# Vulnerability Detection in Mobile Applications Using State Machine Modeling

Wesley van der Lee

Delft University of Technology

Faculty of Electrical Engineering, Mathematics & Computer Science
Department of Intelligent Systems
Cyber Security Group

MSc. Thesis

# Vulnerability Detection in Mobile Applications Using State Machine Modeling

Wesley van der Lee

to obtain the degree of Master of Science in Computer Science
Data Science & Technology Track
with a specialization in Cyber Security
to be defended publicly on January 16th, 2018

Thesis Committee

| | |
|---|---|
| Dr. ir. J.C.A. van der Lubbe | Full Professor TUDelft |
| Dr. ir. M. Loog | Assistant Professor, TUDelft |
| Dr. ir. S. Verwer | Assistant Professor, TUDelft |
| R. van Galen | Cyber Security Consultant, KPMG |

**Wesley van der Lee**
*Vulnerability Detection in Mobile Applications Using State Machine Modeling*
MSc. Thesis, January 8, 2018

**Delft University of Technology**
*Cyber Security Group*
Department of Intelligent Systems
Faculty of Electrical Engineering, Mathematics & Computer Science

**KPMG**
*Cyber Security*
IT Advisory Services
Risk Consulting, Advisory

# Abstract

Mobile applications play a critical role in modern society. Although mobile apps are widely adopted, everyday news shows that the applications often contain severe security vulnerabilities. Recent work indicates that state machine learning has proven to be an effective method for vulnerability detection in software implementations. The state machine that can be learned about a software implementation provides additional insight into the internal software structure. The insight can then be used as input for security assessment which most of the times is performed by manual evaluation of the learned model.

In this thesis, we aim to extend state machine learning to improve the security of mobile applications in an automated way, solving two problem. The first problem is the lack of a methodology to learn state machines for mobile apps. The second problem is the need for an approach that detects vulnerabilities from the inferred models. To the best of our knowledge, there exists no framework that automatically infers behavioral state machine models on mobile Android applications, nor does there exist a methodology for automatic vulnerability detection on the inferred models.

We propose two solutions to the aforementioned problems. For the former, a framework for inferring a state machine model on general mobile Android applications is presented, which uses active state machine learning algorithms to ensure time optimization and model correctness on the learning process. For the latter, we designed algorithms that use the inferred models and determine the presence of vulnerabilities. We combine both solutions and propose a novel testing methodology that gains new insights into the behavior of an app and achieves the goal of vulnerability detection. The methodology identified relevant security weaknesses in numerous Android apps. Moreover, the solution can detect rogue applications such as a malicious WhatsApp version in the Android Play Store, which affected over a million devices in three days on November 2017.

*For family and friends, and friends that are family.*

- Wesley van der Lee
Delft, 2018

# Contents

# 1 Introduction

> *What we have to learn to do we learn by doing*
>
> — **Aristotle**
> The Nicomachean Ethics

Mobile applications play an ever-more important role in modern society. Mobile apps are the gateways to use social media, to perform banking transactions, to schedule trips and much more [1]. The same type of applications are not only limited to run on mobile phones but can also run on other smart devices such as smart televisions, smart watches and smart cars. The multi-platform deployment of applications is particularly true for applications that are designed to run on the Android operating system since Android is the most popular operating system for smart devices[1]. As a result, Android applications facilitate a multitude of services for everyday life. Although the Android platform is well-established, the security of Android applications is not. The conclusion became especially true when in 2016 researchers of the Norwegian firm Promon were able to exploit Tesla's Android application and achieve full control of the Tesla car paired with the application[2]. Also in 2016, the research institute Fraunhofer SIT found exploitable vulnerabilities in 9 popular password managers for Android that could compromise the passwords locally stored by the user [3]. These examples illustrate that although today's society moves towards a pervasive adoption of mobile applications, the applications are insufficiently secured.

The main reason why applications fail to meet modern security standards is that software security is part of a trade-off where development methodology and time-to-market play an essential role [4]. An early time-to-market brings an economical strategic advantage for companies because of two reasons [5]. First of all, when a company launches an application as soon as possible inside a niche domain, they can become market leaders, with the ability to lock in users. Secondly, publishing software expeditiously also generates an early revenue. On the other hand, a software development life cycle that implements security, such as test-driven development, might consume more time thus delaying the time-to-market, but results in a more secure application.

Modern security testing tools aid the process of discovering bugs by applying a multitude of automated testing techniques that come in three flavors: white-box, grey-box and black-box testing [6]. White-box testing examines the application's internal logic by code reviews or specific tests. Grey-box testing tests the software's logic using metadata, such as documentation or file structure. Black-box testing interacts with the software as an application and determines whether a given input returns the correct output.

[1] `https://developer.android.com/about/android.html`

Black-box testing techniques can also be fine-tuned to understand the application's internal logic, by modeling the application logic as a state machine. A state machine visually graphs the software behavior, shows which application input lead to which state and depicts the corresponding application responses for a given input. Modeling a state machine with black-box testing is precisely what a state machine learning algorithm does by observing a large number of *traces*: a combination of inputs and outputs. The inferred state machine reveals detailed information about the application's logic, and can hence function as input for the identification of vulnerabilities and weaknesses in the software application. The identification of bugs or vulnerabilities from inferred state machines has been achieved for a wide range of software systems, such as various driver implementations for the TLS protocol [7]. The last mentioned study assessed models for the existence of extraneous transitions or states and concluded that the visual insight of the state machine aided the identification of vulnerabilities in the implementation. Another way to use state machine learning is the to verify if a software implementation meets specific requirements. A state machine can be modeled to function as a reference model that is based on the predefined software requirements. The reference model can be compared to the inferred state machine of the application. Discrepancies between the two state machines can then be further investigated to assess whether the implementation concurs the requirement model. One could also learn a state machine model for formal documentation. Learning a model for the purpose to establish documentation, has been performed for the chip in the Dutch biometric passport. It was more time efficient to infer a state machine than having a team of experts establish such a model [8].

State machine models can be learned from a set of existing traces (*passive learning*) or a set of traces that are generated while learning (*active learning*). The traces are generated by interacting with an application and observing input and output combinations. The drawback of passive learning is that the model is incomplete in the variety of existing traces, i.e., when the set of traces does not describe particular application behavior, the inferred model does not specify this behavior either. Active learning overcomes the shortcoming by querying for the information it needs to know. Software systems, and therefore also Android applications, can function as a data source to generate the required traces because the applications can respond interactively. The drawback of applying active learning to software systems is that interacting with an application consumes time, as each input needs to be simulated and each output is required to be observed. To improve the learning process, different active learning algorithms have been developed, such as the L* and TTT algorithm.

Active state machine learning is often implemented according to the *Minimally Adequate Teacher* (MAT) framework as first proposed by Angluin [9], which is composed of a learner and a teacher. The goal of the learner is to infer the state machine model of a system under test (SUT), by posing membership queries and equivalence queries to the teacher. Membership queries ask whether the SUT recognizes a specific behavior input. The combination of a query and answer form a trace, and a hypothesized model can be constructed after a sufficient amount of traces are generated. The model learner poses an equivalence query to the teacher for the built hypothesis. The query determines if the model correctly describes the SUT's input/output behavior. Learning halts when the hypothesized model

is equivalent. If the model incorrectly describes the SUT's behavior, the teacher provides a trace which distinguishes the model and the SUT. The trace is also called a counterexample because it invalidates the hypothesized model. The learner utilizes the counterexample to refine the hypothesis. The process repeats itself until an equivalence query yields success. Figure 1.1 visually depicts the discussed methods of the MAT framework.

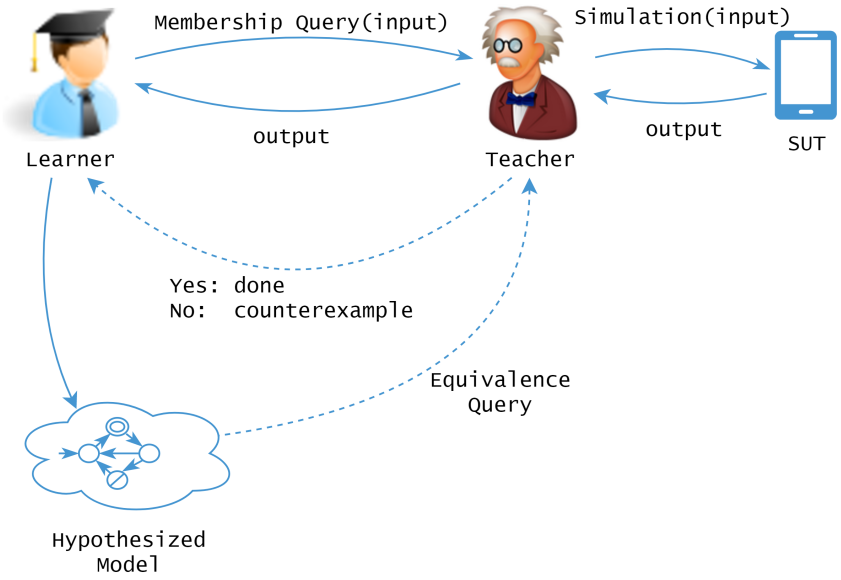



**Fig. 1.1:** Active Learning with the MAT Framework

A vital component of the discussed framework is the ability of the teacher to verify equivalence between the hypothesized model and the unknown model that is implemented by the SUT. Equivalence between a hypothesized model and the SUT is determined through approximation. The teacher generates a number of test cases and verifies whether the model's output is equal to the outputs of the SUT. If a given test case yields a different result, the model is inequivalent and the test case itself forms the counterexample. Numerous algorithms generate the test cases such as randomized input sequences and the W-method [10]. A common drawback of the test case generation algorithms is that the tests are insufficiently diverse or excessively long. Insufficiently diverse test cases prematurely determine a model to be equivalent to the implementation, whereas the model is inequivalent to the implementation. Excessively long test cases consume too much time, which makes equivalence testing infeasible. Test case generation is a study on its own and this thesis will only touch upon some model conformance algorithms that we apply for equivalence approximation.

Because the model that is inferred by active state machine learning manifests additional information about the application, the model might also be used as input to assess the application's security. Up until now, most research that involves active learning stops when the model is automatically inferred and continues with manual model inspection to conclude a security assessment, such as the identification of extraneous behavior. The security assessment thus also depends on the reviewer and may yield different results for different reviewers, as there exists no standardized methodology. Furthermore, research on retrieving such a model for mobile Android

applications is insufficient. There is only one study performed by Lampe et al. that developed a tool for a specific application to infer a state machine model [11]. This model was also manually reviewed, and although the inferred model nor the review results were publically published, the authors were able to apply state machine learning to a single mobile application successfully. The research provides a limited framework for model inference of mobile applications. Despite the limitations, the proposed framework can be used as a basis for this research, but the framework's restrictions need to be overcome. Furthermore, we aim to utilize the inferred model as a data source for an automated security assessment, such that the security evaluation is not prone to human error when attempting to discover weaknesses in the model.

## 1.1 Outline

This thesis describes the conducted research on how to infer a correct state machine model for mobile Android applications and assess its security in an automated way. To infer an accurate state machine model we review active automata learning algorithms and solve challenges such as the equivalence approximation between a model and an application. Furthermore, we need to establish algorithms that identify vulnerabilities on the input of an inferred model. The primary goal of this thesis is to use these building blocks to answer the following main research question:

*How can one identify weaknesses in mobile Android applications through feasible behavioral state machine learning?*

To aid the process of answering the main question, the question has been divided into the following sub-questions:

**RQ 1. How can model learning be extended to apply to mobile Android applications?**
This research question mainly focuses on reducing or mitigating the limitations of the framework provided by Lampe et al. [11]. This question deals with the practical obstacles that arise when introducing active learning to the mobile application domain.

**RQ 2. How can the feasibility of model learning for of Android applications be improved?** Active learning from simulations is time-consuming. Different active learning algorithms reduce the time complexity by limiting the number of queries and the overall query length. This question focuses on the different learning algorithms and corresponding attributes such as model equivalence approximation.

**RQ 3. How can the learned model be used to assess the application's security?** The novelty of this thesis lies in the application of active learning on Android applications and the automatic processing of the model as a new data source to assess the application's security. The latter inquires a set of identification algorithms that determine the presence of vulnerabilities on the input of the inferred model.

My initial expectation is that an extension of the framework proposed by Lampe et al. can achieve model inference of Android applications. The tool has not been maintained since its publication from 2015, therefore to be able to launch the tool is already a starting requirement. Moreover, we hypothesize that the inferred model can function as a data source for identifying vulnerabilities. Before we can discover vulnerabilities from state machines, we likely need to add additional information to the model to precisely describe a state. For example, the type of the incurred network requests when performing a specific action. Given the assumption that the inferred model describes the entire application, certain invariants for the application must hold. An example of such an invariant could be that all network requests performed by the application are done over an encrypted connection, i.e., connections over SSL.

The research that is presented in this thesis has also be submitted as a scientific paper to the first workshop on Security Protocol Implementations: Development and Analysis (SPIDA)[2]. The workshop is organized in conjunction with IEEE EuroS&P 2018. At the moment of publishing this thesis, the paper is in the process of external review.

The structure of this thesis is as follows. Chapter 2 gives an overview of the building blocks of active state machine learning, where various learning algorithms are discussed, as well as techniques that solve the model equivalence problem. Chapter 3 reviews the prior work of Lampe et al. by elaborating on the framework they proposed and identifying its limitations. Chapter 4 discusses requirements to overcome the recognized limitations and introduces a solution framework that is developed based on these requirements. Chapter 5 describes algorithms that identify security vulnerabilities in the learned models. Chapter 6 depicts the results of the resulting proof of concept running on various mobile Android applications. Chapter 7 discusses validation and the results. Chapter 8 answers the research questions and provides a perspective ahead by presenting future work references.

[2] https://spida.cs.ru.nl/

# 2 Building Blocks of Model Inference

> *Quid opus est verbis?*
> *(What need is there for words?)*
>
> — **Terence**

*Up until now, we discussed in what way active state machine learning can contribute to the security of mobile applications. Active state machine learning utilizes a learner and a teacher, where the learner is guided by active learning algorithms to ask the right questions and the teacher is responsible for providing correct answers. For the teacher to be able to answer the questions posed by the learner, the teacher must have access to an oracle that solves the equivalence relation between an inferred model and a mobile application.*

This chapter discusses methodologies that are essential to achieve state machine learning. Because an active state machine learning algorithm guides the learner, the algorithm that first proposed the MAT framework, the L* algorithm, is explained, as well as an improved version of L*, the TTT algorithm. Another technique that is imperative in the MAT framework is the equivalence oracle that is utilized by the teacher. The oracle is responsible for refinement of the learner's hypothesis state machine and eventually ascertains the halting condition of learning. There exist different algorithms that enact the oracle to answer equivalence relations, such as the RandomWalk algorithm and the W-method.

This chapter is organized as follows. To consistently discuss varied literature, Section 2.1 establishes a uniform terminology that is used throughout this thesis and provides the prerequisite knowledge that is required for components of the MAT framework. Section 2.2 and 2.4 reviews the L* and TTT active state machine learning algorithms respectively. At last, building blocks that can contribute to the equivalence oracle are examined in Section 2.5.

## 2.1 Prerequisite terminology

When discussing the building blocks of model inference, the related work applies their terminology. To keep the terminology consistent, this section establishes a formal mathematical notation. The remainder of this thesis uses the presented formal mathematical notation. The notation will be consistent with the notation that is proposed by Sipser [12].

The deterministic finite automaton (DFA) $U$ is used to formalize state machines. $U$ can be defined as follows:

**Definition 1** (*Deterministic Finite Automaton*)**.** A DFA can be formalized by a 5-tuple $U = (Q, \Sigma, \delta, q_0, F)$, where

1. $Q$ is a finite set called the ***states***,
2. $\Sigma$ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \to Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

An example DFA $\mathcal{A}$ is depicted in Figure 2.1. At this point and throughout this chapter, $U$ is an abstract DFA that is utilized for generic statements about DFAs, whereas $\mathcal{A}$ is an actual example DFA and has defined all fields as described in Definition 1.

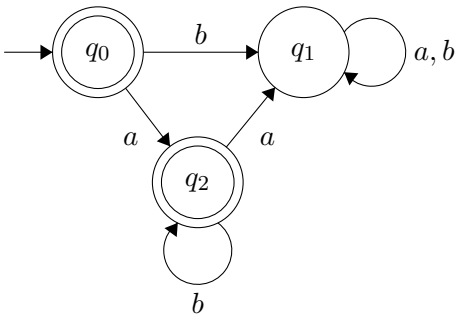



**Fig. 2.1:** Example DFA $\mathcal{A}$.

The example DFA $\mathcal{A}$ shows three states: $q_0$, $q_1$ and $q_2$ depicted by the circles. Double edged circles indicate that the state is an accepting state, which in the example of $\mathcal{A}$ is the case for $q_0$ and $q_2$. Arrows and their labeled input symbol indicate transitions from one state to another state. From the example DFA it can be observed that the input alphabet $\Sigma$ of $\mathcal{A}$ solely consists of the input symbols $a$ and $b$.

$\Sigma^*$ is the set of words over symbols in $\Sigma$, including the empty word $\varepsilon$. The $^*$-notation follows from the unary operation, which attaches any number of strings in $\Sigma$ together: $\Sigma^* = \{x_1, x_2 \ldots, x_{k-1}, x_k | k \geq 0 \text{ and each } x_i \in \Sigma\}$. For all input sequences in $\mathcal{A}$ one can see that word $w_1 = abbb \in \Sigma^*$, because $w_1$ leads to an accepting state[1]. A word $w_2 = abab$ can be identified as not a member of $\Sigma^*$: $w_2 \notin L(\mathcal{A})$[2]. For words $w, w' \in \Sigma^*$, $w \cdot w'$ is a concatenation-operation of the two words. The concatenation-operation of two words can also be written by omitting the operator $\cdot$ and just write $ww'$.

[1] $\delta(q_0, abbb) = \delta(\delta(q_0, a), bbb) = \delta(\delta(\delta(q_0, a), b), bb) = \delta(\delta(\delta(\delta(q_0, a), b), b), b) = \delta(\delta(\delta(q_1, b), b), b) = \delta(\delta(q_1, b), b) = \delta(q_1, b) = q_2 \in F \to abbb \in L(\mathcal{A})$.

[2] $\delta(q_0, abab) = \delta(\delta(q_0, a), bab) = \delta(\delta(\delta(q_0, a), b), ab) = \delta(\delta(\delta(\delta(q_0, a), b), a), b) = \delta(\delta(\delta(q_2, b), a), b) = \delta(\delta(q_2, a), b) = \delta(q_1, b) = q_1 \notin F \to abab \notin L(\mathcal{A})$.

In general, a state $q' \in Q$ which is the result of a transition from another state $q \in Q$ with input letter $a \in \Sigma$, is also called the $a$-successor of $q$, i.e. $q' = \delta(q, a)$ is the $a$-successor of $q$. In the example of DFA $\mathcal{A}$, state $q_1$ is the $b$-successor of state $q_0$. The successor-notation can also be extended for words by defining $\delta(q, \varepsilon) = q$ and $\delta(q, wa) = \delta(\delta(q, w), a)$ for $q \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$. For words $w \in \Sigma^*$ the transition function $\delta(q, w)$ implies the extended transition function: $\delta(q, w) = \delta(q_0, w)$ unless specifically specified. Moreover, by also utilizing the notation of Vazirani et al. [13]. A state of a general DFA $U$ can be denoted as $U[w]$ for $w \in \Sigma^*$, where $U[w]$ corresponds to the state in $U$ reached by $w$, i.e. $U[w] = \delta(q_0, w)$. For $q \in Q$ if $U[w] = q$ then $w$ is also called an *access sequence* for $q$. The singleton transition without an associating input letter, indicates the empty transition, which points to the initial starting state of $\mathcal{A}$.

Furthermore, for words $w \in \Sigma^*$ the state $U[w]$ of a DFA $U$ either results in an accepting state or a rejecting state. Double-edged nodes represent accepting states, whereas single-edged nodes represent rejecting states. For states $q \in Q$ that are accepting states, it also holds that $q \in F$. The accepting states of $\mathcal{A}$ are $q_0$ and $q_2$. The set of all words $w \subseteq \Sigma^*$ that DFA $U$ accepts is called the language of $U$ indicated by $L(U)$. A language can be infinite as $\Sigma^*$ is unbounded. This is the case for DFA $\mathcal{A}$ as it accepts an input existing of any number of $b$'s after a single $a$, i.e. $\{\varepsilon, a, ab, abb, abbb, \cdots, ab^*\}$. The $\lambda$-function also evaluates if a DFA accepts an input sequence. For a word $w \in \Sigma^*$ the $\lambda$-evaluation $\lambda(w)$ returns 1 iff the DFA in question accepts word $w$, that is if the extended transition function for $q_0$ concludes in an accepting state. The function $\lambda(w)$ returns 0 if the extended transition function results in a rejecting state.

**Example 2.1.1.** DFA $\mathcal{A}$ from Figure 2.1, can be formally written as the 5-tuple $\mathcal{A} = \{Q, \Sigma, \delta, q_0, F\}$ where

$Q = \{q_0, q_1, q_2\}$,
$\Sigma = \{a, b\}$,
$\delta$ is described as:

| | $a$ | $b$ |
|---|---|---|
| $q_0$ | $q_2$ | $q_1$ |
| $q_1$ | $q_1$ | $q_1$ |
| $q_2$ | $q_1$ | $q_2$ |

$q_0$ is the starting state, and
$F = \{q_0, q_2\}$.

## 2.2 The L* Algorithm: The Basic Building Block of Model Inference

*Learning Regular Sets from Queries and Counterexamples* by Dana Angluin [9] forms the basis of many modern state machine inference algorithms. Her research introduces the polynomial L* algorithm for learning a regular set, a task which before was computationally intractable because it was proven to be NP-hard [14]. A regular set represents the value of expressions that describe languages or regular expressions. Expressions are regular if they are created by regular operators, such as union and

intersection [12]. The reason to infer a regular language is that if a set is regular, it can be modeled by a DFA.

**Example 2.2.1.** The regular expression to express DFA $\mathcal{A}$ is $(\varepsilon \cup ab^*)$, since it accepts the empty string and a single $a$ followed by any number of $b$'s.

The basic idea of the L* algorithm is a learner whose goal is to create a conjecture state machine model by utilization of an expert system, the Minimally Adequate Teacher (MAT). A conjecture is a hypothesized DFA that approximates the SUT's behavior and is either equivalent or not. Two types of queries that are posed to the teacher enable a learner to infer a state machine model:

- **membership queries** that answer *yes* or *no* for an input word $w$ depending on whether $w$ is a member of the to be hypothesized DFA. This is equivalent to the $lambda$-evaluation for a word $w$ that depicts whether $w$ is recognized by the set U: $\lambda(w) \rightarrow \{0, 1\}$.
- **equivalence queries** that take as input a conjecture DFA and then answers *yes* if the conjecture is equal to the set $U$. If this is not the case, the MAT provides a counterexample, which is a string $w'$ in the symmetric difference of the conjecture and the unknown language.

The learner keeps track of the queried strings, classified by either a member or non-member of the unknown regular set U. This information is organized in an observation table that consists of three fields: a nonempty finite prefix-closed set $S$ of strings, a nonempty finite suffix-closed set $E$ of strings and a finite function $T$ that maps all entries that are formed by concatenating the prefix and suffix together, see Figure 2.2. A set is *prefix-closed* if all prefixes of every member of the set is also a member of the set. Suffix-closed is defined analogously. A word $u$ recognized by the set $U$ can be typical of the form $u = sae$, for $s \in S$, $e \in E$ and $a \in \Sigma$, meaning a word starts with a prefix and ends with a suffix. Sometimes a one-letter extension in $\Sigma$ is added to the prefix. The one-letter prefix extension identifies transitions in the hypothesized model. The latter will become apparent in the running example of the L* algorithm. Angluin applies the notation of words in the form $u = sae$ as $u \in ((S \cup S \cdot \Sigma) \cdot E)$, thus if $u \in U$ then $T(u)$ will return $1$ and $0$ otherwise. Function T thus corresponds to the earlier mentioned $\lambda$-function: $T(w) \mathrel{\widehat{=}} \lambda(w)$. To maintain consistency with Angluin's work, the remainder of this section uses the $T$ notation for to depict a lambda evaluation. In conclusion, an observation table can be denoted as a 3-tuple $(S, E, T)$.

To clarify the terminology, imagine the example where a learner is required to learn the behavior of DFA $\mathcal{A}$ described in Figure 2.1. Initially, any information except $\mathcal{A}$'s alphabet $\Sigma$ is unknown to the learner, but the learner has access to a MAT that can answer membership and equivalence queries. The learner starts by first creating the observation table. Set $S$ initially contains the input alphabet of $\mathcal{A}$ and the empty string $\varepsilon$ as one-letter prefixes. Thus $S = \{\varepsilon\} \cup \Sigma = \{\varepsilon, a, b\}$ The commencing distinguishing suffix is $\varepsilon$, therefore set $E = \{\varepsilon\}$. The learner then fills the entries in observation table $S \times E$ with the output of membership queries, depending on whether an entry $e \in S \times E$ is accepted by $\mathcal{A}$ or not. This leads to the initial observation table depicted in Figure 2.2a. Note that the table vertically distinguishes two sections that are separated by an additional line. That is, set $S$ is divided into

two subsets. The top section of $S$ represents all distinct rows with respect to the outcome of $T$. Since $row(\varepsilon) = 1$ and $row(b) = 0$, those distinct rows are put in the top part of $S$. $Row(a)$ results to 1, which is equivalent to $row(\varepsilon)$, hence $row(a)$ is put in the bottom part of $S$.

