Vulnerability Detection in Mobile Applications Using State Machine Modeling

Wesley van der Lee

November 30, 2017

Delft University of Technology

TUDelft

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 x, 2018

Thesis Committee

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

Wesley van der Lee
Vulnerability Detection in Mobile Applications Using State Machine Modeling
MSc. Thesis, November 30, 2017

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 an ever-more important role in modern society lives. Although the applications are widely adopted, their security is often not guaranteed. State machine learning has proven to be an effective method for vulnerability detection in software implementations and can thus be extended to improve the security of mobile applications. The extension requires a method to learn state machines for mobile applications and the establishment of 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 general mobile applications and neither does there exist a methodology for automatic vulnerability detection on the inferred models. This thesis proposes two solutions to the above-mentioned challenges. First, a framework for inferring a model on general mobile Android applications is presented, that uses active state machine learning algorithms to ensure model correctness and time minimization on the learning process. Secondly, algorithms are designed that utilize the inferred model and determine the presence of vulnerabilities. Both solutions can be combined to provide a new insight into an application's behavior and achieve the goal of vulnerability detection. Moreover, the solution is able to detect rogue applications such as a malicious WhatsApp version in the Android's Play Store, which infected more than 1 million devices in three days.

For family and friends, and friends that are family.

- Wesley van der Lee
Delft, 2018

Contents

Introduction | 1

Draft version: 4, ext. reviews: FSP

Mobile applications play an ever-more important role in modern society lives, they are the gateways to use social media, to perform banking transactions, to schedule trips and much more [7]. The same type of mobile applications are not only limited to run on mobile phones, but can also run on other smart devices such as smart televisions, -watches and -cars. This is in particular true for applications that are designed to run on the operating system Android, since it is the most popular operating system for smart devices[1]. As a result Android applications facilitate a multitude of services for everyday's life. Although the Android platform is well established, the security of its applications is not. This attribution became especially true when last year 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[6]. A few months after, 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 [33]. These examples illustrate that although today's society moves towards a pervasive adoption of mobile applications, the application are insufficiently secured.

The main reason why applications deficit security is because software security in general always has been part of a tradeoff where development methodology and time-to-market play an essential role [3]. An early time-to-market brings an economical strategic advantage for companies because of two reasons [8]. First of all, when a company launches an application as soon as possible inside a niche domain, they have the ability to become market leaders, with the ability of locking 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 [21]. 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 the correct output is given for the correct input.

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 and shows which application input lead to

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

which state and depicts the corresponding application responses. Establishment of a state machine with black-box testing, is exactly 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 a lot of information about the application's logic, and might as such function as a data source for identifying bugs or vulnerabilities in the application as illustrated in the following examples. The identification of bugs or vulnerabilities from inferred state machines has been achieved for a wide range of software systems, such as various TLS driver implementations [11]. The last mentioned research assessed models for the existence of extraneous transitions or states, and concluded that the visual aid of the state machine helped them to identify vulnerabilities in the implementation. Another way to use state machine learning is the to verify if a software implementation meets certain requirements. The requirements can also be modeled as a state machine, which can be compared to the inferred state machine of the implementation. Discrepancies between the two machines can then be further investigated to assess whether the implementation meets its requirements. One could also learn a state machine model for documentation. This has been performed for the chip in the Dutch biometric passport, where it was more time efficient to infer a state machine than having a team of experts establish such a model [1].

Models can be learned from a set of existing traces (*passive learning*) or a set of traces that is generated while learning by interacting with an application (*active learning*). The drawback of passive learning is that the model is incomplete in the variety of existing traces, i.e. when certain application behavior is not described in the set of traces, the inferred model does not describe this behavior either. Active learning overcomes this problem by querying for the information it needs to know. Software systems, and thus also Android applications, can function as a data source to generate the required, because the systems are able to 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. In order 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, which composes the entities 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 asking membership queries and equivalence queries to the teacher. Membership queries ask whether a certain behavior input is recognized by the SUT. The query and answer combination together form a trace, and a hypothesized model can be established after a sufficient amount of traces are generated. The learner then queries an equivalence query to the teacher for the established hypothesis, which determines if the model correctly describes the SUT's input/output behavior. If this is the case, learning halts as at that point a sufficiently correct model has been inferred. If the model incorrectly describes the SUT's behavior, a trace will be provided by the teacher 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. This process is visually depicted in Figure 1.1.

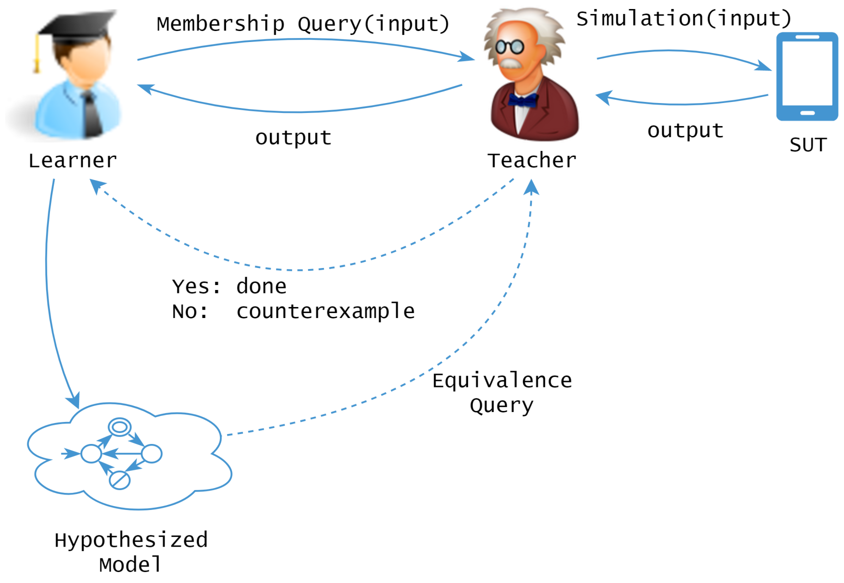


Fig. 1.1: Active Learning with the MAT Framework

A key component of the discussed framework is the teacher's ability to verify equivalence between the hypothesized model and the model implemented by the SUT. Equivalence between an hypothesized model and the SUT is often established through approximation, where the teacher generates a number of test cases and verifies whether the model's output is equal to the SUT. If a given test case yields a different result, the model is inequivalent and the test case itself forms the counterexample. There are numerous algorithms that generate these test cases like randomized input sequences and the W-method [9]. A common drawback of the test case generation algorithms is that the tests are insufficiently diverse or excessively long. Test case generation is a study on its own and this thesis will only touch upon some algorithms to be applied 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 a new data source 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 very limited. There is only one study performed by Lampe et al. that developed a tool for a specific application to infer a state machine model [22]. This model was also manually reviewed, and although the inferred model nor the review results were published, Lampe et al. was able to successfully apply state machine learning to a mobile application. The aforementioned research provides a limited framework for model inference of mobile applications. Due to its limitations, the framework can however be used as a basis for this research to first of all overcome these limitations and therefore utilize the inferred model as a data source for an automated security assessment.

1.1 Outline

This thesis describes the conducted research on how to infer a correct state machine model for a generic mobile Android application and automatically assess its security in an automated way. To infer a correct model one has to review active automata learning algorithms and solve challenges such as the equivalence approximation between a model and an application. Furthermore, algorithms that identify vulnerabilities on the input of an inferred model need to be established. The primary goal of this thesis is to utilize 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 be applicable to mobile Android applications?
This research question mainly focusses on reducing or mitigating the limitations of the framework provided by Lampe et al. [22]. 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 a certain vulnerability on input of the inferred model.

My initial expectation is that model inference of Android applications can be achieved by extending the tool developed by Lampe et al. The tool has not been maintained since publication its publication from 2015, so be able to start the tool with all its dependencies is already a starting requirement. Moreover, my hypothesis is that the inferred model can function as a data source for identifying vulnerabilities, but this requires additional contextual information, such as the type of API calls. 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 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 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 establishes requirements to overcome the identified limitations and proposes a solution framework that is developed based on these requirements. Chapter 5 establishes 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 the results, answers the research questions and provides a perspective ahead on future work references. This thesis closes by the conclusion stated in Chapter 8.

Building Blocks of Model Inference

2

Draft version: 4.1, ext. reviews: SSRS

The previous chapter introduced the goal of this thesis, where it has been 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. In order 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 the learner is guided with an active state machine learning algorithm, the active learning algorithm that first proposed the MAT framework, the L* algorithm, is explained, as well as the 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 establishes the halting condition of learning. There exists 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. In order 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 are able to contribute to the equivalence oracle are examined in Section 2.5.

2.1 Prerequisite terminology

This chapter discusses notorious algorithms for inferring state machines. In order to keep consistency between all discussed work, this section establishes a formal mathematical notation which will be used in the remainder of this research. The notation will be consistent with the notation proposed by Sipser [30].

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 $U = (Q, \Sigma, \epsilon, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \rightarrow Q$ 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.

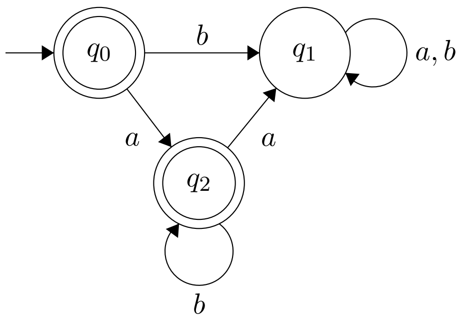


Fig. 2.1: Example DFA $\mathcal{A}$.

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. Transitions from one state to another are indicated by arrows and their corresponding input letter. From this follows that the input alphabet Σ of $\mathcal{A}$ solely consists of the elements a and b.

Σ^* is the set of words over symbols in Σ, including the empty word ε. The *-notation follows from the unary operation, which attaches any number of strings in Σ together: $\Sigma^* = \{x_1, x_2 \ldots, x_{k-1}, x_k | k \geq 0$ 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 Σ^*: $w_2 \notin L(\mathcal{A})$[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'.

