Vulnerability Detection on Mobile Applications Using State Machine Inference

van der Lee, Wesley; Verwer, Sicco

DOI
10.1109/EuroSPW.2018.00008

Publication date
2018

Document Version
Accepted author manuscript

Published in
Proceedings - 3rd IEEE European Symposium on Security and Privacy Workshops, EUROS&PW 2018

Vulnerability Detection on Mobile Applications Using State Machine Inference

Wesley van der Lee* and Sicco Verwer†
Department of Intelligent Systems
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, The Netherlands
*E-mail: *wesleyvdlee@gmail.com, †S.E.Verwer@tudelft.nl*

***Abstract*—Although the importance of mobile applications grows every day, recent vulnerability reports argue the application's deficiency to meet modern security standards. Testing strategies alleviate the problem by identifying security violations in software implementations. This paper proposes a novel testing methodology that applies state machine learning of mobile Android applications in combination with algorithms that discover attack paths in the learned state machine. The presence of an attack path evidences the existence of a vulnerability in the mobile application. We apply our methods to real-life apps and show that the novel methodology is capable of identifying vulnerabilities.**

1. Introduction

Mobile applications play an ever-more important role in modern society. We do not only use the applications for a multitude of purposes on our smartphones but also deploy the applications to other types of smart devices. The multi-platform deployment of applications is, in particular, true for applications that are designed to run on the Android operating system, because Android is the most popular operating system for smart devices [1]. Although the Android platform is well-established regarding utility, new vulnerabilities are being discovered in mobile Android applications every week. One of the main reasons why mobile applications unsuccessfully meet modern security standards is because application developers fail to adopt adequate security testing during the development process of the application. Testing tools that aid the process of security examination come in the three flavors of white-box, grey-box, and black-box testing [2]. White-box and grey-box testing require prerequisite knowledge about the application's internal structure or documentation before adequate test cases can be defined. Black-box testing does not demand the application's insight because the technique by only observing the output that corresponds to an input.

Black-box testing techniques can also be fine-tuned to understand the applications internal logic, by modeling the application logic as a state machine. A state machine visually shows the application behavior and what input leads to what state and depicts the corresponding application responses. Modeling of a state machine with black-box testing is what state machine learning algorithms do. By observing a large number of *traces*, a combination of inputs and outputs, a state machine can be constructed that is consistent with the observed behavior. The inferred state machine reveals detailed information about the applications logic, and might as such function as input for identifying bugs or vulnerabilities in the application.

Learning state machines through active learning has gained increasing popularity in the last years, due to its success in revealing insight information on black box implementations. State machine learning showed promising results when inferring a model of a botnet command and control server [3] or drivers that implement the TLS protocol [4]. Active state machine learning is often implemented according to the Minimally Adequate Teacher (MAT) framework as first proposed by Angluin [5], which is composed of a learner and a teacher. The goal of the learner is to infer the state machine model of a System Under Test (SUT), by posing *membership queries* and *equivalence queries* to the teacher. Membership queries ask whether the SUT recognizes a particular input sequence. The input and answer combination together form a *trace*, and a sufficient amount of traces enables the learner to hypothesize a state machine model. The learner then poses an equivalence query to the teacher for the established hypothesis, which determines

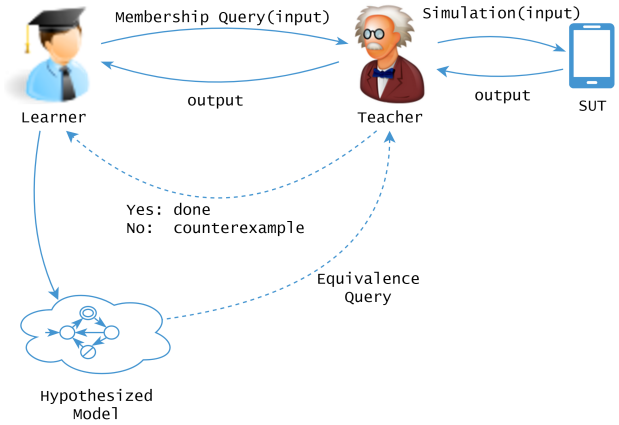


Figure 1: Active Learning with the MAT Framework

whether the model correctly describes the SUTs input/output behavior. If this is the case, learning halts because at that point a sufficiently correct model has been inferred. If the model is inequivalent to the SUT, the teacher provides a trace that distinguishes the model and the SUT. The trace that is returned by the teacher is also called a counterexample because the trace invalidates the hypothesized model. The learner uses the counterexample to refine the hypothesis. The process repeats itself until an equivalence query yields success. Figure 1 visually shows the discussed MAT framework.

The goal of this research is to improve the mobile application security by providing a testing methodology that applies vulnerability detection on inferred state machine models. To achieve this goal, a framework for learning state machines of mobile Android applications is required, and secondly, an approach that analyzes the inferred state machines to identify vulnerabilities must be established. To the best of our knowledge, there exists no other framework that automatically learns behavioral state machines on mobile Android applications and neither does there exist a method for automatic vulnerability detection on the learned models. The code of the framework's implementation has been published open source [1].

2. Model Learning

State machine models can be learned with active learning algorithms. The real-time execution of sequences of inputs on the SUT distinguishes active learning from passive learning. Hence active learning allows for the inference of more complete models. Because the generation of traces can be steered, the equivalence query that must be answered by the learner entity can be optimized as well to fulfill our mobile application domain's needs. This section discusses the principles of model learning. First, a formal notation to keep the terminology from different types of work consistent is established. Second, multiple active learning algorithms are presented, and third, various approaches are explained that the teacher entity can apply to determine the equivalence problem between the SUT and its hypothesized model.