The learner queries for the right amount of data from the MAT, by ensuring that the observation table is both closed and consistent. An observation table is *closed* if for every entry $e \in S \times \Sigma$, there exists an element $s \in S$ such that $row(e) = row(s)$. If the table is not closed, it lacks information for transitions. Table 2.2a is not closed, because this property is not ensured for the state $\mathcal{A}[b]$. The table depicts at least two states because there are two distinct rows. However, $\mathcal{A}[b]$ requires information on where to transit from that state. Hence the access sequence of the state $\mathcal{A}[b]$ appended with one-letter extensions are added to the set of prefixes. In other words, $\{ba, bb\}$ are to be added to the set $S$. This results in a bigger observation table, depicted in Table 2.2b. The learner identifies which suffix distinguishes the rows and adds the word to the set $S$. An observation table is *consistent* when for all $s_1, s_2 \in S$ where $row(s_1) = row(s_2)$, if for all $a$ in the alphabet $\Sigma$ holds that $row(s_1 \cdot a) = row(s_2 \cdot a)$. If the table is inconsistent, the language would be non-deterministic and can thus not be modeled by a deterministic finite automaton. The learner identifies for which $s_1, s_2 \in S$ distinguishes the result of output $T$ and adds the word to the set $E$. Given the observation table in 2.2b, the observation table is consistent.

| | $\varepsilon$ |
|---|---|
| $\varepsilon$ | 1 |
| $b$ | 0 |
| $a$ | 1 |

**(a)**

| | $\varepsilon$ |
|---|---|
| $\varepsilon$ | 1 |
| $b$ | 0 |
| $a$ | 1 |
| $ba$ | 0 |
| $bb$ | 0 |

**(b)**

| | $\varepsilon$ | $b$ |
|---|---|---|
| $\varepsilon$ | 1 | 0 |
| $b$ | 0 | 0 |
| $a$ | 1 | 1 |
| $ba$ | 0 | 0 |
| $bb$ | 0 | 0 |
| $aa$ | 0 | 0 |
| $ab$ | 1 | 1 |

**(c)**

**Fig. 2.2:** Gradually growing observation tables corresponding to various steps of the L* algorithm: (a) initial observation table $T_0$, (b) observation table $T_1$ that is a closed and consistent version of the initial observation table, (c) final observation table $T_2$ describing DFA $\mathcal{A}$.

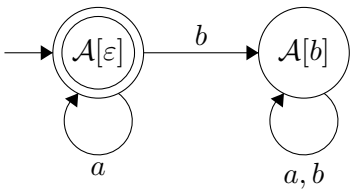



**Fig. 2.3:** Hypothesized conjecture DFA $H_0$ corresponding to observation table $T_1$ (Table 2.2b).

If $(S, E, T)$ is closed and consistent, an acceptor $H(S, E, T)$ can be defined which is consistent with function $T$. $H$ then forms the hypothesized model based on the

contents of the observation table. A conjecture $H = (\Sigma, Q, \delta, q_0, F)$ can be modeled as follows:

1. $Q = \{row(s) | s \in S_H\}$
2. $q_0 = row(\varepsilon)$
3. $F = \{row(s) | s \in S_H \text{ and } T_H(s, \varepsilon) = 1\}$
4. $\delta(row(s), a) = row(sa)$

**Example 2.2.2.** A hypothesized DFA $H_0$ generated according to the steps above consistent to the observation table 2.2b that has the following contents: $Q = \{\mathcal{A}[\varepsilon], \mathcal{A}[b]\}$, $q_0 = \mathcal{A}[\varepsilon]$, $F = \mathcal{A}[\varepsilon]$. Conjecture $H_0$ is visualized in Figure 2.3. The learner then verifies the conjecture to the MAT and receives a counterexample back, indicating that the conjecture is inequivalent to U. Although the learner was unable to see this, the model depicted in Figure 2.3 clearly differs from the SUT shown in Figure 2.1. Suppose that the learner received the counterexample word $w = ab$. This is a valid counterexample because $\lambda_{\mathcal{A}}(ab) = 1$, $\lambda_{H_0}(ab) = 0$ and thus $\lambda_{\mathcal{A}}(ab) \neq \lambda_{H_0}(ab)$ holds.

The L* algorithm adds the entire counterexample and all its prefixes to $S$ to guarantee that the right suffix is added. Since the observation table is not consistent anymore, it will find a breakpoint on the distinguishing suffix for $b$. Letter $b$ is a breakpoint because analysis of the counterexample $w = ab$ shows that if suffix $b$ is added, $H_0$ predicts the wrong output. Element $b$ is therefore identified as a distinguishing suffix and hence added to set $E$. The observation table must now be updated again: the new entries caused by the appearance of the $b$-column are filled and to make the table closed again, new prefixes for the new state $\mathcal{A}[a]$ have to be determined. This process results in the observation table depicted in Table 2.2c. Following the steps to establish a DFA from an observation table yields the original DFA shown in Figure 2.1 which is the correct DFA that corresponds to the observation table $T_2$ shown in Table 2.2c. After the learner poses an equivalence query with this DFA, the MAT answers $yes$, indicating that the learner inferred the correct DFA and hence learning stops.

### 2.2.1 Why the inferred DFA is minimal

Up until now, the focus was for the conjecture to be consistent with $T$. Since a DFA with the state size equal to the number of observations, that is each observation leads to a new state in a conjecture, such a DFA would be incorrect because it possibly incorrectly models unforeseen future observations. A hypothesized conjecture $H$ should thus not only be consistent with $T$, but also have the smallest set of states to model the SUT's behavior. The L* algorithm only learns a DFA consistent with $T$ and that has the smallest set of states. Angluin proves this as follows. Let $q_0$ be the starting state ($row(\varepsilon)$) and $\delta$ be the transition function from one state to another in the acceptor, then for $s \in S$ and $a \in \Sigma$ holds that because $\delta(row(s), a) = row(s \cdot a)$ follows $\forall s \in (S \cup S \cdot \Sigma) : \delta(q_0, s) = row(s)$, thus the closed property ensures that any row in the observation table corresponds with a valid path in the acceptor. $\forall s \in (S \cup S \cdot \Sigma) \forall e \in E \rightarrow \delta(q_0, s \cdot e)$ is an accepting state if and only if $T(s \cdot e) = 1$, thus due to the consistency with finite function $T$, a word will be accepted by the acceptor if it is in the regular set. To see that $H(S, E, T)$ is the acceptor with the

least states, one must note that any other acceptor $H'$ consistent with $T$ is either isomorphic or equivalent to $H(S, E, T)$ or contains more states.

Algorithm 1 provides a complete overview of the L* algorithm.

**Algorithm 1** The L* Algorithm

**Input:** Access to the teacher functions $MQ$ and $EQ$ for respectively computing membership and equivalence queries
**Output:** Hypothesis DFA $H$

$S \leftarrow \{\varepsilon\}$
$E \leftarrow \{\varepsilon\}$
$(S, E, T) \leftarrow MQ$ for $\varepsilon$ and $\Sigma$ ▷ $(S, E, T)$ is the observation table
**while** $H$ is incorrect **do** ▷ $H$ is the conjecture
    **while** $(S, E, T)$ is not consistent or not closed **do**
        **if** $(S, E, T)$ is not consistent **then**
            find $s_1$ and $s_2$ in $S$, $a \in \Sigma$ and $e \in E$ such that:
            $row(s_1) = row(s_2)$ and $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$
            add $a \cdot e$ to $E$
            extend $T$ to $(S \cup S \cdot \Sigma) \cdot E$ using MQ
        **end if**
        **if** $(S, E, T)$ is not closed **then**
            find $s_1$ in $S$ and $a \in \Sigma$ such that:
            $row(s_1 \cdot a)$ is different for all $s \in S$
            add $s_1 \cdot a$ to $S$
            extend $T$ to $(S \cup S \cdot \Sigma) \cdot E$ using $MQs$
        **end if**
    **end while** ▷ $(S, E, T)$ is closed and consistent
    $H = H(S, E, T)$ ▷ $H$ is the conjecture
    **if** $EQ(H)$ is a counter example $t$ **then**
        add $t$ and all its prefixes to $S$
        extend $T$ to $(S \cup S \cdot A) \cdot E$ using $MQ$
    **end if**
**end while**
**return** $H$

## 2.3 Counterexample Decomposition

In the previous section, the learner was tasked to infer the behavior of set $\mathcal{A}$ illustrated in Figure 2.1. At some point the learner inferred an incorrect conjecture $H_0$, depicted in Figure 2.3. The key step to improve $H_0$ towards $\mathcal{A}$ was to utilize a given counterexample word $w = ab$. This section discusses how the counterexample can be decomposed by the learner and elaborates on the process of how this decomposition leads towards the appearance of a new state.

Suppose at some point the learner poses an equivalence query to the teacher for an incorrect conjecture $H$ and receives a counterexample word $w$. Word $w$ is composed of a prefix and a suffix part. The suffix part can be written as $av$, where $v$ is the distinguishing character. Since $H$ is incorrect, there exist two access sequences $u, u'$

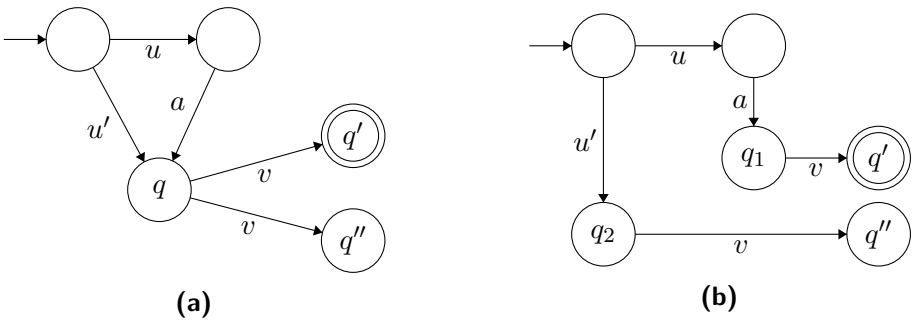



**Fig. 2.4:** Formal progression of an incorrect conjecture: (a) inconsistent model for distinguishing suffix $v$ from state $q$, (b) consistent model after splitting $q$ into new states $q_1$ and $q_2$.

from the set of prefixes, such that $ua$ and $u'$ reach the same state in $H$. The latter is true because input $uav$ (the given counterexample) yields a different result than input $u'v$ (hypothesis). Because $w$ is a counterexample it holds that $\lambda^H(w) \neq \lambda^{\mathcal{A}}(w)$, or in other words $\lambda^H(ua \cdot v) \neq \lambda^{\mathcal{A}}(u' \cdot v)$. This is also visually depicted in Figure 2.4a where input word $uav$ leads to state $q'$ and input word $u'v$ leads to $q''$. One could digest this even further by concluding that the state with access sequence $u$ trailed with letter $a$ is a different state in comparison to the state with access sequence $ua$. In other words $\lambda([u] \cdot a \cdot v) \neq \lambda([ua] \cdot v)$.

Since a word $w = \langle prefix, suffix \rangle$ and the suffix is modeled as $av$, a counterexample word $w$ can also be decomposed in a three-tuple $w = \langle u, a, v \rangle$. Element $v$ is called a distinguishing suffix, because it distinguishes the states $U[ua]$ and $U[u']$ from each other. Angluin's L* algorithm adds $v$ to the set $E$, such that in the observation table the rows $ua$ and $u'$ differ from each other. This results in one more distinct row in the observation table, hence that state $q$ is split into two states: one state $q_1$ that leads $uav$ to $q'$ and one state $q_2$ that leads $u'v$ to $q''$. The latter is depicted in Figure 2.4b.

**Example 2.3.1.** In the running example of the former section the counterexample word $w = ab$ led to the appearance of a new state in the conjecture. Word $w$ is a valid counterexample because $\lambda_H(ab) \neq \lambda_{\mathcal{A}}(ab)$. According to the decomposition from above, this must be true because $\lambda_H(ua \cdot v) \neq \lambda_H(u' \cdot v)$ with the counterexample decomposition $w = \langle u, a, v \rangle = \langle \varepsilon, a, b \rangle$. In Figure 2.3 $H[ua \cdot v] = H[\varepsilon \cdot a \cdot b] = H[ab]$ is equal to the state $H[u'v] = H[\varepsilon \cdot b] = H[b]$, whilst $\lambda_{\mathcal{A}}(ab) \neq \lambda_{\mathcal{A}}(b)$. The state $H[\varepsilon \cdot a] = H[a]$ should thus be added. This is achieved by splitting $H[\varepsilon]$, because both $H[\varepsilon]$ and $H[a]$ have the same output for $\lambda$. This results into two new states: $H[\varepsilon]$ and $H[a]$.

## 2.4 The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning

In *The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning* [15] Isberner et al. recognize that one of the computational expensive disadvantages of Angluin's L* algorithm is a consequence of redundant entries in the observation table. The redundant entries are a result of suboptimal and possibly excessively

long counterexamples provided by the MAT to the learner's equivalence queries. The prefix-part and the suffix-part of the counterexample contain elements that do not contribute to the discovery of new states even the distinguishing suffix possibly include redundant symbols. The L* algorithm can only establish a conjecture when the observation table is closed and consistent. As a result, the learner also poses membership queries to fill out each new entry field that is created by the product of the counterexample's redundant prefix and suffix elements. To illustrate the presence of the redundant fields: the observation table of the employed example DFA shown in Table 2.2c contains the value $0$ in the $\varepsilon$-column, which suffices to distinguish the $\mathcal{A}[b]$-state from the other two states. The remaining fields for these particular rows denoted by the $b$-suffix do not contribute to the distinctiveness of these rows, hence the suffix to distinguish the state $\mathcal{A}[b]$ from $\mathcal{A}[\varepsilon]$ and $\mathcal{A}[a]$ are redundant.

To overcome performance impacts caused by redundancies in counterexamples, Isberner et al. propose the TTT algorithm. The TTT algorithm does so by utilization of a redundancy-free organization of observations in a discrimination tree. A discrimination tree adopts two sets $S, E \subset \Sigma^*$ that are non-empty and finite like Angluin's L* algorithm does, the only difference being that $S$ consists of state access strings opposed to prefixes. Set $E$ still contains distinguishing suffixes, which are referred to by Isberner as discriminators. The tree can then be modeled by a rooted binary tree where leafs are labeled with access strings in $S$ and inner nodes are labeled with suffixes in $E$. The two children of an inner node correspond to labels $\ell \in \{\top, \bot\}$ that represent the $\lambda$-evaluation. Other studies apply different terminology, like $\ell \in \{+, -\}$, $\ell \in \{1, 0\}$ or adopt consistency in the direction their children have: a child to the left coincides with $\ell = \top$ and a child to the right coincides with $\ell = \bot$ [15, 16].

The discrimination tree is shaped in a way such that words can be *sifted* through the tree. Sifting is required to determine the transitions when determining the transitions of a conjecture. The process of sifting is administered to a word $w \in \Sigma^*$ that starts at the root of the tree. For every internal node $v \in E$ of the discrimination tree the sifting process passes one branch to the $\top$- or $\bot$- child depending on the value $\lambda(w \cdot v)$ until the process reaches a leaf node. The leaf node then represents the resulting state with the corresponding access sequence for word $w$. The relation of the leaf node to its direct parent, either $\top$ or $\bot$, depict whether $w$ is accepted or not. Each pair of states have then precisely one distinguishing suffix, which is the lowest common ancestor of the two leafs.

The discrimination tree $DT$ corresponding to $\mathcal{A}$ is depicted in Figure 2.5. How the TTT algorithm gradually builds this tree will be discussed at the end of this section. Note that the state names $q_0, q_1$ and $q_2$ are replaced by their access sequence notation: $[\varepsilon], [b]$ and $[a]$ respectively.

**Example 2.4.1.** Sifting is used to establish the transitions of a conjecture from a discrimination tree. If for example the $b$-transition from state $\mathcal{A}[a]$ should be determined i.e. $\delta(\mathcal{A}[a], b)$, one needs the access sequence for state $\mathcal{A}[a]$ which is $a$, and the one-letter extension which in this case is $b$. Together they form the word $w = ab$ and this is sifted through $DT$. Starting at the root of the tree, one must evaluate $\lambda(ab \cdot \varepsilon)$ which results to 1. Thus one follows the $\top$-branch and finds the inner-node $b$. Then $\lambda(ab \cdot b)$ is evaluated, which also results to 1. Again following

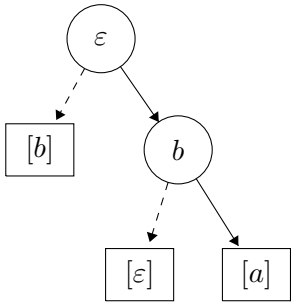



**Fig. 2.5:** Discrimination Tree $DT$ corresponding to DFA $\mathcal{A}$.

the $\top$-branch of the tree, results in the leaf $[a]$, at which point the process shows that $\delta(\mathcal{A}[a], b) = \mathcal{A}[a]$.

The TTT algorithm follows the following steps to infer a correct model:

1. Hypothesis Construction
2. Hypothesis Refinement
3. Hypothesis Stabilization
4. Discriminator Finalization

**1. Hypothesis Construction**
The initial discrimination tree is constructed by evaluating $\lambda(\varepsilon)$. The initial evaluation results in either a tree with as root node $\varepsilon$ and a single leaf with access string $[\varepsilon]$ on either the $\top$- or $\bot$-child of the root node depending on the $\lambda$-evaluation. A conjecture DFA can be created, such that:

1. $Q$ are all the leaf nodes in the discrimination tree,
2. $\Sigma$ is already known,
3. $\delta$ is determined by sifting all words $w \in Q \times \Sigma$,
4. $q_0$ is $[\varepsilon]$ and
5. $F$ are all leaf nodes that are a $\top$-child in the tree.

**Example 2.4.2.** Following the example where DFA $\mathcal{A}$ from Figure 2.1 should be learned again, but now according to the TTT algorithm, the algorithm starts with evaluating $\lambda(q_0, \varepsilon)$. The evaluation results in 1, thus an initial discrimination tree is established where the $\top$-child points to the initial state since it is an accepting state. The initial discrimination tree $DT_0$ is depicted in Figure 2.6a. In order to create an initial conjecture DFA, one takes all leafs from $DT_0$, in this case only $H[\varepsilon]$ and determines transitions by sifting the access sequence with all one-letter extensions. Thus $\varepsilon a$ and $\varepsilon b$ are sifted in order to respectively determine the $a$ and $b$ transitions from the initial state $H[\varepsilon]$. Furthermore, a state $q \in Q^H$ is in $F^H$ if the associated leaf is on the $\top$-side of the $\varepsilon$-node. The initial state is the only state in the hypothesis, hence $S = \{\varepsilon\}$, and since $q_0$ is an accepting state, the $q_0$-leaf is the $\top$-child of the $\varepsilon$-root. The initial conjecture DFA $H_0$ corresponding to $DT_0$ is depicted in Figure 2.6b.

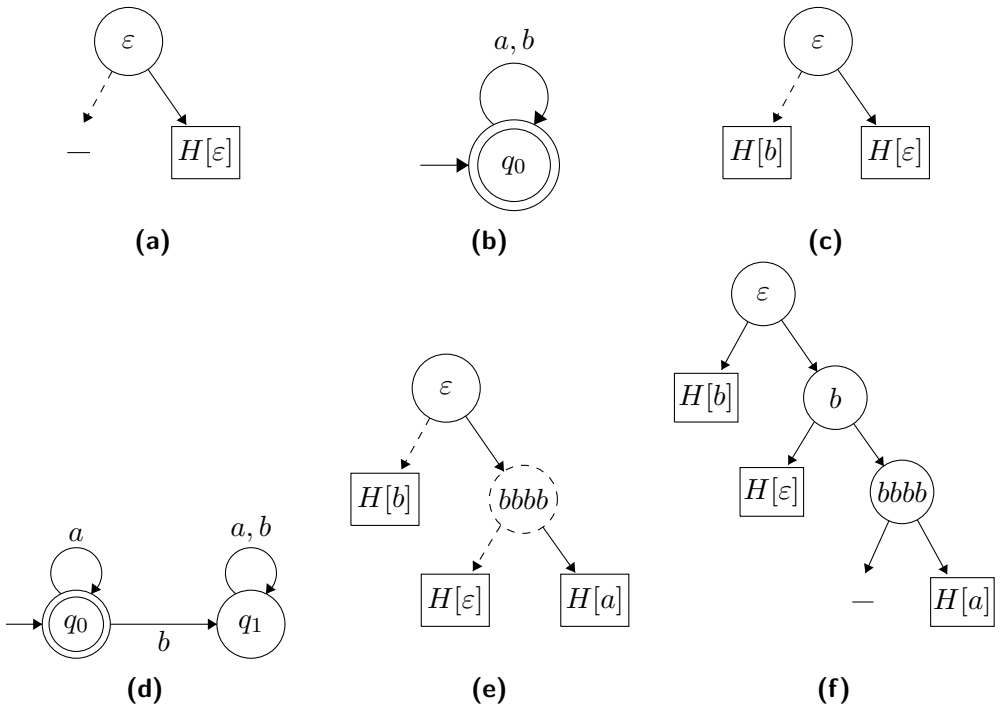



**Fig. 2.6:** Evolution of discrimination trees and conjectures towards learning the DFA $\mathcal{A}$ with the TTT algorithm: (a): initial discrimination tree $DT_0$, (b) conjecture DFA $H_0$ corresponding to $DT_0$, (c) discrimination tree $DT_1$ after processing counterexample $w = b$, (d) the conjecture DFA $H_1$ corresponding to $DT_1$, (e) discrimination tree a temporary node, (f) discrimination tree where the temporary node is pushed down.

**2. Hypothesis Refinement**
The initial hypothesis is likely to be inequivalent to the SUT, in which case the teacher will return a counterexample. As discussed in section 2.3, a counterexample can be decomposed as $\langle u, a, v\rangle$. The $a$-part is called a breaking point because the conjecture predicts ambiguous results for the $v$-part after $a$. Th breaking point is added as a node in the tree, depending on the evaluation of $\lambda(ua)$ on the $\top$- or $\bot$ branch.

**Example 2.4.3.** Suppose that conjecture $H_0$ was submitted as an equivalence query to the teacher and the learner receives a counterexample $w = b$. Word $w$ is a valid counterexample since $\lambda^{\mathcal{A}}(b) = 0 \neq \lambda^{H}(b) = 1$. As discussed in section 2.3, a counterexample can be decomposed as $\langle u, a, v\rangle$. Word $w$ can thus also be read as $w = \varepsilon b \varepsilon$, thus the complete decomposition of the counterexample is: $u = \varepsilon, a = b$ and $v = \varepsilon$. Since state $H[ua] = H[\varepsilon b] = H[b] = q_0^{H_0}$ should be split to a new state $[u]_H a \to [\varepsilon]_H b$, a new state $q_1$ is introduced that can be reached by the transition $\delta(q_0, b)$. The outbound transitions from the new state are determined by sifting the access sequence of the new state with $\Sigma$. These adjustments are depicted in $DT_1$ and $H_1$ (Figure 2.6c and 2.6d).

**3. Hypothesis Stabilization**
Although the previous step of hypothesis refinement constructed a discrimination tree and a DFA conjecture that are consistent with each other, consistency does not necessarily need to be the case. There is a possibility that for a word that can be formed by concatenating an element from the set of prefixes $S$ and the set of suffixes $E$. A $\top$-child predicts the output $1$ but the hypothesized conjecture $0$ or a $\bot$-child predicts the output $0$, but the hypothesized conjecture predicts the output $1$. Word $w$ then forms a new counterexample and is dealt with likewise the former counterexample. This step is done until the hypothesis is stable and the discrimination tree and the conjecture are consistent.

**4. Hypothesis Finalization**
Often the counterexample retrieved is not minimal, such that the counterexample, for example, contains redundant information. The discriminator is added as a new node in the discrimination tree, but the algorithm adds non-elementary (more than one element in the alphabet) discriminators as a temporary node. To prevent redundancy, trees that contain temporary nodes must be refined until all discriminators consist of a single element of the alphabet. The subtree generated with the temporary node as root must be split by subsequential replacement of the temporary discriminator closest to the subtree's root by its final discriminator. The final discriminators are obtained by prepending a symbol from the alphabet to an existing final discriminator. Prepending a symbol to an existing final discriminator results in the addition of a parent node in the discrimination tree above the temporary node, thus shifting the temporary node a level down the tree. The temporary node can now be assessed for redundant behavior and if the node does not provide distinguishable behavior, it can be removed from the tree. This scenario is also illustrated in example 2.4.4.

**Example 2.4.4.** Suppose $H_1$ was subjected to an equivalence query, which resulted in the counterexample $abbbb$. Because $\lambda^{\mathcal{A}}(H[u]a \cdot v) \neq \lambda^{\mathcal{A}}(H[ua] \cdot v)$ only holds for $a = a$, the corresponding $\langle u, a, v\rangle$-decomposition of this counterexample thus is $\langle \varepsilon, a, bbbb\rangle$. Splitting $H[a] = q_0^{H_1}$ into a new state $H[a]$ a new state $q_2$ is introduced

that can be reached by the transition $\delta(q_0, a)$. The discriminator $bbbb$ is added as a temporary node, indicated by the dashed surroundings depicted in Figure 2.6e. The algorithm finds $b$ to be a final discriminator, as for all elements in $\Sigma$ this yields the same behavior, hence $b$ is added as a node above the temporary discriminator, shown in Figure 2.6f. Finally, since the temporary discriminator does not provide distinguishable behavior compared to the new final discriminator, the temporary discriminator is removed from the discrimination tree, which results in $DT$ Figure 2.5. Following the steps as before to construct a DFA from the discrimination tree results in the example DFA $\mathcal{A}$.