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: $\delta(q, w) =$

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

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

$\delta(q_0, w)$ unless specifically specified. Moreover, by also utilizing the notation of Vazirani et al. [20], 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. Accepting states are modeled with a double edged node, whereas rejecting states are modeled with a single edged node. 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 Σ^* 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 λ-function also evaluates if an input sequence is accepted by a DFA. For a word $w \in \Sigma^*$ the λ-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

$Q = \{q_0, q_1, q_2\}$,

$\Sigma = \{a, b\}$,

δ 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 [4] 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 with regular operators, such as union and intersection [30]. 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. The establishment of a conjecture is achieved by posing two types of queries to the MAT:

- **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 typically 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 Σ is added to the prefix, they identify transitions. 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 λ-function: $T(w) \mathrel{\widehat{=}} \lambda(w)$. To maintain consistency with Angluin's work, the function T notation will be used for the remainder of this section. 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 Σ 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 ε as one-letter prefixes. Thus $S = \{\varepsilon\} \cup \Sigma = \{\varepsilon, a, b\}$ The commencing distinguishing suffix is ε, 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 clearly 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 Σ 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.

	ε
ε	1
b	0
a	1

(a)

	ε
ε	1
b	0
a	1
ba	0
bb	0

(b)

	ε	b
ε	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 $\mathcal{A}$.

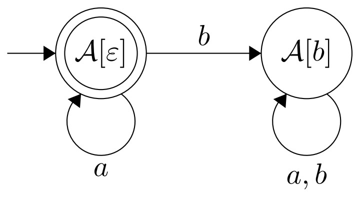


Fig. 2.3: Hypothesized conjecture DFA H_0 corresponding to observation table T_1 (Table 2.2b).

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. $F = \{row(s) | s \in S_H$ 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.

To guarantee that the right suffix is added, the L* algorithm adds the entire counterexample and all its prefixes to S. 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 in order to make the table closed again, new prefixes for the new state $\mathcal{A}[a]$ have to be determined. This 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 correct DFA has been inferred and the learner can stop learning.

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. An 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 δ 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.

A complete overview of the L* algorithm is provided in Algorithm 1.

