig, Kim Guldstrand Larsen, and Florian Lorber. Time to learn - Learning timed automata from tests. In Étienne André and Mariëlle Stoelinga, editors, *Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings*, volume 11750 of *Lecture Notes in Computer Science*, pages 216–235. Springer, 2019. ISBN 978-3-030-29661-2. doi: 10.1007/978-3-030-29662-9_13. URL https://doi.org/10.1007/978-3-030-29662-9_13.

[172] Martin Tappler, Bernhard K. Aichernig, Giovanni Bacci, Maria Eichlseder, and Kim G. Larsen. L^*-based learning of Markov decision processes (extended version). *Formal Aspects of Computing*, 33(4-5):575–615, 2021. doi: 10.1007/s00165-021-00536-5. URL https://doi.org/10.1007/s00165-021-00536-5.

[173] Martin Tappler, Edi Muškardin, Bernhard K. Aichernig, and Ingo Pill. Active model learning of stochastic reactive systems. In Radu Calinescu and Corina S. Pasareanu, editors, *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings*, volume 13085 of *Lecture Notes in Computer Science*, pages 481–500. Springer, 2021. ISBN 978-3-030-92123-1. doi: 10.1007/978-3-030-92124-8_27. URL https://doi.org/10.1007/978-3-030-92124-8_27.

[174] Martin Tappler, Bernhard K. Aichernig, and Florian Lorber. Timed automata learning via SMT solving. In Deshmukh et al. [51], pages 489–507. ISBN 978-3-031-06772-3. doi: 10.1007/978-3-031-06773-0_26. URL https://doi.org/10.1007/978-3-031-06773-0_26.

[175] Martin Tappler, Filip Cano Córdoba, Bernhard K. Aichernig, and Bettina Könighofer. Search-based testing of reinforcement learning. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 503–510. ijcai.org, 2022. doi: 10.24963/ijcai.2022/72. URL https://doi.org/10.24963/ijcai.2022/72.

[176] Martin Tappler, Andrea Pferscher, Bernhard K. Aichernig, and Bettina Könighofer. Learning and repair of deep reinforcement learning policies from fuzz-testing data. In *46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. IEEE, 2024. accepted on August 23, 2023.

[177] Peter Tiňo and Jozef Sajda. Learning and extracting initial Mealy automata with a modular neural network model. *Neural Computation*, 7(4):822–844, 1995. doi: 10.1162/neco.1995.7.4.822. URL https://doi.org/10.1162/neco.1995.7.4.822.

[178] Masaru Tomita. Dynamic construction of finite-state automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Ann Arbor, MI, USA, 4–6 August 1982*, pages 105–108, 1982. URL https://apps.dtic.mil/sti/pdfs/ADA120123.pdf.

[179] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software-Concepts and Tools*, 17(3):103–120, 1996.

[180] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. ISBN 978-3-540-78916-1. doi: 10.1007/978-3-540-78917-8_1. URL https://doi.org/10.1007/978-3-540-78917-8_1.

[181] Petar Tsankov, Mohammad Torabi Dashti, and David A. Basin. Semi-valid input coverage for fuzz testing. In Mauro Pezzè and Mark Harman, editors, *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 56–66. ACM, 2013. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2483787. URL https://doi.org/10.1145/2483760.2483787.

[182] Jeff Turner, Mark J. Schertler, Mark S. Schneider, and Douglas Maughan. Internet Security Association and Key Management Protocol (ISAKMP). RFC 2408, November 1998. URL https://www.rfc-editor.org/info/rfc2408.

[183] Vladimir Ulyantsev, Ilya Zakirzyanov, and Anatoly Shalyto. BFS-based symmetry breaking predicates for DFA identification. In Adrian-Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, volume 8977 of *Lecture Notes in Computer Science*, pages 611–622. Springer, 2015. ISBN 978-3-319-15578-4. doi: 10.1007/978-3-319-15579-1_48. URL https://doi.org/10.1007/978-3-319-15579-1_48.

[184] Frits W. Vaandrager, Roderick Bloem, and Masoud Ebrahimi. Learning Mealy machines with one timer. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *Language and Automata Theory and Applications - 15th International Conference, LATA 2021, Milan, Italy, March 1-5, 2021, Proceedings*, volume 12638 of *Lecture Notes in Computer Science*, pages 157–170. Springer, 2021. ISBN 978-3-030-68194-4. doi: 10.1007/978-3-030-68195-1_13. URL https://doi.org/10.1007/978-3-030-68195-1_13.

[185] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984. doi: 10.1145/1968.1972. URL https://doi.org/10.1145/1968.1972.

[186] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, 1973. doi: 10.1007/BF01068590. URL https://doi.org/10.1007/BF01068590.