### 2.4.1 Speedup of TTT opposed to L*

Because the TTT algorithm can omit redundant entries during active learning, the algorithm implies a polynomial speedup opposed to the L* algorithm. The remainder of this section will analyze the theoretical speedup of both algorithms, and quantify the difference in complexity between the algorithms.

Three variables matter when computing the time complexity of both learning algorithms. The variables are the size of the state machine of the actual implementation, which is unknown in advance, and the maximum length of the counterexamples provided by the teacher. Furthermore, the size of the input alphabet also determines the number of queries. However, the input alphabet may be assumed to be fixed. Let the size of the state machine of the target system be $n$, the maximum length of all counterexamples be $m$ and the cardinality of $\Sigma$ be $k$. For both L* and TTT it is essential to realize that in the worst case $n - 1$ counterexamples are necessary to learn a state machine because each counterexample introduces a new state in the conjecture and the conjecture can have at most $n$ states.

#### Complexity of L*

Angluin argues that the L* algorithm is a polynomial algorithm, which means that the algorithm's time complexity can be defined by a polynomial function based on the learning input variables: $m$ and $n$ when $k$ is treated as a constant. The function can be determined by analysis of the implementation of L*. Recall that the observation table $(S, E, T)$ has two dimensions: the prefixes $S$ and the suffixes $E$.

At the start of the algorithm, the sets $S$ and $E$ only contain the empty word $\varepsilon$ and the initial goal is to make $(S, E, T)$ closed and consistent. When the observation table is not closed a single element is added to the set $S$ and when the observation table is not consistent a single element is added to the set $E$. Once the observation table is closed, consistent and inequivalent to the target model, a counterexample word $w$ with at most size $m$ is treated. Because the counterexample and all new prefixes are added to the observation table, at most $m$ new entries are added to $S$ for each counterexample.

The conjecture will contain at most $n$ states because the system under test has a maximum number of $n$ states and the L* algorithm guarantees a minimal equivalent

state machine. The maximum length of a suffix in $E$ is initially $0$ and can be increased by $1$ for every suffix that is added to $E$. This only happens $n-1$ times as the size of $E$ is at most $n-1$, hence the maximum length of a suffix in $E$ is also $n-1$.

There are two cases when an element is added to set $S$, either when the table is inconsistent or when a counterexample is received. The observation table is discovered to be not closed at most $n-1$ times. The learner can also receive only $n-1$ counterexamples, which have a maximum upper bound of $m$. Since all prefixes of the counterexample can be added to the observation table, a maximum of $n-1$ times $m$ elements can be added to $S$. The maximum length of any element in $S$ is increased by at most $1$ for every string that is added. Thus the maximum length of an element in $S$ is at most $m+n-1$.

In terms of $\Sigma$, $S$ and $E$, the set of all entries in the observation table can be written as $(S \cup S \cdot \Sigma)E$. If one would substitute the sets with their corresponding maximum cardinality, the maximum complexity in terms of $n$ and $m$ is:

$$(k+1)(n+m(n-1))n = \mathcal{O}(mn^2)$$

Analogous substitution of the sets by the maximum length shows an upper bound of:

$$m+2n-1 = \mathcal{O}(m+n)$$

The upper bound polynomial for the observation table in terms of $m$ and $n$ is the product of the cardinality and the maximum length of an element in the table:

$$mn^2 \cdot (m+n) = \mathcal{O}(mn^3 + m^2n^2)$$

The L* algorithm stops learning if the observation table if closed, consistent and the teacher does not provide a counterexample, which happens within the above established period. The complexity of the L* algorithm is thus $\mathcal{O}(mn^3 + m^2n^2)$.

### Complexity of TTT

The TTT algorithm requires in the worst case scenario $n-1$ equivalence queries and the most pessimistic discrimination tree is entirely unbalanced and thus of height $n$. Analogous to the L* algorithm, a discrimination tree consists of prefixes and suffixes. Ideally, suffixes are distinguishing suffixes, but before a suffix is elementary, they might contain redundant elements, due to redundant elements in the counterexamples.

It is clear to see that the discrimination set $E$ comprises each node in the discrimination tree. Hence $E$'s cardinality is $n$.

Set $S$ depicts all transitions in the discrimination tree. Since the eventual and correct conjecture contains $n$ states, at least $kn$ transitions are required to model the deterministic behavior. Because of the redundant free storage in the discrimination tree, the transitions between discriminator nodes are also included in the $kn$ upper bound. The cardinality of $S$ can thus be noted as $kn$.

Because the most degenerate discrimination tree can be of height $n$, sifting the longest counterexample of size $m$ thus consumes at most $n + m$ queries. The value $n + m$ thus forms an upper bound for the size of a term in $S$.

In terms of $k$, $S$ and $E$, the tree's cardinality can be computed in $S \times E$, hence the maximum complexity in terms of $n$ and $m$ is:

$$kn \cdot n = \mathcal{O}(n^2)$$

The maximum length of a query has shown to be of $\mathcal{O}(m + n)$. The upper bound polynomial for the observation table in terms of $m$ and $n$ is the product of the cardinality and the maximum length of an element in the table:

$$n^2 \cdot (m + n) = \mathcal{O}(n^3 + mn^2)$$

#### Discussion

At this point it has been established that the polynomial complexity of L* is $\mathcal{O}(mn^3 + m^2n^2)$ and the polynomial complexity of TTT is $\mathcal{O}(n^3 + mn^2)$. The TTT algorithm is thus a factor $m$ less complex opposed to the L* algorithm. The $m$-factor reduction is in practice an under-estimation because the worst case scenario for the TTT algorithm is less likely to occur opposed to the worst case scenario for the L* algorithm. The situation depicted for TTT's worst-case scenario is improbable because the tree will never be completely unbalanced in practice. As a consequence, the height of the tree will not be of size $n$. The unlikeliness of entirely unbalanced trees is also shown in the experimental results provided by Isberner et al. (Figure 5 in [15]). In conclusion, the TTT algorithm is superior over L* concerning time complexity.

## 2.5 Equivalence Testing

Once a learning algorithm converges to a stable hypothesis, a counterexample is needed to ensure further progress. Counterexamples are the result of an equivalence queries that test the equality between a hypothesized model and the actual SUT. Equivalence queries in particular return a positive result, indicating that both models are equal or provide a symmetric difference between the hypothesis and the unknown model of the SUT. This conclusion is drawn after running a series of exhaustive or trivial test cases. This section discusses the two most popular DFA equivalence testing methods: the *RandomWalk* and the *W-method*. The W-method is an improvement over RandomWalk regarding the determination of the equivalence between two DFA's,

but the method has a gradual drawback in performance due to the large number of test sequences it generates. Hence, a recently developed method for finding separating sequences for all pairs of states will also be discussed as a conformance testing method.

### 2.5.1 RandomWalk

To test whether two DFAs $\mathcal{A}$ and $H$ are equivalent, one could perform a series of random 'walks' over $H$ and compare if the SUT yields the same output. A walk is an arbitrary input sequence that either concludes in an accepting state or a rejecting state. If for enough sequences both $\mathcal{A}$ and $H$ return the same output, the two DFAs are equivalent. If one test case fails, then the sequence functions as a counterexample and refinement of $H$ starts until the RandomWalk oracle is used again. The RandomWalk algorithm is depicted in Algorithm 2.

**Algorithm 2** The RandomWalk Algorithm

**Input:** $H$ the hypothesis DFA, $\Sigma$ the input alphabet, $maxSteps$ the maximum number of steps to be performed
**Output:** A word $walk$ which is a counter example or `null` if no counter example can be found

```
1:  step ← 0
2:  current ← H's initial state
3:  while steps < maxSteps do
4:      walk ← ε
5:      if random then
6:          increment step by 1
7:          w ← random element in Σ
8:          append w to walk
9:          if δ^H(current, w) is not equal to δ^SUT(current, w) then
10:             return walk
11:         end if
12:         current ← δ^H(current, w)                ▷ traverse 1 step
13:     else                                         ▷ restart the walk
14:         walk ← ε
15:         current ← H's initial state
16:     end if
17: end while
18: return null
```

### 2.5.2 W-method

The W-method was first proposed by Chow in *Testing Software Design Modeled by Finite-State Machines* [10] as a method of testing the correctness of control structures that can be modeled by a finite state machine. The method embodies a test suite development strategy based on a *transition cover set* $P$ of inputs and a *characterization set* $W$ of input sequences that can distinguish every pair of states in a model. A transition cover set is a set $P$ of input sequences such that for each state $q \in Q$ and each input $a \in \Sigma$ there exists an input sequence $w \in P$ starting from the

initial state $q_0$ and ending with the transition that applies $a$ to state $q$. Moreover, the transition cover set also includes the empty word $\varepsilon$, since that transition leads to the initial state. Set $P$ is thus composed of all short prefixes for state identification in case all observations are stored in an observation table or access sequences of all states any other case.

Chow constructs the transition cover set $P$ with the aid of a *testing tree* of the DFA $H$. A testing tree of $H$ depicts $H$'s control flow in a linear and non-cyclic manner. The tree is generated by induction as follows:

1. The root of the testing tree is the initial state of $H$.
2. Suppose the tree is built to a level $k$. Level $(k+1)$ is built by examining all nodes on the $k$'th level from left to right as follows. A node is terminated, meaning that its branch will halt at that node, if the node appeared at a lower level $j$ for $j \leq k$. The labels correspond to the transition symbol between the states.

The above process always terminates, as the DFA only has a limited number of states. Because a branch ends if a node occurred on a lower level, each level contributes to at least one terminating branch. Hence if a DFA has $n$ nodes, a testing tree of maximum $n+1$ levels contain all paths to reach all nodes. One level more than the number of states is needed because the root node at level $1$ does excludes a terminating branch.

If the testing tree is constructed as described above, the transition cover set $P$ can be constructed by obtaining the input sequence of all branches including all partial paths. A partial path of a testing tree is a sequence of consecutive branches, that start from the root of the tree and ends in a terminal or nonterminal node [17].

The characterization set $W$ consists of input sequences that distinguish the behavior between every pair of states in a minimal automaton. In other words, for every two distinct states $q, q' \in Q$, $W$ contains at least one input sequence that produces different outputs when applied from $q$ and $q'$ respectively. Gill et al. [18] describes definitions and methods for constructing such sets.

**Example 2.5.1.** A testing tree for conjecture DFA $H_1$ (Fig. 2.6d) is constructed as follows:

1. The root of the tree is the initial state of $H_1$: $q_0$.
2. The next branch is created by determining the resulting state for each input symbol: $a, b$. Since on input $a$ from $q_0$ one goes to $q_0$, e.g. $\delta(q_0, a) = q_0$, one node labeled $q_0$ is connected to the root node. The $b$-successor of $q_0$ is $q_1$, so a new node labeled $q_1$ is added and connected to the root node. Because this label differs from the layer above, this branch continues to grow. Because both the $a$- and $b$-successor of $q_1$ result in the $q_1$ state, two children are added, both labeled with $q_1$.

The testing tree that corresponds to conjecture $H_1$ from Figure 2.6d is depicted in Figure 2.7. From this figure one can derive that there are $3$ branches, as there are $3$

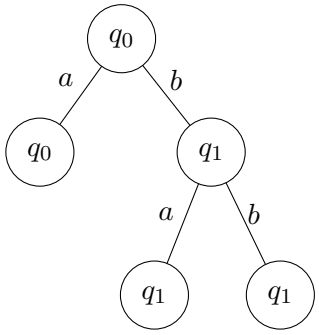



**Fig. 2.7:** The testing tree conform conjecture DFA $H_1$.

leafs. The set of all paths and all partial paths are $\{a, b, ba, bb\}$. The transition cover set can thus be defined as $P = \{\varepsilon, a, b, ba, bb\}$.

**Example 2.5.2.** In the example of conjecture $H_1$, finding a characterization set is trivial, as the only pair of states is the pair $q_0$ and $q_1$ and their output differs for input symbol $a$ as $\delta(q_0, a) = q_0 \in F$ and $\delta(q_1, a) = q_1 \notin F$, thus the output for $a$ differs in both states and hence the characterization set $W = \{a\}$.

One drawback of the W-method is that it requires knowledge about the maximum number of states $m$ that a correct version conjecture might have. Chow solves this variable as to be determined by human judgment. Since $m$ functions as an upper-bound estimation, $m$ can be any number as long as it is larger or equal to the number of states in the hypothesized conjecture $n$. The test cases are then derived by the concatenation of $P$ with $(\bigcup_{i=0}^{m-n} \Sigma^i \cdot W)$. Chow defines $(\bigcup_{i=0}^{m-n} \Sigma^i \cdot W)$ to be the set $Z$ and because $\Sigma^0 = \{\varepsilon\}$, $Z$ can be written as $W \cup \Sigma \cdot W \cup \cdots \cup \Sigma^{m-n} \cdot W$.

**Example 2.5.3.** To exemplify the test suite $P \cdot Z$ that would be generated by the W-method for the hypothesized conjecture $H_1$, one should recall that $P = \{\varepsilon, a, b, ba, bb\}$, $W = \{a\}$ and $\Sigma = \{a, b\}$. Because $Q = \{q_0, q_1\}$ the size of $Q$: $|Q| = n = 2$. Let $m$ then be an arbitrary estimation that satisfies $m \leq n$, one could choose $m = 4$. Set $Z$ can then be determined by $Z = \bigcup_{i=0}^{m-n} \Sigma^i \cdot W = \bigcup_{i=0}^{2} \{a, b\}^i \cdot \{a\} = \{a\} \cup \{a, b\}^1 \cdot \{a\} \cup \{a, b\}^2 \cdot \{a\} = \{a\} \cup \{a, b\} \cdot \{a\} \cup \{aa, ab, ba, bb\} \cdot \{a\} = \{a\} \cup \{aa, ba\} \cup \{aaa, aba, baa, bba\} = \{a, aa, ba, aaa, aba, baa, bba\}$.

The final test suite is created by concatenating the transition coverage set with set $Z$:

$$P \cdot Z = \{\varepsilon, a, b, ba, bb\} \cdot \{a, aa, ba, aaa, aba, baa, bba\} =$$
$$\{a, aa, ba, aaa, aba, baa, bba, aaaa, aaba, abaa, abba, baaa, baba,$$
$$bbaa, bbba, baaaa, baaba, babaa, babba, bbaaa, bbaba, bbbaa, bbbba\}$$

The cardinality of the test suite that is created by $P \cdot Z$ is 23 and all 28 test cases are executed on both the SUT and the hypothesized model. If the generated test suite is executed in the order that is depicted above, the second input sequence $aa$ returns a different result as $\lambda^{H_1}(aa) = 1 \neq \lambda^{\mathcal{A}}(aa) = 0$. The input sequence $aa$ is then provided as a counterexample to the learner by the equivalence oracle. In example 2.2.2 the learner received the counterexample word $w = ab$ which also shows the

different behavior as the test sequence $aa$ that is identified as a counterexample with the W-method does.

### 2.5.3 Minimal Separating Sequences for All Pairs of states

One drawback of the W-method is that the number of test cases rapidly grows in the size of the alphabet, number of states, the maximum depth and especially the length of the characterizing set. Instead of each combination in the permutation of the alphabet up until a certain depth, one can utilize smarter techniques for finding separating sequences. In *Minimal Separating Sequences for All Pairs of States*[19] Smetsers et al. propose an improved modification based on Hopcroft's framework [20] for finding the shortest input sequence that distinguishes two inequivalent states in a DFA. Minimal separating sequences play a central role in conformance testing methods and hence can be applied to establish an equivalence oracle for learning automata. Separating sequences function as an input for test suite generation, which like Chow's W-method can determine whether a hypothesized conjecture is equivalent to an abstraction of a system under test.

Smetsers et al. identify minimal separating sequences by systematically refining state partitions to ensure a minimal DFA minimal access sequence. The operational refinement information is maintained in a tree-like data structure called a splitting tree, which was first introduced by Lee and Yannakakis[21]. The remainder of this section elaborates on how Smetsers et al. utilize partitions and splitting trees to determine the minimal separating sequences.

Let the SUT's behavior be abstracted to the DFA $\mathcal{A} = \{Q, \Sigma, \delta, q_0, F\}$ and let $H$ be the hypothesized model of $\mathcal{A}$. A state *partition* $P$ of $Q$ is a set of pairwise disjoint non-empty subsets of $Q$ whose union is exactly $Q$. Each subset of $P$ is called a block. If $P$ and $P'$ are partitions of $Q$ and every block of $P'$ is contained in a block of $P$, then $P'$ is called a refinement of $P$. The algorithm starts with the trivial partition $P = \{Q\}$ and refines $P$ until the partition is valid, that is when all equivalent states are contained in the same block. Let $B$ be a block in $P$ and $a$ be an input. The partition refinement algorithm splits blocks because of two reasons. $B$ can be split concerning the output after $a$ if the set $\lambda(B, a)$ contains more than one output. In this instance each distinct output in $\lambda(B, a)$ defines a new block. Alternatively, $B$ can be split concerning the succession state after input $a$. In the latter instance, each block that contains a state of $\delta(B, a)$ defines a new block. The refinement process is continued until for all pairs of states $q, q' \in Q$ that are contained in the same block and for all inputs $a \in \Sigma$ hold that $\lambda(q, a) = \lambda(q', a)$. At this point, the partition is classified as *acceptable*. Final refinement is reached when a partition is acceptable and for all $a \in \Sigma$ hold that for all $q, q' \in Q$ in the same block, the new states $\delta(q, a)$ and $\delta(q', a)$ are also in the same block. At this final point, the partition is classified as *stable*.

Separating sequences are determined by the type of split and the information is maintained in a splitting tree. Smetsers et al. redefine the splitting tree to apply to situation where it stores minimal separating sequences, as follows:

**Definition 2** (*Splitting Tree*)**.** A splitting tree for $\mathcal{A}$ is a rooted tree $T$ with a finite set of nodes with the following properties:

- Each node in $T$ is labelled by a subset of $Q$, denoted $l(u)$
- Each leaf nodes $u$, $l(u)$ corresponds to a block in the stable partition $P$.
- Each non-leaf node $u$, $l(u)$ is partitioned by the labels of its children, thus the root node is labeled $Q$.
- Each non-leaf node $u$ is associated with a sequence $\sigma(u)$ that separates states contained in different children of $u$.

**Example 2.5.4.** Figure 2.8b shows an example splitting tree that satisfies these properties.

- The nodes $\{q_0, q_1, q_2\}$ and $\{q_0, q_1\}$ are subsets of $Q$.
- The root node is labeled with $Q$ as $Q = \{q_0, q_1, q_2\}$ is the label of the root node.
- The non-leaf children of the root node are labeled as a partition of $Q$: all elements in the non-leaf children of the root together form $Q$.

$C(u)$ denotes the set of children of a node $u$ in $T$ and the lowest common ancestor for a set $Q' \subseteq Q$ is a node $u$ denoted by $lca(Q')$. For a pair of states, the shorthand notation $lca(s, t)$ is used instead of $lca(\{s, t\})$ to denote the lowest common ancestor of $s$ and $t$. At any given time, the labels of the leafs of $T$, denoted as $P(T)$ together form a partition of $Q$. A tree $T$ is valid, if $P(T)$ is valid as well. A leaf $u$ within block $B = l(u)$ can be split in the same way partition blocks are split, either based on output or the consecutive state for an input $a$. If the block is split based on output, $\sigma(u)$ is set to $a$ and a new node for each subset of $B$ that produces the same output for $a$ are appended as children for $u$. If the block is split based on the consecutive state, then the node $v = lca(\delta(B, a))$ has at least two children whose labels contain elements of $\delta(B, a)$. This information is utilized to create a new child of $u$ labelled $\{s \in B | \delta(s, a) \in l(w)\}$ for each node $w$ in $C(v)$. The state separator $\sigma(u)$ is set to $a \cdot \sigma(v)$.

To create a stable splitting tree for the example DFA $\mathcal{A}$ shown in Figure 2.1 one should note that the partition refinement algorithm only works for Mealy Machines. The reason for this is because splitting concerning output only works if every transition produces an output, which is not the case for a general DFA. To map the algorithm to the running example, $\mathcal{A}$ can be regarded as a mealy machine where transitions output $1$ if the resulting state is an accepting state and $0$ if the resulting state is a rejecting state. The conversion results in a Mealy Machine version of $\mathcal{A}$ depicted in Figure 2.8a.

**Example 2.5.5.** Figure 2.8b shows the splitting tree for mealy machine $\mathcal{A}'$, it is generated by Smetsers al.'s algorithm as follows. The first step is setting the root node to $Q$, hence the root node is labeled $\{q_0, q_1, q_2\}$. Since $q_2$ gives another output for input $b$ opposed to the output of states $q_0$ and $q_1$, the root node is split based on this output after $b$. The node labeled $\{q_2\}$ cannot be split anymore, as it contains one single element, the algorithm determines that states $q_0$ and $q_1$ yield a different output for input $a$. Since the nodes cannot be split anymore, the tree is complete.

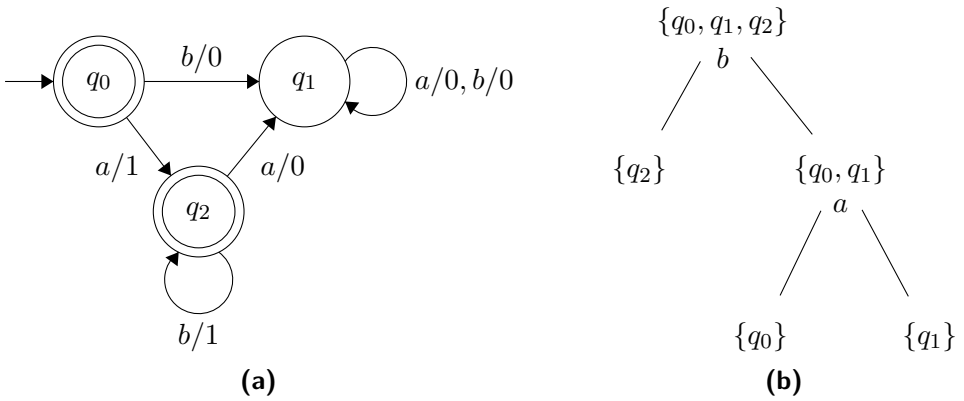



**Fig. 2.8:** (a): $\mathcal{A}'$ the mealy machine representation of $\mathcal{A}$ (b): splitting tree representation of $\mathcal{A}'$

The algorithm for generating splitting trees can be used to obtain separating sequences, but they are not necessarily minimal separating sequences. The scenario where the sequence is not minimal occurs when child nodes have a smaller sequence than their parents. The sequences can be shortened, as the parents' can be split first. This does not appear in the elementary example from Figure 2.8b, but it shows in Smetsers et al.'s example DFA and splitting tree (Figures 2.9a and 2.9b). The labels of the child node are arbitrarily larger than the labels of the parent nodes. Splitting trees that are obtained in a way that ensures that each $k$'th level has a label of size $k$, are *layered* splitting trees. Each layer in the tree consists of nodes for which the associated separating sequences have the same length. During the construction of the splitting tree, each layer should be as large as possible, before continuing to the next one. The following two definitions aid the process of obtaining such splitting trees:

**Definition 3** (*k-stable Splitting Trees*)**.** A splitting tree $T$ is $k$-stable if for all states $s$ and $t$ in the same leaf holds that $\lambda(s, x) = \lambda(t, x)$ for all $x \in I^{\leq k}$

**Definition 4** (*Minimal Splitting Trees*)**.** A splitting tree $T$ is minimal if for all states $s$ and $t$ in the same leaf holds that $\lambda(s, x) = \lambda(t, x)$ implies $|x| \geq |\sigma(lca(s, t))|$ for all $x \in I^{\leq k}$

The recipe for establishing a minimal splitting tree is to create a splitting tree splitting blocks only concerning output and next assuring that for $k = 1 \ldots |Q|$ the tree is $k$-stable and minimal.

**Example 2.5.6.** The example of Smetsers yield Figure 2.10 as a result of this process. Basically it starts splitting blocks with respect to output as shown in Figure 2.9b until the node labeled $\{s_0, s_2\}$ appears. This node cannot be split based on output because in DFA figure 2.9a it shows that all outgoing transitions of both states yield the same output. At this point $k = 1$ and one can observe that the sequence of $lca(\{s_0, s_2\}, a))$ has length 2, which is too long for the value of $k$. One thus has to move on to the next input. It is then possible to split this block concerning the state after $b$, thus the associated sequence is $ba$. If this procedure is continued on all levels and for all blocks, the splitting tree and partition are identical to the earlier splitting tree, except that the labels are shorter.