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

```
1: $S \leftarrow \{\varepsilon\}$
2: $E \leftarrow \{\varepsilon\}$
3: $(S, E, T) \leftarrow MQ$ for $\varepsilon$ and $\Sigma$        ▷ $(S, E, T)$ is the observation table
4: while $H$ is incorrect do        ▷ $H$ is the conjecture
5:     while $(S, E, T)$ is not consistent or not closed do
6:         if $(S, E, T)$ is not consistent then
7:             find $s_1$ and $s_2$ in $S$, $a \in \Sigma$ and $e \in E$ such that:
8:             $row(s_1) = row(s_2)$ and $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$
9:             add $a \cdot e$ to $E$
10:            extend $T$ to $(S \cup S \cdot \Sigma) \cdot E$ using MQ
11:        end if
12:        if $(S, E, T)$ is not closed then
13:            find $s_1$ in $S$ and $a \in \Sigma$ such that:
14:            $row(s_1 \cdot a)$ is different for all $s \in S$
15:            add $s_1 \cdot a$ to $S$
16:            extend $T$ to $(S \cup S \cdot \Sigma) \cdot E$ using $MQs$
17:        end if
18:    end while        ▷ $(S, E, T)$ is closed and consistent
19:    $H = H(S, E, T)$        ▷ $H$ is the conjecture
20:    if $EQ(H)$ is a counter example $t$ then
21:        add $t$ and all its prefixes to $S$
22:        extend $T$ to $(S \cup S \cdot A) \cdot E$ using $MQ$
23:    end if
24: end while
25: 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 exists two access sequences u, u' from the set of prefixes, such that ua and u' reach the same state in H. The latter is true because input uav (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''. This

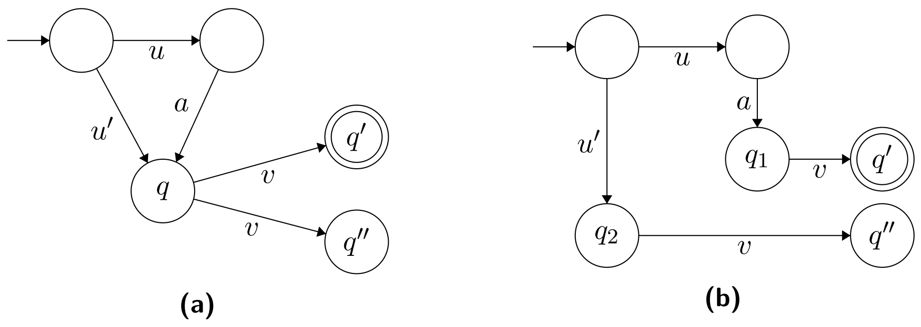


Fig. 2.4: Formal progression of an incorrect conjecture: (a) inconsistent model for distinguishing suffix v from state q, (b) consistent model after splitting q into new states q_1 and q_2.

is noticeable since both states yield a different output. 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 λ. 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* [19] Isberner et al. recognizes 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 contain redundant elements. 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: in the observation table of the employed example DFA shown in Table 2.2c, the value 0 in the ε-column 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.

In order to overcome performance impacts caused by redundancies in counterexamples, Isberner et al. proposes 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 λ-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 $\ell = \bot$ [19, 5].

The discrimination tree is shaped in a way such that words can be *sifted* through the tree. Sifting is required in order to determine the transitions when establishing 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 inner node $v \in E$ of the discrimination tree the sifting process passes one branches to the $\top$- or $\bot$- child depending on the value $\lambda(w \cdot v)$ until a leaf node is reached. The leaf node then indicates 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 exactly 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 in 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 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 in order to infer a correct model:

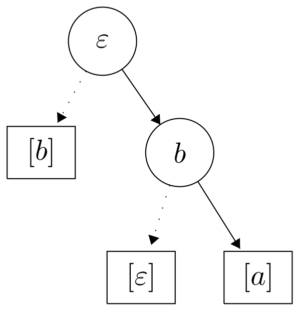


Fig. 2.5: Discrimination Tree DT corresponding to DFA $\mathcal{A}$.

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

1. Hypothesis Construction
The initial discrimination tree is constructed by evaluating $\lambda(\varepsilon)$. This results in either a tree with as root node ε and a single leaf with access string $[\varepsilon]$ on either the $\top$- or $\bot$-child of the root node depending on the λ-evaluation. A conjecture DFA is established where:

1. Q are all the leaf nodes in the discrimination tree,
2. Σ is already known,
3. δ 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 $\lambda(q_0, \varepsilon)$. This results to 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 εa and εb 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 ε-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 ε-root. The initial conjecture DFA H_0 corresponding to DT_0 is depicted in Figure 2.6b.

2. Hypothesis Refinement
The initial hypothesis is likely to be inequivalent to the SUT, in which case a counterexample will be given by the teacher. 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. This breaking point is added as a node in the tree, depending on the evaluation of $\lambda(ua)$ on the $\top$- or $\bot$ branch.

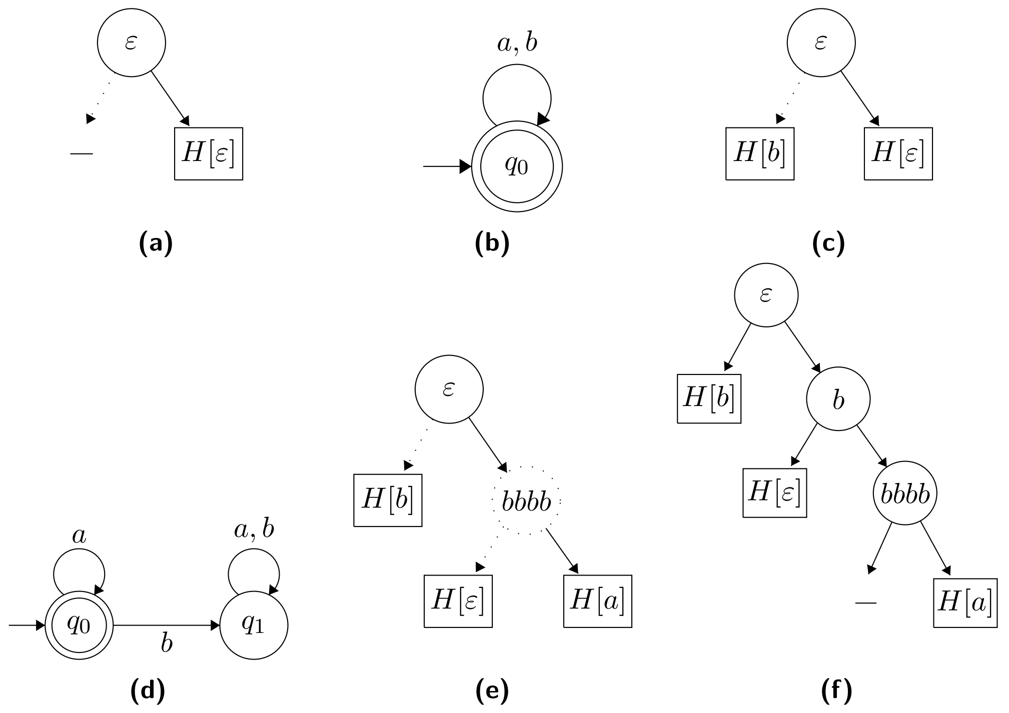


Fig. 2.6: Evolution of discrimination trees and conjectures towards learning the DFA $\mathcal{A}$ with the TTT algorithm: (a): initial discrimination tree DT_0, (b) conjecture DFA H_0 corresponding to DT_0, (c) discrimination tree DT_1 after processing counterexample $w = b$, (d) the conjecture DFA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

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 $[u]_H a \rightarrow [\varepsilon]_H b$, 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 Σ. 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, this 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 $\perp$-child predicts the output 0, but the hypothesized conjecture 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. In order to prevent redundancy, trees that contain temporary nodes must be refined until all discriminators consist of a single element from 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. This yields the result of adding an additional parent node in the discrimination tree above the temporary node, thus shifting the temporary node a level down. 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 $\lambda(H[u]a \cdot v) \neq \lambda^A(H[ua] \cdot v)$ 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 $H[\varepsilon]a$ 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 Σ 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*

a note on speedup.

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. In particular equivalence queries return a positive result indicating that both models are equal or provide a symmetric difference of 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 in terms of determining 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 an 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 counter example 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, Σ 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: $step \leftarrow 0$
 2: $current \leftarrow H$'s initial state
 3: while $steps < maxSteps$ do
 4:     $walk \leftarrow \varepsilon$
 5:     if random then
 6:         increment $step$ by 1
 7:         $w \leftarrow$ random element in $\Sigma$
 8:         append $w$ to $walk$
 9:         if $\delta^H(current, w)$ is not equal to $\delta^{SUT}(current, w)$ then
10:             return $walk$
11:         end if
12:         $current \leftarrow \delta^H(current, w)$                ▷ traverse 1 step
13:     else                                                          ▷ restart the walk
14:         $walk \leftarrow \varepsilon$
15:         $current \leftarrow H$'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* [9] 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 *state cover set* P of inputs and a *characterization set* W of input sequences that can distinguish between the behaviors of every pair of states. A state cover set is a set of input sequences for which hold that it contains an input sequence to reach a state $q \in Q$ for all q. Moreover, the state cover set also includes the empty word ε, since this 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 state 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 terminates if a node has been seen 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 are needed, because the root node at level 1 does not contain a terminating branch.

The characterization set W consists of input sequences that distinguishes 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. [13] 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 corresponding 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 leafs, all that can be reached with inputs a, ba, bb. The state cover set can thus be defined as $P = \{\varepsilon, a, ba, bb\}$.

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 judgement. Since m functions as an

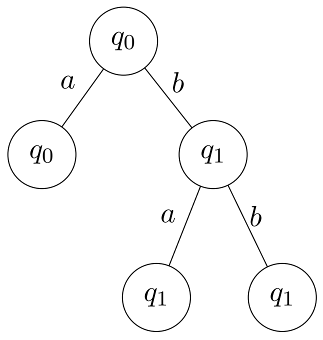


Fig. 2.7: The testing tree conform conjecture DFA H_1.

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 in $P \cdot (\bigcup_{i=0}^{m-n} \Sigma^i \cdot W)$. Chow summarizes $\bigcup_{i=0}^{m-n} \Sigma^i \cdot W$ 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 for the hypothesized conjecture H_1, one should recall that $P = \{\varepsilon, a, ba, bb\}$, $W = \{a\}$ 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 $Z = W \cdot \bigcup_{i=0}^{m-n} \Sigma^i = \{a\} \cdot \bigcup_{i=0}^{2} \{a, b\}^i = \{a\} \cup \{a, b\}^1 \cdot \{a\} \cup \{a, b\}^2 \cdot \{a\} = \{a\} \cup \{a, b\} \cdot \{a\} \cup \{aa, ab, ba, bb\} \cdot \{a\} = \{a\} \cup \{aa, ba\} \cup \{aaa, aba, baa, bba\} = \{a, aa, ba, aaa, aba, baa, bba\}$.

The test suite $P \cdot Z$ created with the W-method is run on both the SUT and the hypothesized model. If the generated test suite is executed in the order depicted above, input sequence aba yields a different result. This input sequence 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 demonstrates the same difference in behavior as the test input sequence aba generated with Chow's method.

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*[32] Smetsers et al. propose an improved modification based on Hopcroft's framework [16] for finding the shortest input sequence that distinguishes two inequivalent states in a DFA. Minimal separating sequences play an 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[23]. 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 in 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 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 with respect to 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 with respect to the succession state after input a. In the latter instance each block that contains a state in $\delta(B, a)$ 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. Smetser et al. redefine the splitting tree, in order to be applicable for the 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 lable 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.

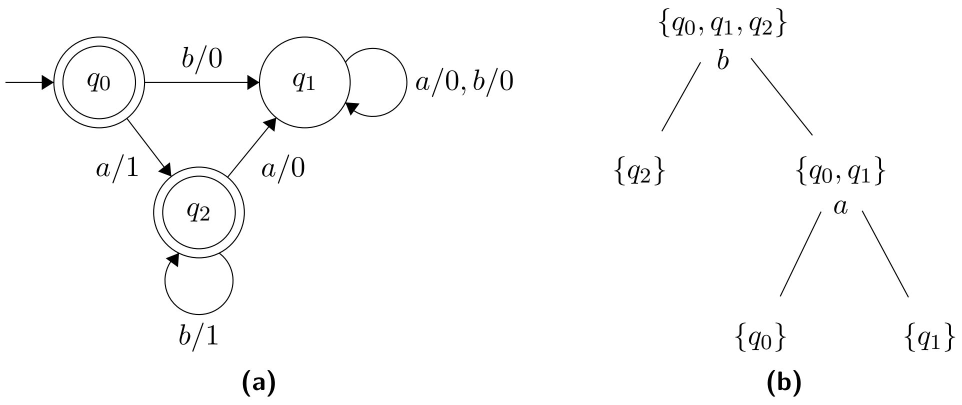


Fig. 2.8: (a): $\mathcal{A}'$ the mealy machine representation of $\mathcal{A}$ (b): splitting tree representation of $\mathcal{A}'$

$C(u)$ 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, iff $P(T)$ 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 $a \cdot \sigma(v)$.

In order 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 with respect to 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 an mealy machine where transitions output 1 if the resulting state is an accepting state and 0 if the resulting state is a rejecting state. This 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 Smetser et al.'s algorithm as follows. The first step is setting the root node to Q, hence the root node is labelled $\{q_0, q_1, q_2\}$. 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.

The algorithm for generating splitting trees can be used to obtain separating sequences, but they are not necessarily minimal separating sequences. This is the case 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

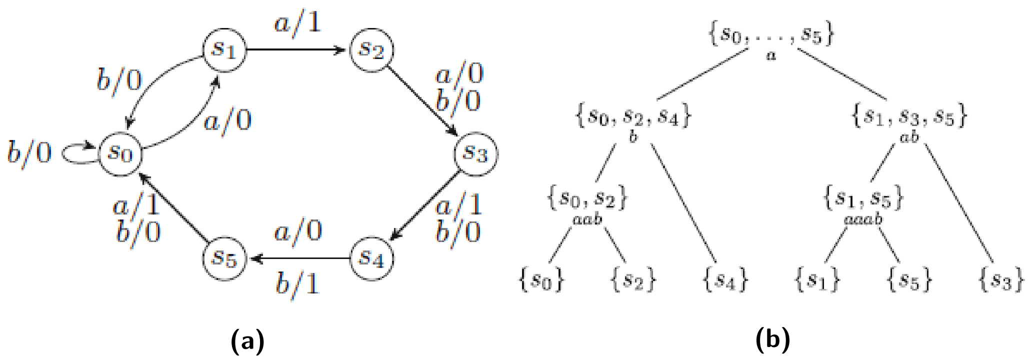


