Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning

Wontae Choi
EECS Department
University of California, Berkeley
wtchoi@cs.berkeley.edu

George Necula
EECS Department
University of California, Berkeley
necula@cs.berkeley.edu

Koushik Sen
EECS Department
University of California, Berkeley
ksen@cs.berkeley.edu

Abstract

Smartphones and tablets with rich graphical user interfaces (GUI) are becoming increasingly popular. Hundreds of thousands of specialized applications, called apps, are available for such mobile platforms. Manual testing is the most popular technique for testing graphical user interfaces of such apps. Manual testing is often tedious and error-prone. In this paper, we propose an automated technique, called *SwiftHand*, for generating sequences of test inputs for Android apps. The technique uses machine learning to learn a model of the app during testing, uses the learned model to generate user inputs that visit unexplored states of the app, and uses the execution of the app on the generated inputs to refine the model. A key feature of the testing algorithm is that it avoids restarting the app, which is a significantly more expensive operation than executing the app on a sequence of inputs. An important insight behind our testing algorithm is that we do not need to learn a precise model of an app, which is often computationally intensive, if our goal is to simply guide test execution into unexplored parts of the state space. We have implemented our testing algorithm in a publicly available tool for Android apps written in Java. Our experimental results show that we can achieve significantly better coverage than traditional random testing and $\mathcal{L}^*$-based testing in a given time budget. Our algorithm also reaches peak coverage faster than both random and $\mathcal{L}^*$-based testing.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Algorithm, Design, Experimentation

Keywords GUI testing, Learning, Automata, Android

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA..

http://dx.doi.org/10.1145/2509136.2509552

1. Introduction

Smartphones and tablets with rich graphical user interfaces (GUI) are becoming increasingly popular. Hundred of thousands of specialized applications, called *apps*, are already available for these mobile platforms. The complexity of these apps lies often in the user interface, with data processing either minor, or delegated to a backend component. A similar situation exists in applications using software-as-a-service architecture, where the client-side component consists mostly of user interface code. Testing such applications involves predominantly user interface testing.

We focus on user interface testing for Android apps, although many of the same challenges exist on other mobile and browser platforms. The only two tools that are widely used in practice for testing Android apps are Monkeyrunner [2], a framework for manually scripting sequences of user inputs in Python, and Monkey [3], a random automatic user input generation tool. A typical use of the Monkey tool involves generating clicks at random positions on the screen, without any consideration of what are the actual controls shown on the screen, which ones have already been clicked, and what is the sequence of steps taken to arrive at the current configuration. It is not surprising then that such testing has trouble exploring enough of the user interface, especially the parts that are reached after a specific sequence of inputs.

In this paper, we consider the problem of automatically generating sequences of test inputs for Android apps for which we do not have an existing model of the GUI. The goal is to *achieve code coverage quickly* by learning and exploring an abstraction of the model of the GUI of the app. We discuss further in the paper various techniques that have been explored previously to address different aspects of this problem. However, our experience shows that previous learning techniques ignore a very significant practical constraint in testing apps. All automatic exploration algorithms will occasionally need to restart the app, in order to explore additional states reachable from the initial state. The only reliable way to restart an app is to remove and reinstall it. This is necessary, for example, when the application has persistent data. Our experiments show that the restart operation takes 30 sec-

onds, which is significantly larger than the time required to explore any other transition, such as a user input. Currently, our implementation waits for up to 5 seconds after sending a user input, to ensure that all handlers have finished. We expect that with more support from the operating system, or a more involved implementation, we could reduce that wait time to well under a second.

Since the cost of exploring a transition to the initial state (a restart) is an order of magnitude more than the cost of any other transition, we must use an exploration and learning algorithm that minimizes the number of restarts. Standard regular language learning algorithms are not appropriate in this case. For example, Angluin's $\mathcal{L}^*$ [10] requires at least $O(n^2)$ restarts, where n is the number of states in the model of the user interface. Rivest and Schapire's algorithm [38] reduces the number of restarts to $O(n)$, which is still high, by computing homing sequences, and it also increases the runtime by a factor of n, which is again not acceptable when we want to achieve code coverage quickly.

In this paper we propose a testing algorithm based on two key observations:

1. It is possible to reduce the use of app restarts, because most user interface screens of an Android app can often be reached from other screens just by triggering a sequence of user inputs, e.g., using "back" or "home" buttons.
2. For the purpose of test generation, we do not need to learn an exact model of the app under test. All we need is an approximate model that could guide the generation of user inputs while maximizing the code coverage. Note that for real-world apps a finite state model of the GUI may not even exist. Some apps may require push-down model and others may require more sophisticated infinite models.

Based on these two observations, we propose a testing algorithm, called *SwiftHand*, that uses execution traces generated during the testing process to learn an approximate model of the GUI. *SwiftHand* then uses the learned model to choose user inputs that would take the app to previously unexplored states. As *SwiftHand* triggers the newly generated user inputs and visits new screens, it expands the learned model, and it also refines the model when it finds discrepancies between the model learned so far. The interplay between on-the-fly learning of the model and generation of user inputs based on the learned model helps *SwiftHand* to quickly visit unexplored states of the app. A key feature of *SwiftHand* is that, unlike other learning-based testing algorithms, it minimizes the number of restarts by searching for ways to reach unexplored states using only user inputs.

We have implemented *SwiftHand* for Android apps written in Java. The tool is publicly available at `https://github.com/wtchoi/SwiftHand`. We applied *SwiftHand* to several free real-world Android apps and compared the effectiveness of *SwiftHand* with that of random testing and $\mathcal{L}^*$-based testing. Our results show that *SwiftHand* can achieve significantly better coverage on these apps within a given time budget. Moreover, *SwiftHand* achieves branch coverage at a rate faster than that of random and $\mathcal{L}^*$-based testing. We report the results of our investigation in the empirical evaluation section.

2. Overview

In this section we introduce a motivating example, which we will use first to describe several existing techniques for automated user interface testing. Then we describe *SwiftHand* at a high level in the context of the same example. The formal details of the *SwiftHand* algorithm are in Section 3.

We use part of *Sanity*, an Android app in our benchmark-suite as our running example. Figure 1 shows the first four screens of the app. The app starts with three consecutive end-user license agreement (EULA) screens. To test the main app, an automated testing technique must pass all the three EULA screens.

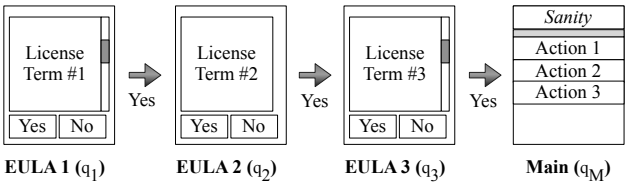


Figure 1: The first four screens of *Sanity* App. The screen names in parentheses are for cross-reference from the models of this app discussed later.

The first and the third EULA screens have four input choices: (a) **Yes** button to accept the license terms, (b) **No** button to decline the license terms, (c) **ScrollUp** the license terms, and (d) **ScrollDown** the license terms. Pressing **No** at any point terminates program. **ScrollDown** and **ScrollUp** do not change the non-visual state of the program. The second EULA screen doesn't have the scrolling option. Pressing **Yes** button three times leads the user to the main screen of the app. For convenience, in the remainder of this paper we are going to use short name q_1, q_2, q_3, and q_M, instead of EULA1, EULA2, EULA2, and Main.

2.1 Goals and Assumptions

We want to develop an algorithm that generates sequences of user inputs and feeds them to the app in order to *achieve high code coverage quickly*.

The design of our testing algorithm is guided by the following practical assumptions.

Testing Interface. We assume that it is possible to dynamically inspect the state of the running app to determine the set of user inputs enabled on a given app screen. We also assume that our algorithm can restart the app under test, can send a

user input to the app, and can wait for the app to become stable after receiving a user input.