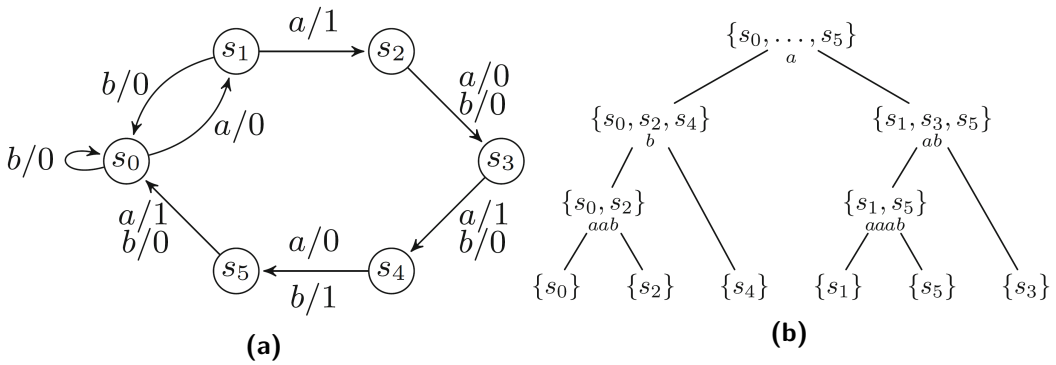



**Fig. 2.9:** (a): Smetsers et al.'s example mealy machine and (b) complete splitting tree for the mealy machine.

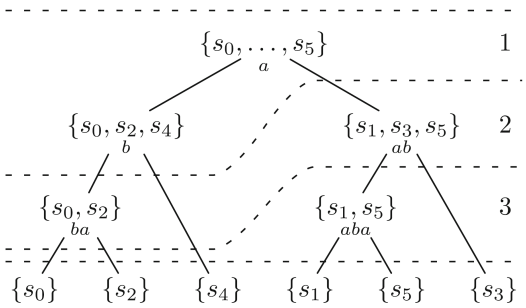



**Fig. 2.10:** Minimal splitting tree for Smetsers mealy machine

In conclusion, following this recipe for the establishment of a $k$-stable and minimal splitting trees, results in the shortest separating sequence.

The complete and minimal splitting trees can henceforth be used to extract relevant information for the characterization set for the W-method.

**Lemma 1.** *Let be a complete splitting tree, then* $\{\sigma(u)|u \in T\}$ *is a characterization set.*

*Proof* Let $W = \{\sigma(u)|u \in T\}$and let $s, t \in Q$ be inequivalent states. A set $W \subset \Sigma^*$ is called a characterization set if for every pair of inequivalent states s,t there is a sequence $w \in W$ such that $\lambda(s, w) \neq \lambda(t, w)$ ([19], definition 17). Because $s, t$ are inequivalent and $T$ is complete, $s$ and $t$ are contained in different leaves of $T$. Hence $u = lca(s, t)$ exists and furthermore $\sigma(u) \in W$. This shows that $W$ is a characterization set. □

# Prior Art on Application Modeling

# 3

> *. . . each larger pattern comes into being as the end product of a long sequence of tiny acts.*
>
> **— Christopher Alexander**

Part of this research is to infer a state machine model on a mobile Android application. Although prior work on model inference of a mobile applications is limited, earlier work performed by Lampe et al. [11] constructed a tool for state machine learning: the *fsm-learner*. Their work serves as a building block for the final model inference system which Chapter 4 discusses. Lampe et al. implemented the MAT framework as proposed by Angluin, by integration of a library for automata learning and experimentation: LearnLib [22]. The tool built by Lampe et al. was designed to infer the model for a single specific banking application: the Dutch bunq bank mobile Android application[1]. The tool was fully tailored to bunq's application, hence the tool must be extended in a way such that other Android applications can be automatically modeled.

This chapter reviews the discussed tool regarding functionality and design choices. Section 3.1 examines the LearnLib library, its capabilities and the reason why the library was incorporated. Section 3.2 discusses how the MAT framework is implemented to achieve active learning. Section 3.3 elaborates on how active learning elements are used to infer a state machine model. Lastly Section 3.4 concludes with a focus on what fundamental components can be utilized in the final design for generic application modeling and what components must be changed.

## 3.1 LearnLib

The library for state machine learning, LearnLib, is popular for numerous reasons. The library supports user-configured learning scenarios, is modularly designed, contains many learning algorithms and as of 2015, the library became open source [23, 24]. LearnLib features active learning algorithms such as the previously discussed L* (Section 2.2) and the TTT algorithm (Section 2.4). The library also provides a number of conformance testing methods, such as an implementation of the W-method and modifications of this method. The developers of LearnLib argue that the direct search of a series of arbitrary sequences that should be true is often the cheapest and fastest way of approximating equivalence queries [24]. Hence the RandomWalk equivalence oracle is implemented as well. A good reason to utilize LearnLib is that the library is actively maintained and is becoming more popular to be implemented

[1] `https://play.google.com/store/apps/details?id=com.bunq.android`

for academic research[2]. As a consequence, the community that can support LearnLib increases.

Apart from LearnLib, there also exist other libraries that aid the process of state machine learning, such as LibAlf [25] and iCRAWLER [26]. Lampe et al. also reviewed these other libraries., but LearnLib was decided to be the most promising library due to the number of algorithms that are featured and the presence of an active community.

## 3.2 The MAT Framework Implementation

LearnLib is not capable of inferring a state machine model on its own. To achieve model inference, LearnLib needs to communicate with a so-called system under test (SUT). Steffen et al. explain that the main obstacle to active learning is the implementation of the idealized form of interrogation regarding membership and equivalence queries [27]. To apply LearnLib to the Android application domain an interplay layer to communicate with the SUT was required. Lampe et al. chose to use Appium [28] as a layer to communicate with the USB tethered mobile device because Appium is actively supported by a supporting community as well and Appium supports both interactions with the Android and the iOS platform.

The MAT framework, as first proposed by Angluin, requires the two entities of a teacher and a learner, where the teacher can answer membership queries and equivalence queries. The Appium mapper resolves membership queries by simulation of the query input on the real USB tethered device. The teacher can answer equivalence queries by the utilization of an equivalence oracle. The fsm-learner tool implements a RandomWalk oracle. This type of oracle tries to find a counterexample directly, which means that the oracle does not construct an entire test set before testing. Contrarily, the oracle executes random steps on both the hypothesized conjecture model and the SUT and tests whether both instances return the same output.

An overview of the way the MAT framework has been implemented in the fsm-learner is depicted in Figure 3.1. The figure shows in what way the learner interacts with different components of the teacher. The SUT is present as a third component, which in this instance is the Android application bunq. The USB connected mobile device that is used to simulate the tests must have the application installed. The teacher is then able to interact with the SUT through the Appium mapper. The component overview also shows how LearnLib is intertwined in the fsm-learner: the LearnLib library provides several sub-entities of the learner and teacher, such as the 'Learning Algorithms', 'Observation Table' and 'RandomWalk Equivalence Oracle'.

To keep state machine learning terminology consistent, this chapter assumes that the L* algorithm is used for active learning. Since the fsm-learner tool enables a multitude of learning algorithms, the actual component overview does not have a fixed data type to store its traces. The component overview shows the type of an observation table that corresponds to the L* algorithm. If for example the

[2]A growing number of citations for LearnLib (Google Scholar): 2003-2007: 39 citations, 2008-2012: 112 citations, 2013-2017: 179 citations

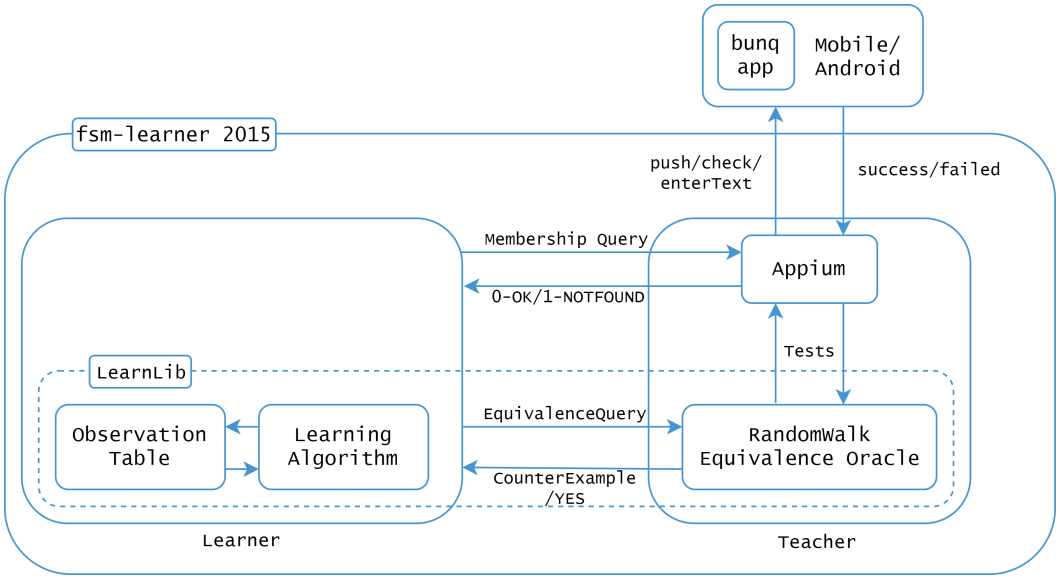



**Fig. 3.1:** A component overview of the fsm-learner's implementation of the MAT framework.

TTT algorithm is used, the observation table in the component overview is to be substituted with a discrimination tree. The following section explains how all the different components are chronologically intertwined.

## 3.3 Learning

This section elaborates on the entire learning process through the MAT implementation. At first one must determine the input alphabet that can be used by the learner. Secondly, all steps that the learner and teacher perform are chronologically explained conform the L* algorithm. Lastly, several techniques have been proposed by Lampe et al. to enhance the time feasibility of learning. These optimization techniques are discussed at the end of this section.

### 3.3.1 Alphabet Establishment

Before the learning process starts, an alphabet must be predefined. The alphabet is conceived by manually traversing the application, where buttons, checkboxes and text fields are gathered from screenshots. The observed elements are mapped to the following user actions:

- **push**, simulates a click on a button or other visual element,
- **check**, toggles the checked-property of a checkbox,
- **enterText**, inserts text in a text area.

The fsm-learner tool transforms all elements from the screen dump to one of the actions mentioned above. The collection of all actions represents the input alphabet. A symbol in the input alphabet, i.e. a single action, is of the following format $\texttt{action\%param}_1\texttt{\#param}_2\texttt{\#...\#param}_n$ for $n$ parameters. The `action` represents one

of the earlier specified user actions: push, check or enterText. The first parameter passed with the action is the corresponding *Xpath* of the element. Xpath is a language that describes the location of elements within a certain page on the screen. For example, `push%button1.Xpath` simulates a push action on the button `button1`. The enterText action requires an additional parameter that specifies the text that should be inserted. The action that enters the text 'active_learning' in text field `field1` looks for example like `enterText%field1.Xpath#active_learning`.

### 3.3.2 Learning Steps

After the input alphabet is established, the learning algorithm component performs the following sequence of steps:

1. The learner initiates an empty observation table.
2. The learner performs membership queries for all actions in the input alphabet to the Appium driver in the Teacher component. The Appium driver simulates the actions on the bunq application on the USB tethered device and returns the observed result to the learner.
3. The learner stores the query and the corresponding result in the observation table.
4. Until the observation table is closed and consistent, the learner continues posing membership queries and thus gradually enlarging the observation table.
5. Once the observation table is closed and consistent, the learner generates a conjecture DFA from the observation table, and poses an equivalence query for the conjecture to the teacher. The teacher then invokes the equivalence oracle as follows:

    a) The equivalence oracle generates a test set, that is a set of input sequences, that is in accordance with the equivalence method.

    b) The test set is run on both the conjecture and simulated on the mobile application.

    c) If one test yields an inequivalent result between the conjecture and the mobile application, the test is returned as it represents a counterexample that invalidates the conjecture.

    d) If all the tests yield an equivalent result between the conjecture and the mobile application, `YES` is returned, which indicates that the conjecture is equal to the mobile application.
6. If the learner receives a counterexample, the test and all of the test's suffixes are added to the observation table, inducing the observation table to be open or inconsistent again. At this point, the process iterates again from step 4 by making the observation table again closed and consistent and a new equivalence query can be posed.
7. If the learner receives `YES` from the equivalence oracle, learning stops as the conjecture is equivalent to the bunq application.

### 3.3.3 Feasibility Techniques

Time feasibility is an important aspect when adopting security tools. Because the actions from the input alphabet need to be simulated on the physical device, the runtime performance drops dramatically. The MAT framework implementation from Lampe et al. adopts three techniques that enhance the time feasibility of the active learning process.

The first technique uses specific resets to restart the application. The most time-consuming reset is a hard reset, which stops all services, deletes the application cache database and restarts the application to its startup activity. Although this type of reset is capable, it might be too preposterous for specific actions. Situations where the latter property holds are for example scenarios where pressing the `back`-button several times also suffices to reach the startup activity.

The second technique that is implemented to enhance time feasibility is called the 'fast-forwarding' of obsolete queries. This technique embraces the assumption that for a given sequence of actions, at some point an element might not be found, which causes the query to be obsolete. The remainder of this query does not have to be simulated since at this point an earlier action could not be resolved. Hence the query is fast-forwarded to a negative result.

An example of fast-forwarding is depicted in Figure 3.2 for word $w = w_0w_1w_2$. Situation $(a)$ depicts the behavior from the application if all elements are found: starting from state $q_0$ action $w_0$ can be satisfied and thus moves on to the next state, etc. All actions in $w$ can thus be satisfied. Situation $(b)$ shows that action $w_0$ can be resolved, but $w_1$ cannot be satisfied. The reason why an action fails can have numerous reasons. The most prominent reason is that a certain element cannot be located within the specific state. The simulation will just move on to the third action: $w_2$, which causes the application to move to another state. The problem with scenario $(b)$ is that $w$ is not an access sequence to $q_2$: this would be word $w' = w_0w_2$, hence the query is obsolete. When processing the action $w_1$, the teacher does not process $w_2$ anymore and instead returns a negative result.

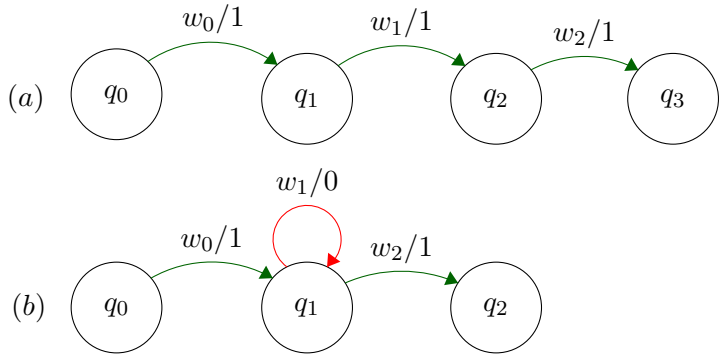



**Fig. 3.2:** Fast forwarding of word $w = w_0w_1w_2$

The third technique that enhances time feasibility of learning is the adoption of a cache. The cache stores the input sequence and the corresponding output in a

dictionary data type. The cache is not persistent, so the cached data is not available between different executions of learning.

## 3.4 Discussion

The tool fsm-learner has demonstrated to produce useful results concerning state machine learning on mobile Android applications. Some aspects of are not suitable to our needs, whereas other aspects form a suitable basis to be modified. Functionality that is irrelevant and must be erased are for example logging on to an application after each reset. Another practical obstacle of the tool is that fsm-learner is only able to connect to an outdated and unsupported version of Appium. Over the course of two years, the Appium driver has changed gigantically. Most of the practical obstacles are not discussed in this report, as they do not have an academic origin. These obstacles are listed online[3]. Various other attributes of the fsm-learner are suitable for state machine learning of general applications, amongst which is the LearnLib integration and the component architecture. The tradeoff to choose for a USB tethered device instead of an emulator is also well substantiated. Hence this research will apply the same technique to connect the SUT.

[3] Available at `http://www.github.com/wesleyvanderlee/`

# Model Inference Tool

# 4

> *Our Age of Anxiety is, in great part, the result of trying to do today's job with yesterday's tools . . .*
>
> **— Marshall McLuhan**

*The previous two chapters discussed notorious active state machine learning algorithms and an initial approach, the fsm-learner, that applies the MAT framework to a single mobile application. To apply model learning to generic mobile Android applications in a time feasible way, the deficiencies discussed in Section 3.4 are addressed and mitigated before the tool can be extended.*

This chapter proposes solutions that solve two practical problems that arise when applying state machine learning to Android applications. The first solution deals with the issue that the tool fsm-learner cannot infer a model from an application other than the bunq application the tool was designed for. As a result, the proposed solution can infer a correct model for an Android application. The improvement composes an answer to the first research question: in what way model learning can be applied to generic mobile Android applications. The second solution comprehends a proposition to advance the time-feasibility of learning state machines, by employing two capabilities. First, the state machine learner integrates active learning algorithms with a lower time complexity and secondly, other techniques that induce algorithmic speedup are incorporated, such as a caching system. The second proposed solution embodies an answer for the second research question on the methods that allow model learning of Android applications to improve time-feasibility. This chapter presents the work that has been performed to establish the two mentioned perspectives.

## 4.1 Android Application Model Inference

Chapter 3 stated all technical deficiencies in the source code of the tool. After these shortcomings have been tempered[1], the tool can be improved and extended for mobile learning. To correctly infer a model, multiple free mobile Android applications are used as the system under test. One application that has been prominently used during this phase is the Dutch application for public transportation journey scheduling: 9292 [2]. At this point the size of the input alphabet $\Sigma$ is $14$. This section discusses how various learning algorithms and equivalence oracles are utilized to infer a state machine model that describes the application's behavior as thoroughly as possible.

[1] a track record of how the shortcomings are mitigated can be found here: `https://github.com/wesleyvanderlee/Thesis/`

[2] `play.google.com/store/apps/details?id=nl.negentwee`

| Learning Algorithm | L* |
|---|---|
| Equivalence Oracle | RandomWalk |
| Membership Queries | 1778 |
| Equivalence Queries | 1 |
| States | 9 |
| Transitions | 25 |
| Learning Time | 26:06 $(hh:mm)$ |

**Tab. 4.1:** Statistics for active learning the inferred state machine for the 9292 application using L* and RandomWalk

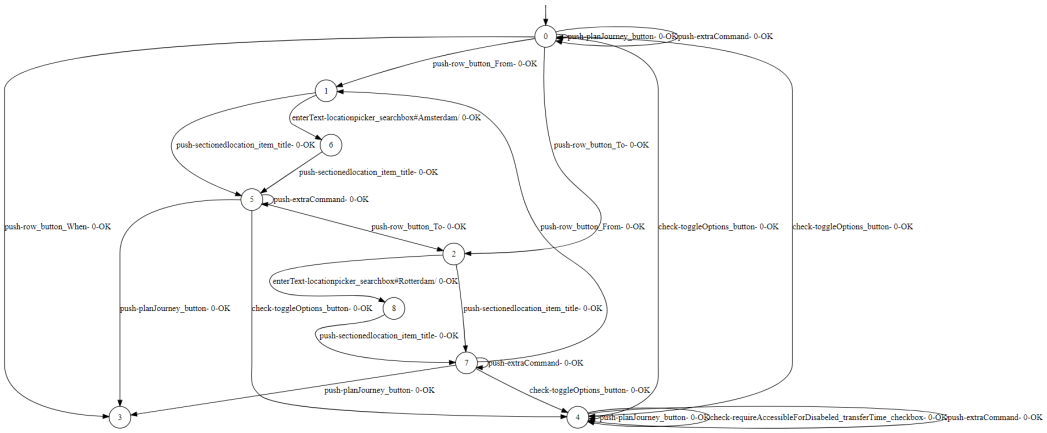



**Fig. 4.1:** The inferred machine for the 9292 application using L* and RandomWalk

### 4.1.1 L* and RandomWalk

The tool created by Lampe et al. [11] only works for the L* algorithm. Running the tool with its default setting yields the state machine depicted in Figure 4.1 with the corresponding learning statistics shown in Table 4.1. The learning algorithm posed only one equivalence query and thus did not receive a counterexample at all. The absence of a counterexample indicates that at the first time the observation table became closed and consistent, the RandomWalk algorithm was not able to find a counterexample. Hence, the DFA that can be generated based on the observation table is believed to be the right one. One should also note that the DFA depicted in Figure 4.1 does not completely correspond to the final, closed and complete observation table because the figure only portrays positive results, i.e., behavior that can be executed. As discussed in the previous chapter, the fsm-learner returns two types of return values: `0-OK` for actions that are allowed to be performed in a certain state and `1-NOTFOUND` for actions that are not allowed to be performed in a certain state. The `1-NOTFOUND` value can be for example returned when an action depicts the push-action on an element that is not present for a specific state of the application. Because the observation table has the closed property, the observation table describes all one-letter extensions from each state. The DFA that fully corresponds to the observation table, has $k$ outgoing transitions for each state, where $k$ is equal to the cardinality of the input alphabet. To remain a clear overview, those transitions have been filtered.

### 4.1.2 TTT and RandomWalk

One disadvantage of the L* algorithm is the storage of redundant information in the observation table. Because the observation table needs to be complete, redundant entries cause superfluous membership queries and as a consequence, the entries negatively influence the learning time performance. The learning time of the model inference discussed above consumes more than 26 hours. The feasibility of model inference increases if the learning algorithm does not query for superfluous data. Another algorithm that is discussed in Chapter 2 is the TTT algorithm. TTT uses a discrimination tree to overcome the problem of redundant data entries. The TTT algorithm can be started by instantiating another class from LearnLib. However, Lampe et al. argue that the TTT algorithm does not work in the fsm-learner because their inferred model did not approximate the SUT close enough. Learning the state machine model with the TTT algorithm of the 9292 application shows that our inferred model is nonequivalent to the SUT as well. Therefore the applying the TTT algorithm induces an identical conclusion. The model that is learned with the TTT algorithm yields the model that is depicted in Figure 4.2 and the corresponding learning statistics are shown Table 4.2.

| Learning Algorithm | TTT |
|---|---|
| Equivalence Oracle | RandomWalk |
| Membership Queries | 55 |
| Equivalence Queries | 2 |
| States | 2 |
| Transitions | 4 |
| Learning Time | 00:21 $(hh:mm)$ |

**Tab. 4.2:** Statistics for active learning the inferred machine for the 9292 application using TTT and RandomWalk

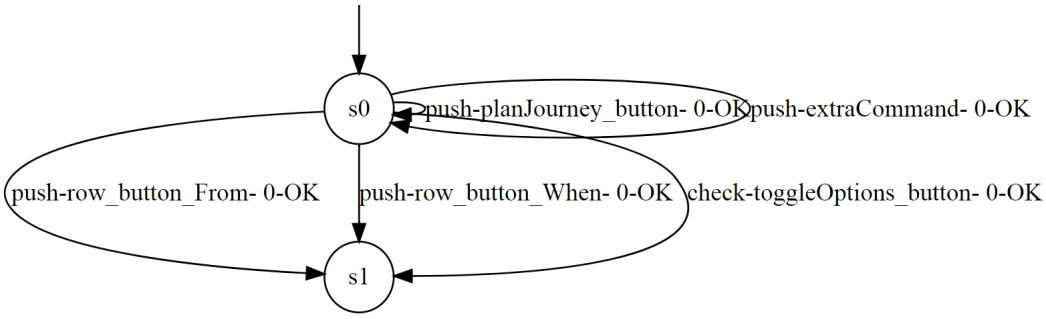



**Fig. 4.2:** The Inferred Machine for the 9292 application using TTT and RandomWalk

The inferred DFA in Figure 4.2 naturally yields an incorrect result as the model does not depict the behavior from the 9292 application at all. The difference between the models illustrated in Figures 4.1 and 4.2 show a decrease in modeled behavior when using the TTT algorithm. At this point Lampe et al. contend that the TTT algorithm yields an incorrect result for the learning process. In this instance, the inferred model is incorrect because the equivalence oracle was not able to find a

valid counterexample and hence the learning process stopped prematurely. The reason why the same equivalence oracle results in a more detailed model is that the L* algorithm generates more traces than the TTT algorithm due to its redundancy. If at the point where the oracle determined that the conjecture is equivalent, the equivalence oracle returned a counterexample, the inferred model would depict more correct behavior. The test set for which the equivalence oracle determines if the conjecture and the SUT return the same result must thus be extended or changed to advance the inferred model.

### 4.1.3 TTT and RandomWalk-HappyFlow

A method to expand the test cases is to test for a pre-defined input sequence that is guaranteed to be accepted by the SUT. The input sequence could, for example, be a *happy flow* of the application. Software applications are often developed with domain-specific use cases as a functional requirement [29]. The use cases describe how a software system should respond and the behavioral path that is consistent with the application is called a happy flow. Since the happy flow describes correct behavior, by definition, a happy flow sequence should be accepted by the SUT. Hence a happy flow can function as a counterexample if the conjecture does not accept the happy flow. Furthermore, if the input sequence $w$ is a happy flow than for each $n = 1 \dots |w|$ the sequence $w' = w_0 w_1 \dots w_{n-1} w_n$ is also a valid counterexample if the conjecture rejects $w'$. Because of all steps up until $|w|$ are legally executable as well. The RandomWalk equivalence oracle has been extended, dubbed as RandomWalk-HappyFlow, such that the equivalence oracle was able to search for a counterexample in the SUT's happy flow when the oracle otherwise would conclude that the conjecture is equivalent.