2.1. Prerequisites

A *Deterministic Finite Automaton* (DFA) U is used to formalize state machines. A DFA can be described by a 5-tuple $U = (Q, \Sigma, \delta, q_0, F)$, where

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

Let q be a state in Q and a be a symbol in Σ, the transition function $\delta(q, a)$ returns the state q' that is reached from q with input a. We also say that q' is the a-successor of q. By defining $\delta(q, \varepsilon) = q$ and $\delta(q, wa) = \delta(\delta(q, w), a)$ for $q \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$, words $w \in \Sigma^*$ the transition function $\delta(q, w)$ implies the *extended* transition function: $\delta(q, w) = \delta(q_0, w)$ as proposed by [6]. For input words $w \in \Sigma^*$, DFA U accepts w if $\delta(q_0, w) \in F$. If U accepts w, the lambda-evaluation $\lambda(w)$ returns 1, i.e. $\lambda(w) \iff \delta(q_0, w) \in F$, and 0 otherwise. The set of all words that are accepted by U is denoted as the language of U: $L(U)$. Furthermore, $U[w]$ denotes the state q in U that can be accessed by input w, i.e. $U[w] \iff \delta(q_0, w)$.

2.2. Learning Algorithms

Learning Regular Sets from Queries and Counterexamples by Angluin [5] 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 [7].

2.2.1. L* Algorithm. The L* algorithm implements the MAT-framework that describes a learner and a teacher. The learner's task is to establish a hypothesized DFA $H = (Q_H, \Sigma, \delta_H, q_{0H}, F_H)$ for a system whose behavior can be characterized by an unknown DFA $M = (Q_M, \Sigma, \delta_M, q_{0M}, F_M)$. Initially, the learner only knows the input alphabet Σ and the learner's assignment is to learn M by posing two types of queries to the teacher:

- A *membership query* asks for the output of the SUT to an input sequence $w \in \Sigma^*$. The teacher ascertains the output for w by simulating the actions depicted in w on the SUT and the teacher returns the SUT's response.
- An *equivalence query* asks if the hypothesized DFA H is equivalent to M. The teacher is able to compute the equivalence between a DFA and an actual implementation through *conformance testing* methods. The teacher answers *yes* if H is equivalent to M or supplies a *counterexample*, which is a sequence $t \in \Sigma^*$ for which holds $\lambda_H(t) \neq \lambda_M(t)$.

The learner keeps track of all the traces information in an observation table (S, E, T). Each input word of a trace consists of a prefix part and a suffix part. The observation table vertically labels the set of prefixes S and horizontally labels the set of suffixes E. Each entry (s, e) in the table for $s \in S$ and $e \in E$ is assigned the value 0 or 1 depending on the membership query response for word $w = s \cdot e$. For each row of values in the table, the *row(s)*-function returns an array of outputs for words $s \cdot e$ for all $e \in E$. If the observation table is closed and consistent, a hypothesis DFA can be constructed. An observation table is *closed* if for each state that is characterized by the table a one-letter prefix extension is present that describes the transition, i.e. $\forall w \in S \times \Sigma$ there is a $s \in S$ such that $row(w) = row(s)$. An observation table is *consistent* if all by the table characterized states have the same one-letter behavior, i.e. $\forall s_1, s_2 \in S$ where $row(s_1) = row(s_2)$ all $a \in \Sigma$ depict $row(s_1 \cdot a) = row(s_2 \cdot a)$.

1. git@github.com:wesleyvanderlee/AppSecurity.git

If the observation table is closed and consistent, a DFA $H = (Q_H, \Sigma, \delta_H, q_{0H}, F_H)$ can be constructed as follows:

- $Q_H = \{row(s) | s \in S_H\}$
- $q_{0H} = row(\varepsilon)$
- $F_H = \{row(s) | s \in S_H \text{ and } T_H(s, \varepsilon) = 1\}$
- $\delta(row(s), a) = row(sa)$

According to the L* algorithm, the hypothesized DFA is then subjected to an equivalence query to the teacher, who either returns *yes* in which case learning halts, or returns a counterexample word w that distinguishes both H and M. The learner adds all prefixes of the counterexample to set S, which causes the observation table not to be closed and consistent anymore because the hypothesis' output for w needs to be corrected. The learning process continues until the table again is closed and consistent, and the teacher returns *yes* to an equivalence query.

2.2.2. TTT Algorithm. A drawback of the observation table is that entries $S \times E$ in the observation table (S, E, T) contain redundant information. An approach to overcome redundant entries is proposed in the TTT algorithm that replaces the observation table with a *discrimination tree* [8]. When using the TTT algorithm, the learner maintains a set S of access sequences to states. The states of the hypothesis correspond to leaves of a discrimination tree T, where inner nodes are labeled with elements from a set of discriminators E. A hypothesis DFA is constructed by *sifting* words $w \in S \times \Sigma$ through the discrimination tree that starts at the root of the tree. For every inner node $v \in E$ of the tree the sifting process passes one branch to the left- or right- child of v depending on the value $\lambda(w \cdot v)$, until a leaf node is reached. By convention, if $\lambda(w \cdot v)$ returns 0 the process branches to the left child of v, otherwise the process branches to the right child. The leaf node indicates the resulting state for the sequence w.