Cost Model. We assume that restarting the app takes significantly more time than sending a user input and waiting for the app to stabilize. Note that a few complicated tasks are performed when an app is restarted: initializing a virtual machine, loading the app package, verifying the app byte-code, and executing the app's own initialization code. Testing tools have to wait until the initialization is properly done. Our experiments show that the restart operation takes 30 seconds. Sending a user input is itself very fast, but at the moment our implementation waits for up to 5 seconds for the app to stabilize. We expect that with a more complex implementation, or with some assistance from the mobile platform we can detect when the handlers have actually finished running. In this case we expect that the ratio of restart cost to the cost of sending a user input will be even higher.

User-Interface Model. We assume that an abstract model of the graphical user interface of the app under test is not available a priori to our testing algorithm. This is a reasonable assumption if we want to test arbitrary real-world Android apps.

When learning a user interface model we have to compare a user interface state with states that are already part of the learned model. For this purpose, we consider two user interface states equivalent if they have the same set of enabled user inputs. An enabled user input is considered according to its type and the bounding box of screen coordinates where it is enabled. This means that we do not care about the actual content of user elements such as colors or text content. This abstraction is similar to the one proposed by MacHiry et al. [25].

Test Coverage Criteria. We assume that the app under test is predominantly concerned with the user interface, and a significant part of the app state is reflected in the state of the user interface. Thus we assume that a testing approach that achieves good coverage of the user interface states also achieves good code coverage. This assumption is not always entirely accurate, e.g., for apps that have significant internal application state that is not exposed in the user interface.

2.2 Existing Approaches

Random Testing. Random testing [25] tests an app by randomly selecting a user input from the set of enabled inputs at each state and by executing the selected input. Random testing also restarts the app at each state with some probability. In *Sanity* case, after a restart, random testing has a low probability ($\frac{1}{2}*\frac{1}{2}*\frac{1}{2} = 0.125$) of reaching the main app screen. User inputs that do not change the non-visual state, for example **ScrollUp** and **ScrollDown** or clicking outside the buttons, do not affect this probability. Expected number of user inputs and restarts required to reach the main app screen are 24 and 7, respectively.[1] This will take about 330 seconds according to our cost model. In summary, random testing has a hard time in achieving good coverage if an interesting screen is reachable only after executing a specific sequence of user inputs. This is true for our example and is common in real-world apps.

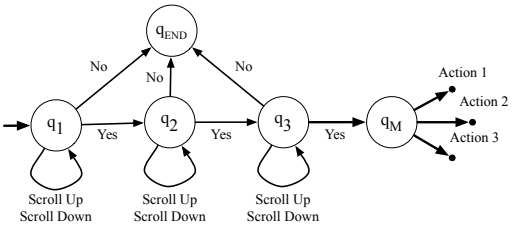


Figure 2: A partial model of Android app *Sanity*.

Note that this analysis, and our experiments, use a random testing technique that is aware of the set of enabled user inputs at each state and can make a decision based on this set. A naïve random testing technique, such as the widely-used Monkey tester, which touches random coordinates on the screen, will do nothing meaningful because most of the screen coordinates have no associated event handler.

Model-based Testing. Model-based testing [8, 11, 28, 37, 39, 43] is a popular alternative to automatically test GUI programs and other event-driven programs. Model-based testing assumes that a finite state model of the GUI is provided by the user. The idea behind model-based testing is to create an efficient set of user input sequences from the model of the target program. The generated test cases could either try to maximize coverage of states or try to maximize coverage of short sequences of user inputs.

Figure 2 is a partial model for the *Sanity* app. The model describes a finite state machine. A finite state machine abstracts the infinite state space of the app into a finite number of interesting states and describes equivalence classes of user input sequences leading to those states. A key advantage of using a finite state machine (FSM) model is that an optimal set of test cases can be generated based on a given coverage criterion.

[1] Let R be the expected number of restarts. At the first EULA screen, if **No** is chosen, the app will terminate and testing should be restarted. This case happens with probability $\frac{1}{4}$. The expected number of restarts is $R_1 = \frac{1}{4}(1 + R)$. If either **ScrollUp** or **ScrollDown** is picked, the app is still in the first EULA screen. Therefore, the expected number of restarts in this case is $R_2 = \frac{1}{2}R$. After considering two more cases (**Yes,No** and **Yes,Yes,Scroll*****,No**), we can construct an equation of R.

$$\begin{aligned} R &= R_1 + R_2 + R_3 + R_4 \\ &= \tfrac{1}{4}(1 + R) + \tfrac{1}{2}R + \tfrac{1}{8}(1 + R) + \tfrac{1}{16}(1 + R) \\ &= \tfrac{7}{16} + \tfrac{15}{16}R \end{aligned}$$

Solving the equation, we have $R = 7$. We can perform a similar analysis to get the expected number of all events including restarts, which is 31. The number of events except restarts is 24.

For our running example, if we want to avoid a restart, a model-based testing algorithm could generate the sequence **ScrollDown**, **ScrollUp**, **Yes**, **Yes**, **ScrollDown**, **ScrollUp**, **Yes** to obtain full coverage of non-terminating user inputs and to lead the test execution to the main screen.

Model-based testing can generate optimal test cases for GUI apps, if the model is finite and accurate. Unfortunately, it is a non-trivial task to manually come up with a precise model of the GUI app under test. For several real-world apps, a finite model may not even exist. Some apps could require a push-down automaton or a Turing machine as a model. Moreover, manually generated models may miss transitions that could be introduced by a programmer by mistake.

Testing with Active Learning. Testing with model learning [18, 34, 35] tries to address the limitations of model-based testing by learning a model of the app as testing is performed. An active learning algorithm is used in conjunction with a testing engine to learn a model of the GUI app and to guide the generation of user input sequences based on the model.

A testing engine is used as a teacher in active learning. The testing engine executes the app under test to answer two kinds of queries: 1) membership queries—whether a sequence of user inputs is *valid* from the initial state, i.e. if the user inputs in the sequence can be triggered in order, and 2) equivalence queries—whether a learned model abstracts the behavior of the app under test. The testing engine resolves equivalence queries by executing untried scenarios until a counter-example is found. An active learning algorithm repeatedly asks the teacher membership and equivalence queries to infer the model of the app.

A case study: $\mathcal{L}^*$. Angluin's $\mathcal{L}^*$[10] is the most widely used active learning algorithm for learning finite state machine. The algorithm has been successfully applied to various problem domains from network protocol inference to functional confirmation testing of circuits.

We applied $\mathcal{L}^*$ to the running example. We observed that $\mathcal{L}^*$ restarts frequently. Moreover, $\mathcal{L}^*$ made a large number of membership queries to learn a precise and minimal model. Specifically, testing with $\mathcal{L}^*$ required 29 input sequences (i.e. 29 restarts) consisting of 64 user inputs to fully learn the partial model in Figure 2. This translates to spending around 870 seconds to restart the app under test (AUT) and 320 seconds for executing the user inputs. 73% of running time is spent on restarting the app. It is important to note that $\mathcal{L}^*$ has to learn the partial model completely and precisely before it can explore the screens beyond the main screen. We show in the experimental evaluation section that $\mathcal{L}^*$ has similar difficulties in actual benchmarks.

2.3 Our Testing Algorithm: *SwiftHand*

SwiftHand combines active learning with testing. However, unlike standard learning algorithms such as $\mathcal{L}^*$, *SwiftHand* restarts the app under test sparingly. At each state, instead of restarting, *SwiftHand* tries to extend the current execution path by selecting a user input enabled at the state. *SwiftHand* uses the model learned so far to select the next user input to be executed.

Informally, *SwiftHand* works as follows. *SwiftHand* installs and launches the app under test and waits for the app to reach a stable state. This is the initial *app-state*. For each app-state, we compute a *model-state* based on the set of enabled user inputs in the app-state along with the bounding boxes of screen coordinates where they are enabled. Initially the model contains only one state, the model-state corresponding to the initial app-state.

If a model-state has at least one unexplored outgoing transition, we call it a *frontier model-state*. At each app-state s, *SwiftHand heuristically picks* a frontier model-state q that can be reached from the current model-state without a restart.

Case 0. If such a state is not found, *SwiftHand* restarts the app. This covers both the case when a frontier-state exists but is not reachable from the current state with user inputs alone, and also the case when there are no frontier-states, in which case our algorithm restarts the app to try to find inconsistencies in the learned model by exploring new paths.