**Algorithm 3** RandomWalk-HappyFlow

**Input:** Hypothesis $H$
Array of happy flows $F$
**Output:** Counterexample $w$ ▷ `null` if $H \equiv$ SUT
1: $w \leftarrow RandomWalk(H)$ ▷ Original RandomWalk (See Alg. 2)
2: **if** $w =$ `null` **then**
3: **for** Sequence $f : F$ **do** ▷ Single Happy Flow
4: **for** $i = 0 \dots |f|$ **do**
5: $test \leftarrow f_0 \dots f_i$
6: **if** $H[test] \neq$ `accepting` **then**
7: return $test$ ▷ $H$ and SUT differ for $test$
8: **end if**
9: **end for**
10: **end for**
11: **end if**
12: **return** $w$

The extended RandomWalk algorithm has been formally written down in Algorithm 3. The algorithm first executes the original RandomWalk algorithm to discover a counterexample. When the routine of the original RandomWalk algorithm results in the value `null`, learning would normally stop. At this point, the extended part of the RandomWalk-HappyFlow algorithm engages, by searching for a counterexample for

each sub-word of all happy flow words. This procedure has the following advantages regarding the speedup of identifying a counterexample:

1. Because happy flow words and the corresponding subwords are computed only for the hypothesized conjecture, no simulation on the SUT is needed. Hence identification of a valid counterexample from a set of happy flows can be performed instantaneously.
2. The test words are generated from the smallest subsequence to the largest subsequence. If the hypothesis accepts a subsequence $w = w_0 \dots w_n$ but the subsequence $w \cdot w_{n+1}$ is rejected by the hypothesis, symbol $w_{n+1}$ is most likely a distinguishing suffix in the case of the L* learning algorithm or a discriminator in the case of the TTT learning algorithm. Because the counterexample is as small as possible, it reduces the appearance of redundant entries in the observation table and temporary nodes in the discrimination tree.

Although the extended RandomWalk algorithm yields a better result opposed to the original RandomWalk algorithm, a downside is that the happy flow must be defined in advance. The happy flows can originate from various sources, such as automated tests and application logs. The source and format, if any, differ per application and must thus be collected and processed for each SUT individually. Gathering happy flows requires knowledge about the application before learning starts and is therefore out of the scope of black box testing methodology. In conclusion, because of the necessary knowledge before testing, an equivalence oracle such as RandomWalk-HappyFlow is not the preferred solution. To summarize, the extended RandomWalk has proven to be useful for learning a more complete model, but the required application-specific knowledge deters the purpose of inferring a model in the first place.

### 4.1.4 TTT and the W-method

Another method to discover counterexamples is to substitute the equivalence oracle from RandomWalk to the W-method, which is a more systematic and exhaustive method. The W-method, has been thoroughly discussed in Section 2.5.2 and one must recall that it establishes a set of test cases by concatenating the state coverage set $P$, a middle part $M$ and a characterizing set $W$. Note that set $M$ is a subset of $\Sigma^*$, and $M$ equals to $\Sigma^{m-n}$, where $m$ is an upper-bound estimation of the maximum number of states the conjecture should have. The number of states in the conjecture is $n$. The W-method thus works by traversing each state, because of the state coverage set, to all other states, because of the middle part and lastly distinguishes each state, which is guaranteed to happen because of the characterization set. If the SUT behaves the same for the maximum depth $m$, that is the W-method is not able to discover new states in the conjecture, the conjecture and the SUT are equal.

Two drawbacks of the W-method are the explosion of the number of test cases as $m$ increases and the excessive amount of symbols the test queries contain, i.e., query length. Both factors negatively influence the speedup. After multiple test runs with the W-method equivalence oracle, no test run could finish, due to the quantity and size of the test cases generated by the W-method. The run required more time than $60$ hours and was halted in during the third equivalence query where it had to loop

through more than $38$ million test cases. The hypothesis conjecture consisted of $6$ states, which is known to be incorrect since the combination of L* and RandomWalk already was able to find $9$ states, as depicted in Figure 4.1. To reduce the number of test cases, Smeenk et al. define a novel heuristic for randomly selecting the middle part $M$ by manually establishing sub-alphabets [30]. Multiple smaller alphabets strongly reduce the size of the middle part $M$. To reduce the test queries and still be able to infer a correct model, Smeenk et al. argue that these sub-alphabets are required to be crafted by hand. For this reason, reducing the test set by applying sub-alphabets are not fit for automatic model inference. The other disadvantage of the W-method is the size of the test words. To decrease the length of the test words, i.e., the number of symbols in a sequence, Smetsers et al. propose a method for establishing the shortest distinguishing sequence for each pair of states [19]. The latter method has been thoroughly discussed in Section 2.5.3 where it was proposed to replace Chow's characterizing set by a set of minimal separating sequences for all pairs of states. As a result, this process would not only reduce the size of the characterizing set but can also eliminate unnecessary elements in the distinguishing suffixes and discriminators to limit redundant entries in the observation table and omit temporary nodes in the discrimination tree respectively. The algorithm that is the result of combining Chow's W-method and Smetsers et al.'s procedure for identifying minimal separating sequences is the WMethod-Minimal algorithm listed in Algorithm 4.

**Algorithm 4** WMethod-Minimal

**Input:** Hypothesis $H$
**Output:** Counterexample $w$ ▷ `null` if $H \equiv$ SUT
1: $P \leftarrow$ transitionCover($H$) ▷ as described in Sec. 2.5.2
2: $M \leftarrow$ allTuples($\Sigma$)
3: $T \leftarrow$ split($H$) ▷ stable and minimal splitting tree (Alg. 1,2,3 from [19])
4: $W \leftarrow \emptyset$
5: **for** $source$ : $H.states$ **do**
6: **for** $destination$ : $H.states$ **do**
7: $w \leftarrow T.lca(source, destination).\sigma$ ▷ lowest common ancestor in $T$
8: **if** $source \neq destination$ and $W$ does not contain $w$ **then**
9: add $w$ to $W$
10: **end if**
11: **end for**
12: **end for**
13: **for** all combinations $w \in P \times M \times W$ **do**
14: **if** $H[w] \neq SUT[w]$ **then**
15: **return** $w$ ▷ this is a counterexample
16: **end if**
17: **end for**
18: **return** $null$ ▷ $H$ and $SUT$ are equivalent

The WMethod-Minimal algorithm starts by establishing the transition cover set $P$ and the middle part set $M$ as the normal W-method would do. The algorithm creates a stable splitting tree according to the procedure of Smetsers et al. Next, the characterizing set is formed by adding all distinct labels between each pair of states. The algorithm proceeds as the normal W-method would continue, by comparing the test results of all generated test cases. The model that can be inferred with the

WMethod-Minimal equivalence oracle is depicted Figure 4.3. The learning statistics that accompany the model inference are presented in Table 4.3.

Reviewing these results shows that the learning time has dropped to 23 hours and the method has discovered an additional state. The fact that a new state has been discovered, means that the WMethod-Minimal was able to find at least one other counterexample opposed to the RandomWalk oracle. The discovery of the additional state shows the superior performance of the WMethod-Minimal equivalence oracle.

| Learning Algorithm | TTT | L* |
|---|---|---|
| Equivalence Oracle | WMethod-Minimal | WMethod-Minimal |
| Membership Queries | 15619 | 16966 |
| Equivalence Queries | 10 | 2 |
| States | 10 | 10 |
| Transitions | 30 | 30 |
| Learning Time | 23:14 $(hh:mm)$ | 27:37 $(hh:mm)$ |

**Tab. 4.3:** Statistics for Active Learning the Inferred Machine for the 9292 application using TTT, L* and WMethod-Minimal

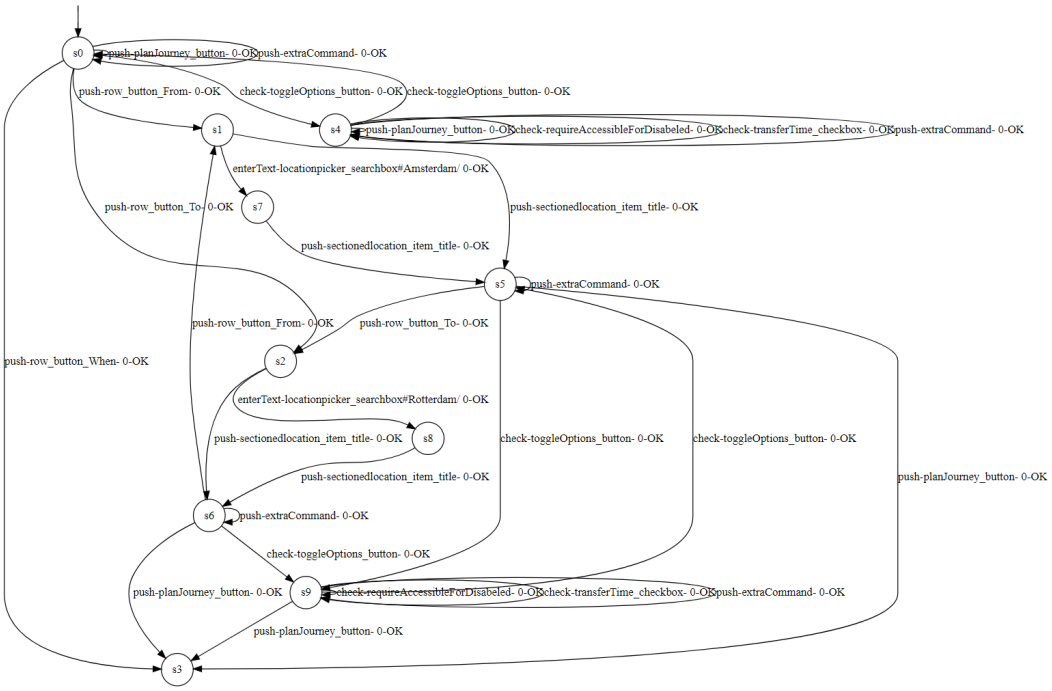



**Fig. 4.3:** The Inferred Machine for the 9292 application using TTT and RandomWalk

The L* learning algorithm has also been executed in conjunction with the WMethod-Minimal equivalence oracle and the corresponding results are also depicted in Table 4.3. The L* algorithm was able to learn a model equivalent to TTT's model depicted in Figure 4.3, but the algorithm requires more membership queries and thus also more learning time. When this run is compared with the initial run which combined the L* learning algorithm and the RandomWalk equivalence oracle, the last run

requires one additional counterexample. If the RandomWalk equivalence oracle could discover this counterexample, the initial run would also infer the correct model, but searching for more complex counterexamples is precisely the property what makes the WMethod, and thus also the WMethod-Minimal, a more advanced and better-suited equivalence oracle.

## 4.2 Mobile Variables

So far, state machine modeling requires consistency in the output that the system under test returns for a set of equivalent input sequences. Inference of the models presented in this chapter has shown that there are factors that are external to the application under test, which influence the output result. The mobile environment introduces a large number of factors that cause the results for equivalent input sequences to be inconsistent. Section 4.2.1 discusses this problem further by also elaborating on the impact of inconsistent results and providing a solution to the stated problem.

The mobile application domain does not solely introduce uncertainty and adversity when learning state machines. The domain also provides the opportunity to specify actions that incur specific behavior. The input alphabet can thus be extended to explore a broader perspective of behavior. Section 4.2.2 examines different opportunities to extend the input alphabet and discusses their effects on the learning process.

### 4.2.1 Non-deterministic Application Behavior

The behavior of mobile applications depends on a set of environmental influences. The most well-known example is the loss of internet connectivity. At arbitrary moments, due to a large number of factors, the mobile device or the application instance can experience a loss of connectivity with the internet. This information is often prompted to the user in a special view, such as shown in Figure 4.4. The red rectangle has emphasized the message.

The reason why the external influences on the behavior are essential to consider when learning a DFA is that different behavior in time causes the model to be non-deterministic. Since the goal is to infer a deterministic automaton, the non-deterministic behavior forms a contradiction to the earlier observed behavior. Example 4.2.1 illustrates the impact of non-deterministic behavior.

**Example 4.2.1.** Consider an action $a$ to be `push%button1` which is permissible at the start of the application. Action $a$ changes the screen to a new screen if there is an internet connection because pushing `button1` queries for server side-data. Note that with an internet connection, action $a$ is permissible, whereas $a \cdot a$ is not, as the object `button1` is not available on the second screen. When the learning algorithm performs the action $a \cdot a$, it observes a negative result and stores this result in the observation table. When at some point further in time the internet connection is lost and the learner queries for a word with $a \cdot a$ as a prefix, this results in a positive

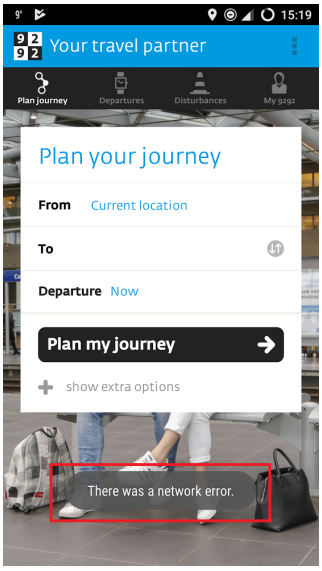



**Fig. 4.4:** The 9292 application without a network connection

result which contradicts the earlier observed behavior. Furthermore, actions that are permissible at the state after $a$ can only be executed if $a$ is successfully executed.

When the learner discovers ambiguous and contradictory behavior, the observation table becomes irregular. The ambiguous behavior cannot be modeled in a deterministic way and as a consequence, the learning application crashes. The environmental influences do not necessarily need to originate from an improper network connection, but can also arise due to inconsistencies in GPS positioning details, Bluetooth connections, the battery level and others. The external influences that induce the non-deterministic behavior limit the behavior that otherwise could be invoked. Given the assumption that behavior can be limited due to the external influences, the behavior that is not limited defines the correct behavior. Example 4.2.1 illustrates the latter conclusion because actions after $a$ are only executable after first successfully executing action $a$, which happens if there is a working Internet connection.

To overcome the occurrence of non-deterministic behavior, we propose and implement a roll-back solution that uses the cache in the following way. Figure 4.5 illustrates the roll-back process. The illustration depicts a sequence $w$ of input symbols for which at two points $t_0$ and $t_1$ in time $\lambda_{t_0}(w) \neq \lambda_{t_1}(w)$. This situation is depicted in Figure 4.5a, where at time $t_0$ the output from $\lambda(w)$ is $0$, whereas at $t_1$ the output from $\lambda(w)$ is $1$. Because the lambda evaluation differs for the same input word, one of the two traces is incorrect. We have seen in example 4.2.1 that if an activity after $a$ can be successfully executed, the corresponding trace has not been affected by external influences, in this example the loss of Internet connection. Therefore, the learner assumes that the trace that depicts new behavior is correct instead of the trace that contends the opposite. There are numerous of dependencies that depend on the environment and thus change in time, that cause an action to be inadmissible. However, if the additional behavior can be executed once, the action must thus be legal and not be subjected to external influences.

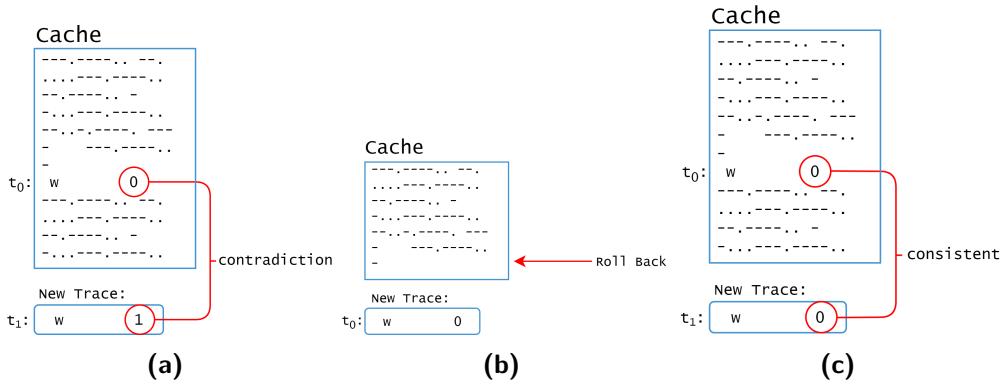



**Fig. 4.5:** Cache rollback methodology to resolve observed non-deterministic behavior. At (a) a contradiction is discovered, (b) the cache is reverted and (c) the contradiction is resolved.

During the learning period between $t_0$ and $t_1$, any number of queries could have been posed by the learner that (partially) depended on the cached result of $w$ computed at $t_0$. Because the result of $w$ is assumed to be uncertain, the queries posed between $t_0$ and $t_1$ that incorporate $w$'s cached result could also be false. For that reason, the cache is reverted to the last trace before $t_0$, which is shown in Figure 4.5b. The positive result for $w$ is cached at $t_0$ and learning continues. If for the new cache at time $t_1$ the query result is equivalent, the learning process continues.

### 4.2.2 Extended Input Alphabet

The tool fsm-learner allow three different types of actions to aid the learning process. The actions represent the user input behavior to click on items, check checkboxes and enter text to a text field. Regarding all possible methods to interact with a mobile application, the three mentioned action types appear to be limited.

Other types of actions that the application under test or mobile device that hosts the application can execute might discover additional new behavior. The types of actions one can distinguish are as follows:

- device specific buttons such as: `back`, `home` and `camera`;
- device settings, for example: toggling the WiFi and Bluetooth modus and the change of landscape setting;
- Android specific key events, in example: `LEFT`,`CALENDAR` and `ESCAPE` and
- Android intents, such as calling another activities in internal or external applications;

Especially Android intents have proven to introduce weaknesses in Android applications and are ideal to be subjected to fuzz testing, as performed by Ye et al. [31]. The research leads us to believe that an application presents different behavior for different intents. If the goal is to infer a state machine model that is as complete as possible, one could thus also include actions in an alphabet that induces the difference in behavior.

| **Device Info** | |
|---|---|
| Mobile Device | Samsung I9505 Galaxy S4 |
| Operating System | Android 5.0.1 (Lollipop) |
| API Level | 21 |
| CPU | Octa-core (4x1.6 GHz Cortex-A15 & 4x1.2 GHz Cortex-A7) |
| **Developers Options** | |
| Window animation scale | off |
| Transition animation scale | off |
| Animator duration scale | off |

**Tab. 4.4:** Mobile Device Configuration

Active automata learning defines new states if the behavior for some inputs is different. If the initial fsm-learner was able to discover web objects in a specific set of order, the modeled behavior corresponds to the application's internal structure, since an object can either be found or be not found. Although an application external command such as pressing the `home` button that is part of the Android operating system might invoke new behavior, the modeled behavior indicates useful information about the application under test. For example, pressing the `home` button once or any number of times returns a successful result as this action is legal at all times. Note that the legality of an action depends on whether or not Appium raises an exception or not. This information does not reveal any insight into the application. Moreover, the information invalidates the inferred model by adding a sinkhole state. A sinkhole state is a state which always can be reached, but cannot be escaped. In other words, once the sinkhole state is entered, application behavior cannot be executed anymore until the application is reset.

Pressing the *home* button yields an equivalent result as pushing the other external actions, such as toggling the Bluetooth modus. Because this type of external actions clutters the model that is learned with sinkhole states, external actions have not been included in the final proof of concept that learns Android application models.

## 4.3 Hardware Specifications

This chapter optimizes time consumption by using improved active learning algorithms and adopting computative improving techniques such as a cache and early query termination. The adopted techniques reduce the number simulations that are to be performed on the physical mobile device by reducing the number of membership queries. The improvement that can be achieved by advancing the hardware options is negligible since the simulation will not experience a speedup. To make the results reproducible, the hardware configuration is presented in Table 4.4.

# 5 Vulnerability Identification on Models

*Up until now we achieved to infer behavioral models on Android applications. Chapter 2 discussed the MAT framework and how the equivalence relation between a model and a software application can be solved. The previous chapter proposed a methodology and an accompanying proof of concept that implements a solution to the equivalence problem, such that we can infer correct state machine models from Android applications.*

The state machine models that are inferred so far depict how logical components of the application connect. The models present a logical skeleton of the mobile application. This research aims to use the skeleton as a guide to eventually determine the presence of certain vulnerabilities. To be able to achieve the identification of vulnerabilities in an application, the inferred model needs to be enriched with properties before one can address the security properties. This chapter discusses techniques to enrich the inferred model in Section 5.1 and considers what type of vulnerabilities materialize in the resulting model in Section 5.2. At last, Section 5.3 provides algorithms that identify the presence of vulnerabilities based on the enriched model.

## 5.1 Model Enrichment

To determine the presence of vulnerabilities through identification algorithms, the inferred model, first of all, needs to be enhanced with supplementary information. Enhancement of the models is achieved by adding labels to the transitions and states that provide information on the corresponding action or a corresponding state. An example label could be the network requests to an external API that are performed for certain actions. The process of establishing those labels and adding them to the model is called model enrichment.

Model enrichment has been separated from the vulnerability identification part, such that one can maintain a clear overview of the algorithms. Else the algorithms would all be cluttered with multiple model instances and in some cases even numerous SUT instances. Another reason to separate the two processes is to reduce time, because the entire model needs to be traversed only once, instead of each time per algorithm. Since enrichment adds new information from the SUT to the inferred model, the actions that are illustrated by the model need to be simulated on the target application to retrieve the relevant information per state. While traversing the learned state machines $M = \{Q, \Sigma, \delta, q_0, F\}$, all states and transitions are traversed to retrieve the information that is discussed in the following subsections.

### 5.1.1 Text

The state of a mobile application yields a graphical response to the end user. The screen portrays text that has a meaning which most of the time is application domain specific and not relevant from a security perspective. However, sometimes text has a semantic meaning that provides information for the application security domain. Examples are keywords such as 'error' or 'fault' to identify a state that correlates to an error state. The text can be gathered by traversing over all visible elements on the screen.

#### Limitation

Not all the text can be gathered from the mobile screen. Most of the time text elements are contained in a TextView instance (class: `android.view.TextView`). Sometimes an application posts messages in a Toast instance (class: `android.widget.Toast`), which is an Android system view that contains a quick and small message for the user. This message disappears after a few seconds. The Appium driver relies on the UIAutomator framework for interaction with the device. A limitation in UIAutomator, and therefore in Appium as well, is that toast messages cannot be gathered because they are not part of the application and do not inherit the `View` class. Instead, they are part of the operating system and inherit directly from the `Object` class. Hence toast messages cannot function as a data source for model enrichment.

### 5.1.2 Activity per state

An activity in Android is an overarching mechanism that handles a process initiated by an application. An activity, for example, generates the view for the user, deals with the process' logic, performs requests to an external server and directs the user to another activity. When an application starts, the launchable activity is executed. Each state can thus also be assigned to an activity and during the enrichment process, the states are labeled with the corresponding activity.

### 5.1.3 Network requests

Applications regularly communicate to a back-end server. The client-server connection is often the case for processes such as authentication verification and retrieving information results. Communication to a back-end server is often done over the Internet through HTTP methods such as GET and POST requests. Each action depicted in the inferred model could generate a request, but not all actions initiate a back-end request. It is thus very likely that one part of the transitions could be labeled with network requests and the other part could be not labeled. Because these requests play a role in assessing the application's security, more insight can be gained by labeling the transitions with the actual request.

Appium can execute commands and retrieve system information, but it cannot read out the requests that are made from the device to the Internet. To be able to read the web requests, the requests have to be intercepted by a proxy. A proxy acts as an

intermediary for network traffic between the mobile application and the back-end server. The mobile phone connects to the proxy and pushes requests as if the proxy was the Internet. The proxy then forwards the requests to the Internet and returns the result, but also logs the requests.

A ubiquitous proxy tool that is widely supported and available for Android clients is Mitmproxy[1]. Mitm is an acronym for 'man in the middle', which is a type of attack where data interception is the primary purpose. Network requests that are sent over HTTP, interception of readable data is as simple as reading out all the requests. For network requests sent over HTTPS, data interception becomes more difficult, since the connection between the device and the server is encrypted.

### HTTPS Traffic

The goal of mitmproxy is to sit in the middle of the data stream between the mobile device and the back-end server and be able to read the data understandably. HTTPS traffic prevents exactly this process by providing an end-to-end encrypted session, meaning that only the mobile device and the server can decrypt the traffic. A Certificate Authority (CA) system is designed to ensure the end-to-end encryption, by signing the server's certificate that comes along with the session. If the signature doesn't match with a known signature or is from a non-trusted part, the client drops the connection. The CA system becomes active in the case HTTPS is used. Hence the CA system increases the difficulty to understand the network traffic if it is sent over HTTPS.

To overcome the problem of not being able to read HTTPS network traffic, mitmproxy needs to become a trusted CA on the mobile device itself. If mitmproxy becomes a trusted CA, the handshake between the application and the proxy that establishes the encrypted session looks legitimate from the application's perspective. Enabling mitmproxy as a rogue CA is achieved by installing the mitmproxy certificate on the device. If the client now connects to a server through the proxy over HTTPS, the proxy connects to the server to retrieve its certificate. The genuine server certificate is copied to a forged certificate and then signed by mitmproxy itself. This forged certificate is then returned to the client as part of the process of establishing a TLS connection. The client validates the certificate as trusted because mitmproxy's CA certificate is installed on the device and the proxy can decrypt HTTPS network traffic.

At this point, the proxy performs the requests that are generated by the application on the application's behalf without their knowledge. On top of that, the device does not know it is communicating to the proxy instead of the genuine server.