[187] Sicco Verwer and Christian A. Hammerschmidt. flexfringe: A passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 638–642. IEEE Computer Society, 2017. ISBN 978-1-5386-0992-7. doi: 10.1109/ICSME.2017.58. URL https://doi.org/10.1109/ICSME.2017.58.

[188] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. An algorithm for learning real-time automata. In Maarten van Someren, Sophia Katrenko, and Pieter Adriaans, editors, *Proceedings of the Sixteenth Annual Machine Learning Conference of Belgium and the Netherlands (Benelearn)*, pages 128–135, 2007. URL http://resolver.tudelft.nl/uuid:a202b4cf-5153-4ad5-b41d-5d0332bf04f2.

[189] Neil Walkinshaw, John Derrick, and Qiang Guo. Iterative refinement of reverse-engineered models by model-based testing. In Cavalcanti and Dams [35], pages 305–320. ISBN 978-3-642-05088-6. doi: 10.1007/978-3-642-05089-3_20. URL https://doi.org/10.1007/978-3-642-05089-3_20.

[190] Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 5244–5253. PMLR, 2018. URL http://proceedings.mlr.press/v80/weiss18a.html.

[191] Gail Weiss, Yoav Goldberg, and Eran Yahav. Learning deterministic weighted automata with queries and counterexamples. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8558–8569, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/d3f93e7766e8e1b7ef66dfdd9a8be93b-Abstract.html.

[192] Valentina Wieser. Property-based testing of a web application using JavaScript, 2022. Bachelor's thesis, Graz University of Technology.

[193] Konstantin Windisch. Optimizations on active automata learning for observable nondeterministic finite state machines, 2022. Bachelor's thesis, Graz University of Technology.

[194] Martin Woolley. The Bluetooth Low Energy primer. Technical report, Bluetooth SIG, 2023. URL https://www.bluetooth.com/bluetooth-resources/the-bluetooth-low-energy-primer/. Accessed: 2023-05-18.

[195] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave (Jing) Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. BLESA: Spoofing attacks against reconnections in Bluetooth Low Energy. In Yarom and Zennou [198]. URL https://www.usenix.org/conference/woot20/presentation/wu.

[196] Benjamin Wunderling. Model learning and fuzzing of the IPsec-IKEv1 VPN protocol, 2023. Master's thesis (not yet submitted).

[197] Huan Yang, Yuqing Zhang, Yu-pu Hu, and Qixu Liu. IKE vulnerability discovery based on fuzzing. *Security and Communication Networks*, 6(7):889–901, 2013. doi: 10.1002/sec.628. URL https://doi.org/10.1002/sec.628.

[198] Yuval Yarom and Sarah Zennou, editors. *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, 2020. USENIX Association. URL https://www.usenix.org/conference/woot20.

[199] Anastasia Yastrebova, Ruslan Kirichek, Yevgeni Koucheryavy, Aleksey Borodin, and Andrey Koucheryavy. Future networks 2030: Architecture & requirements. In *10th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops, ICUMT 2018, Moscow, Russia, November 5-9, 2018*, pages 1–8. IEEE, 2018. ISBN 978-1-5386-9361-2. doi: 10.1109/ICUMT.2018.8631208. URL https://doi.org/10.1109/ICUMT.2018.8631208.

[200] Daniel M. Yellin and Gail Weiss. Synthesizing context-free grammars from recurrent neural networks. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and*

Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I, volume 12651 of *Lecture Notes in Computer Science*, pages 351–369. Springer, 2021. ISBN 978-3-030-72015-5. doi: 10.1007/978-3-030-72016-2_19. URL https://doi.org/10.1007/978-3-030-72016-2_19.

[201] Michael Zalewski. american fuzzy lop (AFL), 2023. URL https://lcamtuf.coredump.cx/afl/. Accessed: 2023-04-17.

[202] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. URL https://www.fuzzingbook.org/. Retrieved 2023-01-07 14:37:57+01:00.

[203] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Search-based fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. URL https://www.fuzzingbook.org/html/SearchBasedFuzzer.html. Retrieved 2023-01-07 15:11:42+01:00.

[204] Yingpei Zeng, Mingmin Lin, Shanqing Guo, Yanzhao Shen, Tingting Cui, Ting Wu, Qiuhua Zheng, and Qiuhua Wang. Multifuzz: A coverage-based multiparty-protocol fuzzer for IoT publish/subscribe protocols. *Sensors*, 20(18):5194, 2020. doi: 10.3390/s20185194. URL https://doi.org/10.3390/s20185194.

[205] Yousaf Bin Zikria, Rashid Ali, Muhammad Khalil Afzal, and Sung Won Kim. Next-generation Internet of Things (IoT): Opportunities, challenges, and solutions. *Sensors*, 21 (4):1174, 2021. doi: 10.3390/s21041174. URL https://doi.org/10.3390/s21041174.