Otherwise, *SwiftHand* avoids restart by *heuristically finding a path* in the model from the current model-state to q and executes the app along the path. *SwiftHand* then executes the unexplored enabled input of state q. Three scenarios can arise during this execution.

Case 1. The app-state reached by *SwiftHand* has a corresponding model-state that has not been encountered before. *SwiftHand* adds a fresh model-state to the model corresponding to the newly visited app-state and adds a transition to the model.

Case 2. If the reached app-state is equivalent based on enabled user inputs to the screen of a previously encountered app-state, say s', then *SwiftHand* adds a transition from q to the model-state corresponding to s'. This is called state *merging*. If there are multiple model-states whose corresponding app-states have equivalent screens, then *SwiftHand picks one of them heuristically for merging*.

Case 3. During the execution of the user inputs to reached the frontier state, *SwiftHand* discovers that an app-state visited during the execution of the path does not match the corresponding model-state along the same path in the model. This is likely due to an earlier merging operation that was too aggressive. At this point, *SwiftHand* runs a passive learning algorithm using the execution traces observed so far, i.e. *SwiftHand* finds the smallest model that can explain the app-states and transitions observed so far.

Note that *SwiftHand* applies heuristics at three steps in the above algorithm. We discuss these heuristics in Section 3.2.

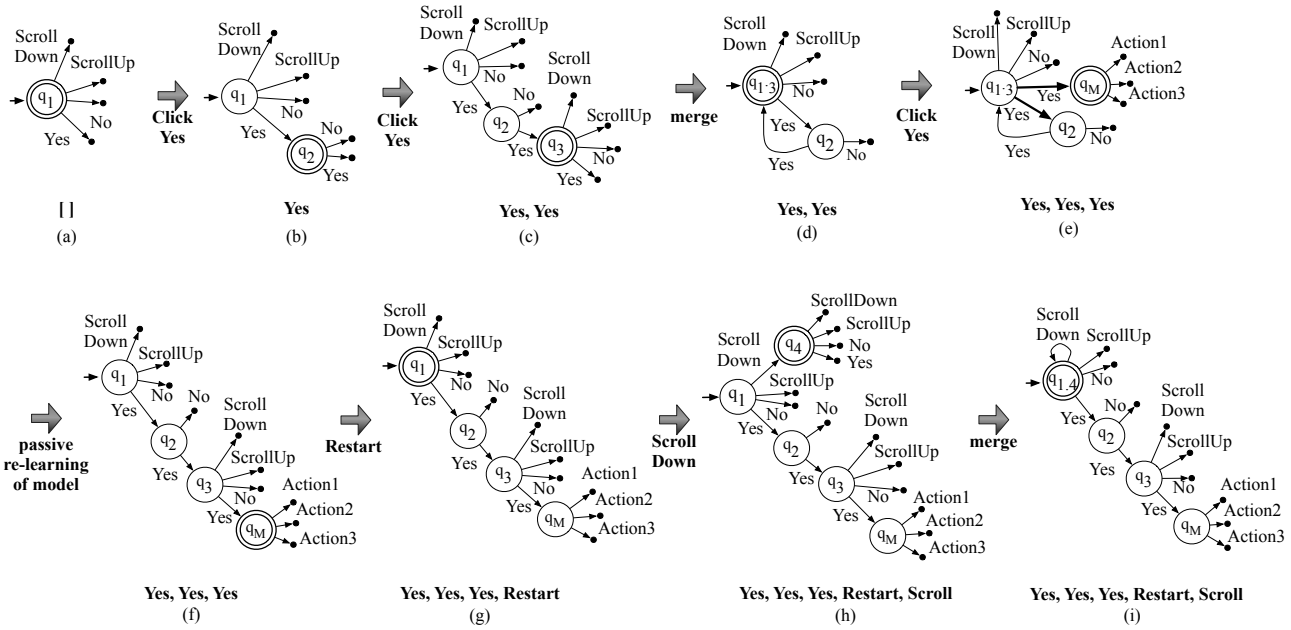


Figure 3: Progress of learning guided testing on *Sanity* example. A circle with solid line is used to denote a state in the model. The initial state is marked with a short incoming arrow. A circle with double line denotes the current model-state.

If the target model is finite, *SwiftHand* will be able to learn the target model irrespective of what heuristics we use at the above three decision points. However, our goal is not necessarily to learn the exact model, but to achieve coverage quickly. Therefore, we apply heuristics that enable *SwiftHand* to explore previously unexplored states quickly. In order to avoid *SwiftHand* from getting stuck in some remote part of the app, we allow *SwiftHand* to restart when it has executed a predefined number of user inputs from the initial state.

Figure 3 illustrates how *SwiftHand* works on the running example. For this example, we pick user inputs carefully in order to keep our illustration short, yet comprehensive. For this reason we do not pick the **No** user input to avoid restart. In actual implementation, we use various heuristics to make such decisions. A model-state with at least one unexplored outgoing transition is called a *frontier* state. A solid line with arrow denotes a transition in the model. The input sequence shown below the diagram of a model denotes an input sequence of the current execution from the initial state.

- *Initialization*: After launching the app, we reach the initial app-state, where the enabled inputs on the state are {**Yes**, **No**, **ScrollUp**, **ScrollDown**}. We abstract this app-state as the model-state q_1 (using the same terminology as in Figure 1 and Figure 2). This initial state of the model is shown in Figure 3(a).
- *1st Iteration*: Starting from the initial state of the model, *SwiftHand* finds that the state q_1 is a frontier state and chooses to execute a transition for the input **Yes**. The resulting state has a different set of enabled inputs from the initial state. Therefore, according to Case 1 of the algorithm, *SwiftHand* adds a new model-state q_2 to the model. The modified model is shown in Figure 3(b).
- *2nd Iteration*: The app is now at an app-state whose corresponding model-state is q_2, as shown in Figure 3(b). Both q_1 and q_2 have unexplored outgoing transitions. However, if we want to avoid restart, we can only pick a transition from q_2 because according to the current model there is no sequence of user inputs to get to q_1 from the current model state. *SwiftHand* chooses to execute a transition on **Yes**, and obtains a new app-state for which it creates a new model-state q_3, as shown in Figure 3(c). However, the new app-state has the same set of enabled inputs as the initial app-state q_1. Therefore, *SwiftHand* merges q_3 with q_1 according to Case 2. This results in the model shown in Figure 3(d). If you compare the partial model learned so far with the actual underlying model shown in Figure 2, you will notice that the merging operation is too aggressive. In the actual underlying model the state reached after a sequence of two **Yes** clicks is different than the initial state. This will become apparent to *SwiftHand* once it explores the app further.
- *3rd Iteration*: The app is now at an app-state whose corresponding model-state is q_1, as shown in Figure 3(d). *SwiftHand* can now pick either q_1 or q_2 as the next frontier state to explore. Assume that *SwiftHand* picks q_2 as the frontier state to explore. A path from the cur-

rent model-state q_1 to q_2 consists of a single transition **Yes**. After executing this input, however, *SwiftHand* encounters an inconsistency—the app has reached the main screen after executing **Yes**,**Yes**,**Yes** sequence from the initialization (see Figure 2). In the current model learned so far (Figure 3(d)), the abstract state after the same sequence of events ought to be q_2. Yet the set of enabled inputs associated with this screen (**Action1**, **Action2**, and **Action3**) is different from the set of enabled inputs associated with q_2.

We say that *SwiftHand* has discovered that the model learned so far is inconsistent with the app, and the input sequence **Yes**,**Yes**,**Yes** is a counter-example showing this inconsistency. Figure 3(e) illustrates this situation; notice that there are two outgoing transitions labeled **Yes** from state q_2. This is Case 3 of the algorithm. The inconsistency happened because of the merging decision made at the second iteration. To resolve the inconsistency, *SwiftHand* abandons the model learned so far and runs an off-the-shelf passive learning algorithm to rebuild the model from scratch using all execution traces observed so far. Figure 3(f) shows the result of passive learning. Note that this learning is done using the transitions that we have recorded so far, without any transition in the actual app.