Fig. 2.9: (a): Smetser et al.'s example mealy machine and (b) complete splitting tree for the mealy machine.

example from Figure 2.8b, but it shows in Smetser 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 we have $\lambda(s, x) = \lambda(t, x)$ 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 we have $\lambda(s, x) = \lambda(t, x)$ 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 with respect to 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 w.r.t. the state after b, thus the associated sequence is ba. If this is continued for all levels and all blocks, the splitting tree and partition are identical to the former splitting tree, except that the labels are shorter.

In conclusion, following this recipe for the establishment of k-stable and minimal splitting trees, result 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.

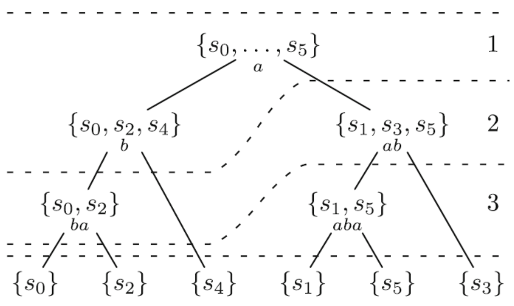


Fig. 2.10: Minimal splitting tree for Smetsers mealy machine

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 $\lambda(s, w) \neq \lambda(t, w)$ ([32], 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

Draft version: 1.1, ext. reviews: -

Part of this research is to infer a state machine model on a mobile Android application. Although prior work on the model inference of a mobile application is very limited, there is one research performed by Lampe et al. [22] that successfully established a framework for state machine learning: the fsm-learner. Their research functions as a building block for the eventual model inference system which will be discussed in Chapter 4. Lampe et al. implemented the MAT framework as proposed by Angluin, by integration of a library for automata learning and experimentation: LearnLib [27]. The tool build by Lampe et al. was designed to specifically infer the model for a single banking application: the Dutch bunq bank mobile Android application[1]. Because the tool was fully tailored to bunq's application, the tool must be extended to be working on general applications. This chapter reviews the discussed tool on functionality and design choices. Section 3.1 examines the LearnLib library, its capabilities and why it has been chosen to be implemented. 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 discussion on what fundamental components can be utilized in the final design for generic application modelling and what components must be changed.

3.1 LearnLib

The library for state machine learning, LearnLib, is a popular library for numerous reasons. The library supports user-configured learning scenarios and, is modularly designed and contains many learning algorithms and as of 2015 it became open source [17, 18]. LearnLib features many active learning algorithms among which the previously discussed L* (Section 2.2) and the TTT algorithm (Section **??**). It also provides a number of conformance testing methods, such as the W-method and modifications of this method. The developers of LearnLib argue that the cheapest and fastest way of approximating equivalence queries is a series of arbitrary sequences that should be true, hence RandomWalk is implemented as well. A good prospect of LearnLib is that it is actively maintained and becoming more popular to be implemented for research[2]. This results in a larger community that is able to support LearnLib. There exists also other libraries that aid the process of state machine learning, such as LibAlf [**bollig2010libalf**] and iCRAWLER [**joorabchi2012reverse**], which have also been reviewed by Lampe et al., but LearnLib was decided to be the most promising library due to the active community.

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

[2]A growing number of citations for LearnLib (Google Scholar): 2003-2007: 39 citations, 2008-2012: 112 citations, 2013-2017: 169 citations

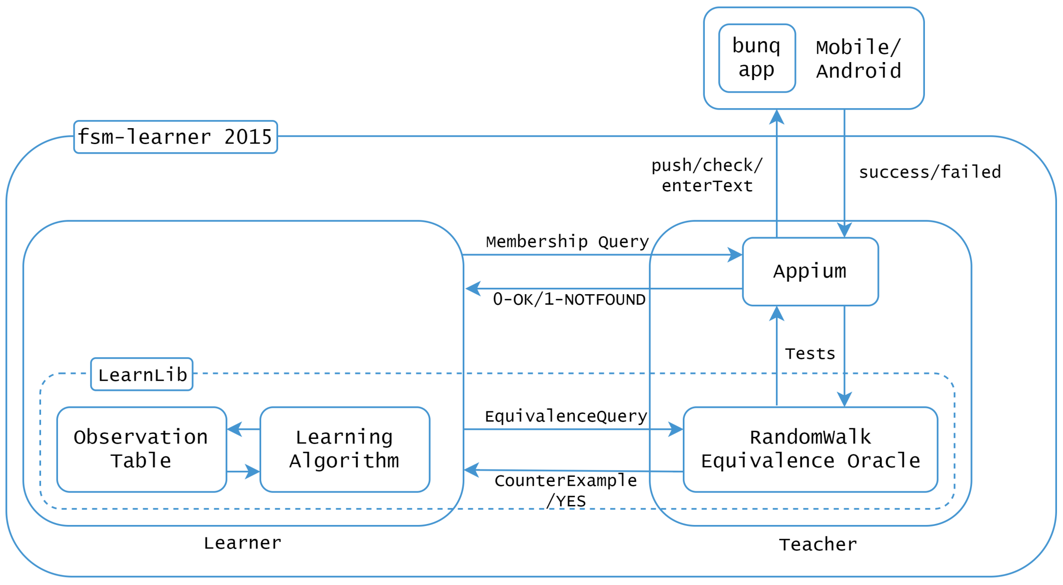


Fig. 3.1: A component overview of the fsm-learner's implementation of the MAT framework.

3.2 The MAT Framework Implementation

LearnLib is not capable of inferring a state machine model on its own, to achieve the inference LearnLib needs to communicate with a system under test. Steffen et al. explain that the main obstacle of active learning is the implementation of the idealized form of interrogation in terms of membership and equivalence queries [24]. In order to apply LearnLib to the Android application domain, an interplay layer for direct communication was needed. Lampe et al. chose to use Appium [15] as a layer to communicate with the USB tethered mobile device. Their main reason to utilize Appium was because Appium is actively supported and supports both interaction 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 is able to answer membership- and equivalence queries. Membership queries are determined through the Appium mapper, which simulates the queries on the real USB thethered device. The teacher can answer equivalence queries by utilization of an equivalence oracle. The oracle that has been implemented for the fsm-learner is a RandomWalk oracle. This type of oracle executes a random sequence of steps on both the SUT and the hypothesized conjecture, and tests whether the conjecture returns the same output as the SUT.

An overview of how the MAT framework is implemented is depicted in Figure 3.1. This figure shows the interaction between the learner and different components of the teacher. The SUT is present as a third component, which in this instance is the android application installed on a USB connected mobile device. The teacher interacts with the SUT through the Appium mapper. The component overview also shows how LearnLib is intertwined in the fsm-learner: several sub-entities of the learner and teacher, such as the 'Learning Algorithms', 'Observation Table' and 'RandomWalk Equivalence Oracle' are part of the LearnLib package.

This chapter assumes that the generic L* algorithm is utilized 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 TTT algorithms 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. This also assumes that the L* algorithm is used.

3.3 Learning

This section elaborates on the entire learning process through the MAT implementation. At first one must establish 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 applied by the developers of fsm-learner to enhance the time feasibility of learning. This is discussed at the end of this section.

3.3.1 Alphabet Establishment

Before the learning process starts, an alphabet must be predefined. This is achieved by manually traversing the application, where buttons and text fields are collected. These buttons and text fields are mapped to the following user actions:

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

A symbol in this alphabet, i.e. a single action, is of the following format `action%param`$_1$`#param`$_2$`#...#p` 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. E.g. `push%xpath(button1)` simulates a push action on the button `button1`. The enterText action requires an additional parameter to specify the text that should be inserted. The action symbol that enters the text 'active learning' in text field `field1` looks for example like `enterText%xpath(field1)#active learning`.

3.3.2 Learning Steps

After the input alphabet is established, the component learning algorithm 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 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 this conjecture to the equivalence oracle.

   a) The Equivalence Oracle generates a test set (a set of input sequences) according to its type.
   b) The test set is run on both the conjecture and the mobile application.
   c) If one test yields an inequivalent result between the conjecture and the mobile application, the test is returned as this is a counterexample to invalidate 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, this is added to the observation table, causing the observation table again to be open or inconsistent. 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, 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 effective, it might be too preposterous for certain 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 they have adopted to enhance time feasibility is called the ’fast-forwarding’ of obsolete queries. This 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 continuance 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. Situation (b) shows that action 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 found for the specific state. The simulation will just move on with 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.

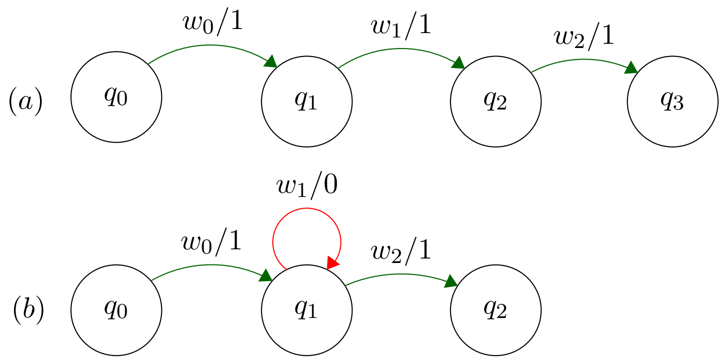


Fig. 3.2: Fast forwarding of word $w = w_0w_1w_2$

The third technique that has been implemented for enhancing time feasibility of learning is the adoption of a cache. This 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 with respect to 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 practicle 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 acedemic 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 thethered 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 active state machine learning and an initial approach that applies the framework on a single mobile application. In order to apply model learning to generic 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 modifications that improve model inference on Android applications with respect to two perspectives. The first perspective considers the problem that the tool fsm-learner cannot infer a model from an application other than the bunq application it was designed for. As a result, this perspective provides insight to a solution that is able to infer a correct model for a generic Android application. The insight composes an answer to the first research question: in what way model learning can be applied to generic mobile Android applications. The second perspective comprehends a proposition to advance the time-feasability of learning, by applying active learning algorithms with a lower time complexity and various other techniques that induces algorithmic speedup. This perspective embodies an answer for the second research question on how the feasibility can be improved of model learning on Android applications. This chapter discusses the work that has been performed to establish the two mentioned perspectives.

4.1 Android Application Model Inference

The previous chapter stated all technical deficiencies in the source code of the tool. After these deficiencies have been assuaged[1], the tool can be improved and extended for mobile learning. In order to correctly infer a model, multiple free mobile Android applications are used as the system under test. One application that has been prominently used at this phase is the Dutch application for public transportation journey scheduling: 9292 [2]. Details on the 9292 application such as the alphabet composition are detailed in Appendix A. At this point the size of the input alphabet Σ is 14. This section discusses how various learning algorithms and equivalence oracles are utilized to infer a model as complete as possible.

[1] a track record of how the deficiencies are assuaged can be found here: `https://github.com/wesleyvanderlee/Thesis/blob/master/Literature/BSc.FSMLearner/Modifications.md`

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

4.1.1 L* and RandomWalk