[1] `https://github.com/mitmproxy/mitmproxy`

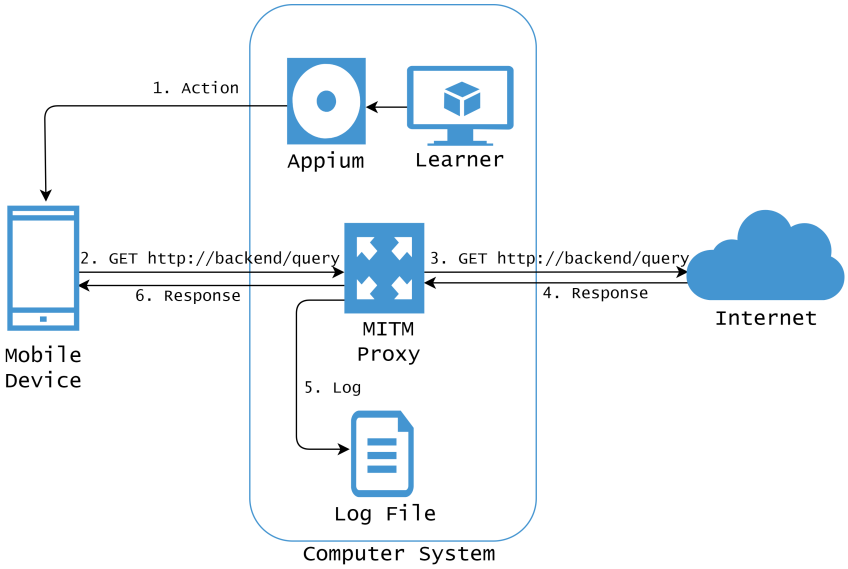



**Fig. 5.1:** Proxy Setup Scenario 1

### Physical Device or Emulator

The proxy can be deployed in two settings, either directly on the physical and USB tethered device that is used for learning or on a new emulated device that has the sole purpose of retrieving network information.

The first option would be to use the physical device since there is already a real USB tethered device connected to Appium. This scenario would require that all Internet connections of the mobile device are forwarded over the USB connection to the proxy on the computer system. The proxy will then forward the request to the Internet and return the response to the mobile device. Moreover, it will log every action. This setup has been visually depicted in Figure 5.1.

### Limitation

Mobile device users can install new certificates, such as the mitmproxy certificate, on Android as 'user-trusted'. Configuring a certificate as 'user-trusted' can only be done if the device meets additional safety requirements such as using a PIN-code or password to unlock the device. Although complying with the security requirements appears to be straightforward and easily applicable, the requirements pose a problem, as Appium is not able to unlock a PIN/password protected device, even if the PIN/password is known to Appium. As a result, meeting Android's safety measures to install 'user-trusted' certificates, renders the device useless for the state machine learner as it cannot access the device anymore.

To overcome the mentioned limitation, a setup that uses an emulator has been proposed. The emulator will be explicitly spawned for retrieving the Internet requests from the application. Figure 5.2 visually depicts the proposed configuration. In the top left part of the figure, the familiar setup can be recognized, where the

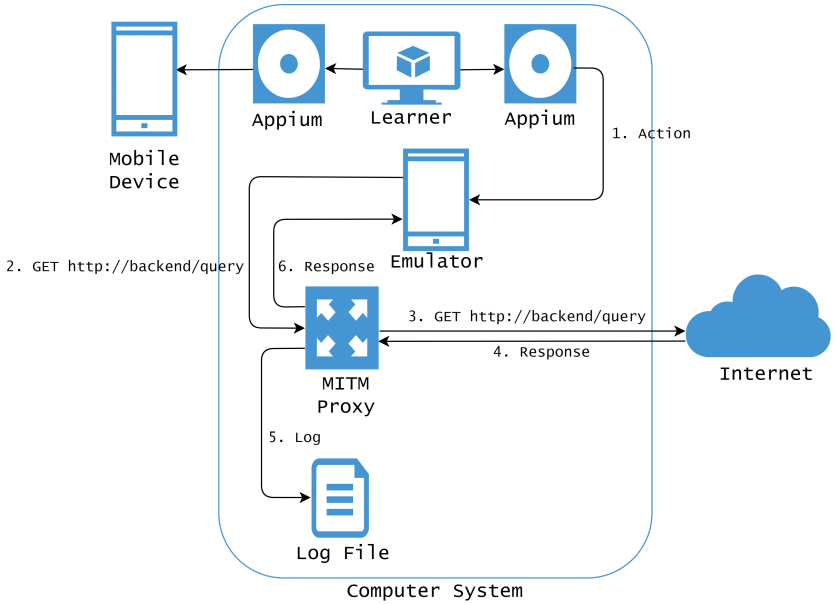



**Fig. 5.2:** Proxy Setup Scenario 2

learner communicates with the mobile device through Appium. The proposed setup includes the emulator as a new device. To connect to the emulated device to execute the application commands, an additional Appium server is required. For that reason, two Appium servers are depicted in the figure. The emulator's network traffic is forwarded to mitmproxy which can decrypt the traffic, log the traffic's contents and forward the traffic to the backend server over the Internet.

In conclusion, we are now able to retrieve the text elements from the screen, the activities that correspond to a state and the network requests that correspond to specific actions. The text is extracted using Appium, which in turn uses UIAutomator, which has the shortcoming that Toast messages cannot be intercepted. The text elements are used to identify error messages and identify login states. The activities that correspond to a state are also retrieved with Appium's functionality. The network requests are intercepted by a proxy from an emulated mobile phone. The additional virtual machine increases this project's complexity regarding dependencies. But the emulator is a necessity because a certificate needs to be installed on a PIN/password protected device.

## 5.2 Mobile Application Security

The inferred models are enriched to identify vulnerabilities that impact the security of an application. The term *security* describes techniques that control what entity is allowed to use or modify the system or the information a system contains [32]. Computer security, and as a subset software security, is the protection of items that one values, also known as the *assets* of an application [33]. To determine in what ways assets can be defended, one must think about ways to attack them. Attacking can be achieved by compromising one of the three following properties that together form the CIA triad: *confidentiality*, *integrity*, *availability* [34]. The following explains these properties.

1. **Confidentiality** is the ability of a system to ensure that any authorized party can access sensitive data. An example of this property is insurance that the username and passwords are well encrypted before storing it on the phone or sending it to a back-end server because adversaries are not allowed to access this type of information.
2. **Integrity** is the ability of a system to ensure that an asset is only modifiable by authorized parties in a way that is foreseen and consistent with a predefined policy. An example is that the back-end's response data is not changeable by an adversary.
3. **Availability** is the ability of a system to ensure that any authorized parties can use an asset at the times where it was agreed to be available. An example of availability is that video streaming through the Netflix application remains possible as long as there is the Internet and a subscription that has been paid for.

The properties mentioned above can be violated on numerous levels of interaction, such as the levels of the application's client, the application's back-end, other installed applications, the operating system and even the device's hardware. The scope that applies to the research conducted for this thesis focuses on the application's client and its interaction with the back-end. The security properties can be assessed by exploration of ways to violate them. Violation is typically done by misusing the application by exploitation of weaknesses. Identification of attack paths is paramount to conclude the presence of vulnerabilities. To determine what type of attack paths and thus also what type of vulnerability categories are present, one could look to the Open Web Application Security Project (OWASP) Foundation [35]. OWASP is an organization that actively maintains documents and guidelines that improve the capability to produce secure code. One of these documents is the OWASP Mobile Top 10 2016 a top 10 list of categories that describe ways to compromise or misuse a mobile application [36]. The OWASP lists are widely adopted by the security community in the entire world to guide identification of vulnerabilities. Subsection 5.2.1 will briefly describe the top 10 list for mobile exploitation categories. Subsection 5.2.2 discusses which of the mentioned categories are fit for identification in models.

### 5.2.1 OWASP Top-10

The OWASP Top 10 is the flagship project founded by OWASP to set a standard for most critical application security risks. The project's contents is a list that has been created by proactively interviewing security industry experts and extensively reviewing the results in public. Because the list can enable a software security tester to identify vulnerabilities in a structured way, the vulnerabilities in the top 10 list will also be reviewed if they can be identified from a model. The following describes a brief overview of the list's contents.

1. **Improper Platform Usage**
   The category of improper platform usage covers the application's failures to meet the platform security controls and other scenarios where platform features are misused. An example of an improper platform usage could be a

wrong implementation of Android intents. Intents are abstract descriptions of an operation to be performed, such as the management of screen activities. The mobile application can listen to events and send the messages to other components of the application or other applications entirely. A failure to meet the correct implementation of an Android intent could, for instance, be the absence of correctly performing the `null` check of the intents origin. If this is not implemented correctly, any instance can initiate the operation that succeeds the intent [37]. Since secure operations can be invoked from an illegitimate session when this category is not met, the application's confidentiality is at stake. Because intents can also be generated to keep the application busy, i.e., a denial of service attack, inadequately implementing intent control also compromises the availability of the application.

2. **Insecure Data Storage**
   The insecure data storage category occurs when information processed by the application, is stored in an insecure way. Android applications can store data in the mobile device's file system. Other applications or users can access the file system. If the data being processed is improperly secured, this data can be compromised or even changed. An example of misuse could be that a raw database file is stored unencrypted. If the database stores all the users account names and passwords, these assets can be read out by other adversaries, thus compromising the confidentiality of data, or changed for malicious purposes and thus compromising integrity. If the data is stored by using a secure methodology, such that the data is encrypted before storage, confidentiality and integrity can be preserved.

3. **Insecure Communication**
   As has been discussed in the previous section, most applications function according to a client-server framework, where the application (client) communicates with a server (back-end). The communication between the client and the server often encloses sensitive data, such as authentication credentials or sensitive files. Sending and receiving sensitive data should be done on a cryptographically secured communication, such as the transport layer security (TLS) protocol describes, or apply custom cryptographic solutions, such as certificate pinning. If communication is insecurely applied, the security properties confidentiality and integrity can be possibly breached, as the data might be accessible to view and change by a third party in between the client and server connection through a man-in-the-middle attack.

4. **Insecure Authentication**
   Sometimes a part of the application is shielded for anonymous users and requires authentication to grant access to user specific or sensitive information and operations. An authorization scheme is required to shield a part of the application effectively. Essential to this scheme is the authentication process, where a user verifies its identity. If verification is not established correctly, the part of the application that performs confidential operations is accessible to anyone, thus breaching the confidentiality property. An example of insecure authentication is a login screen that can be bypassed by a SQL-injection [38]. Through such an attack, an adversary can divert the process of the authentication process and appears as another identity.

5. **Insufficient Cryptography**
   To enable secure data storage, cryptographic fundamentals are needed to store

the data securely. This category covers the components of the application that implement these cryptographic fundamentals to ensure the same security properties as secure data storage: confidentiality and integrity. Cryptography can be insufficient in two ways. First of all, the cryptography algorithm can be obsolete, meaning that the algorithm does not provide an appropriate level of security. Secondly, poor key management may lead to the compromise of the decryption key. This key can be stored in plain text in the source code of the application, be stored in memory or even appear on the project's GitHub website [39].

6. **Insecure Authorization**
   Authorization associates the appropriate level of operations to an authenticated identity. A poor or missing authorization scheme allows attackers to exploit functionality that is above their privilege. Depending on the type of operations an attacker could inappropriately access, the attacker could misuse the entire CIA triad. For example, a low-level user can create a TLS session with the back-end server. The session has the corresponding cookie, which indicates that the user is not from an administrative level. Changing this cookie value opens an administrative part of the application. Depending on the administrative controls, the attacker can now view, change and delete data in the back-end, thus compromising confidentiality, integrity and availability respectively.
7. **Poor Code Quality**
   The poor code quality category defined by the OWASP can enable an attacker to exploit any of the other vulnerability categories. The reason why poor code quality is defined as a separate category in the top 10 list, is because it also embodies non-exploitable vulnerabilities, such as system crashes or logically unnatural activities.
8. **Code Tampering**
   Modified forms of applications that are changed by a third party are commonly distributed through official and unofficial marketplaces. In this case, an application is either torn apart, the code is tampered with and then reassembled again or an entirely new application is made from scratch that tries to impersonate the original application. The modified version tries to trick users into thinking it is a benign application, but besides, the tampered application can perform malicious activities. The malicious activities can include excessive advertisement generation, stealing personal information or exploit other applications. This category classifies ways to prevent tampering an application.
9. **Reverse Engineering**
   Reverse engineering an application is the process of extracting knowledge or design information from an application. This knowledge can even go back to the original source code of the application. The source code is especially easy to retrieve for Android applications since they are written in Java, which allows dynamic introspection at runtime. A mechanism to protect the inference of application knowledge is source code obfuscation, which makes understanding it more difficult. If the application's logic is seen as an asset, then this category protects the confidentiality property of the application's source code, as it should not be viewable or understandable by third parties.
10. **Extraneous Functionality**
    The category of extraneous functionality describes functionality that enables a user to perform operations that is not directly exposed via the user interface.

For example, during development, an application might have a legitimate developer's backdoor to bypass initial authentication to make software testing easier. If at the public release of the application, the superfluous logic is not removed from the application, it serves as extraneous functionality. Depending on the operations that are included in the extraneous functionality, all security properties could be at stake.

In conclusion, we have seen and exemplified how different classes vulnerability can violate security properties. In the mobile application domain, confidentiality and integrity are most at risk of being compromised. No vulnerability class addresses the availability security property. Availability is less discussed because we are dealing with mobile applications which are clients and thus only impact the local device. If an attacker wants to exploit the availability of an application server, he or she needs to attack either the backend of the application or other applications through their client application. A model that is inferred from a single client application would not depict such scope and as a consequence, availability is almost not addressed.

### 5.2.2 Detectable Through Models

The discussed vulnerability categories above are all critical to mitigate during application development to conserve the security properties. Some of the categories do not appear in a behavioral model of a mobile application, as the types of vulnerability are not triggered by behavior, or are too low level. The application models that are inferred for this thesis will be on the user interaction level. There are many levels at which a vulnerability materializes. Some vulnerabilities materialize after a source code inspection, such as a review of the cryptographic methods that are used. Other vulnerabilities materialize after one checks static properties, such as whether or not an application is reverse engineerable. Inference of the application's behavior does not reveal any insights into the static property of reverse engineering, nor on the quality of the code.

The OWASP Top 10 has been reviewed and determined which type of vulnerability is detectable in a behavioral model of the application. Those categories are depicted in Table 5.1.

| OWASP Category | Detectable in a model |
|---|---|
| Improper Platform Usage | Yes |
| Insecure Data Storage | No |
| Insecure Communication | Yes |
| Insecure Authentication | Yes |
| Insufficient Cryptography | No |
| Insecure Authorization | No |
| Poor Code Quality | No |
| Code Tampering | Yes |
| Reverse Engineering | No |
| Extraneous Functionality | Yes |

**Tab. 5.1:** OWASP categories applicable for identification in models

The why and how some of the vulnerability classes can be detected from an inferred model is discussed in the next section. The remaining classes cannot be identified in the inferred model because the detection method requires information from the application or system on a level at which we cannot communicate. As explained before, some of the required information resides on a too low level that is not the user interaction level. Other security classes cannot be detected through observed behavior, but require for example source code reviews. The vulnerability categories that can be exploited by invoking actions on the user interaction level such that the behavior can be modeled, are detectable in a model. These vulnerabilities are depicted in Table 5.1 and algorithms to detect these vulnerabilities are discussed in Section 5.3.

## 5.3 Vulnerability Algorithms

The former sections discussed enhancement techniques on inferred models and consider classes of mobile application vulnerabilities and how they breach security properties. This section combines the two pieces by providing approaches for identifying the vulnerabilities through the enriched models.

1. **Error States**
   Frequently, when an attacker performs actions that are illegal or cause a failure in the application, the result of the action is notified to the end user. Although the action can be legal according to the application's logic, the operation might not be semantically correct. The application then raises a notification, which often has a semantic meaning. As a result, certain keywords in the notification can function as a classifier for the corresponding state being an error state or not. Keywords that have been used to identify an error state are: `error`, `failure`, `fault`, `unexpected` and `fail`. Algorithm 5 uses the classifier to determine whether the state is an error state which might indicate a security flaw and therefore a possible vulnerability.

**Algorithm 5** Error State Identification

**Input:** Inferred State Machine $M = \{Q, \Sigma, \delta, q_0, F\}$
**Output:** Returns a set $R$ of transitions that are error states in model $M$. There possibly exists a vulnerability if $R$ is non-empty.

```
1: R ← ∅
2: for all q ∈ Q do
3:     if q is classified as error then
4:         add q to R
5:     end if
6: end for
7: return R
```

2. **Dead ends**
   A state that cannot be escaped is called a dead end. The presence of a dead end does not necessarily imply that a software vulnerability is present in the application, but it might show unexpected behavior that the developer has not thought of while programming. In this scenario, one can best have a look at

what causes the unexpected behavior, as more unexpected actions are possible at the critical point that destabilizes the system.

**Algorithm 6** Dead End Identification

**Input:** Inferred State Machine $M = \{Q, \Sigma, \delta, q_0, F\}$
**Output:** Returns a set $R$ of states that are dead ends in model $M$. There possibly exists a vulnerability if $R$ is non-empty.

```
1:  R ← copy Q
2:  for all q ∈ Q do
3:      for all t ∈ δ do
4:          if q equals t.source then
5:              remove q from R
6:              continue                          ▷ skip remaining transitions
7:          end if
8:      end for
9:  end for
10: return R
```

3. **Improper Platform Usage** *OWASP-16-1*
The risk of improper platform usage is listed first on the OWASP Mobile Top 10. In essence, the risk categorizes misuse of platform security controls that are part of the, in our case Android, operating system. The controls also include Android intents such as the calling of activities. Under the assumption that the inferred model describes normal application behavior, any new behavior that can be accessed by calling an activity is superfluous and should not be accessible by end users. A violation of this vulnerability can be best exemplified with an application that is secured with a login screen. The inferred state machine, which describes normal application behavior, shows that activity after the login screen is only accessible after successfully logging in. If however an application activity can be called that executes code behind the login screen, the activity is regarded as the improper use of the platform and thus results in a severe application vulnerability. The algorithm to detect paths in the inferred model is depicted in Algorithm 7.

The algorithm first constructs a set $A$ of all activities that are callable and can henceforth potentially be misused. Next, all activities that are callable and can be reached by invoking normal system behavior, i.e. the inferred model, are removed from the set of activities that could potentially exploit additional behavior. The remaining activities cannot be invoked by normal interaction with the application, but can be executed through a debugger and therefore be misused. Set $R$ contains these activities that are callable and are not reached by invoking normal application behavior.

**Algorithm 7** Improper Platform Usage Identification

**Input:** Inferred State Machine $M = \{Q, \Sigma, \delta, q_0, F\}$
**Output:** Returns a set $R$ of activities that induce supplementary behavior in machine $M$. There possibly exists a vulnerability if $R$ is non-empty.

```
R ← ∅
A ← activities from M
for all a in A do
    if a is a callable activity then
        add a to R
    end if
end for                                  ▷ R contains all callable activities
for all q in Q and R is not empty do
    remove q's activity from R
end for
return R
```

4. **Insecure Communication** *OWASP-16-3*
   Most applications exchange data according to a client-server framework. Although the mobile application (*client*) and the back-end (*server*) are to be trusted, every entity in between the line of communication is not. The entities include other malicious applications that are installed on the phone and are listening to broadcast requests and rogue access points (AP) that control and monitor all network data passing through the AP. Modern security standards include that mobile applications apply at least the use of a secured communication (TLS) and at best exercise *certificate pinning*. Since the insecure communication category has gradations in itself (i.e., plain-text communication, the application of TLS, certificate pinning, etc.) a special function `request_insecure()` has been created that assesses a backend request for its security level.

The identification of this vulnerability is thus determined by the function `request_insecure()`. The function returns true if the request is made over HTTP and false if the request is secured through encryption protocols such as TLS and certificate pinning. The specification for the method `request_insecure()` is also depicted in Table 5.2. The algorithm that uses the method is presented in Algorithm 8.

| Request Type | Result |
|---|---|
| HTTP | true |
| HTTPS | false |
| Certificate Pinning | false |

**Tab. 5.2:** Specification of the `request_insecure()` method

**Algorithm 8** Insecure Communication Identification

**Input:** Inferred State Machine $M = \{Q, \Sigma, \delta, q_0, F\}$
**Output:** Returns a set $R$ of requests that made by actions in machine $M$ that do not adhere to common network security standards. There possibly exists a vulnerability if $R$ is non-empty.

```
1: R ← ∅
2: for all t in δ do
3:     r ← request_insecure(t.request)
4:     if r then
5:         add r to R
6:     end if
7: end for
8: return R
```

5. **Insecure Authentication** *OWASP-16-4*
Authentication is the process of distinction of confidential services or data in an application to verified and authorized end-users. It is listed fourth in the OWASP Top 10 Mobile and is thus seen as one of the most important security categories to be applied in a mobile application. One of the consequences of improper authentication is a possible breach of confidentiality as classified services or private data is accessible to anyone.

The inferred model can be used to assess the authentication of an application. By searching for paths to states that should only be accessible after authentication, one can identify an authentication bypass and diagnose an improper authentication vulnerability in the application. The algorithm that corresponds to detect insecure authentication is presented in Algorithm 9.

Algorithm 9 first determines the authentication state in the application. Classification of the login state is done based on the presence of keywords such as: `login`, `log in`, `authenticate` and `sign in`. Moreover, the access sequence to the authentication state should at least depict one occurrence of the action where text has been entered. Entering the text is necessary to enter the password at least. If no such state can be found, the application likely does not support authentication and a result, authentication cannot be bypassed.

If an authentication state exists, the algorithm searches for paths that can access a state after the authentication state by avoiding the authentication state. The states that should be secured by authentication are marked by the membership of the set $Marks$. Discovery of an authentication bypass path is achieved by creating a submachine $M'$ that is a copy of the inferred model but excludes the authentication state. If in $M'$ a path to one of the marked states is discovered, the path can bypass the authentication state and hence serves as evidence that the application is vulnerable to insecure authentication. The same process holds for executing activities that are callable.

**Algorithm 9** Insecure Authentication Identification

**Input:** Inferred State Machine $M = \{Q, \Sigma, \delta, q_0, F\}$
**Output:** Returns a set $R$ of authentication bypass techniques in machine $M$. There possibly exists a vulnerability if $R$ is non-empty.

```
1: a ← authentication state of M                                   ▷ a ∈ Q
2: if a is null then          ▷ no authentication → no authentication bypass
3:     return R
4: end if
5: Marks ← subset of nodes possible to reach after a
6: M' ← M − a                  ▷ the machine without the authentication state
7: for all m in Marks do
8:     if a path from the q_0 to m exists in M' then
9:         add path to R
10:    end if
11: end for
12: Q'' ← Q − Marks
13: A ← callable activities
14: for all q in Q'' and A is not empty do
15:    remove q's activity from A
16: end for
17: add A to R
18: return R
```

6. **Code Tampering** *OWASP-16-8*
Code tampering is the process of altering the application's source code by unauthorized third parties. The source code of an application can be changed by attackers, such that a benign looking application will perform malicious activities. To that extent, an application can be modified to collect data which is sold on the black market (*spyware*), perform activities on that phone, so it will generate revenue for the attacker (*ad-clicking fraud*) or bypass security measures in the original application to access paid/limited services.

Under the assumption that tampering source code yields different behavior, code tampering can be identified by comparison of the inferred model to a reference model of the application under test. The reference model can either be inferred from a legitimate data source, such as the Google Play Store, or inferred from an application that should be genuine and benign. If the SUT's model is the result of a tampered application, Algorithm 10 will identify the difference between the two of them. The behavior that the application under test contains in addition to the reference model can potentially be malicious.

The algorithm computes the transition cover set of access sequences, which is also used by the W-method. The transition cover set is created in the same way Chow [10] creates a transition cover set by the aid of a *testing tree*. Let `TCS(M)` be the function that returns the transition cover set for a DFA $M$, as proposed by Chow. The difference in behavior is then computed by assessing the outputs for each input sequence in the transition cover set by the inferred model and the reference model.

**Algorithm 10** Code Tampering Identification

**Input:** Inferred machine $M = \{Q, \Sigma, \delta, q_0, F\}$ and reference machine $M' = \{Q', \Sigma', \delta', q_0', F'\}$

**Output:** Finds difference in $M$ and $M'$ Returns a set $R$ of sequences that yield a different output for the two machines $M$ and $M'$. The sequences are divided into sets $R_1$ and $R_2$ which depict what machine models the sequence. There possibly exists a vulnerability if $R$ is non-empty.