- *4th Iteration*: *SwiftHand* is forced to restart the app when it has executed a predefined number of user inputs from the initial state. Assume that *SwiftHand* restarts the app in the 4th iteration so that we can illustrate another scenario of *SwiftHand*. After restart, q_1 becomes the current state (see Figure 3(g)). *SwiftHand* now has several options to execute an unexplored transition. Assume that *SwiftHand* picks **ScrollDown** transition out of q_1 state for execution. Since scrolling does nothing to the screen state and the set of enabled inputs, we reach an app-state that has the same screen and enabled inputs as q_1 and q_3. *SwiftHand* can now merge the new model-state with either q_1 or q_3 (see Figure 3(h)). In practice, we found that the nearest ancestor or the nearest state with the same set of enabled inputs works better than other model states. We use this heuristics to pick q_1. The resulting model at this point is shown in Figure 3(i).

After the first four iterations, *SwiftHand* will execute 4 restarts and 11 more user inputs, in the worst case, to learn the partial model in Figure 2. The restarts are necessary to learn the transitions to the terminal state, `End`. If *SwiftHand* wants to explore states beyond the main screen after a restart, it can consult the model and execute the input sequence **Yes**, **Yes**, and **Yes** to reach the main screen. Random testing will have a hard time reaching the main screen through random inputs. In terms of our cost model, *SwiftHand* will spend 190 seconds or 60% of the execution time in restarting the app. The percentage of time spent in restarting drops if the search space becomes larger. In our actual benchmarks, we observed that *SwiftHand* spent about 10% of the total time in restarting.

3. Learning guided Testing Algorithm

In this sections, we describe formally the *SwiftHand* algorithm. We first introduce a few definitions that we use in our algorithm. Then we briefly describe the algorithm. *SwiftHand* uses a variant of an existing passive learning algorithm to refine a model whenever it observes any inconsistency between the learned model and the app. We describe this passive learning algorithm to keep the paper self-contained.

Models as ELTS. We use *extended deterministic labeled transition systems* (ELTS) as models for GUI apps. An ELTS is a deterministic labeled transition system whose states are labeled with a set of enabled transitions (or user inputs). Formally, an ELTS M is a tuple

$$M = (Q, q_0, \Sigma, \delta, \lambda)$$

where

- Q is a set of states,
- $q_0 \in Q$ is the initial state,
- Σ is an input alphabet,
- $\delta : Q \times \Sigma \to Q$ is a partial state transition function,
- $\lambda : Q \to \wp(\Sigma)$ is a state labeling function. $\lambda(q)$ denotes the set of inputs enabled at state q, and
- for any $q \in Q$ and $a \in \Sigma$, if there exists a $p \in Q$ such that $\delta(q, a) = p$, then $a \in \lambda(q)$.

The last condition implies that if there is a transition from q to some p on input a, then a is an enabled input at state q.

Arrow. We use $q \xrightarrow{a} p$ to denote $\delta(q, a) = p$.

Arrow*. We say $q \xrightarrow{l} p$ where $l = a_1, \ldots, a_n \in \Sigma^*$ if there exists $q_1, \ldots, q_{n-1}$ such that $q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \ldots q_{n-1} \xrightarrow{a_n} p$.

Trace. An execution trace, or simply a trace, is a sequence of pairs of inputs and sets of enabled inputs. Formally, a trace t is an element of $(\Sigma \times \wp(\Sigma))^*$.

Trace Projection. We use $\pi(t)$ to denote the sequence of inputs in the trace t. Formally, if $t = (a_1, \Sigma_1), \ldots, (a_n, \Sigma_n)$, then $\pi(t) = a_1, \ldots, a_n$.

Arrow Trace. We say $q \xrightarrow{t} p$ if $q \xrightarrow{\pi(t)} p$.

Consistency. A trace $t = (a_1, \Sigma_1), \ldots, (a_n, \Sigma_n)$ is consistent with a model $M = (Q, q_0, \Sigma, \delta, \lambda)$ if and only if

$$\exists q_1, \ldots q_n \in Q. \bigwedge_{i \in [1,n]} q_{i-1} \xrightarrow{a_i} q_i \wedge \lambda(q_i) = \Sigma_i$$

Frontier State. A state in an ELTS is a *frontier-state* if there is no transition from the state on some input that is enabled on the state. Formally, a state q is a frontier-state if there exists a $a \in \lambda(q)$ such that $q \xrightarrow{a} p$ is not true for any $p \in Q$.

Terminal State. When the execution of an app terminates, we assume that it reaches a state that has no enabled inputs.

3.1 Learning Guided Testing Algorithm

Interface with the App under Test. *SwiftHand* treats the app state as a black box. However, it can query the set of enabled inputs on an app state. We assume that $\lambda(s)$ returns the set of enabled inputs on an app state s. *SwiftHand* can also ask the app to return a new state and trace after executing a sequence of user inputs from a given app state. Let s be an app state, t be a trace of executing the app from the initial state s_0 to s, and l be a sequence of user inputs. Then $\text{EXECUTE}(s, t, l)$ returns a pair containing the app state after executing the app from state s on the input sequence l and the trace of the entire execution from the initial state s_0 to the new state.

Description of the Actual Algorithm. The pseudo-code of the algorithm is shown in Algorithm 1. The algorithm maintains five local variables: 1) s denotes the current app-state and is initialized to s_0, the initial state of the app, 2) p denotes the current model-state, 3) t denotes the current trace that is being executed, 4) T denotes the set of traces tested so far, and 5) M denotes the ELTS model learned so far.

At each iteration the algorithm tries to explore a new app state. To do so, it finds a frontier-state q in the model, then finds a sequence of transitions l that could lead to the frontier-state from the current model-state, and a transition a enabled at the frontier-state (lines 9–10). It then executes the app on the input sequence l from the current app state s and obtains a trace of the execution (line 11). If the trace is not consistent with the model, then we know that some previous merging operation was incorrect and we re-learn the model using a passive learning algorithm from the set of traces observed so far (lines 21–24). On the other hand, if the trace is consistent with the model learned so far, the algorithm executes the app on the input a from the latest app state (lines 12–13) . If the set of enabled inputs on the new app state matches with the set of enabled inputs on an existing model-state (line 14), then it is possible that we are revisiting an existing model-state from the frontier state q on input a. The algorithm, therefore, merges the new model-state with the existing model-state (line 15). Note that this is an approximate check of equivalence between two model-states, but it helps to prune the search space. If the algorithm later discovers that the two states are not equivalent, it will do a passive learning to learn a new model, effectively undoing the merging operation. Nevertheless, this aggressive merging strategy is key to prune similar states and guide the app into previously unexplored state space. On the other hand, if the algorithm finds that the set of enabled inputs on the new app-state is not same as the set of enabled inputs of any existing model-state, then we have visited a new app-state. The algorithm then adds a new model-state corresponding to the app-state to the model (line 18). In either case, that is whether we merge or we add a new model-state, we update our current model-state to the model-state corresponding to the new app state and repeat the iteration (lines 16 and 19).

During the iteration if we fail to find a frontier state, we know that our model is *complete*, i.e. every transition from every model-state has been explored. However, there is a possibility that some incorrect merging might have happened in the process. We, therefore, need to now confirm that the model is equivalent to the target model of the app. This is similar to the equivalence check in the $\mathcal{L}^*$ algorithm. The algorithm picks a sequence of transitions l such that l is not a subsequence of any trace that has been explored so far (lines 28–30). Moreover, l should lead to a state q from the current state in the model. If such a l is not found, we know that our model is equivalent to the target model (lines 35–36). On the other hand, if such an l exists, the algorithm executes the app on l and checks if the resulting trace is consistent with the model (lines 30–31). If an inconsistency is found, the model is re-learned from the set of traces executed so far (lines 32–33). Otherwise, we continue the refinement process over any other existing l.

During the iteration, if the algorithm finds a frontier state, but fails to find a path from the current state to the frontier state in the model, it restarts the app (lines 25–26). We observed that in most GUI apps, it is often possible to reach most screen states from another screen state after a series of user inputs while avoiding a restart. As such, this kind of restart is rare in *SwiftHand*. In order to make sure that our testing does not get stuck in some sub-component of an app with a huge number of model states, we do restart the app if the length of a trace exceeds some user defined limit (i.e. MAX_LENGTH) (lines 7–8).