2.3. Equivalence Problems

The active learning algorithms explain how to construct a hypothesis DFA H, but do not answer how the teacher can verify equivalence between H and the machine corresponding to the implementation of M. Conformance testing offers methods to discover this equivalence [9]. A straightforward implementation of a conformance method is the direct search for a counterexample by performing random actions on both the hypothesis machine and the SUT. This technique is also called a *RandomWalk*. If for a high number of actions both machines yield the same result, they are determined to be equivalent. If at some points the outputs differ, the sequence of actions that led to the difference in output is regarded as the counterexample.

A more comprehensive, but intensive, approach to identify counterexamples is the W-Method [10]. In essence, the method creates a large test set of input queries for which the DFA and the SUT should yield the same behavior to be equivalent. The test set is created by appending a *transition cover set* $(S \cdot \Sigma)$ with a *traversal set* $(\Sigma^l$ that contains all input sequences of length $l = m - n + 1$, where $m = |Q_M|$ and $n = |Q_H|$ and a *characterizing set* (E) that distinguishes all pairs of states [10], [11]. Note that since M is unknown, $|Q_m|$ is unknown as well and has to be estimated by human judgment. If one test word $w \in S \cdot \Sigma \times \Sigma^l \times E$ yields a different output for H and M, then w is a counterexample. If no such w exists, H and M are equivalent.

3. Setup

The goal of this research is to improve mobile security by proposing a method to detect vulnerabilities in mobile applications. The method constructs an abstraction of a mobile app. The inferred model is then examined for the presence of patterns that coincide with patterns of vulnerabilities. The vulnerability detection technique is thus twofold. First of all, state machine models that describe the behavior of the mobile application need to be inferred. Because the system under test can be interactively queried for outputs, active state machine learning can be applied to generate the set of traces that are required to learn a deterministic and finite state machine. Secondly, a detection methodology is proposed that identifies the presence of vulnerabilities based on the inferred state machine. The method applies vulnerability-based pattern detection algorithms that detect the presence of a weakness from the model. The twofold parts of the technique are setup as follows.

Modeling Mobile Applications. The previous section discussed two active state machine learning algorithms that can infer a model by interaction with an application. The LearnLib library accommodates the learning algorithms and auxiliary helper classes that alleviate the active learning process [12]. Learning models with LearnLib requires a *mapper* that associates the queries which are posed by LearnLib to commands that are understandable by the system under test. The system under test is in this instance an Android application, and as a consequence, the mapper also needs to execute the commands on the system and observe the application's behavior. One of the available drivers that can perform the types of commands required for active learning is Appium [13].

All mentioned components can be integrated according to the MAT framework. The learner poses membership and equivalence queries to the teacher. The teacher translates both queries to a simulation command which is mapped to an Appium understandable command by the mapper instance. Appium then executes the test on the mobile device and observes the system's response. The result is returned to the learner in reverse order. Figure 2 visually depicts the integration of all the mobile learning components.

Vulnerability Detection in Models. After the application is abstracted to a state machine model, the model functions as a new source of information to assess the application's security. The state machine model shows what type of actions are allowed within the application and illustrates the cohesion of logical components. To detect vulnerabilities from a state machine model, paths that constitute to a

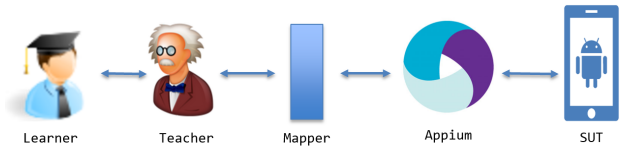


Figure 2: Mobile learning components integrated according to the MAT framework.

violation of a class of vulnerabilities must be identified. The paths in the state machine then represent an attack scenario that violates a predefined vulnerability class.

The malicious paths can be identified by designing detection algorithms that detect the exposure of predefined vulnerability classes. For demonstration and within the scope of this research, we limit ourselves to the vulnerabilities that are discussed in the infamous OWASP Top 10 Mobile 2016 [14]. The list is the most recently published list of top 10 crucial vulnerability classes that need to be addressed when examining mobile security. For each class that is enumerated in the top 10 list, the question of *how the specific vulnerability class would materialize in a state machine model* needs to be answered. This question can only be answered for vulnerability classes that are behavior-driven, such that paths of violations would appear in a behavioral state machine model.

4. Model Inference on Android Applications

Before active learning starts, two challenges that are specific to the mobile environment must be solved. Firstly, the input alphabet must be defined before the learner and the teacher can start interacting. Secondly, a mobile application introduces behavior that might be inconsistent/non-deterministic due to external mobile factors. The learning algorithms can only cope with deterministic behavior, and as a consequence, we must mitigate this problem. At the end, this section presents multiple models that are learned using different types of learning algorithms and equivalence oracles.

4.1. Input Alphabet

The learner and the teacher share one set of common knowledge: the input alphabet Σ. Σ must be defined before learning starts and cannot be changed during learning according to the L* and TTT learning algorithms. We want to infer a model that describes the application's behavior on the interaction level. Inference of such a model can be achieved by applying the appropriate level of actions as the input alphabet. The mobile application responds to *internal* and *external* actions. The application facilitates internal actions, such as navigating through the application by pressing buttons. Commands that are external to the application are facilitated by the operating system or other applications, such as toggling the WiFi mode on the mobile device or pressing the `home`-button. The application's behavior can change when such settings are modified.