1: $R_1, R_2 \leftarrow \emptyset$
2: $TCS_1 \leftarrow TCS(M)$
3: $TCS_2 \leftarrow TCS(M')$
4: **for all** $w \in TCS_1$ **do**
5: **if** $\lambda^M(w) \neq \lambda^{M'}(w)$ **then**
6: $R_1 \leftarrow w$
7: **end if**
8: **end for**
9: **for all** $w \in TCS_2$ **do**
10: **if** $\lambda^M(w) \neq \lambda^{M'}(w)$ and $w \notin R$ **then**
11: $R_2 \leftarrow w$
12: **end if**
13: **end for**
14: $R \leftarrow R_1, R_2$
15: **return** $R$

# Results

# 6

> *. . . as if a magic lantern threw the nerves in patterns on a screen . . .*
>
> **— T. S. Elliot**
> The Love Song of J. Alfred Prufrock

*The previous chapters discussed how we could apply state machine inference to mobile Android applications for learning models in feasible learning time. Moreover, we proposed a set of detection techniques that utilize the inferred model to detect the presence of vulnerabilities.*

This chapter applies the earlier proposed methodology for model inference and vulnerability detection on two types of applications and presents the corresponding results. The first application that is tested for the presence of vulnerabilities is a mobile banking application. The banking application has been deliberately made vulnerable to a number of attack vectors for testing purposes. Some of the attack vectors that can be applied to this application violate classes defined in the OWASP Top 10 Mobile, hence the tool is expected to produce results that depict identification of multiple vulnerabilities. The assessment of the vulnerable banking application is presented in Section 6.1. The second test examines a malicious version of a chat application. The inferred model can then be compared to a reference model to identify extraneous behavior. If the benign version of the chat application is used as a reference model, the extraneous behavior that is depicted in the inferred model describes the malicious behavior that is implemented in the fake chat version. The analysis of the malicious mobile chat application is presented in section 6.2

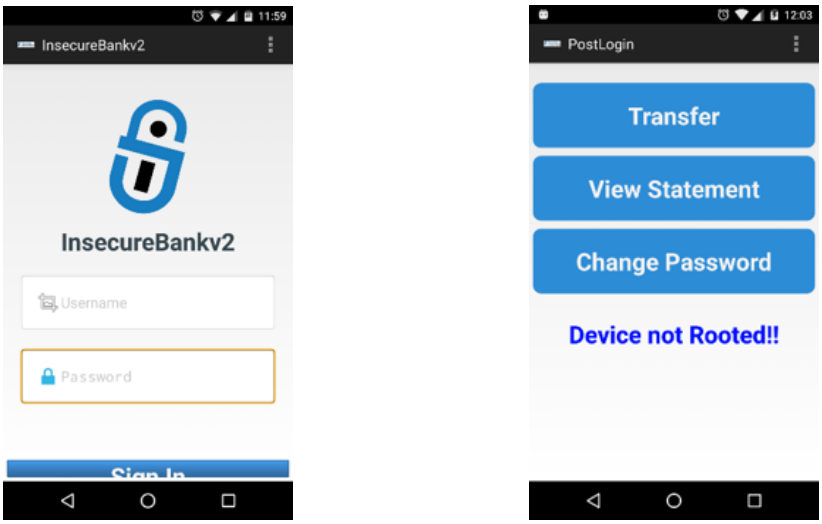
## 6.1 Banking Application

The first application to be tested is the fictitious banking application InsecureBankv2[1]. This application is originally developed with the intentions of being vulnerable. By doing so, the author of InsecureBankv2 provides an environment to teach exploitation of Android applications to Android developers and the security community. One of the best reasons to test this application is because it is known in advance what type of vulnerabilities are present and thus should also be detected by the developed tool.

The general idea of the vulnerable banking application is as follows. At the start of the application, a login screen is shown as depicted in Figure 6.1a. Valid credentials for the application are the username and password combination of `jack` and `Jack123$`.

[1] `https://github.com/dineshshetty/Android-InsecureBankv2`

After logging in with the valid credentials, a second screen is shown as depicted in Figure 6.1b, that allows the user to perform banking operations. A login screen thus protects the banking operations.




(a) (b)

**Fig. 6.1:** The user interface of InsecureBankv2: (a) the initial screen and (b) the screen after successfully logging in.

The minimal recorded time to infer the correct model of the banking application required 3 hours and 13 minutes. The learning settings and statistics that accompany the model inference procedure are presented in Table 6.1. The complete model is presented in Figure 6.2.

| InsecureBankv2 | |
|---|---|
| Learning Time | 3:13 |
| Learning Algorithm | TTT |
| Equivalence Oracle | WMethod-Minimal |
| Equivalence Queries | 7 |
| Membership Queries | 11627 |
| States | 6 |
| Alphabet Size | 6 |
| Cache Hit | 49% |
| Fast Forwarded | 42% |

**Tab. 6.1:** Learning statistics for InsecureBankv2

After the testing tool inferred the state machine model, the enrichment process of the model starts. One of the sources for model enrichment is the Android activities that can be called for an application. The activities that have been implemented by the banking application and whether or not the activities are callable are depicted in Table 6.2. In total, the application has discovered 5 activities to be executable that possibly divert the application's logic. Moreover, the enrichment process has classified state 3 to be a *login*-state, because of the presence of all user interface utilities that are required for authentication.

| Activity | Callable |
|---|---|
| `LoginActivity` | *yes, launchable* |
| `FilePrefActivity` | *no* |
| `DoLogin` | *no* |
| `PostLogin` | *yes* |
| `WrongLogin` | *no* |
| `DoTransfer` | *yes* |
| `ViewStatement` | *yes* |
| `TrackUserContentProvider` | *no* |
| `MyBroadCastReceiver` | *no* |
| `ChangePassword` | *yes* |

**Tab. 6.2:** Discovered activities for InsecureBankv2

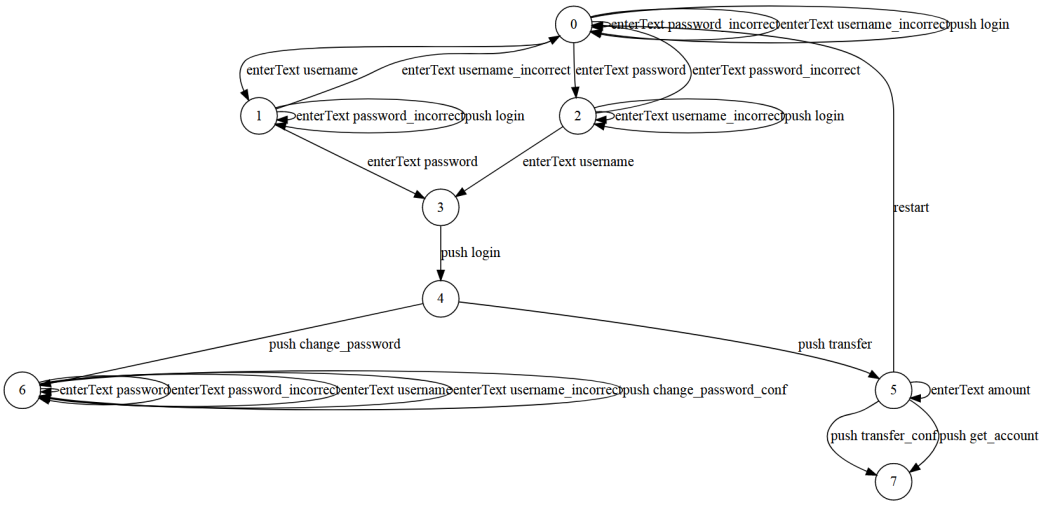



**Fig. 6.2:** State machine model of the InsecureBankv2 application.

After the enrichment process has completed, vulnerability identification can start. The vulnerability identification process returns a report with three found vulnerabilities in the application. This report is depicted in the following vulnerability report 1.

**Vulnerability Report 1** (InsecureBankv2)**.**

- **Improper Platform Usage**
  There were new discovered states for activity `./ViewStatement`
- **Insecure Authentication**
  Authentication can be bypassed for activities `./PostLogin` (to state 4) and `./DoTransfer` (to state 5).
- **Insecure Communication**
  Data is send the back-end unencrypted. Transition from state 3 to 4 requests: `POST http://57.97.2.11:8888/login`.

The three classes of vulnerabilities that have been identified by the tool are thus Improper Platform Usage, Insecure Authentication and Insecure Communication. A simplified version of the inferred model is shown in Figure 6.3. The simplified

model depicts three classes of arrows represented by different colors. Black arrows correlate to normal application behavior. These transitions have been inferred before the model enrichment process starts. The red and green colored arrows correspond to paths that can be taken by the invocation of callable activities. The red and green colored paths are emphasized by a different color in the figure because the colored paths represent a sequence of inputs that exploits a vulnerability.

The set of red arrows illustrate a bypass of the authentication state. The tool has classified state 3 as a login state and has determined that the states that are accessible after the login state should be restricted to unauthenticated users. From the inferred model can be derived that state 4 and 5 are restricted states because they are normally only accessible after successfully logging in. The red arrows depict the existence of a path to a restricted state without proper authentication and therefore the presence of insecure authentication is detected as a vulnerability. The second class of arrows that are shown in green represents a path to a state that is unknown to the model. Since it is assumed that the model represents the abstraction of the entire application, new functionality that can be invoked by controls in the platform that are not initiated through the application's user interface, is the result of improperly using the platform controls. Hence the application is vulnerable to improper platform usage. The green arrow illustrates the presence of improper platform usage in Figure 6.3. A callable activity can execute a process in the banking application that has not been discovered while learning the model. In this example, running the callable activity `ViewStatement` invokes new behavior, which leads to a new state. The third and last vulnerability which is identified by the vulnerability detection algorithms is the implementation of an insecure communication channel. One of the detected vulnerabilities is the following network request: `POST http://57.97.2.11:8888/login`. This request corresponds to the login action that is executed when transitioning from state 3 to 4 and sends the login credentials to the backend server for validation. The POST request is performed over the insecure HTTP protocol instead of an encrypted channel and is, therefore, an exemplification of insecure communication. The vulnerability can also be seen in the data that is sent along with the network request. A snippet of the insecure session is shown in Figure 6.4. The red box surrounds the sensitive information that is sent unencrypted to the backend and thus causes the application to be vulnerable to man-in-the-middle attacks.

## 6.2 WhatsApp

The second set of results presented in this report regards the two remaining vulnerability categories from the OWASP Top 10 Mobile: code tampering and extraneous functionality. Both categories describe functionality that should not be present according to a specification model. For that reason, a vulnerability identification algorithm has been developed that compares an inferred model to a reference model, which for example could have been inferred earlier in the process of testing. Functionality that the application under test accommodates in addition to the reference model could be an indication of tampered code or extraneous functionality.

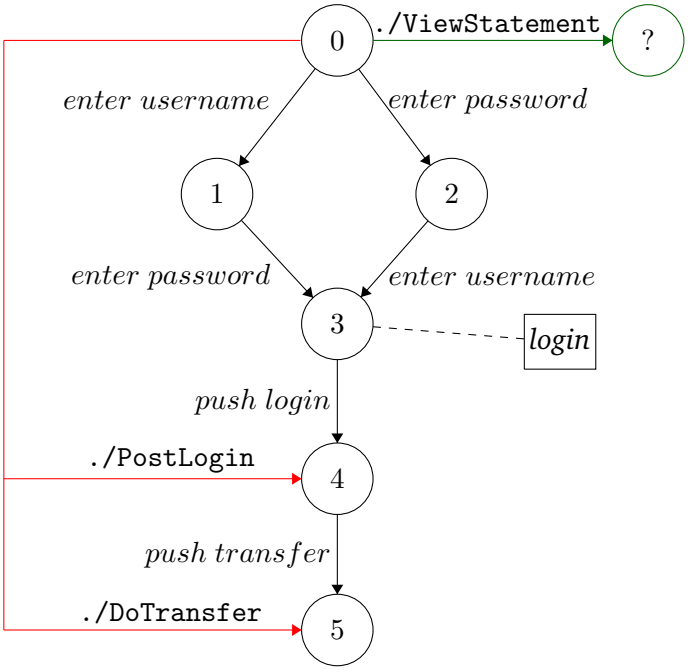



**Fig. 6.3:** Inferred and enriched model of the InsecureBankv2 application

```
7:regular;2:id;36:3e8a2917-f91b-483a-b163-ac0bcb4777d;7:request;508:6:scheme;
4:http,4:port;4:8888#4:host;10:57.97.2.11,12:http_version;8:HTTP/1.1,13:times
tamp_end;18:1512133633.9018605^7:content;29:username=root&password=secret,9:i
s_replay;5:false!15:timestamp_start;18:1512133633.9001265^6:method;4:POST,17:
first_line_format;8:relative;7:headers;208:23:14:Content-Length,2:29,]
```

**Fig. 6.4:** Snippet of the network traffic caused by the InsecureBankv2 application.

The use cases to compare two software implementations are interminable. For example, to show its potential, the vulnerability identification algorithm depicted the difference between a benign version and a malicious version of the popular chat application WhatsApp. The benign version of WhatsApp has been used as a reference model and differences in both models should be the malicious behavior.

On November 2017 a malicious version of WhatsApp was discovered in the Google Play Store that intended to hinder the mobile device with an overdose of advertisements and also tries to trick the user to install additional mobile malware on the mobile device. The malicious application was disguised as a genuine WhatsApp application in the Google Play Store, where the logo and developer title mimicked the benign application's original details, at least so far the naked eye was able to see. The developers of the malicious application (the attackers) were able to do so, by adding a Unicode character space at the end of the developer's name field where the application was advertised. Because of the presence of the additional character, the Google Play Store considered the developers to be a new name for the development team, whereas all Google Play Store users would not be able to spot the difference and are thus convinced of the origin's benevolence. Although the malicious application was only distributed by the Google Play Store for a week, over a million devices, have downloaded the fake version of WhatsApp.

After the state machine model of the benign WhatsApp application has been inferred, the model of the malicious WhatsApp is learned and the vulnerability identification algorithms are executed with the benign model as a reference. The inferred models of the benign and the malicious applications are shown respectively in Figure 6.5 and 6.6.

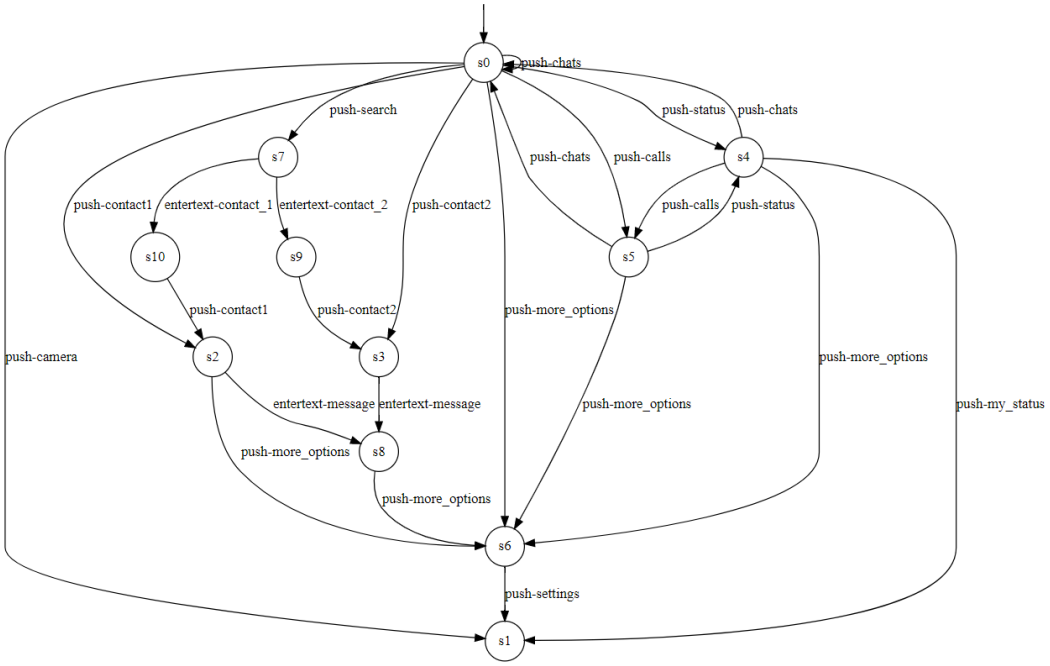



**Fig. 6.5:** State machine model of the benign WhatsApp application.

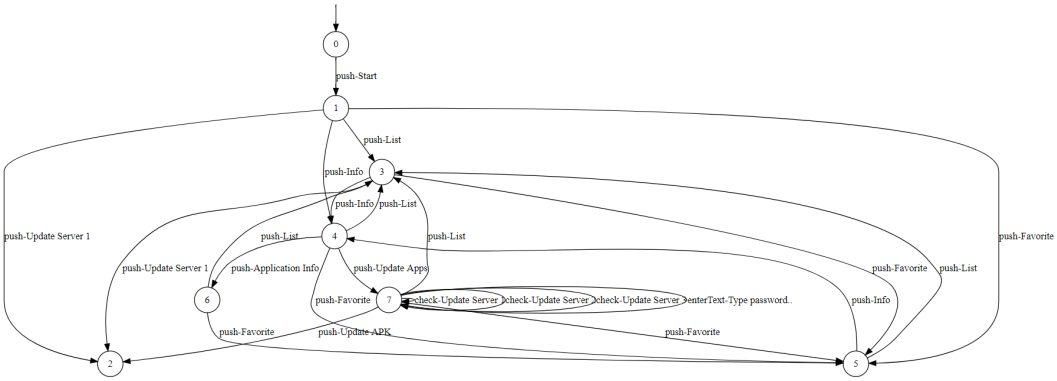



**Fig. 6.6:** State machine model of the malicious WhatsApp application.

After the testing tool inferred the state machine model of the malicious WhatsApp application, the vulnerability identification phase starts. The process returns a report with two found vulnerabilities in the application. Vulnerability report 2 depicts the

| | Fake WhatsApp | Real WhatsApp |
|---|---|---|
| Learning Time | 7:22 | 5:47 |
| Learning Algorithm | TTT | TTT |
| Equivalence Oracle | WMethod-Minimal | WMethod-Minimal |
| Equivalence Queries | 7 | 6 |
| Membership Queries | 29.751 | 25.083 |
| States | 22 | 11 |
| Alphabet Size | 12 | 11 |
| Cache Hit | 71% | 69% |
| Fast Forwarded | 67% | 62% |

**Tab. 6.3:** Learning statistics for both WhatsApp versions

two vulnerabilities.

**Vulnerability Report 2** (Fake-WhatsApp)**.**

- **Extraneous Functionality**
  Extraneous functionality found: `100%` of the functionality in the test application is not represented in the reference model. Moreover `100%` of the functionality in the reference model is not represented in the test application.

- **Insecure Communication**
  Data is send the back-end unencrypted. Transition from state 0 to 1 requests: `GET hxxp://req.startappservice.com/1.4/....` (URL is shortened).

The report shows that two classes of vulnerabilities are present in the malicious WhatsApp version. First of all the tool determined the presence of extraneous functionality when the model of the benign WhatsApp version serves as a reference model. In particular, the tool determines that 100% of the malicious' version functionality does not exist in the benign version. The difference is evident for the reason that the input alphabet is also 100% different. Therefore the set of access sequences for all states are different and no functionality can be simulated on the reference model and vice versa. Because the referenced model does not describe the behavior from the SUT, the application under test exhibits additional functionality, which is characterized as extraneous functionality.

The second vulnerability class that is identified by the tool, is the adoption of insecure communication. The malicious WhatsApp application communicates with several backend servers for retrieving advertisements and storing data about the user and device. All traffic is performed over a plain HTTP connection and is thus unencrypted, instead of an encrypted SSL connection. Although the application communicates for malicious purposes since user and device data is sent to the backend server an SSL connection prevents leakage of sensitive data to possibly additional malicious third parties that perform a man-in-the-middle-attack.

| FreeCola | |
|---|---|
| **Vulnerability** | **Detected** |
| Improper Platform Usage | Yes |
| Insecure Communication | No |
| Insecure Authentication | No |
| Extraneous Functionality | No |

**Tab. 6.4:** Detected vulnerabilities for the FreeCola application

| 9292 | |
|---|---|
| **Vulnerability** | **Detected** |
| Improper Platform Usage | No |
| Insecure Communication | No |
| Insecure Authentication | No |
| Extraneous Functionality | No |

**Tab. 6.5:** Detected vulnerabilities for the 9292 application

## 6.3 Remaining Results

The two discussed applications InsecureBankv2 and the fake Whatsapp show how the proposed testing methodology is able to determine their level of security. The model learning and vulnerability identification algorithms have been executed on various other mobile applications as well. The results of the remaining applications are presented in this section and will be thoroughly discussed in the next chapter.

**FreeCola**
The mobile application FreeCola has been developed to contain security vulnerabilities on purpose as well as the InsecureBankv2 application. KPMG developed the FreeCola application for training purposes. Because we know in advance what type of vulnerabilities are present in the application, the detection phase should identify a number of weaknesses.

Table 6.4 depicts the results of the FreeCola application.

**9292**
The Dutch transportation application that has been used to show the influence of different learning algorithms when learning a model, has also been subjected to the vulnerability identification algorithms. The results of the security assessment are presented in Table 6.5.

# 7 Discussion

*The proposed vulnerability identification algorithms results that presented in the previous chapter. The algorithms examined two different cases of mobile applications, a vulnerable banking application and a malicious chat application that was found in the wild in the Google Play Store.*

Based on the results observed in Chapter 6, we can conclude that vulnerabilities can be discovered from abstracted state machines. However, the observations also raise important questions, such as "*Are all expected vulnerabilities discovered?*" and "*How an incomplete model still lead to the discovery of vulnerabilities?*". This chapter addresses questions alike, by evaluating vulnerability class that has been detected and validating the observed results.

## 7.1 Evaluation

The implemented tool was designed to identify only a subset of all the possible vulnerability classes that exist for mobile applications. Only a subset of the vulnerability classes is discoverable because the abstracted models describe the application behavior on a user interaction level. As a consequence, vulnerabilities that are violated on a nonuser interaction level, such as poor code quality, cannot be detected through the behavioral model. From the OWASP Top 10, we have provided a detection strategy for the following behavior-driven vulnerability classes.

1. Improper Platform Usage
2. Insecure Communication
3. Insecure Authentication
4. Code Tampering and Extraneous Functionality

To analyze the detection performance of the vulnerability identification algorithms, statistical measures can be used as a metric. The detection algorithms solve a binary classification test because a vulnerability can either be detected or not be detected. Because of the binary classification property, the following classification metrics aid the process of reviewing the results. If all vulnerability classes were able to be identified when they should be identified because the vulnerability resides in the application, the type of result is *true positive* (Table 7.1). When a vulnerability should not be detected by the tool because the vulnerability is not present in the mobile application, this type of result is called *true negative* (Table 7.1). A vulnerability can be detected when the vulnerability is not present at all (*false positive*) and a vulnerability can be ignored whereas it is present in the mobile application (*false negative*). The number of true positives and true negatives are two metrics that should be as high as possible opposed to false positives and false

| | | Vulnerability present | |
|---|---|---|---|
| | | Yes | No |
| Vulnerability detected | Yes | True Positive | False Positive |
| | No | False Negative | True Negative |

**Tab. 7.1:** Definitions of True/False Positive and True/False Negative

| | Improper Platform Usage | Insecure Communication | Insecure Authentication | Extraneous Functionality |
|---|---|---|---|---|
| InsecureBank | +/+ | +/+ | +/+ | −/− |
| FreeCola | +/− | −/− | −/+ | −/− |
| 9292 | −/− | −/− | −/− | −/− |
| Benign WhatsApp | −/− | −/− | −/− | −/− |
| Malicious WhatsApp | +/+ | +/+ | −/− | +/+ |

**Tab. 7.2:** The expected results versus the expected results for all applications under test.

negatives respectively because both values resemble the accuracy and the correctness of the tool. The two remaining scenarios depicted in Table 7.1 conform to the level of inefficacy of the detection tool.

This section discusses for each of the vulnerability classes to what extent the results influence the above-defined accuracy terminology. An overview that depicts for each assessed application what vulnerabilities have been detected is presented in Table 7.2. The symbol + or − before a slash sign represents if the specified vulnerability has or has not been detected respectively. The symbol after the slash sign details analogous to the symbol before the slash sign if the vulnerability is present (+) or absent (-) in the mobile application.

### 7.1.1 Improper Platform Usage

The vulnerability class Improper Platform Usage has been correctly detected for the insecure banking application and the malicious WhatsApp application. The vulnerability has been detected because the algorithm encountered a path that infers additional functionality by misusing platform controls. Both paths are driven by executing a callable activity. Moreover, the assessed mobile applications that are well secured and thus do not contain the vulnerability, such as the Dutch public transport travel planner and the malicious WhatsApp application, are not determined to contain the vulnerability. Table 7.2 shows that one of the test applications, the FreeCola application, yields a false positive result for the Improper Platform Usage class. The detection algorithm thus detected a path that invokes behavior that is new to the inferred model. The false positive result can easily be explained because the behavior is, in fact, a violation of the insecure authentication vulnerability that the FreeCola application has implemented. The execution path that invokes new behavior should thus not be new behavior, but be labeled as post-login behavior. The reason why the behavior has not be labeled as such by the identification algorithm

is that the application under test does not allow the action of logging in. As a consequence, the state machine learning tool was not able to infer the states after the login state. Hence the actual post-login behavior is labeled as new behavior, which causes the detection of improper platform usage. If during model inference the state machine would describe the behavior after the login state, the vulnerability would have been classified to the correct class.

The false negative result is less severe because it causes a false positive in another class. If no false positive result would appear for another vulnerability class, the exploitable activities remain undetected. The latter is not the case when testing the FreeCola application. All other instances of the improper platform usage vulnerability were correctly discovered, when present, or disregarded when absent. Hence, this vulnerability class is always discovered when the vulnerability is present in an application or the attack path is labeled to a different vulnerability otherwise. Mislabeling of this vulnerability class happened at the FreeCola application. Although the vulnerability should be classified as insecure authentication, due to the absence of post-login behavior in the model an improper platform usage is identified.

### 7.1.2 Insecure Communication

The Insecure Communication vulnerability has been correctly detected for the InsecureBankv2 application and the malicious WhatsApp application. The unencrypted network requests which have been intercepted by the proxy are proof of the insecure communication. The remaining test applications do not implement this type of vulnerability. The FreeCola application applies certificate pinning for the communication to the back-end server. The 9292 application uses a secured TLS session for communication and the benign WhatsApp application uses a custom protocol for establishment of an encrypted session [40].

Sessions that are encrypted are detectable because the communication cannot be read out as plain text. Sessions that are unencrypted lack a handshake for key exchange and certificates. Secure communication is thus either present or absent and therefore does not generate false results if the inferred model is complete.