3.2 Heuristics and Decision Points

The described algorithm has four decision points and we use the following heuristics to resolve them. We came up with these heuristics after extensive experimentation.

- If there are multiple frontier-states and if there are multiple enabled inputs on those frontier-states (lines 9–10), then which frontier-state and enabled transition should we pick? *SwiftHand* picks a random state from the set of frontier-states and picks a random transition from the frontier state. We found through experiments that effectiveness of *SwiftHand* does not depend on what heuristics we use for this case.
- If there are multiple transition sequences l from the current model-state to a frontier state (line 10), which one should we pick? We found that picking a short sequence

Algorithm 1 *SwiftHand*: Learning Guided Testing

```
1: procedure TESTING(s_0)                                   ▷ s_0 is the initial state of the app
2:     M ← ({q_0}, q_0, Σ, ∅, {q_0 ↦ λ(s_0)}) for some fresh state q_0   ▷ M is the current model
3:     T ← ∅                                                ▷ T accumulates the set of traces executed so far
4:     p ← q_0 and s ← s_0 and t ← ϵ
5:                                                          ▷ p, s, and t are the current model-state, app-state, and trace, respectively
6:     while ¬timeout() do                                  ▷ While time budget for testing has not expired
7:         if |t| > MAX_LENGTH then                         ▷ Current trace is longer than a maximum limit
8:             p ← q_0 and s ← s_0 and T ← T ∪ {t} and t ← ϵ   ▷ Restart the app
9:         else if there exists a frontier-state q in M then   ▷ Model is not complete yet
10:            if there exists l ∈ Σ* and a ∈ Σ such that p -l-> q and a ∈ λ(q) then
11:                (s, t) ← EXECUTE(s, t, l)
12:                if t is consistent with M then
13:                    (s, t) ← EXECUTE(s, t, a)
14:                    if there exists a state r in M such that λ(r) = λ(s) then   ▷ Merge with an existing state
15:                        add q -a-> r to M
16:                        p ← r                            ▷ Update current model-state
17:                    else                                 ▷ Add a new model-state
18:                        add a fresh state r to M such that q -a-> r and λ(r) = λ(s)
19:                        p ← r                            ▷ Update current model-state
20:                    end if
21:                else                                     ▷ Inconsistent model. Need to re-learn the model.
22:                    T ← T ∪ {t} and M ← PASSIVELEARN(T)
23:                    p ← r where q_0 -t-> r               ▷ Update current model-state
24:                end if
25:            else                                         ▷ A frontier-state cannot reached from the current state
26:                p ← q_0 and s ← s_0 and T ← T ∪ {t} and t ← ϵ   ▷ Restart the app
27:            end if
28:        else if there exists state q in M and l ∈ Σ* such that p -l-> q and l is not a subsequence of π(t) for any t ∈ T then
29:                                                         ▷ Model is complete, but may not be equivalent to target model
30:            (s, t) ← EXECUTE(s, t, l)
31:            if t is not consistent with M then           ▷ Inconsistent model. Need to re-learn the model.
32:                T ← T ∪ {t} and M ← PASSIVELEARN(T)
33:                p ← r where q_0 -t-> r                   ▷ Update current model-state
34:            end if
35:        else                                             ▷ Model is complete and equivalent to target model
36:            return T                                     ▷ Done with learning
37:        end if
38:    end while
39:    return T
40: end procedure
```

is not necessarily the best strategy. Instead, *SwiftHand* selects a sequence of transitions from the current model state to the frontier state so that the sequence contains a previously unexplored sequence of inputs. This helps *SwiftHand* to learn a more accurate model early in the testing process.

- If multiple states are available for merging (at line 14), then which one should *SwiftHand* pick? If we pick the correct model-state, we can avoid re-learning in future. We experimented with a random selection strategy and with a strategy that selects a nearby state. However, we discovered after some experimentation that if we prefer the nearest ancestor to other states, our merge operations are often correct. Therefore, *SwiftHand* uses a heuristics that first tries to merge with an ancestor. If an ancestor is not available for merging, *SwiftHand* picks a random state from the set of candidate states.
- If there are multiple transition sequences l available for checking equivalence (line 28), which one should we pick? In this case, we again found that none of the strate-

gies we tried make a difference. We therefore use random walk to select such an l.

- We set the maximum length of a trace (i.e. MAX_LENGTH) to 50. We again found this number through trial-and-error.

3.3 Rebuilding Model using Passive Learning.

We describe the passive learning algorithm that we use for re-learning a model from a set of traces. The algorithm is a variants of Lambeau et al.'s [21] state-merging algorithm. We have modified the algorithm to learn ELTS. We describe this algorithm to keep the paper self-contained.

Prefix Tree Acceptor. A prefix tree acceptor [9] (or a PTA) is an ELTS whose state transition diagram is a tree with the initial state of the ELTS being the root of the tree. Given a set of traces T, we build a prefix tree acceptor PTA_T whose states are the set of all prefixes of the traces in T. There is a transition with label a from t to t' if t can be extended to t' using the transition a. The $\lambda(t)$ is Σ' if the last element of t has Σ' as the second component.

Partitioning and Quotient Model. $\Pi \subseteq \wp(Q)$ is a partition of the state space Q if all elements of Π are disjoint, all elements of Q are a member of some element of Π, and λ of all elements of a given element of Π are the same. An element of Π is called a partition and is denoted by π. $\texttt{element}(\pi)$ denotes a random single element of π. M/Π is a quotient model of M obtained by merging equivalent states with respect to Π:

$$\frac{\begin{array}{c}\pi_0 \text{ is the partition containing } q_0 \\ \delta' \stackrel{\text{def}}{=} \{(\pi, a) \mapsto \pi' \mid \exists q \in \pi \text{ and } \exists q' \in \pi' .(q, a) \mapsto q' \in \delta\} \\ \forall \pi \in \Pi.\lambda'(\pi) \stackrel{\text{def}}{=} \lambda(\texttt{element}(\pi))\end{array}}{M/\Pi = (\Pi, \pi_0, \Sigma, \delta', \lambda')}$$

Note that a quotient model can be non-deterministic even though the original model is deterministic.

The Algorithm. Algorithm 2 describes the state-merging based learning algorithm. Conceptually, the algorithm starts from a partial tree acceptor PTA_T and repeatedly generalizes the model by merging states. The merging procedure first checks whether any two states agree on the λ function, and then tries to merge them. If merging results in a non-deterministic model, the algorithm tries to eliminate non-determinism by merging target states of non-deterministic transitions provided that the merged states have the same λ. This is applied recursively until the model is deterministic. If at some point of the procedure, merging of two states fails because λs of the states are different, the algorithm unrolls the entire merging process for the original pair of states.

The `ChoosePair` function decides the order of state merging. The quality of the learned model solely depends on the implementation of this function. Our implementation uses BlueFringe [22] ordering, which is considered as the best known heuristics.

Algorithm 2 Passive Learning Algorithm

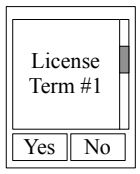
```
1: procedure REBUILD(T)
2: |  Π ← {{q} | q ∈ Q_{PTA_T}}
3: |  while (π_i, π_j) ← ChoosePair(PTA_T) do
4: |  |  Try
5: |  |  |  Π ← MERGE(Π, π_i, π_j)
6: |  |  CatchAndIngnore
7: |  end while
8: |  return PTA_T/Π
9: end procedure

10: procedure MERGE(Π, π_i, π_j)
11: |  M ← PTA_T/Π
12: |  if λ_M(π_i) ≠ λ_M(π_j) then
13: |  |  throw exception
14: |  end if
15: |  π_pivot ← π_i ∪ π_j
16: |  Π ← (Π \ {π_i, π_j}) + π_pivot
17: |  while (π_k, π_l) ← FINDNONDETER(Π, π_pivot) do
18: |  |  Π ← MERGE(Π, π_k, π_l)
19: |  end while
20: |  return Π
21: end procedure

22: procedure FINDNONDETER(Π, π)
23: |  M ← PTA_T/Π
24: |  S ← {(π_i, π_j) | ∃a ∈ λ_M(π).π -a->_M π_i ∧ π -a->_M π_j ∧ π_i ≠ π_j}
25: |  return pick(S)
26: end procedure
```

Our algorithm differs from the original algorithm on two fronts. First, the original algorithm [21] aims to learn DFA with mandatory merging constraints and blocking constraints. Our algorithm learns an ELTS and only uses the idea of blocking constraints. We use the λ function or the set of enabled transitions at any state to avoid illegal merging. Second, DFA learning requires both positive and negative examples. ELTS has no notion of negative examples.