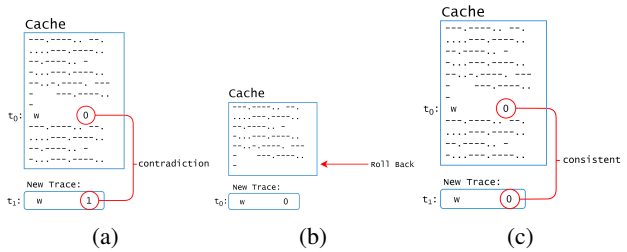
External actions are less suitable to be included in the alphabet because these actions are executable from every state. As a consequence, the external actions clutter the state machine model with transitions that are possible to perform from each state and behavior-wise do not contribute to a better understanding of the model. Only the internal commands are thus used as input alphabet. We distinguish the following user interaction commands: *push* for pushing on buttons and elements, *check* for toggling checkboxes, and *enterText* for entering text in a field.

4.2. Overcoming Non-Deterministic Behavior

The behavior of mobile applications depends on a set of environmental influences. The most well-known example is the loss of Internet connectivity. At arbitrary moments, due to a large number of factors, the mobile device or the application instance can experience a loss of connectivity with the Internet. The reason why the external influences on the behavior are essential to consider when learning a DFA is that different behavior in time causes the model to be non-deterministic. Since the goal is to infer a deterministic finite automaton, the non-deterministic behavior forms a contradiction with the earlier observed behavior. The contradiction causes the learning application to crash. These environmental influences do not necessarily originate from an improper network connection, but can also arise due to inconsistencies in GPS positioning details, Bluetooth connections, the battery level and much more.

To overcome the occurrence of non-deterministic behavior, we propose and implement a roll-back solution that uses the cache. The implemented cache stores combinations of inputs and the corresponding outputs to reduce evaluations of membership queries. The cache can be used in the following way to overcome the occurrence of non-deterministic behavior. Figure 3 illustrates the roll-back process. The figure depicts a sequence w of input symbols for which at two points t_0 and t_1 in time $\lambda_{t_0}(w) \neq \lambda_{t_1}(w)$. This situation is depicted in Figure 3a, where at time t_0 the output from $\lambda(w)$ equals 0, whereas at t_1 the output from $\lambda(w)$ is 1. Because the lambda evaluation differs for the same input word, one of the two traces is incorrect. If an activity after w can be successfully executed, the corresponding trace has not been affected by external influences. Therefore, the learner assumes that the trace that depicts new behavior is correct instead of the trace that contends the opposite. There are numerous of dependencies that depend on the environment and thus change in time, that cause an action to be inadmissible. However, if the additional behavior can be executed once, the action must thus be legal and not be subjected to external influences.

During the learning period between t_0 and t_1, any number of queries could have been posed by the learner that (partially) depended on the cached result of w computed at t_0. Because the result of w is assumed to be uncertain, the queries posed between t_0 and t_1 that incorporate w's



(a) (b) (c)

Figure 3: Cache rollback methodology to resolve observed non-deterministic behavior. At (a) a contradiction is discovered, (b) the cache is reverted and (c) the contradiction is resolved.

cached result could also be false. For that reason, the cache is reverted to the last trace before t_0, which is shown in Figure 3b. The positive result for w is cached at t_0 and learning continues. If for the new cache at time t_1 the query result is equivalent, the learning process continues.

4.3. Learning with L* and RandomWalk

Before active learning can start, two variables of the MAT framework have to be determined. The variables are the active learning algorithm and the technique that is used to determine the equivalence between a hypothesized model and a software implementation. The inferred model is an approximation of the application and different settings for the variables conclude different approximations and consume a different amount of time.

Different values for the active learning variables will be used that aim to result in a complete model by consuming a low amount of time. Without loss of generality, various models will be learned for the 9292[2] Android application, which is a Dutch public transport planning service.

Figure 4 shows the model that is inferred when using the classic L* algorithm for learning and the RandomWalk algorithm to determine the equivalence. The model is learned without receiving a counterexample, which indicates that the first time the observation table became closed and consistent, the RandomWalk algorithm was not able to find any counterexample.

4.4. Learning with TTT and RandomWalk

One disadvantage of the L* algorithm is the storage of redundant information in the observation table. Because the observation table (S, E, T) needs to be complete, each entry in $S \times E$ needs to be computed, hence also redundant fields are evaluated. Redundant entries cause superfluous membership queries, and as a result, the entries negatively influence the learning time performance. The learning time of the model inference discussed above consumes more than 26 hours. The feasibility of model inference increases when the learning algorithm does not query for superfluous data.

2. https://play.google.com/store/apps/details?id=nl.negentwee

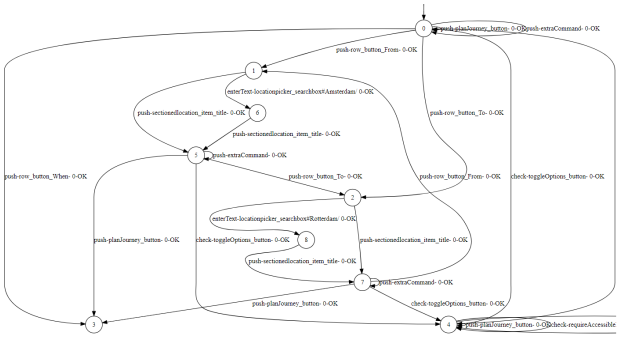


Figure 4: The inferred machine for the 9292 application using L* and RandomWalk.