The tool created by Lampe et al. 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. This indicates that the first time the observation table became closed and consistent, the RandomWalk algorithm was not able to find a counterexample and 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 observation table, because the figure only illustrates positive results. As discussed in the previous chapter, the fsm-learner returns two types of queries: `0-OK` for actions that are allowed for a certain state and `1-NOTFOUND` for actions that are not allowed for a certain state. The `1-NOTFOUND` value can be for example returned when an action depicts the push-action on a certain element that is not present in the 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 n outgoing transitions for each state, where n is equal to the size of the input alphabet. To remain a clear overview, those transitions have been filtered.

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 Machine for the 9292 application using L* and RandomWalk

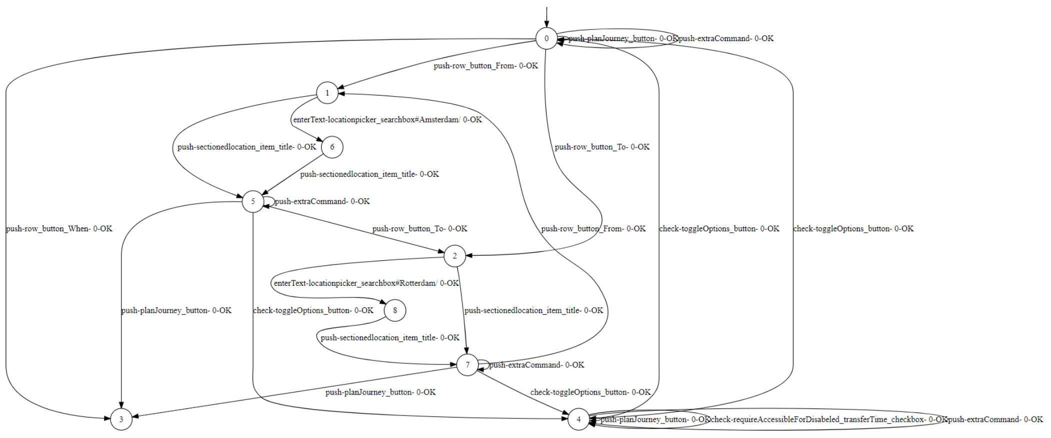

0
push-planJourney_button- 0-OK
push-extraCommand- 0-OK
push-row_button_From- 0-OK
1
enterText-locationpicker_searchbox#Amsterdam/ 0-OK
push-sectionedlocation_item_title- 0-OK
6
push-sectionedlocation_item_title- 0-OK
push-row_button_To- 0-OK
5
push-extraCommand- 0-OK
push-row_button_When- 0-OK
push-row_button_To- 0-OK
push-row_button_From- 0-OK
check-toggleOptions_button- 0-OK
check-toggleOptions_button- 0-OK
2
enterText-locationpicker_searchbox#Rotterdam/ 0-OK
push-planJourney_button- 0-OK
check-toggleOptions_button- 0-OK
8
push-sectionedlocation_item_title- 0-OK
push-sectionedlocation_item_title- 0-OK
7
push-extraCommand- 0-OK
push-planJourney_button- 0-OK
check-toggleOptions_button- 0-OK
3
4
push-planJourney_button- 0-OK
check-requireAccessibleForDisabeled_transferTime_checkbox- 0-OK
push-extraCommand- 0-OK

Fig. 4.1: The Inferred Machine for the 9292 application using L* and RandomWalk

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 result 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, that uses a discrimination tree to overcome the problem of redundant data entries. The TTT algorithm can be started as easily by instantiating another class from LearnLib, however Lampe et al. argue that the TTT algorithm does not work in the fsm-learner, as the inferred model is nonequivalent to the SUT's behavior. Inferring a model with the TTT algorithm yields the model 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 L* and RandomWalk

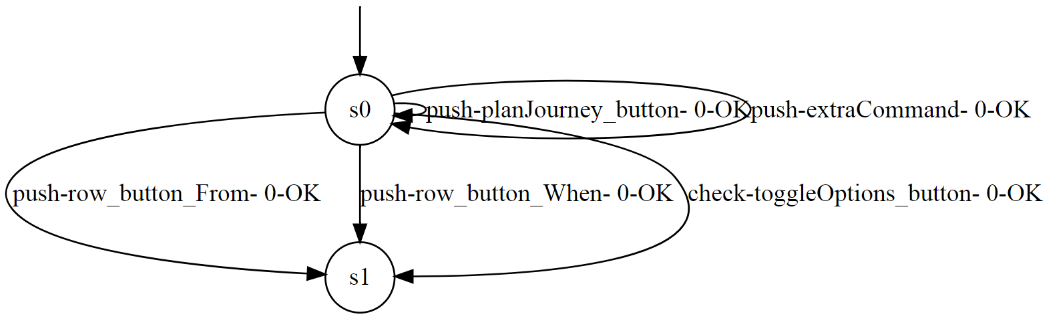

s0
push-planJourney_button- 0-OK
push-extraCommand- 0-OK
push-row_button_From- 0-OK
push-row_button_When- 0-OK
check-toggleOptions_button- 0-OK
s1

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

The inferred DFA in Figure 4.2 obviously yields an incorrect result as the model does not depict the behavior from the 9292 application at all. 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 because 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, it would return a counterexample, the inferred model depicts more precise 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 in order to advance the inferred model.

4.1.3 TTT and RandomWalk-HappyFlow

One method to expand the test cases, is to test for a pre-defined input sequence that is guaranteed to be accepted by the SUT. This is for example a happy flow of the application. Software applications are often developed with domain specific use cases as a functional requirement [28]. These use cases describe how a software system should respond and are thus accepted y the SUT, thus 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 all steps up until $|w|$ are legally executed as well. The RandomWalk equivalence oracle has been extended, dubbed as RandomWalk-HappyFlow, such that it was able to search for a counterexample in the SUT's happy flow, when it otherwise would say 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 calls the original RandomWalk algorithm for finding a counterexample. When this results in the value `null`, learning would normally stop. At this point, the extended part of the algorithm engages, by searching for a counterexample for each subsequence of each happy flow word that includes the first element of the word. This procedure has the following advantages regarding the speedup of identifying a counterexample:

1. Because happy flow words and the corresponding subsequences are computed only on the hypothesized conjecture, no simulation on the SUT is needed. Hence identifying a valid counterexample from a set of happy flows can be done almost instantly.
2. The test words are generated from the smallest subsequence to the largest subsequence. If a subsequence $w = w_0 \dots w_n$ is accepted by the hypothesis

but the subsequence $w \cdot w_{n+1}$ is not accepted 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 possilble, 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 detriment of the procedure 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, differs per application and must thus be collected and processed for each application as a test individually. This requires knowledge about the application before learning starts and thus surpasses the black box testing methodology and is in conclusion not the most optimal solution. To summarize, the extended RandomWalk has proven to be effective 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 for discovering 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 Σ^*, and M equals to Σ^{m-n}, where m is an upper-bound estimation of the maximum number of states the conjecture should have and 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, one can state that the conjecture and the SUT must be 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. Both negatively influence the speedup. After multiple test runs with the W method equivalence oracle, no test run could completely 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. defines a novel heuristic for randomly selecting the middle part M by manually establishing sub-alphabets [31]. Multiple smaller alphabets strongly reduce the size of the middle part M. In order to reduce the test queries and still be able to infer a correct model, Smeenk et al. argues that these sub-alphabets are required to be manually crafted, that is 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 are 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 [32]. The latter method has been thoroughly discussed in Section 2.5.3 where it was proposed to replace Chow's method for establishing a shorter characterizing set. As a result, this process would not only reduce the size of the characterizing set, but can also abolish superfluous 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 outcome 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 [32])
 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. Then the algorithm creates a stable splitting tree according to the procedure of Smetsers et al. Next, the characterizing set is created by adding all distinct labels between each pair of states. The algorithm then proceeds as the normal W method would do. The results of WMethod-Minimal are depicted in Table 4.3 and Figure 4.3. Reviewing these results show 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 additional counterexample opposed to the RandomWalk oracle. This

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

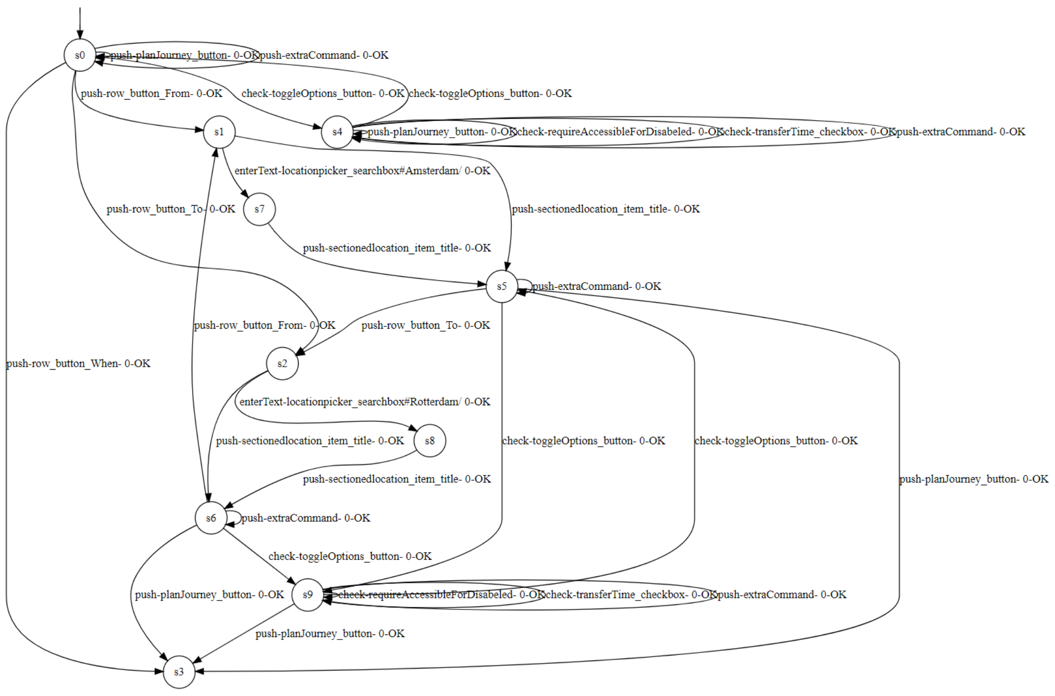

s0
push-planJourney_button- 0-OK
push-extraCommand- 0-OK
push-row_button_From- 0-OK
check-toggleOptions_button- 0-OK
check-toggleOptions_button- 0-OK
s1
s4
push-planJourney_button- 0-OK
check-requireAccessibleForDisabeled- 0-OK
check-transferTime_checkbox- 0-OK
push-extraCommand- 0-OK
enterText-locationpicker_searchbox#Amsterdam/ 0-OK
push-row_button_To- 0-OK
s7
push-sectionedlocation_item_title- 0-OK
push-sectionedlocation_item_title- 0-OK
s5
push-extraCommand- 0-OK
push-row_button_From- 0-OK
push-row_button_To- 0-OK
push-row_button_When- 0-OK
s2
enterText-locationpicker_searchbox#Rotterdam/ 0-OK
push-sectionedlocation_item_title- 0-OK
s8
check-toggleOptions_button- 0-OK
check-toggleOptions_button- 0-OK
push-sectionedlocation_item_title- 0-OK
push-planJourney_button- 0-OK
s6
push-extraCommand- 0-OK
check-toggleOptions_button- 0-OK
push-planJourney_button- 0-OK
s9
check-requireAccessibleForDisabeled- 0-OK
check-transferTime_checkbox- 0-OK
push-extraCommand- 0-OK
push-planJourney_button- 0-OK
s3

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

4.2 Mobile Variables

This section discusses mobile variables.

4.2.1 Non-deterministic Application Behavior

The behavior of mobile applications depend on a set of environmental influences. The most recognizing

In the first place, one must note that the input alphabet is very limited. The alphabet consists of the actions `push` for pushing buttons, `check` for toggling checkboxes and radiobuttons and `enterText` to insert text into text areas. Other actions that the application or mobile device is able to execute 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 fuzzy testing, as performed by Ye et al. [35]. This leads us to believe that an application performs different behavior for different intents. If the goal is to infer a state machine model that is as complete as possible, one should thus also include actions in an alphabet that induces the difference in behavior.

5 Vulnerability Indentification on Models

Draft version: 0, ext. reviews: -

The model that has been inferred so far shows how logical components connect together. They form a logical skeleton of the mobile application. The prospect is to use this skeleton as a guide to eventually determine the presence of certain vulnerabilities. In order to be able to achieve the identification of vulnerabilities in an application, the inferred skeleton 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 based on the enriched model if the original application accommodates vulnerabilities.

5.1 Model Enrichment

In order to determine the presence of vulnerabilities through identification algorithms, the inferred graph first needs to be enhanced with supplementary information. This is achieved by adding labels to the transitions and vertexes that provide information on a current action and a current state. An example label could for example be the network requests to an external API that are made for certain actions. The process of establishing those labels and adding them to the model is called graph enrichment.

Graph enrichment has been separated from the vulnerability identification part, such that one can maintain a clear overview on the algorithms. Else the algorithms would all be cluttered with multiple model instances and in some cases even multiple SUT instances. Another reason to separate the two processes is to reduce time, because the entire graph 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 described in the model need to be simulated in order to retrieve relevant information per state. While traversing the learned model $G = (V, E)$, all vertexes and edges are traversed to retrieve the following information:

Vertexes

- Text on screen per state
  Used to identify *error* states and *login* states.

- Activity corresponding to the state

Edges

- Network requests

5.1.1 Text

The state of a mobile application yields a graphical response to the end user. Text that is depicted on the screen has a meaning that most of the time is application domain specific and not relevant from a security perspective. However, sometimes text has a semantic meaning that concise in the application security domain. Examples are key words such as 'error' or 'fault' to identify a state that correlates to an error state. The text can be carved by traversing over all visible elements on the screen.