4. Implementation

We have implemented *SwiftHand* for Android apps written in Java. The tool is publicly available at `https://github.com/wtchoi/SwiftHand`. The tool itself is implemented using Java and Scala. *SwiftHand* uses `asmdex` [31], a *Dalvik* byte code instrumentation library, `axml` [4], an xml encoded binary manipulation library, and `chimpchat`, an Android app remote control library. *SwiftHand* can test an Android app either on an emulator or an Android phone connected to a computer using ADB (Android Debug Bridge). *SwiftHand* expects a target app package file (*Apk*), a testing strategy, a time budget for testing, and a device on which the target app will be tested as input. It currently supports three strategies: *Random*, $\mathcal{L}^*$, and *SwiftHand*.

For basic app control, such as restarting, sending a user input, installing and removing the app, *SwiftHand* uses the `chimpchat` library. The same library has been used to implement the `monkeyrunner` remote testing framework. The `chimpchat` library is able to control any device connected to the computer through ADB.



(a) EULA screen

DecorView(1)
\+ *LinearLayout*(2)
| + *ScrollableLayout*(3)
| | + *TextBox*(4)
| | *LinearLayout*(5)
| | + *Button*[Yes](6)
| | | *Button*[No](7)

(b) GUI Component Tree

{
Touch(6),
Touch(7),
ScrollDown(3),
ScrollUp(3)
}

(c) Enabled Inputs

Figure 4: Enabled Input Inference Example

Local clean restart can be implemented by sequentially uninstalling, reinstalling, and starting a target app. Note that the Android file system is sandboxed. Removing an app is enough to cleanse the majority of local persistent data. The only exception is an SD card. SD cards are treated as a shared storage. Therefore, removing an application will not remove persistent data from SD card. For simplicity, we choose to use devices without an SD card. A full local cleaning may require checkpointing and restoring the SD card content.

`chimpchat` has two limitations. First, `chimpchat` can only send device level events: to touch a specific coordinate of the screen, to tilt the device, etc. Thus, *SwiftHand* has to infer coordinates from screen information to construct the `chimpchat` requests. Second, `chimpchat` can only send events without waiting for the results. It will never tell programmers that an event-handling is done or an app is terminated.

SwiftHand overcomes these limitations by binary instrumentation. The binary instrumentation basically does two things: First, an app is modified to record necessary runtime information throughout its execution. Second, the instrumented app launches an observer thread when the app is starting. The observer thread periodically checks recorded information and reports to the *SwiftHand* tool running on a separate computer.

The remainder of this section explains how specific information —such as the end of a state transition —is collected through binary instrumentation, how binary instrumentation is implemented, and what limitations remain in the current *SwiftHand* implementation.

4.1 Inferring Enabled Inputs

The exact information about an app's screen is available at runtime as a GUI component tree. The GUI component tree represents the hierarchical structure of GUI components on the screen of the app. Coordinates, size, and type information of each GUI component is also available. *SwiftHand* instruments the target apps to obtain the GUI component tree and computes a representative set of enabled inputs by traversing the tree. If a leaf in the GUI component tree has an event handler, a touch input corresponding to the event handler is added to the set of enabled inputs. If the GUI component has type *EditText*, then a user input capable of generating a pre-defined or random string is added to the set of enabled inputs. The input events for a given GUI component remain the same across executions. For scrollable components, scroll inputs are added to the set of enabled inputs. Inputs corresponding to pressing **Back** and **Menu** buttons, are always added to the enabled input set.

Figure 4 shows an EULA screen from the *Sanity* example, its simplified GUI component tree, and the corresponding enabled inputs. In the GUI component tree, each component is described with its type and identifier. *ScrollableLayout* can be scrolled. Therefore, we add **ScrollDown**(3) and **ScrollUp**(3) to the enabled input set. *TextBox* is a view component without any event handler. We add `touch` events for **Button**s with id 6 and 7 to the enabled input set because they are leaf components and each one has an associated event handler.

To actually fire an event, *SwiftHand* gets the coordinate and size information of the target GUI component from the GUI component tree, and passes this information to the chimpchat library.

4.2 App State Detection

The *SwiftHand* algorithm needs to detect the end of a state transition and the termination of the app.

Detecting App Termination. An Android app is composed of a set of activities. Each activity implements a single application screen and its functionality. When an app is terminated, all activities are stopped. Thus, the problem of checking app termination boils down to the problem of tracking the status of app's activities. The observer thread has an activity-state tracking mechanism and can detect the app termination by periodically checking tracked activity states.

Our activity state tracking mechanism is based on the Android Activity Lifecycle document [1]. According to the Activity Lifecycle, activities have six conceptual states: created, started, resumed, paused, stopped, and destroyed. When an activity state is changing, the Android framework triggers a fixed group of event-handlers in a fixed order. *SwiftHand* tracks the current activity state using a finite state machine. There is a separate state machine for each activity-class instance. Event handlers involved in the Activity Lifecycle are instrumented to trigger a state transition of a corresponding state machine.

Detecting the End of a State Transition. When a state transition happens in Android apps a number of event handlers are executed and the screen content is modified. After sending a command, *SwiftHand* must wait until all event handler executions and screen modifications are finished.

The end of a single event-handler execution can be detected by observing when the call-stack of the event handler becomes empty. However, a single event can trigger multiple event handlers. Therefore, *SwiftHand* waits a while after detecting the end of a single event handler execution to

make sure that no event handler is waiting. In experiments we found that 200 milliseconds are enough for real phones and 1000 milliseconds are enough for emulators.

In the Android framework an event handler could modify the screen content using animation effects. Since *SwiftHand* needs coordinate information of GUI components to trigger events through `chimpchat`, it must wait for the screen coordinates to stabilize. This can be done by periodically checking coordinate and size information of GUI components. If there is no change for a while, *SwiftHand* concludes that the screen is stable. In experiments we use 1100 milliseconds for real phones and 1800 milliseconds for emulators.

The overall 5 seconds wait interval that we use in the cost model is the result of having to wait for one or more event handlers and the animations that these handlers may start. With a more complex implementation, and perhaps some assistance from the platform, we could detect more accurately when a user input has been fully processed, without such long conservative wait intervals.

4.3 Instrumentation

To instrument the app byte code, we use the `asmdex` library. The library provides only parsing and byte code generation functionalities. We implemented all other intermediate steps: control-flow graph (CFG) construction, live register analysis, register type inference, and code injection with free-register acquisition. The CFG construction and live register analysis are standard. Type inference is based on Leroy's Java bytecode verification [24].

Code Injection and Free Registers. The *Dalvik* virtual machine (VM) is a register based machine. The biggest hurdle for code injection is acquiring free registers.

Dalvik VM has following policies to manage registers:

1. Registers are identified by a number from 0 to 65535.
2. Each method has to declare the upper bound of register identifiers.
3. When a method is invoked, arguments are assigned to the higher most registers.
4. Each instruction can accept only a limited range of registers as operands; some accept registers 0 to 15, another accepts 0 to 255, and the others accept the entire range of registers.
5. Registers must have a consistent type through the program's control flow, i.e., at every control flow join point, a register should have a consistent type.