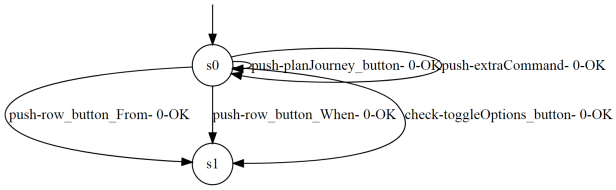


Figure 5: The inferred machine for the 9292 application using TTT and RandomWalk.

An active learning algorithm that uses a space-efficient data structure for storing observations is the TTT algorithm. the TTT algorithm yields the model depicted in Figure 5.

The inferred model in Figure 5 obviously yields an incorrect result as the model does not depict the behavior from the 9292 application at all. The difference between the models presented in Figures 4 and 5 show a decrease in modeled behavior when using the TTT algorithm. The reason why the same equivalence oracle results in a more detailed model is that the L* algorithm generates more traces than the TTT algorithm does. The random walk is even not able to find a counterexample for this small model. The test set constructed during an equivalence query must thus be extended or changed to advance the inferred model.

4.5. Learning with TTT and RandomWalk-HappyFlow

One method to expand the test cases is to test for a predefined input sequence that is guaranteed to be accepted by the SUT. The input sequence could, for example, be a happy flow of the application. Software applications are often developed with domain-specific use cases as a functional requirement [15]. The use cases describe how a software system should respond and are thus by definition accepted by the SUT. Hence a happy flow can function as a counterexample if the conjecture does not accept the happy flow. Furthermore, if the input sequence w is a happy flow, then for each $n = 1 \ldots |w|$ the sequence $w' = w_0 w_1 \ldots w_{n-1} w_n$ is also a valid counterexample if the conjecture rejects w', since all steps until $|w|$ are legally executable as well. The

RandomWalk equivalence oracle has been extended, dubbed as RandomWalk-HappyFlow, such that the equivalence oracle was able to search for a counterexample in the SUT's happy flow when the oracle otherwise would conclude that the conjecture is equivalent.

The equivalence testing algorithm first executes the original RandomWalk algorithm to discover a counterexample. When this results in the value `null`, learning would normally stop. At this point, the extended part of the RandomWalk-HappyFlow algorithm engages, by searching for a counterexample for each sub-word of all happy flow words. This procedure has the following advantages regarding the speedup of identifying a counterexample:

1) Because happy flow words and the corresponding subwords are computed only for the hypothesized conjecture, no simulation on the SUT is needed. Hence identification of a valid counterexample from a set of happy flows can be performed almost instantly.
2) The test words are generated from the smallest subsequence to the largest subsequence. If the hypothesis accepts a subsequence $w = w_0 \dots w_n$ but the subsequence $w \cdot w_{n+1}$ is not accepted by the hypothesis, symbol w_{n+1} is most likely a distinguishing suffix. Because the counterexample is as small as possible, it reduces the appearance of redundant entries in the observation table and temporary nodes in the discrimination tree.

Although the extended RandomWalk algorithm yields a better result opposed to the original RandomWalk algorithm, a 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, differ per application and must thus be collected and processed for each application as a test individually. Gathering happy flows requires knowledge about the application before learning starts and therefore surpasses the black box testing methodology.

4.6. TTT and the W-method

Another method to discover counterexamples is to substitute the equivalence oracle from RandomWalk to the W-method, which is a more systematic and exhaustive method. Two drawbacks of the W-method are the explosion of the number of test cases as the number of unique states in the SUT model increases and the excessive amount of symbols the test queries contain, i.e., the query length. Both factors negatively influence the time performance. After multiple test runs with the W-method equivalence oracle, no test run could finish, due to the quantity and size of the test cases generated by the W-method. The run required more time than 60 hours and was halted 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. The disadvantage of the W-method is the size

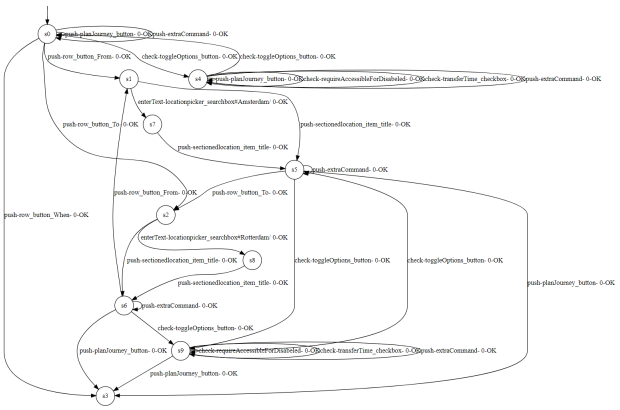


Figure 6: The inferred machine for the 9292 application using TTT and a consolidation between the W-method and Smetsers et al.'s work for creating a characterizing set.

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 [16]. Smetsers et al. also argue that the set of all minimal separating sequences for all pairs of states can replace Chow's W-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 model that can be learned by combining Chow's W-method with Smetsers et al.'s work for creating a characterizing set is shown in Figure 6. The last inferred model depicts the model that best approximates the application's structure, as it has the highest number of states and transitions. The final model could be learned within four hours. The remainder of this paper uses TTT as the learning algorithm and the modified W-method as the equivalence oracle.

5. Vulnerability Identification on Models

After the determination of the most optimal parameter values to successfully infer state machines of Android applications, a detection technique must be established that identifies vulnerabilities from the inferred models. The detection techniques attempt to discover a path that exploits a given class of vulnerabilities as specified in the OWASP Top 10. If such a malicious path exists in the inferred model, the application must contain the specific vulnerability.