### 7.1.3 Insecure Authentication

Insecure Authentication was detected in the InsecureBankv2 as expected, because of the inferred and enriched model (Fig. 6.3) depicts paths that bypass the login state. These results completely coincide with the intended method for detecting paths of exploitation. For all applications where it is known in advance that the Insecure Authentication class should not be detected because it is not present, the vulnerability was not detected. In other words, the Insecure Authentication class does not generate false positive results.

The Insecure Authentication class does accommodate a false negative result for the FreeCola application. The false negative result indicates that the vulnerability should be detected, whereas it was not detected. The vulnerability was not identified because the invoked functionality was defined in the inferred model and was

therefore attributed as proof to the improper platform usage vulnerability class. The vulnerability class to which a path of exploitation is assigned is, in this case, a semantic difference. Although the wrong classification of the exploitation path leads to a false negative and a false positive result, the exploit path will eventually be reported.

The latter depicted scenario is the only instance where a path of exploitation produced false results. The result will, however, be classified as an Improper Platform Usage vulnerability. The classification in itself is not entirely incorrect, as the Insecure Authentication class is, in fact, a subclass of Improper Platform Usage.

### 7.1.4 Code Tampering/Extraneous Functionality

Detection of tampered code or the presence of extraneous functionality is determined by comparison of the functionality of a reference state machine to a new inferred model. The difference in functionality is then assumed to originate from an illegitimate source, such that the functionality is either extraneous or the result of the tampered code. The assumption can also be made for different versions of an application during the development process. In between two releases of an application, additional functionality could be added or removed. Comparing the inferred models of an application that changed over time then is able to verify the change.

The presence of extraneous behavior has been assessed for the malicious version of WhatsApp. The assessment has been done to gain insight knowledge about the application. The developers (attackers) of the malicious application, have put in the effort to advertise the malicious app as a genuine WhatsApp application. From a forensic point of view, it is therefore interesting to know to what extent the attackers have put in an effort to mimic the genuine WhatsApp application. Comparison of both models showed that the malicious application differs regarding all functionality, meaning that the attackers did not put in any effort to mimic the benign application concerning behavior. This observation constitutes with the attack vector the attackers applied. If for instance, their method of operation would be to also implement benign WhatsApp functionality such that the malicious activities are more covert, the malicious behavior would still be detected, because loading and presenting advertisements is not performed by the benign application. The state machine similarity would in this instance be lower than $100\%$ but larger than $0\%$.

## 7.2 Validation

In order to verify the obtained results, the vulnerability detection tool has been applied to applications that contain vulnerabilities which are known in advance. The opposite must be true as well, vulnerabilities that have not been detected by the tool should be absent in the actual application. The latter scenario is verified by applying the tool to applications that do not contain any vulnerabilities. Since it cannot be proven that an application does not contain a vulnerability, several selection criteria and techniques are adapted to approximate the absence or presence of vulnerabilities. These criteria and techniques are as follows:

In the first instance, applications are required that accommodate a vulnerability that should be detectable by the tool. These test cases are also called the positive test set. A positive test set is required to determine if a result is true positive in the case a vulnerability is detected or a false negative in the case a vulnerability is not detected. Three applications under test are used to assess the tool for the mentioned type of vulnerability: InsecureBankv2, FreeCola and the malicious WhatsApp. The first two applications are developed with the intention to accommodate vulnerabilities for the purpose of testing or education. The source of these applications is, therefore, a criterion that guarantees the presence of certain vulnerabilities. The second technique that is applied to verify that the positive test set indeed contains the corresponding vulnerabilities is manual verification. For each exploitable technique the application contains according to its source, I manually exploited the vulnerability to determine its presence. To verify that the malicious WhatsApp application is truly fraudulent, the source of the application was assessed. The Android file for the malicious WhatsApp application was difficult to obtain because the Google Play Store does not publicly host the application file anymore. Eventually, a mobile malware database for threat researchers was discovered to host the malicious application. By comparing the hash value of the application with the threat intelligence literature, it was verified that the retrieved application was, in fact, the malicious WhatsApp application. Apart from the threat intelligence literature, the application was also manually verified to be malicious and concluded to contain extraneous functionality opposed to the benign WhatsApp application.

The second type of test sets are the negative test cases, which are the applications that do not accommodate any of the vulnerabilities that are detectable by the tool. A negative test set is required to determine if a result is false positive in the case a vulnerability is detected or a true negative in the case a vulnerability is not detected. Detection of vulnerabilities is an ongoing and maturing field, hence it is impossible to guarantee the complete absence of vulnerabilities in an application. There are, however, properties of an application that positively contribute to the level of security. The absence of vulnerabilities can be approximated by reviewing the properties. The first property that has been considered is the development team of the application and the community's endorsement. Popular applications that are nationally or even globally recommended tend to have a higher incentive to mitigate vulnerabilities. For this reason, popular applications have been used as the negative test set. Moreover, only popular applications from the Google Play Store have been taken as candidates for the negative test set. The application distribution center also reviews the application for vulnerabilities through Google Play Protect. The protection mechanism can be seen as an additional tool that guarantees the application's safety. To approximate the absence of security vulnerabilities, the applications have been tested manually and automatically by utilization of Android vulnerability scanners. To confirm the absence of vulnerabilities we applied the well-recommended Android vulnerability scanning tools `quixxi`[1] and `ostorlab`[2]. The mentioned testing tools were able to identify the presence of vulnerabilities, therefore confirming our suspicion about the absence of vulnerabilities.

[1] `https://quixxi.com/`
[2] `https://www.ostorlab.co/`

## 7.3 Limitations

The discussed results demonstrate that model inference can be combined to identify vulnerabilities in Android applications successfully. We have also seen that the detection method only detect vulnerabilities that are exploited by misusing the application because the model only depicts user interaction behavior. A limitation of the proposed testing methodology is thus that vulnerabilities are only detectable if behavior drives the vulnerability. Section 5.2.2 thoroughly discussed this limitation is. This section discusses other limitations of the testing framework.

The first limitation of the testing methodology is the uncertainty in a complete abstraction of the application. It is unsure to what extent the inferred model thoroughly describes the entire application. One method to measure the completeness is the percentage of code that is executed when invoking the actions that are depicted in the transition coverage set. Due to the difficulty that arises when computing the code coverage of generic Android applications, a method that uses code coverage to efficiently learn a complete model is presented and discussed as a future work reference in Section 8.2. The proposed framework mitigates this problem by assuming that the input alphabet is correct, which depends on the test initiator. The diversity of the inputs determine what part of the application will be reached. Up until now, all buttons, text fields and checkboxes that are presented by the application are added to the input alphabet. Although more types of actions can be added, we have found those three to fulfill the requirements.

When the inferred models are compared to models that have been actively learned in other studies, such as the model of a printer controller [30] and the model of TLS driver implementations [7], we can determine that the models of Android applications are smaller in size. The models are likely smaller in size, because of two reasons. First, as stated before, the actions that are depicted in the input alphabet are not diverse and of a high level. Secondly, a mobile application has less diverse observable state changes than a printer controller or encryption protocol has, because mobile applications are not designed to be state-full, whereas embedded controllers and protocols are. Future research that also infers models on Android applications should compare their models to the ones we present in this paper. Up until now, no automatically inferred model has been publicly published.

# Conclusion

# 8

> *Logic is not the end of wisdom, it is the beginning . . .*
>
> **— Spock**
> Star Trek VI

The objective of this research is to propose a new testing methodology for mobile applications that infers state machine learning in a time-optimized way and applies vulnerability detection algorithms to the inferred models. We divided the objective into three subgoals, which each were guided by a research question. First of all, we must apply active state machine learning to mobile Android applications. Secondly, to achieve time-optimization, we need to administer methodologies that improve the learning time. Thirdly and lastly, the vulnerability identification algorithms that cope with the inferred models need to be established. This final chapter answers the research questions that accompany the subgoals and reflects on the entire research process. Although we can answer all questions and propose the novel testing methodology, there remains ample opportunity to improve the method. The future work references at the end of this chapter subsequently discussed the improvements.

## 8.1 Reflection on Research Questions

The research was guided by the following main research question: **How can one identify weaknesses in mobile Android applications through feasible behavioral state machine learning?**. The question has been divided intro three subquestions, which each answer the main research question from three point of views.

Before models can be assessed for the presence of vulnerabilities, the models need to be inferred. This requirement gave rise to the first research question: "**How can model learning be extended to apply to mobile Android applications?**" In short, the first question can be answered as follows:

*Model learning can be extended to apply to mobile Android applications by:*

- *Creating an input alphabet of internal actions that reside on the user interaction level.*
- *Connecting LearnLib to an interplay layer Appium that is able to execute commands on the mobile device.*
- *Applying the proposed cache roll-back technique to overcome the mobile application's non-deterministic behavior.*

The following led us to the answer to the first research question. First, a framework is required that applies active learning to Android applications before state machine models of Android applications can be inferred. The framework is developed by extending the state machine learner tool *fsm-learner* proposed by Lampe et al. in a way that a model of any Android application can be inferred. To achieve an approach for learning general Android applications we had to overcome challenges that were introduced by the mobile application domain. We selected an input alphabet that is consistent with actions that reside on the user interaction level. Furthermore, the actions must be mapped to commands that can be executed on the application. Mapping these actions introduced the additional interplay layer Appium that can perform the commands to an application. Another challenge was that normal application usage often exhibits non-deterministic behavior. Non-deterministic behavior cannot be learned by an active learning algorithm that learns a *deterministic* finite automaton and hence this challenge must be overcome. By applying a roll-back technique to the cache, the contradictory observations can be amended.

After a framework that applies active learning to mobile Android applications is conceived, we can explore the two variables of active learning: what learning algorithm to use and what method can best be applied to determine the equivalence between a model and an implementation. At first, the classical L* algorithm in combination with a RandomWalk equivalence oracle was applied as proposed by Lampe et al., but the combination produced a limited model of the application and consumed too much time. The shortcomings gave rise to the second research question: "**How can the feasibility of model learning for of Android applications be improved?**" The second research question can be answered as follows:

*The feasibility of model learning of Android applications can be improved by:*

- *Applying learning algorithms that utilize a redundancy-free data structure to store the observations. The optimized algorithm reduces the query complexity.*
- *Extending the RandomWalk equivalence oracle such that a counterexample is also devised from an application's happy flow as explained in Section 4.1.3.*
- *Replacing the characterizing set of the W-method with a set of minimal separating sequences for all pairs of states as proposed by Smetsers et al.*
- *Simulating the actions from a membership query on an USB-tethered physical mobile device.*
- *Adopting the fast-forward technique for early query termination as proposed in Section 3.3.3.*
- *Caching queries and the corresponding outputs.*

To optimize the time-feasibility of active learning, we observed that the classic L* algorithm contains redundant entries in the observation table that lead to superfluous membership queries. To overcome the redundancy, the TTT algorithm that uses a redundancy-free data structure for storing the observations has been incorporated into the learning framework. Although the TTT algorithm reduces the required learning time, the TTT algorithm depends more than the L* algorithm on the equivalence oracle. The latter was discovered, when the RandomWalk equivalence oracle could not devise a counterexample for a model that has fewer states than the model that

is inferred with the L* algorithm. For that reason, we extended the RandomWalk oracle by also identifying possible counterexamples from an application's happy flow. Although the extension introduces increasing success, as a more complete model can be inferred, the expansion requires prerequisite knowledge about the application under test. The requirement of prerequisite knowledge invalidates the goal to infer models of generic Android applications because we then need to exclude the applications that lack the happy flows.

Our objective is to infer a state machine model that approximates the mobile application as close as possible. To achieve this goal, the thoroughness of the method that determines the quality between a model and its corresponding implementation is of importance. The W-method implements the thoroughness by creating test cases that cover all scenarios that depict the same behavior between a software implementation and a hypothesized model, but the W-method is too exhaustive. Within a reasonable amount of time, we were not able to observe the learning process finish due to the excessive amount ($> 60hours$) of time that is required to run all test cases. We reduced the length of the test cases by replacing the characterizing set that is generated by the W-method with a set of minimal separating sequences for all pairs of states as proposed by Smetsers et al. The combination of the W-method with Smetsers et al.'s research enabled us to infer state machines in a time-optimal approach that was able to finish within a reasonable amount of time. Furthermore, the combination learned the most complete models, i.e., the model that contains the highest number of states and transition.

The last approach that has been applied to optimize the feasibility of state machine modeling of Android applications, was the adoption of techniques that are proposed by Lampe et al. First, the mobile device that hosts the application under test, was chosen to be a USB-tethered physical device opposed to a virtual machine. A physical device responds the fastest to commands. Secondly, the fast-forward technique of test queries was implemented that allows early query termination after the query is determined to be obsolete. Thirdly, a cache was implemented that contains queries and the results. All factors reduced the learning time as much as possible.

After we have developed a framework that allows us to infer state machines from Android applications, the inferred models are examined for the presence of vulnerabilities. The last practice gave rise to the third research question: "**How can the learned model be used to assess the application's security?**" The last research question can be answered as follows:

*The following techniques aid and allow vulnerability detection on state machine models:*

- *Vulnerability classes, such as listed in the OWASP Top 10 Mobile, that can be violated by interacting with the application, can be detected in the behavioral model.*
- *Algorithms that discover paths that violate one of the security classes have been proposed. The discovered paths do not only function as evidence for the existence of a vulnerability, but the paths also give insight into the way the application is vulnerable.*

- *Model enrichment aids the process of vulnerability identification by providing additional information to the abstraction. Because the abstraction represents the entire model, certain properties of the enriched information must hold.*

The strategy that we applied to identify the presence of vulnerabilities is to search for a path in the model that exploits a known security class. Exploring such an attack path has been achieved by first of all reviewing classes of vulnerabilities for the mobile domain from the OWASP Top 10 Mobile. We found that some vulnerability classes cannot be detected from a model that shows the user interaction because exploitable behavior cannot violate the vulnerability. The other classes of vulnerabilities that are violated by applying exploitable behavior can be detected by assessing the behavioral model.

Vulnerability identification is made possible by first of all enriching the state machines with supplementary information of the application that belongs to the specific state or transition. Secondly, vulnerability identification algorithms have been proposed that can discover a path of exploitation from the inferred and enriched models. The paths then pose as evidence for the presence of a vulnerability that belongs to a class defined by OWASP.

Results show that for Android applications the paths of exploitation can be found in all test cases. However, due to the semantic connotation of some classes, the identified paths can be assigned to the wrong class of vulnerabilities. The wrongful assignment has been exemplified for an application that contains an improper authentication vulnerability, but the path of exploitation was assigned as an improper platform usage vulnerability. Because the authentication was able to be bypassed through the inappropriate use of platform functionality, the wrongful classification is only induced by the semantic definition that the improper authentication class can be a subset of the improper platform usage class. Notwithstanding the last result, the path of exploitation was defined and the application has been determined to be vulnerable.

In conclusion, the proposed time-optimized active model inference method on Android applications and the automatic identification of vulnerabilities on these models provide the first step towards a new approach for testing and securing the mobile environment. There remain opportunities to cover additional classes of weaknesses and to improve active learning on mobile applications. Nevertheless, given the results that the proposed methodology can produce, the mobile testing community no longer has an excuse not to thoroughly test mobile applications and make the mobile application landscape a safer place.

## 8.2 Future Work

The presented work can identify vulnerabilities in mobile applications based on their behavioral model. There is, however, ample opportunity to enrich the inferred state machine models and apply additional vulnerability detection techniques.

**Model Enrichment** This thesis demonstrates various techniques to enrich inferred models. Information that is added to a model is, for example, the corresponding Android activities for each state and the generated network traffic for each transition. Because this thesis' scope focused on the generation of models in the first place, the provided proof of concept tool is not able to cover all types of vulnerabilities or data sources that are generated by the application. The limitation, however, does not argue that coverage of multiple data sources is not applicable at all. For example, one of the OWASP Top 10 vulnerabilities for which detection has not been implemented, is the category Insecure Data Storage. This type of vulnerability requires information about file storage. The information that is required to assess file storage can be retrieved in two ways. First of all, one could index the entire file system and observe when changes are made to the file system. The changes can be attributed to specific states in the model and henceforth provide a source of information that can be used to detect the insecure data storage. The second technique that can be used to identify file storage is searching for commands in the application code that create, update or remove files in the system. A specific Android agent, such as Frida [41] can be deployed to observe when commands are executed that change the file system. The timing of the detected file change can be used to enrich the model. Both methods provide a technique to enrich the model with information when data storage occurs, after which an algorithm can be established that determines if the data storage is insecure.

**Model Completeness** Within the extent of this research, we assume that the inferred state machine model is complete, such that the model describes the behavior of the entire application. At this moment, model completeness depends on the completeness of the input alphabet and the quality of the equivalence oracle. One way to measure the accuracy of model completeness is to compute the application's code coverage of all actions depicted in the model. If one can compute the code coverage per action, one can guide the learning process by selection of test cases that generate the most or new code coverage. Moreover, an equivalence algorithm can be established that generates a counterexample from the conditions to reach the uncovered code. This methodology has shown to be effective for the discovery of more reachable states by Smetsers et al. in *Complementing Model Learning with Mutation-Based Fuzzing* [42].

**Platform Extension** Another future work reference could be to apply model learning to Apple applications. This reference requires a change in the way that an alphabet is generated and the method where actions are mapped to the system that runs the tests. Especially the part where actions are mapped to the system that hosts the application are straightforward, as the Appium mapper itself can communicate both to Android system and Apple's iOS platforms. One use case to support both platforms for model learning is to verify consistency between the two applications. To reach the largest target audience, applications are written both for Android and iOS. Because a development team is often specialized in code development for a specific platform, the different applications are often developed separately from each other. Although the applications are separate products, they should depict the same behavior. Comparison of state machine models can identify differences or verify equivalence regarding behavior.

# Bibliography


[1] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. „Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage“. In: *Proceedings of the 13th international conference on Human computer interaction with mobile devices and services*. ACM. 2011, pp. 47–56 (cit. on p. 1).

[2] Lars Lunde Birkeland. *Tesla cars can be stolen by hacking the app*. `https://promon.co/blog/tesla-cars-can-be-stolen-by-hacking-the-app/`. Blog. 2016 (cit. on p. 1).

[3] Steven Arzt Stephan Huber Siegfried Rasthofer. *Extracting All Your Secrets: Vulnerabilities in Android Password Managers*. Presentation. 2017 (cit. on p. 1).

[4] Patrick Afflerbach, Simon Kratzer, Maximilian Röglinger, and Simon Stelzl. „Analyzing the Trade-Off between Traditional and Agile Software Development - A Cost/Risk Perspective“. In: *Wirtschafts informatic* (2017) (cit. on p. 1).

[5] Peter Buxmann, Heiner Diefenbach, and Thomas Hess. *The software industry: economic principles, strategies, perspectives*. Springer Science & Business Media, 2012 (cit. on p. 1).

[6] Mohd Ehmer Khan, Farmeena Khan, et al. „A comparative study of white box, black box and grey box testing techniques“. In: *International Journal of Advanced Computer Sciences and Applications* 3.6 (2012), pp. 12–1 (cit. on p. 1).

[7] Joeri De Ruiter and Erik Poll. „Protocol State Fuzzing of TLS Implementations.“ In: *USENIX Security Symposium*. 2015, pp. 193–206 (cit. on pp. 2, 76).

[8] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. „Inference and abstraction of the biometric passport“. In: *Leveraging Applications of Formal Methods, Verification, and Validation* (2010), pp. 673–686 (cit. on p. 2).

[9] Dana Angluin. „Learning regular sets from queries and counterexamples“. In: *Information and computation* 75.2 (1987), pp. 87–106 (cit. on pp. 2, 9).

[10] T. S. Chow. „Testing Software Design Modeled by Finite-State Machines“. In: *IEEE Transactions on Software Engineering* SE-4.3 (May 1978), pp. 178–187 (cit. on pp. 3, 22, 60).

[11] KQ Lampe, JCM Kraaijeveld, and TD Den Braber. „Mobile Application Security: An assessment of bunq's financial app“. In: *Delft University of Technology Research Repository* (2015) (cit. on pp. 4, 29, 36).

[12] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012 (cit. on pp. 7, 10).

[13] Michael J Kearns and Umesh Virkumar Vazirani. *An introduction to computational learning theory*. MIT press, 1994 (cit. on p. 9).

[14] E Mark Gold. „Complexity of automaton identification from given data“. In: *Information and control* 37.3 (1978), pp. 302–320 (cit. on p. 9).

[15] Malte Isberner, Falk Howar, and Bernhard Steffen. „The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning.“ In: *RV*. 2014, pp. 307–322 (cit. on pp. 14, 15, 21).

[16] Therese Berg and Harald Raffelt. „Model Checking“. In: *Model-Based Testing of Reactive Systems* (2005), pp. 77–84 (cit. on p. 15).

[17] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based testing of reactive systems: advanced lectures*. Vol. 3472. Springer, 2005 (cit. on p. 23).

[18] Arthur Gill et al. *Introduction to the theory of finite-state machines*. McGraw-Hill, 1962 (cit. on p. 23).

[19] Rick Smetsers, Joshua Moerman, and David N Jansen. „Minimal separating sequences for all pairs of states“. In: *International Conference on Language and Automata Theory and Applications*. Vol. 9618. Springer. 2016, pp. 181–193 (cit. on pp. 25, 28, 40).

[20] J.E. Hopcroft. „An n log n algorithm for minimizing states in a finite automaton“. In: *Theory of Machines and Computation*. 1971, pp. 189–196 (cit. on p. 25).

[21] David Lee and Mihalis Yannakakis. „Testing finite-state machines: State identification and verification“. In: *IEEE Transactions on computers* 43.3 (1994), pp. 306–320 (cit. on p. 25).

[22] Harald Raffelt, Bernhard Steffen, and Therese Berg. „Learnlib: A library for automata learning and experimentation“. In: *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*. ACM. 2005, pp. 62–71 (cit. on p. 29).

[23] Malte Isberner, Bernhard Steffen, and Falk Howar. „LearnLib Tutorial - An Open-Source Java Library for Active Automata Learning“. In: *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*. 2015, pp. 358–377 (cit. on p. 29).

[24] Malte Isberner, Falk Howar, and Bernhard Steffen. „The Open-Source LearnLib - A Framework for Active Automata Learning“. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 2015, pp. 487–495 (cit. on p. 29).

[25] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, et al. „libalf: The Automata Learning Framework.“ In: *CAV*. Vol. 10. Springer. 2010, pp. 360–364 (cit. on p. 30).

[26] Mona Erfani Joorabchi and Ali Mesbah. „Reverse engineering iOS mobile applications“. In: *Reverse engineering (wcre), 2012 19th working conference on*. IEEE. 2012, pp. 177–186 (cit. on p. 30).

[27] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. „Next generation learnlib“. In: *Tools and Algorithms for the Construction and Analysis of Systems* (2011), pp. 220–223 (cit. on p. 30).

[28] Manoj Hans. *Appium Essentials*. Packt Publishing Ltd, 2015 (cit. on p. 30).

[29] Dorothy Graham Rex Black Erik Van Veenendaal. *Foundations of Software Testing - ISTQB Certification*. Cengage Learning EMEA, 2012 (cit. on p. 38).

[30] Wouter Smeenk, Joshua Moerman, Frits Vaandrager, and David N Jansen. „Applying automata learning to embedded control software“. In: *International Conference on Formal Engineering Methods*. Springer. 2015, pp. 67–83 (cit. on pp. 40, 76).

[31] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. „Droidfuzzer: Fuzzing the android apps with intent-filter tag“. In: *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. ACM. 2013, p. 68 (cit. on p. 44).

[32] Jerome H Saltzer and Michael D Schroeder. „The protection of information in computer systems“. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308 (cit. on p. 51).

[33] Charles P Pfleeger and Shari Lawrence Pfleeger. *Analyzing computer security: a threat/vulnerability/countermeasure approach*. Prentice Hall Professional, 2012 (cit. on p. 51).

[34] Rossouw Von Solms and Johan Van Niekerk. „From information security to cyber security“. In: *computers & security* 38 (2013), pp. 97–102 (cit. on p. 51).

[35] *About The Open Web Application Security Project*. `https://www.owasp.org/index.php/About_The_Open_Web_Application_Security_Project`. Accessed: 2017-11-11 (cit. on p. 52).

[36] *Mobile Top 10 2016-Top 10*. `https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10`. Accessed: 2017-11-11 (cit. on p. 52).

[37] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. „A Study of Android Application Security.“ In: *USENIX security symposium*. Vol. 2. 2011, p. 2 (cit. on p. 53).

[38] Justin Clarke-Salt. *SQL injection attacks and defense*. Elsevier, 2009 (cit. on p. 53).

[39] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. „Harvesting developer credentials in android apps“. In: *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM. 2015, p. 23 (cit. on p. 54).

[40] Filip Karpisek, Ibrahim Baggili, and Frank Breitinger. „WhatsApp network forensics: Decrypting and understanding the WhatsApp call signaling messages“. In: *Digital Investigation* 15 (2015), pp. 110–118 (cit. on p. 73).

[41] Srinivasa Rao Kotipalli and Mohammed A Imran. *Hacking Android*. Packt Publishing Ltd, 2016 (cit. on p. 81).

[42] Rick Smetsers, Joshua Moerman, Mark Janssen, and Sicco Verwer. „Complementing Model Learning with Mutation-Based Fuzzing“. In: *arXiv preprint arXiv:1611.02429* (2016) (cit. on p. 81).

# List of Figures

# List of Tables