Code injection requires free registers. Free registers are not always available due to the second policy and the fact that live registers at the target injection point cannot be used. To relax this situation, *SwiftHand* increases the upper bound of register range and adds a few instructions to move the argument registers to their original position. This creates a number of free registers at the highest position of the register range. Note that the second step is crucial due to the third and the fourth policies: Assume a method with register upper bound 15 and two arguments. Arguments are originally assigned to register 14 and 15. If the upper bound is increased by one, the register assigned to the arguments become 15, 16. Unfortunately, some binary instructions cannot accept the last argument register. Method call instruction, one of the most frequently used instructions, is in this category. Restoring argument registers to their original position solve this particular problem without seriously modifying the method body.

However, having a free register is not enough due to the fourth policy. If a method has the upper bound larger then 15, for example, the acquired free resisters are useless for certain classes of instructions. *SwiftHand* solves this problem by register swapping: adding instructions to swap out a content of low registers to free high registers, injecting the main code utilizing the newly acquired free low registers, and adding instructions to restore the original content of the low registers after after the main code.

More interestingly, not all registers can be swapped out due to the fifth policy and the exception mechanism. *Dalvik* VM supports a try/catch style exception mechanism. *Dalvik* VM considers that every instruction in a try block can raise an exception. In terms of program control flow, the first instruction of a catch block is a direct successor for all instructions in the related try block. Thus, registers used in the catch block should have a single type throughout the entire try block. They are not reusable. *SwiftHand* considers that these registers are constrained and avoids using them. We modified the live-register analysis to additionally calculate groups of constrained registers. This heuristic will not work if all registers in an instruction's operand range are constrained. Fortunately, we have not faced such an extreme situation.

Note that instrumentation can be performed even with a single unconstrained register. This can be done in four steps: (a) Swap out the unconstrained register. (b) Move all values to be used into a globally shared data structure one by one. A static method moving one value at a time, which requires only one low low register to invoke, could be used. (c) Invoke the static method containing the actual instructions to be injected. At the beginning of the method, values should be loaded from the globally shared data structure and casted into their original types. (d) Restore the original content of the unconstraint register.

APK Management. Android apps are distributed as a single *Apk* package file. To perform instrumentation, *SwiftHand* needs to extract a program binary from a package file and inject the modified binary to the package file.

Apk file is a zip archive having a predefined directory structure. The archive contains a `class.dex` file, a `Manifest.xml` file, a `META-INF` directory, and other resource files. `class.dex` file contains *Dalvik* byte code to be

executed. `Manifest.xml` is a binary encoded xml file describing various parameters such as main activity and privilege setting. `META-INF` directory includes signature of the app.

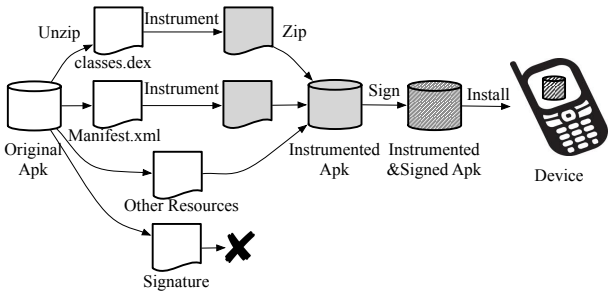


Figure 5: Flow of App Modification Process

Figure 5 shows the flow of the app modification process. First, the app package is unpacked. Then `class.dex` and `Manifest.xml` files are instrumented. The `Manifest.xml` file has to be modified because the observer thread in the instrumented app needs network access privilege to communicate with the *SwiftHand* tool. `META-INF` directory is removed since the original signature conflicts with the modified package. After the modification, the app is repacked, signed, and delivered to experiment devices. For testing purpose, the key for the resigned signature does not need to be identical to the key for the original signature.

4.4 Limitations

The current *SwiftHand* implementation has three limitations. First, the current implementation does not support apps whose main entry routine is native. Several games fall under this category and we have excluded them from our benchmark-suite.

Second, the current implementation works correctly only with devices with Android 4.1 or higher versions.

Third, *SwiftHand* cannot handle apps that use internet connectivity to store data on a remote server. This is a limitation of our algorithm. Our algorithm needs to reset apps occasionally. However, after a reset a previously feasible trace could become infeasible depending on the content stored on a remote server. This violates a key requirement of our algorithm. This limitation could be eliminated by sandboxing the remote server or by mocking the remote server.

5. Empirical Evaluation

In this section, we evaluate *SwiftHand* on several real-world Android apps. We first compare *SwiftHand* with random testing and $\mathcal{L}^*$-based testing. Then we discuss the results and shortcomings of *SwiftHand*.

5.1 Experimental Setup

We use branch coverage to compare the effectiveness and performance of the three testing strategies. We use binary instrumentation to obtain code coverage information.

Our benchmark-suite consists of 10 apps from *F-Droid* (`https://f-droid.org`) open app market. The apps were randomly selected initially, then apps with a native entry routine or frequent internet access are excluded.

name	category	#inst.	#method	#branch
`music note`	educational game	1345	72	245
`whohas`	lent item tracker	2366	146	464
`explorer`	JVM status lookup	6034	252	885
`weight`	weight tracker	4283	222	813
`tippy`	tip calculator	4475	303	1090
`myexpense`	finance manager	11472	482	1948
`mininote`	text viewer	13892	680	2489
`mileage`	car management	44631	2583	3761
`anymemo`	flash card	72145	832	4954
`sanity`	system management	21946	1415	5723

Table 1: Benchmark Apps

Table 1 lists these apps. #inst, #method, and #branch columns report number of bytecode instructions in the app binary, number of methods, and number of branches (excluding framework libraries), respectively.

We performed the experiment on a 8-core *Intel Xeon* 2.0Ghz (E5335) linux machine with 8Gb RAM using 5 emulators. We use 3 hours as our testing budget per app for every strategy. For unknown reasons, `sanity` and `anymemo` did not work smoothly on emulators. We used a *Google Galaxy Nexus* phone to test those apps and used a time budget of 1 hour. Android version 4.1.2 (Jelly Bean) was used on both emulators and phone.

In random testing, we restart an app with probability 0.1 and pick an input from a set of enabled inputs uniformly at random. We have implemented all three strategies as a part of the *SwiftHand* implementation.

5.2 Comparison with Random and $\mathcal{L}^*$-based Testing

Table 2 summarizes the results of applying the three testing strategies to the ten Android apps. In the table we use SH to denote *SwiftHand*. *% Branch Coverage* column reports branch coverage percentage for each strategy. *% Time Spent on Reset* columns show percentage of execution time spent on restarting app. *#Reset/#Input* columns report the ratio of the number restarts and the number of inputs generated for each strategy. *#Unique Prefixes* columns report the number of unique input prefixes tried in each testing strategy. *# States in Model* columns report number of unique states learned within the given time budget.

Figure 6 and Figure 7 plots percentage branch coverage progress against testing time.

- In all cases, *SwiftHand* achieves more coverage than both random and $\mathcal{L}^*$-based testing. Random and $\mathcal{L}^*$-based