Limitation

Not all the text can be carved 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 a system view containing a quick and small message for the user that disappear after a small amount of seconds. The Appium driver relies on the UI Automator framework for interaction with the device. A limitation in UI Automator, and therefore in Appium as well, toast messages cannot be carved, 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 graph enrichment.

5.1.2 Activity per state

An activity is an overarching mechanism that handles an application process. An activity 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 launches the initial main 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. This is often the case for processes such as authentication verification and retrieving query results. Communication to a back-end server is generally done over the Internet through HTTP methods such as GET and POST requests. Each action depicted in the graph could possibly generate a request, but not all actions require a back-end request. It is thus very likely that one part of the transitions are labeled with network requests and the other part is not. 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 is able to execute commands and retrieve trivial system information, but it cannot read out the requests that are made from the device to the Internet. To be able to read the requests, they have to be intercepted by a proxy. A proxy acts as an intermediary for network traffic from the mobile application and the back-end server. The mobile phone connects to the proxy and pushes requests as if it were the Internet. The proxy then forwards the requests to the Internet and returns the result, but also logs the requests.

A very common 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 category of attacks where data interception is the main purpose. For network requests send over HTTP, intercepting readable data is as simple as reading out all the requests. For network requests send over HTTPS, data interception becomes much harder, since the connection between the device and the server is encrypted.

HTTPS Traffic

The goal of the proxy is to sit in the middle of the data stream between the mobile device and the server and be able to read the data in an understandable way. HTTPS traffic prevents exactly that by providing an end-to-end encrypted session, meaning that only the mobile device and the server are able to decrypt the traffic. The 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. Hence it would become impossible to understand the network traffic if it is send over HTTPS.

To overcome the stated problem, mitmproxy needs to become a trusted CA on the mobile device itself. This 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 this certificate as trusted, because mitmproxy's CA certificate is installed on the device.

At this point, the proxy performs the device's request on their 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`

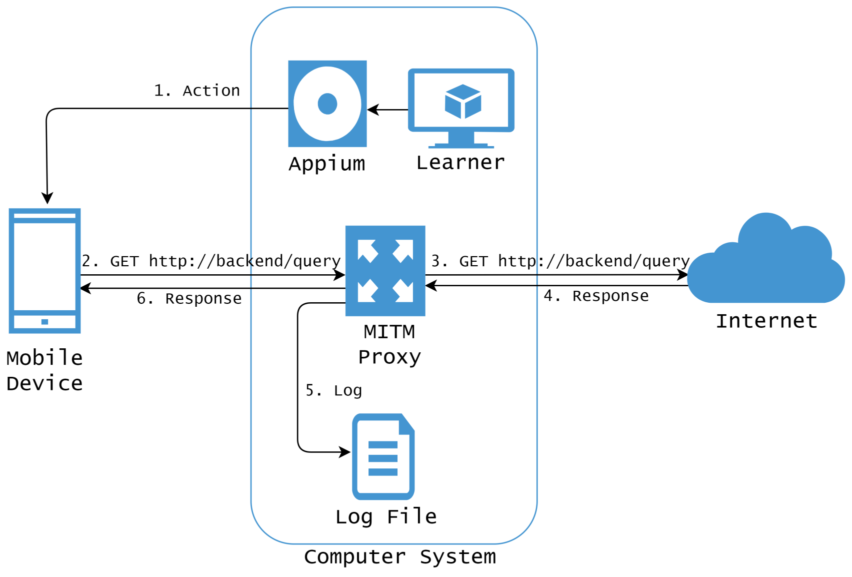


Fig. 5.1: Proxy Setup Scenario 1

Physical Device or Emulator

The proxy can be used in two flavors: on the physical device used for learning or on an emulated device.

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

Users can install new certificates, such as the mitmproxy certificate, on Android as 'user-trusted'. This can only be done if the device meets additional safety requirements such as using a PIN-code or password to unlock the device. Although this appears to be straightforward, it poses as problem as Appium is not able to unlock a PIN/password protected device, even if the PIN/password is known. 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 with an emulator has been proposed. This emulator will be spawned specifically for retrieving the Internet requests from the application. This setup has been visually depicted in Figure 5.2

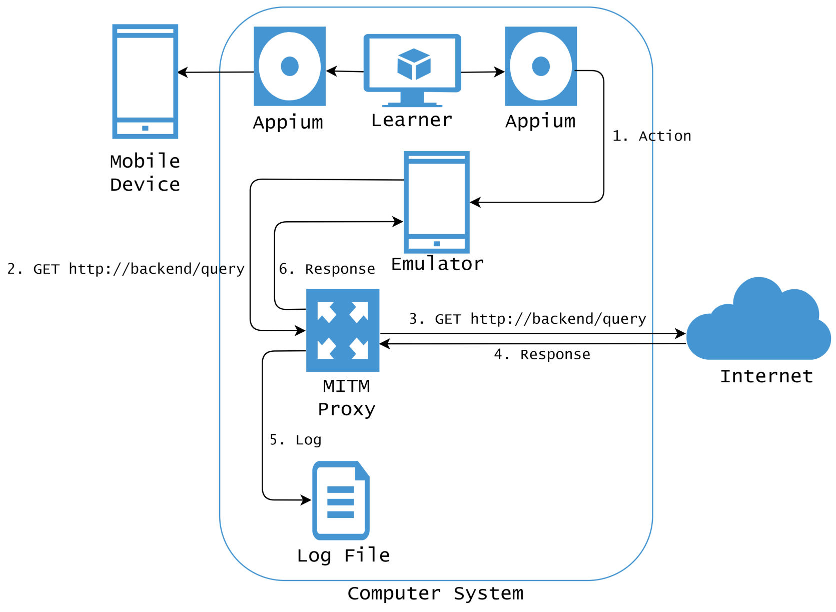


Fig. 5.2: Proxy Setup Scenario 2

5.2 Mobile Application Security

The term *security* describes techniques that control who may use or modify the system or its contained information [29]. Computer security, and as a subset software security, is the protection of items that one values, also known as the *assets* of an application [26]. In order to determine in what ways assets can be defended, one must think about ways to attack them. This can be done by compromising one of the three following properties that together form the CIA triad: *confidentiality*, *integrity*, *availability* [34].

1. **Confidentiality** is the ability of a system to ensure that sensitive data is accessed by authorized parties. 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.
2. **Integrity** is the ability of a system to ensure that an asset is modified only 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 changed by a adverse proxy.
3. **Availability** is the ability of a system to ensure that an asset can be used by any authorized parties 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 Internet and a subscription that has been paid for.

The above mentioned properties can be violated on numerous levels of interaction, such as the levels of the the application client, the application's back-end, other installed applications, the operating system and even the device's hardware. The scope applicable to this research mainly focuses on the application 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 software weaknesses, or the lack of them, is paramount in order to conclude the security properties. The Open Web Application Security Project (OWASP) Foundation [2] 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 [25]. The OWASP lists are widely adopted by the security community in the entire world to group vulnerabilities or weaknesses and provide guidance for identifying them. 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 by OWASP used to set a standard for the most critical application security risks. The contents and order of the list are determined by proactively interviewing security industry experts and extensively reviewing the results in public. Because the list can enable a tester to identify vulnerabilities in a structured way, the items 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 failure to meet the platform security controls or the misuse of general platform features. An example could be the wrong implementation of Android intents. Intents are abstract descriptions of an operation to be performed. 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 [12]. Since secure operations can be invoked from an illegitimate session when this category is not met, the application's confidentiality is at stake. Since 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**
   This 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. This file system can be accessed by other applications or users. 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 confidentiality of data, or changed for malicious purposes, thus compromising integrity.

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). This communication often embodies sensitive data, such as authentication credentials or sensitive files. Sending and receiving of this data should be done on a cryptographically secured communication (TLS) or custom implemented certificates (certificate pinning). Insecure communication can lead to a breach in the security properties confidentiality and integrity, as the data might be accessible to view and change by a third party in between the client and server connection.
4. **Insecure Authentication**
Sometimes a part of the application is shielded for anonymous users and require an authorization scheme. Essential to this scheme is the authentication process, where a user verifies its identity. If this 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 for example a login screen that can be bypassed by a SQL-injection [10]. Through such an attack, an adversary is able to divert the logic of the authentication scheme and appearing with an other identity.
5. **Insufficient Cryptography**
To enable secure data storage, cryptographic fundamentals are needed to store the data in a secure way. This category covers the applications components 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 [36].
6. **Insecure Authorization**
Authorization couples 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 one could inappropriately access, the attacker could misuse the entire CIA triad. For example, an low-level user is able to create a TLS session with the back-end server. The session has 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 it 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 unusual 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 entire 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 in addition the tampered application is able to perform malicious activities. The malicious activities can include excessive advertisement generation, stealing of 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. This is especially easy to achieve 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 an user to perform operations that is not directly exposed via the user interface. For example, during development an application might have a legitimate developers backdoor to bypass initial authentication for easy testing purposes. If at the public release of the application, this logic is not removed from the application, it serves as extraneous functionality. Depending on the operations included in the extraneous functionality, all security properties could be at stake.

5.2.2 Detectable Through Models

The vulnerability categories discussed above are all critical to address during application development in order to conserve the security properties. Some of the categories do not appear in a behavioral model, as they are not triggered by behavior, or are too low level. The inferred model will be on the user interaction level. Some vulnerabilities would materialize after a source code inspection, such as a review on the cryptographic methods that are used. Other vulnerabilities materialize after one checks certain static properties. Inference of the application's behavior does not reveal any insights into the static property of reverse engineering.

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

5.3 Vulnerability Algorithms

The former sections discussed enhancement techniques on inferred models and considers 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**
   Oftentimes when actions are performed that are illegal or cause a failure, this is notified to the end user. Although the action can be legal according to the application's logic, it might not be semantically correct. The notification that is returned is often semantically formed as well. As a result, certain key words in the notification can function as a classifier for the corresponding state being an error state or not. This algorithm 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 Graph $G = (V, E)$
Output: Returns a set $R$ of vertexes that are error states in graph $G$. There
    possibly exists a vulnerability if $R$ is non-empty.
1: $R \leftarrow \emptyset$
2: for all $v \in V$ do
3:     if $v$ is classified as error then
4:         add $v$ 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 this point that destabilize the system.

Algorithm 6 Dead End Identification

Input: Inferred Graph $G = (V, E)$

Output: Returns a set R of vertexes that are dead ends in graph G. There possibly exists a vulnerability if R is non-empty.