The learned state machines are defined by the distinguishing observable application responses to user interaction commands. Because the state machine depicts how the components of the application are interconnected, the model represents the app on an abstract level. The state machine can also be enriched with information and properties that are

static to a given state or transition, such that the combination of dynamic behavior and the static information can be used for attack path discovery.

The following enumerates the OWASP Top 10 Mobile and provides reasoning on their detection in a behavioral model. If the class is capable of being detected in a behavioral model, an algorithm is presented that identifies attack paths that exploit the vulnerability. If it is required to enrich the model with properties that are local to a state, the enrichment is mentioned and explained in the corresponding vulnerability class. To keep consistency between all algorithms, the discovered attack paths are saved in a set R, which is returned at the end of the algorithm. A violation is present if R is non-empty.

1. Improper Platform Usage This category covers the applications failures to meet the platform security controls and other scenarios where platform features are misused. A violation of this class is detectable because the platform controls also include Android *intents* such as the calling of activities. Specifically for Android apps, intents have shown to be a valuable source of input when assessing proper platform usage as advocated by [17]. The intents that are exhibited by the app can be carved from the app's binary. Under the assumption that the inferred model describes normal application behavior, any new behavior that can be invoked by calling an activity is superfluous and should thus not be accessible by end users. The algorithm to detect paths in the inferred model is depicted as Algorithm 1.

Algorithm 1 Improper Platform Usage Identification

Input: Inferred State Machine $M = \{Q, \Sigma, \delta, q_0, F\}$
$R \leftarrow \emptyset$
$A \leftarrow$ activities from SUT's binary
for all a in A **do**
  if a is callable in M **then**
    add a to R
  end if
end for ▷ R contains all callable activities
for all q in Q and R is not empty **do**
  remove q's activity from R
end for
return R

2. Insecure Data Storage The insecure data storage category is violated when information which is processed by the application, is stored locally on the device in an insecure way. Android applications store data, preferably encrypted and in the correct directory, in the mobile devices file system. This category cannot be discovered from the user interaction model abstraction of the application because the data storing activity is not noticeable by the user or the Appium driver.

3. Insecure Communication Most applications function according to a client-server framework, where the application (client) communicates with a server (back-end). The communication between client and server often encloses sensitive data, such as authentication credentials or confidential files. Sending and receiving sensitive data should be done on a cryptographically secured communication, such as the transport layer security (TLS) protocol describes, or apply custom cryptographic solutions, such as certificate pinning [18].

A violation of this vulnerability can be detected because the network requests exit the mobile device. If the web request exits the mobile device, the request can be caught with a proxy and assessed. Each transition in the model is labeled with the web requests that correspond to the action. Set R is then the set of all insecure web requests that are performed by the application. Algorithm 2 depicts the method to identify violations of this vulnerability class.

Algorithm 2 Insecure Communication Identification

Input: Inferred State Machine $M = \{Q, \Sigma, \delta, q_0, F\}$
$R \leftarrow \emptyset$
for all t in δ **do**
  $r \leftarrow$ request_insecure(t.request)
  if r **then**
    add r to R
  end if
end for
return R

4. Insecure Authentication Authentication is the process or action of verifying the identity of a user such that restricted services and data can only be presented to authorized users. Poor or missing authentication schemes allow an adversary to execute functionality within the mobile app or backend server used by the mobile app without the proper restrictions. Functionality-wise, the violation materializes in the same way as an insecure authorization violation and hence detection is discussed in that category.

5. Insufficient Cryptography To enable secure data storage, the implementation of cryptographic fundamentals is required. The correct appliance of cryptography is a static property that can be assessed by reviewing the source code manually. The level of adequate cryptography is not visible on the user interaction level. Hence insufficient cryptography cannot be detected from a behavioral model.

6. Insecure Authorization Authorization associates the appropriate level of operations to an authenticated identity. A poor or missing authorization scheme allows attackers to exploit functionality that is above their privilege. The inferred model can be used to assess the authorization of an application. By searching for paths to states that should only be accessible for authenticated users, one can identify an authorization bypass and diagnose an improper authorization vulnerability in the application. States that are only accessible after the *login* state are decided to be secured. A path to one of the states that avoid the login state is a violation of authentication. The algorithm that corresponds to detect insecure authentication is presented in Algorithm 3. This algorithms determines which state represents the authentication state based on login fields that are present, such as a username field and a password field in the user interface of a specific state.

Algorithm 3 Insecure Authentication Identification

Input: Inferred State Machine $M = \{Q, \Sigma, \delta, q_0, F\}$

$a \leftarrow$ authentication state of M ▷ $a \in Q$
if a is `null` **then** ▷ no authentication → no authentication bypass
  return R
end if
$Marks \leftarrow$ subset of nodes possible to reach after a
$M' \leftarrow M - a$ ▷ the machine without the authentication state
for all m in $Marks$ **do**
  if a path from the q_0 to m exists in M' **then**
    add $path$ to R
  end if
end for
$Q'' \leftarrow Q - Marks$
$A \leftarrow$ callable activities
for all q in Q'' and A is not empty **do**
  remove q's activity from A
end for
add A to R
return R

7. Poor Code Quality The category of poor code quality does not directly appear in a behavioral state machine model because code quality is a static property of the code. Since black box testing does not examined the source code, this property cannot be identified from a state machine model.

8. Code Tampering and Extraneous Functionality Modified forms of applications that are changed by a third party are commonly distributed through official and unofficial marketplaces. In this case, an application is either torn apart, the code is tampered with and then reassembled again, or an entirely new application is made from scratch that tries to impersonate the original application. The code tampering class describes this scenario. The modified version tries to trick users into thinking it is a benign application, but also, the tampered application can perform malicious activities.