```
1: $R \leftarrow$ copy $V$
2: for all $v \in V$ do
3:     for all $e \in E$ do
4:         if $v$ equals $e$.source then
5:             remove $v$ 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 it categorizes misuse of platform security controls that are part of the operating system. This also includes 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 called by an activity is superfluous and should not be accessible by end users. This 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 this login screen, this is seen as improper use of the platform and thus results in a severe application vulnerability.

Algorithm 7 Improper Platform Usage Identification

Input: Inferred Graph $G = (V, E)$

Output: Returns a set R of activities that induce supplementary behavior in graph G. There possibly exists a vulnerability if R is non-empty.

```
1: $R \leftarrow \emptyset$
2: $A \leftarrow$ activities from the SUT
3: for all $a$ in $A$ do
4:     if $a$ is a callable activity then
5:         add $a$ to $R$
6:     end if
7: end for                    ▷ $R$ contains all callable activities
8: for all $v$ in $V$ and $R$ is not empty do
9:     remove $v$'s activity from $R$
10: end for
11: 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, everything in between is not. This embodies 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 data passing through that AP. Security standards of mobile applications should apply at least the use of a secured communication (TLS) and at best exercise *certificate pinning*. Since the insecure communication category has many gradations in itself (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 mostly determined by the function `request_insecure()`. This function returns true if the request is made over HTTP and false if the request is secured through protocols such as TLS and such as certificate pinning. The specification for the method is also depicted in Table 5.2.

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 Graph $G = (V, E)$
Output: Returns a set R of requests that made by actions in G that do not adhere to common network security standards. There possibly exists a vulnerability if R is non-empty.

```
1: $R \leftarrow \emptyset$
2: for all $e$ in $E$ do
3:     $r \leftarrow$ request_insecure(e.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 in 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.

Algorithm 9 Insecure Authentication Identification

```
Input: Inferred Graph $G = (V, E)$
Output: Returns a set $R$ of authentication bypass techniques in graph $G$.
    There possibly exists a vulnerability if $R$ is non-empty.
 1: $a \leftarrow$ authentication state of $G$
 2: $Marks \leftarrow$ subset of nodes possible to reach after $auth$
 3: $G' \leftarrow G - a$                ▷ the graph without the authentication state
 4: for all $m$ in $Marks$ do
 5:     if a path from the root to $m$ exists in $G'$ then
 6:         add $path$ to $R$
 7:     end if
 8: end for
 9: $V' \leftarrow V - Marks$
10: $A \leftarrow$ callable activities
11: for all $v$ in $V'$ and $A$ is not empty do
12:     remove $v$'s activity from $A$
13: end for
14: add $A$ to $R$
15: 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 code is often changed such that a benign looking application will perform malicious activities. To that extent, the application can be modified to collect data which is sold on the black market (*spyware*), perform activities on that phone so it will generate a 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 comparing the inferred model to a reference model of an application. This 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 then is the result of a tampered application, this algorithm will identify the difference between the two of them.

Algorithm 10 Code Tampering Identification

```
Input: Inferred Graph $G = (V, E)$ and reference graph $G' = (V', E')$
Output: Finds difference in $G$ and $G'$ Returns a set $R$ of Vertex and Edges of
    graph $G$ that do not appear in reference graph $G'$. There possibly exists a
    vulnerability if $R$ is non-empty.
 1: $R \leftarrow \emptyset$
 2: for all $v$ in $V$ do
 3:     $v' \leftarrow V'$.getState($v$.property)
 4:     if $v$ is not equal to $v'$ then
 5:         add $v$ to $R$
 6:     end if
 7: end for
 8: for all $e$ in $E$ do
 9:     $e' \leftarrow E'$.getState($e$.property)
10:     if $e$ is not equal to $e'$ then
11:         add $e$ to $R$
12:     end if
13: end for
14: return $R$
```

6 Results

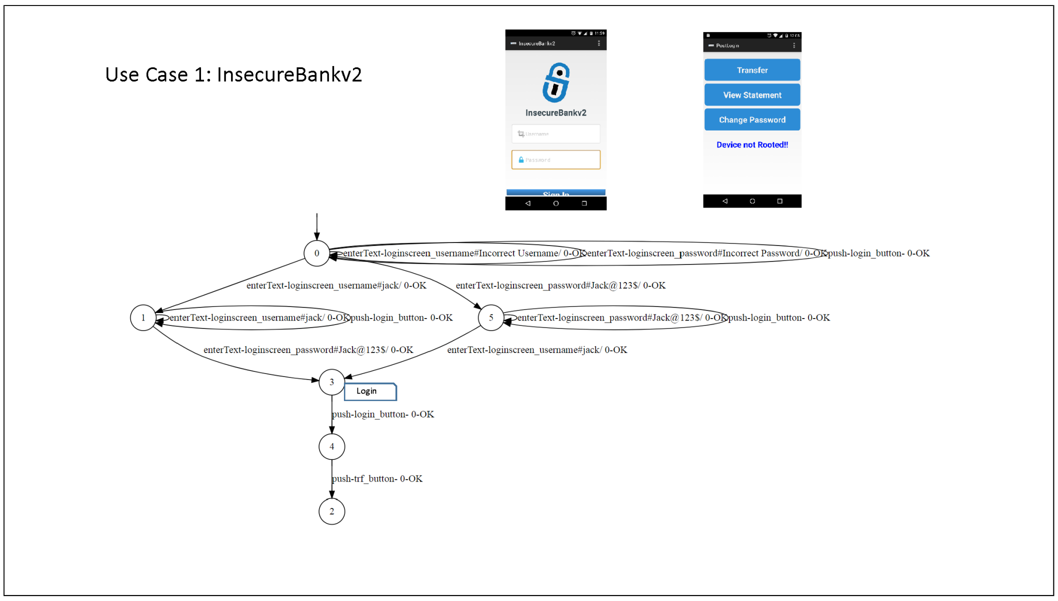
The former three chapters discussed how Android application models can be learned in a time feasible way and how the inferred models can be utilized to determine the presence of vulnerabilities. Establishment of the entire framework has been done in an interactive process where results are devised and analyzed at every step. This chapter shows the results of each aspect of this research, by elaborating the results of i) correct model learning, ii) time feasible model learning and iii) identifying vulnerabilities in the model.

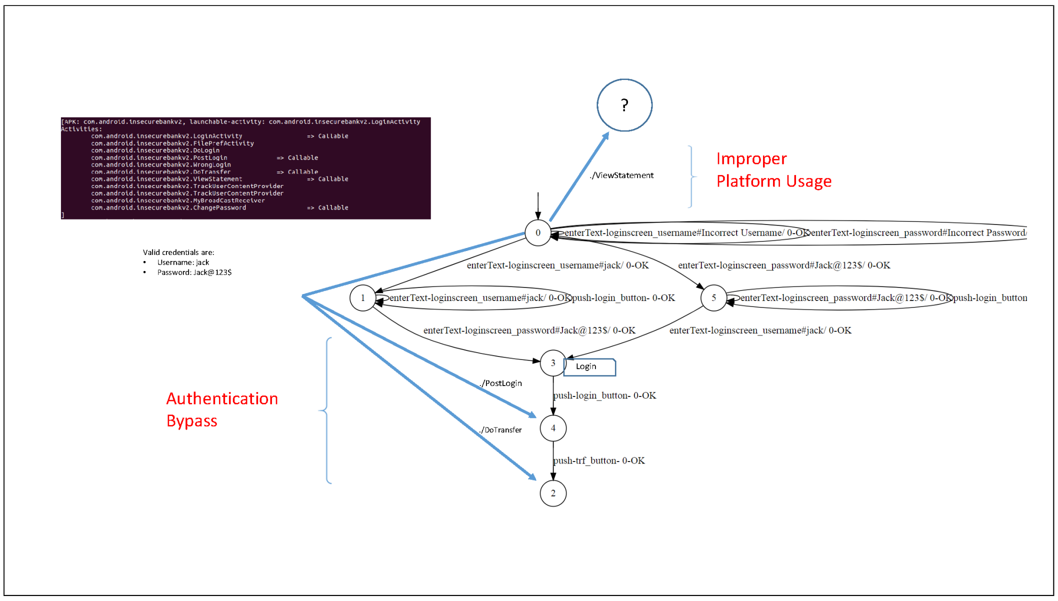
6.1 Correct Model Learning

6.2 Time-Feasible Model Learning

6.3 Vulnerability Identification

The images contained in this chapter give an impression of my final results. This chapter is however not final.


Use Case 1: InsecureBankv2
InsecureBankv2
Username
Password
Sign In
PostLogin
Transfer
View Statement
Change Password
Device not Rooted!!
enterText-loginscreen_username#Incorrect Username/ 0-OK
enterText-loginscreen_password#Incorrect Password/ 0-OK
push-login_button- 0-OK
enterText-loginscreen_username#jack/ 0-OK
enterText-loginscreen_password#Jack@123$/ 0-OK
Login
push-trf_button- 0-OK
0
1
2
3
4
5

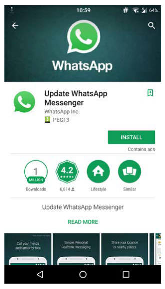