Under the assumption that tampering source code yields different behavior, code tampering can be identified by comparison of the inferred model to a reference model of the application under test. 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 SUTs model is the result of a tampered application, Algorithm 4 will identify the difference between the two of them. The behavior that the application under test contains in addition to the reference model can potentially be malicious.

The category of extraneous functionality describes functionality that enables a user to perform operations that is not directly exposed by the user interface. Detection of this vulnerability assumes that the extraneous functionality is an addition or deduction of functionality when compared to a reference model. Hence, this vulnerability class is detected along with the code tampering vulnerability.

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

Algorithm 4 Code Tampering Identification

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

$R_1, R_2 \leftarrow \emptyset$
$TCS_1 \leftarrow TCS(M)$
$TCS_2 \leftarrow TCS(M')$
for all $w \in TCS_1$ **do**
  if $\lambda^M(w) \neq \lambda^{M'}(w)$ **then**
    $R_1 \leftarrow w$
  end if
end for
for all $w \in TCS_2$ **do**
  if $\lambda^M(w) \neq \lambda^{M'}(w)$ and $w \notin R$ **then**
    $R_2 \leftarrow w$
  end if
end for
$R \leftarrow R_1, R_2$
return R

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 category is also a static property that is not exploitable by interacting with the application on the end user level and can hence not be detected in a model.

6. Results

We combined model learning on Android applications with vulnerability detection algorithms. The combination has been applied to an extensive set of mobile applications. To verify the success rate of the detection algorithms, known-to-be vulnerable applications have been tested. We obtained the vulnerable applications from various sources, such as GitHub repositories with reported vulnerabilities and enterprises that developed applications with the intent to be vulnerable for educative purposes. Because not all applications are publicly distributable, we present and discuss the results of one test application. The results are shown in Table 1.

The `InsecureBankv2` application is developed by Dinesh Shetty for the self-educative of mobile hacker training [19]. The model learner was able to infer and enrich the state machine model as depicted in Figure 7. The vulnerability identification algorithms were able to identify paths that exploit one of the vulnerability classes. In total three vulnerability classes were discovered to be violated.

InsecureBankv2	Violation
Improper Platform Usage	There were new discovered states for activity ./ViewStatement
Insecure Authorization	Authorization can be bypassed for activities ./PostLogin (to state 4) and. /DoTransfer (to state 5).
Insecure Communication	Data is sent the back-end unencrypted. Transition from state 3 to 4 requests: `POST http://57.97.2.11:8888/login`.

TABLE 1: Identified vulnerabilities for the applications *InsecureBankv2*

First of all, a path, depicted in green, induces behavior that is not described by the inferred model by using a platform control. The new behavior is discovered when analyzing the *activities* from the SUT's binary. Because the action is new to the model and we assume that the model represents the entire application, the platform control is misused, and improper platform usage is detected. If the inferred model would be incomplete, the vulnerability detection algorithm raises (a lot of) false positive results. If the latter is the case and substantially occurs, one should aim to infer a model as complete as possible in order to successfully assert the security properties. Therefore the path that exhibits the *new* behavior should be included in the model by extending the input alphabet or adding the path as a happy flow.

The second exploitation path, depicted in red, that has been discovered bypass the *login* state and therefore constitutes with a secure authorization violation. Because post login activities can be reached without going through the proper authentication scheme, insecure authorization is detected. Detection of other ways to reach new behavior is achieved likewise the *Improper Platform Usage* detection, but the property that the actions are known to the model and normally only accessible after the login state are utilized to distinguish the classes of exploitation. The Insecure Authorization class can produce faulty results in two ways. Way one would be that the login state is incorrectly determined, which causes the behavior to be known to the model and not to be restricted. As a result, no vulnerability will be detected which would be a false negative result. The false negative rate can be reduced by optimizing the classification technique that is used to determine if a state is a login state. The second way a result can be faulty is when information for an authorization bypass cannot be obtained from the SUT's binary and can therefore not be detected. These type of results are difficult to reduce and is a limitation in the way we use models for vulnerability detection, because the vulnerability then is not known.

At last, insecure communication was detected because some transitions, such as the transition between state 3 and 4, initiate a web request that does not implement modern security standards. The web requests are not encrypted and are hence vulnerable to a man-in-the-middle attack. All identified violations were expected before testing and manually verified to be present. We conclude that the combination of model learning and vulnerability identification was able to assess the application's security correctly and discovered three significant classes of vulnerabilities to be present.

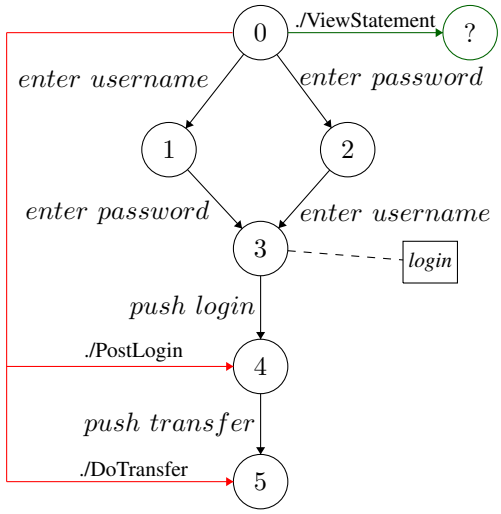


Figure 7: Simplified model of the InsecureBankv2 app

The algorithm to identify code tampering or extraneous functionality can be applied to a scenario where a counterfeit version of the SUT has been learned and the differences in terms of functionality need to be assessed. The detection technique is not applicable to the InsecureBankv2 scenario, since no counterfeit version is available. If such an application would be available, the corresponding detection algorithm identifies the difference in behavior, which if detected confirms that the app is a modified version.