```
[APK: com.android.insecurebankv2, launchable-activity: com.android.insecurebankv2.LoginActivity
Activities:
        com.android.insecurebankv2.LoginActivity                 => Callable
        com.android.insecurebankv2.FilePrefActivity
        com.android.insecurebankv2.DoLogin
        com.android.insecurebankv2.PostLogin             => Callable
        com.android.insecurebankv2.WrongLogin
        com.android.insecurebankv2.DoTransfer            => Callable
        com.android.insecurebankv2.ViewStatement                 => Callable
        com.android.insecurebankv2.TrackUserContentProvider
        com.android.insecurebankv2.TrackUserContentProvider
        com.android.insecurebankv2.MyBroadCastReceiver
        com.android.insecurebankv2.ChangePassword                => Callable
]
```

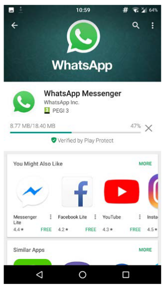
Valid credentials are:
- Username: jack
- Password: Jack@123$

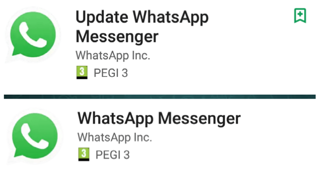
?
./ViewStatement
Improper Platform Usage
Authentication Bypass
./PostLogin
./DoTransfer
enterText-loginscreen_username#Incorrect Username/ 0-OK
enterText-loginscreen_password#Incorrect Password
enterText-loginscreen_username#jack/ 0-OK
enterText-loginscreen_password#Jack@123$/ 0-OK
push-login_button- 0-OK
Login
push-trf_button- 0-OK
0
1
2
3
4
5

Use Case 2: WhatsApp Fake Update (Nov. 2017)

Background: Unicode trick allows malicious developers to impersonate WhatsApp devlopers
> 1M downloads, now deleted from Play Store



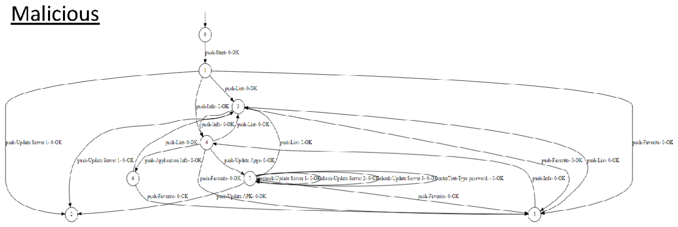




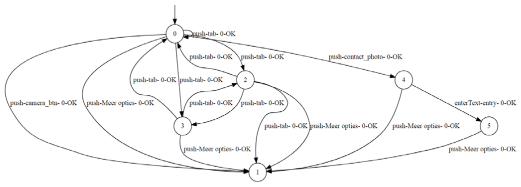
Differences:

- 100% different alphabet
  → Code Tampering/
  Extraneous Functionality

Malicious

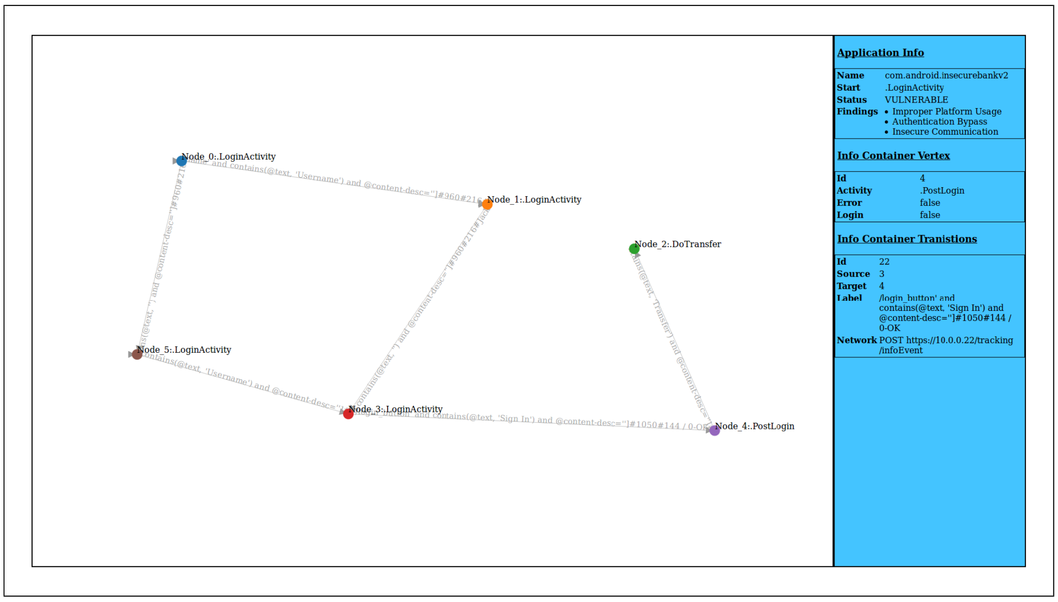


Benign



- Malicious uses http traffic
  - For tracking and advertising
  → Insecure Communication

http://req.startappservice.com/1.4/gethtmlad?productId=209893312&os=android&sdkVersion=3.6.6&flavor=1023&packageId=whyuas.fullversion.update2017&userAdvertisingId=a1daf34f-5392-4bd2-b370-e1871c9a24c8&advertisingIdSource=APP&model=Android%20SDK%20built%20for%20x86&manufacturer=Google&deviceVersion=24&locale=en_US&inputLangs=en_US&isp=310260&ispName=Android&netOper=UiszMjsg%0A&networkOperName=IHRncmJ5YA%3D%3D%0A&roaming=false&grid=13&silev=2&cellSignalLevel=2&outsource=true&width=1080&height=1776&density=3.0&fgApp=true&sdkId=3&clientSessionId=eefaaf6c-8585-42dd-83b4-59c2450cbc5a&appVersion=1.3&appCode=1&placement=INAPP_SPLASH&testMode=false&adsNumber=1&packageExclude=whyuas.fullversion.update2017&offset=0&twoClicks=true&engInclude=true&timeSinceSessionStart=16730&adsDisplayed=0&hardwareAccelerated=true&dts=true&downloadingMode=CACHE&contentAd=false&ts=1510825785034&afh=M%2F%2BKkbil1erQSnobT8AE0g&token=1510825779026-4-H4sIAAAAAAAAADMwIBYYUlUZfmBEDUOoACwo1I*fH3TwJcQKE1I10B4MlhgeBejAAkagRhEJaYjugDK3kaF7MAcGWYDmHhr4*I7qAkpLdmqbgwsMu5RGd0DjEERP2KS3fwY_a6AD0hM1XAcAkYCFzIAKAAA%23


Node_0:.LoginActivity
Node_1:.LoginActivity
Node_2:.DoTransfer
Node_3:.LoginActivity
Node_4:.PostLogin
Node_5:.LoginActivity
contains(@text, 'Username') and @content-desc="]#960#216
contains(@text, '') and @content-desc="]#960#216
contains(@text, '') and @content-desc="]#960#216#Jack
contains(@text, 'Username') and @content-desc="
and contains(@text, 'Sign In') and @content-desc="]#1050#144 / 0-OK
contains(@text, 'Transfer') and @content-desc="

Application Info

Name	com.android.insecurebankv2
Start	.LoginActivity
Status	VULNERABLE
Findings	• Improper Platform Usage • Authentication Bypass • Insecure Communication

Info Container Vertex

Id	4
Activity	.PostLogin
Error	false
Login	false

Info Container Tranistions

Id	22
Source	3
Target	4
Label	/login_button' and contains(@text, 'Sign In') and @content-desc="]#1050#144 / 0-OK
Network	POST https://10.0.0.22/tracking /infoEvent

7 Discussion

8 Conclusion

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Bibliography

[1] 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).

[2] *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. 46).

[3] 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).

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

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

[6] 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).

[7] 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).

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

[9] 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, 19).

[10] Justin Clarke-Salt. *SQL injection attacks and defense*. Elsevier, 2009 (cit. on p. 47).

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

[12] 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. 46).

[13] Arthur Gill et al. *Introduction to the theory of finite-state machines*. McGraw-Hill, 1962 (cit. on p. 20).

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

[16] 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. 21).

[17] 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. 27).

[18] 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. 27).

[19] 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).

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

[21] 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).

[22] 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. 3, 4, 27).

[23] 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. 22).

[24] 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. 28).

[25] *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. 46).

[26] Charles P Pfleeger and Shari Lawrence Pfleeger. *Analyzing computer security: a threat/vulnerability/countermeasure approach*. Prentice Hall Professional, 2012 (cit. on p. 45).

[27] 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. 27).

[28] Dorothy Graham Rex Black Erik Van Veenendaal. *Foundations of Software Testing - ISTQB Certification*. Cengage Learning EMEA, 2012 (cit. on p. 36).

[29] 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. 45).

[30] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012 (cit. on pp. 7, 9).

[31] 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 p. 37).

[32] 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. 21, 25, 38).

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

[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. 45).

[35] 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. 40).

[36] 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. 47).

List of Figures

List of Tables