6.1. Future Work

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

Model Completeness Within the scope of this research, we assume that the inferred state machine model is complete, such that the model describes the behavior of the entire application. Model completeness depends on the completeness of the input alphabet and the thoroughness of the equivalence oracle. One way to measure the accuracy of model completeness is to compute the code coverage of all actions depicted in the model. If we can compute the code coverage per action, the learning algorithm can guide the learning process by selection of test cases that generate the most or new code coverage. Moreover, an equivalence algorithm can be established that generates a counterexample based on the conditions to reach the uncovered code. This methodology has shown to be effective for the discovery of more reachable states by Smetsers et al. [20].

Platform Extension Another lookout is to apply model learning to other mobile platforms, such as iOS. One use case to support both platforms for model learning is to

verify consistency between the two applications. To reach the largest target audience, applications are written both for Android and iOS. Because a development team is often specialized in code development for a specific platform, the different applications are often developed separately from each other [21]. Although the applications are separate products, they should depict the same behavior. Comparison of state machine models can identify differences or verify equivalence regarding behavior.

7. Conclusion

The objective of this research was to apply time-optimized state machine learning to mobile Android applications and establish algorithms that identify the presence of vulnerabilities in the application by assessment of the inferred model. The objective has been divided into three subgoals. First of all, we must apply active state machine learning to mobile Android applications. Secondly, to achieve time-optimization, we need to administer methodologies that improve the learning time. Lastly, the vulnerability identification algorithms that cope with the inferred models need to be established.

The methodology and the accompanying proof of concept that are presented in this research, achieve the research objective by combining modern active learning algorithms for model inference and applying techniques that specifically enable state machine learning for the mobile environment. The combination of modern algorithms, such as the TTT algorithm and Smetsers et al.'s algorithm for finding minimal separating sequences for all pairs of states, can curtail redundant queries and reduce the query space and hence optimize the time that is required to answer model equivalence. The mobile application environment also introduces non-deterministic behavior. A method to overcome the non-deterministic behavior is presented, by applying the cache rollback technique.

To achieve the last subgoal, one must be able to identify vulnerabilities on inferred models. Vulnerability identification is made possible by first of all enriching the state machines with supplementary information. Secondly, vulnerability identification algorithms have been proposed that can determine paths of exploitation in the inferred and enriched models. The paths then pose as evidence for the presence of a vulnerability that belongs to a class defined by OWASP.

The proposed testing methodology provides the first step towards a new approach for testing and securing the mobile environment. There remain opportunities to cover additional classes of vulnerabilities and to improve active learning on mobile applications. Nevertheless, we demonstrated that state machine learning can be a valuable asset when searching for vulnerabilities in mobile applications.

References

[1] T. Vidas, C. Zhang, and N. Christin. Toward a general collection methodology for android devices. *digital investigation*, 8:S14–S24, 2011.

[2] M. Khan, F. Khan, et al. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Sciences and Applications*, 3(6):12–1, 2012.

[3] C. Cho, E. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 426–439. ACM, 2010.

[4] J. De Ruiter and E. Poll. Protocol State Fuzzing of TLS Implementations. In *USENIX Security Symposium*, pages 193–206, 2015.

[5] D. Angluin. Learning regular sets from queries and counterexamples. *Machine learning*, 2(4):319–342, 1988.

[6] M.J. Kearns and U. Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[7] E.M. Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978.

[8] M. Isberner, F. Howar, and B. Steffen. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. pages 307–322, 2014.

[9] M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[10] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 5 1978.

[11] R. Dorofeeva, K. El-Fakih, S. Maag, A. Cavalli, and N. Yevtushenko. Fsm-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology*, 52(12):1286–1297, 2010.

[12] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. Learnlib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(5):393–407, 2009.

[13] M. Hans. *Appium Essentials*. Packt Publishing Ltd, 2015.

[14] Z. Lanier et al. OWASP Top 10 Mobile, 2016. [Online] Available: www.owasp.org/index.php/Mobile_Top_10_2016-Top_10.

[15] Black R., E. Van Veenendaal, and D. Graham. *Foundations of Software Testing - ISTQB Certification*. Cengage Learning EMEA, 3 edition, 2012.

[16] R. Smetsers, J. Moerman, and D.N. Jansen. Minimal separating sequences for all pairs of states. In *International Conference on Language and Automata Theory and Applications*, volume 9618, pages 181–193. Springer, 2016.

[17] H. Ye, S. Cheng, L. Zhang, and F. Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, page 68. ACM, 2013.

[18] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl. To pin or not to pin-helping app developers bullet proof their tls connections. In *USENIX Security Symposium*, pages 239–254, 2015.

[19] S. Dinesh. Android-InsecureBankv2, 2017. [Online] Available: github.com/dineshshetty/Android-InsecureBankv2.

[20] R. Smetsers, J. Moerman, M. Janssen, and S. Verwer. Complementing model learning with mutation-based fuzzing. *arXiv preprint arXiv:1611.02429*, 2016.

[21] S. Allen, V. Graupera, and L. Lundrigan. *Pro smartphone cross-platform development: iPhone, blackberry, windows mobile and android development and distribution*. Apress, 2010.