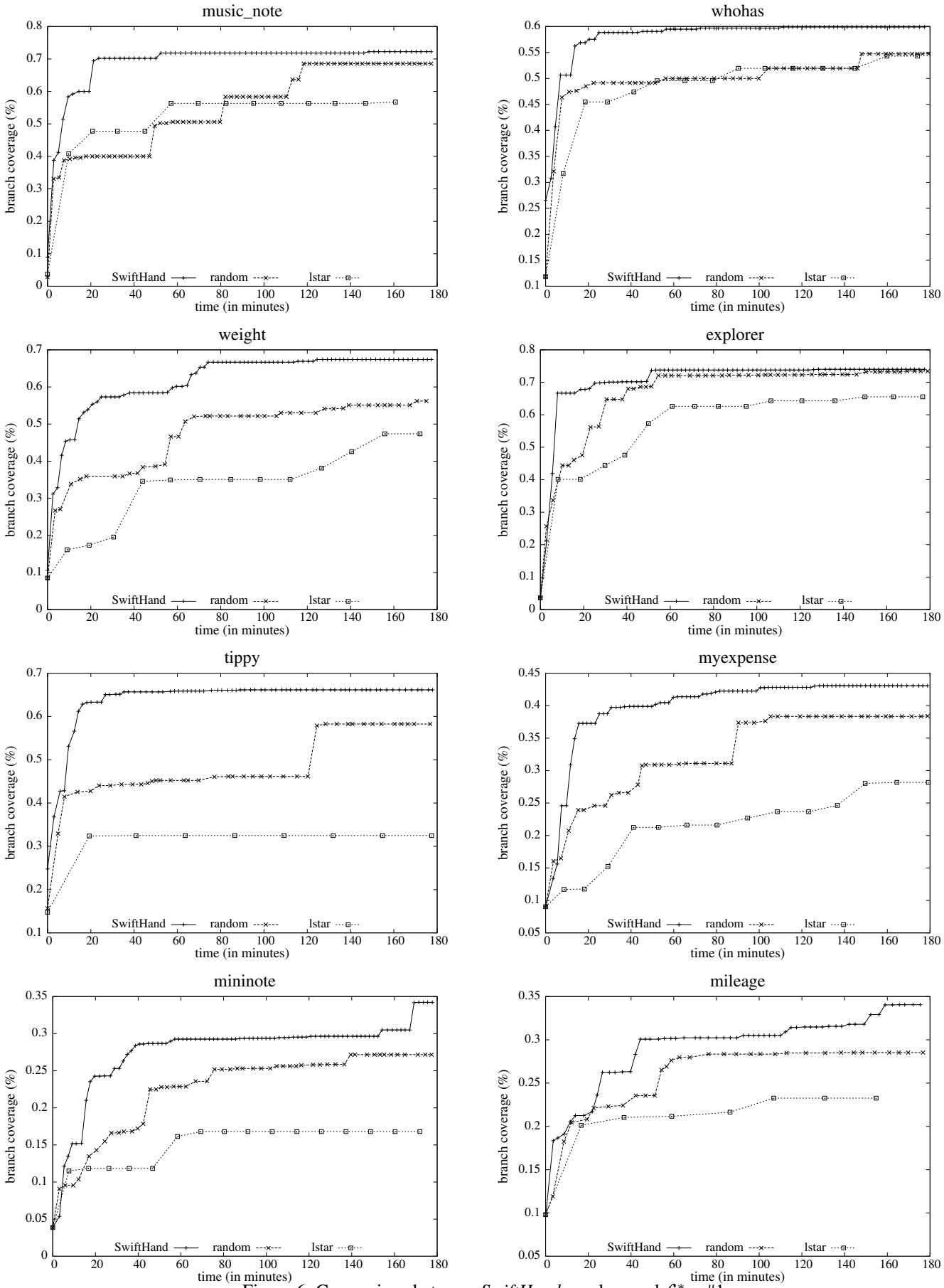


Figure 6: Comparison between *SwiftHand*, random and $\mathcal{L}^*$, #1

App	% Branch Coverage			% Time Spent on Reset			#Resets / #Inputs Ratio			#Unique Input Prefixes			#State in Model	
	SH	Rand	$\mathcal{L}^*$	SH	Rand	$\mathcal{L}^*$	SH	Rand	$\mathcal{L}^*$	SH	Rand	$\mathcal{L}^*$	SH	$\mathcal{L}^*$
`music note`	72.2	68.6	56.7	10.5	36.9	43.0	0.05	0.24	0.33	1419	712	275	46	15
`whohas`	59.3	54.7	54.3	10.0	26.2	41.6	0.05	0.14	0.33	1481	983	289	97	73
`explorer`	74.0	73.4	65.5	2.1	18.2	42.0	0.02	0.11	0.40	1271	1047	305	195	150
`weight`	62.1	57.6	48.2	8.3	25.6	42.4	0.04	0.12	0.24	1341	855	271	109	94
`tippy`	68.5	60.3	32.9	17.6	49.2	81.6	0.02	0.11	0.54	1435	812	162	71	17
`myexpense`	41.8	38.4	23.6	4.0	32.3	46.1	0.02	0.15	0.25	1740	926	281	149	63
`mininote`	34.1	27.2	16.8	9.8	25.7	49.1	0.19	0.13	0.37	1562	1051	334	169	72
`mileage`	34.6	28.5	23.3	12.7	32.4	58.8	0.03	0.18	0.55	1109	599	160	130	72
`anymemo`	52.9	37.9	26.6	3.45	34.6	50.8	0.01	0.19	0.33	1366	632	247	169	50
`sanity`	19.6	17.0	13.1	15.2	32.5	46.3	0.06	0.16	0.27	1012	501	230	78	99

Table 2: Summary of Comparison Experiments

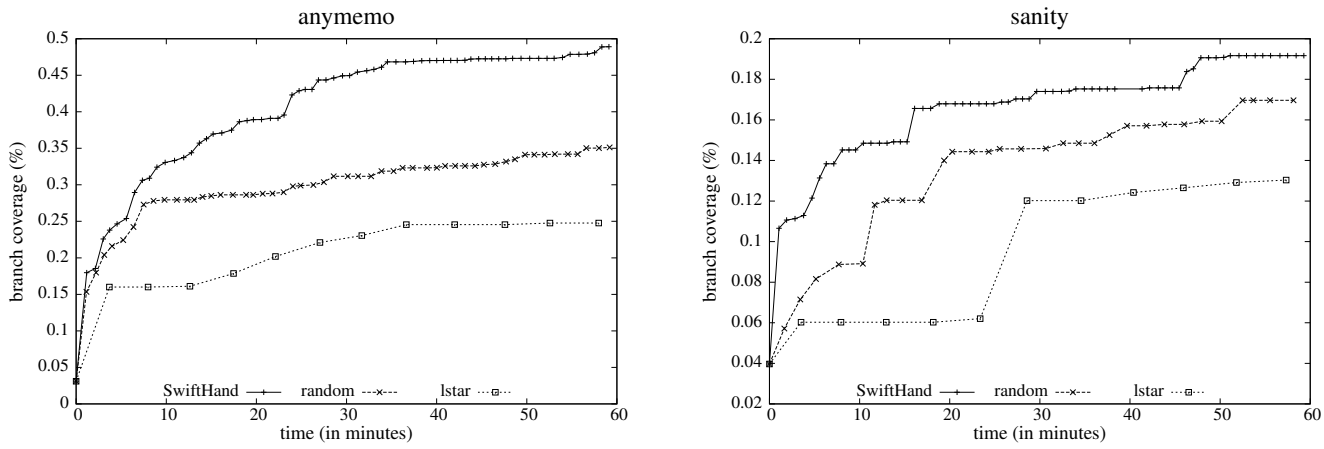


Figure 7: Comparison between *SwiftHand*, random, and $\mathcal{L}^*$, #2

testing performs relatively well for simple apps such as *whohas* and *explorer*. However, for more complex apps, such as `anymemo`, *SwiftHand* outperforms both random and $\mathcal{L}^*$-based testing.

- For all apps, *SwiftHand* achieves branch coverage at a rate that is significantly faster than that of random and $\mathcal{L}^*$-based testing. For example, in `explorer` *SwiftHand* reached almost 65% branch coverage within 10 minutes whereas both random and $\mathcal{L}^*$-based testing failed to reach 45% coverage in 10 minutes. This implies that *SwiftHand* is a far better choice when the time budget for testing is limited.
- Random testing restarts more frequently than *SwiftHand*. Android apps, in comparison with desktop apps, have fewer GUI components on screen. Therefore, the probability of random testing to terminate an app, by accidentally pushing **Back** button or some other quit button, is relatively high. *SwiftHand* can avoid this by merging states and by avoiding input sequences that lead to a terminal state.
- Random testing catches up with the coverage of *SwiftHand* given an additional period of time for some small apps; `musicnote`, `explorer`, and `tippy`. However, random testing fails to close the coverage gap for complicated apps; `mininote`, `mileage`, `anymemo`, and `sanity`.
- Our experiments confirm inadequacy of $\mathcal{L}^*$ as a testing strategy when restart is expensive. $\mathcal{L}^*$ spent about half of its execution time in restarting the app under test (Table 2). This resulted in the low branch coverage over time (Figure 6, Figure 7). Furthermore, note that $\mathcal{L}^*$ has relatively small numbers in the *#Unique Input Prefixes* column compared to random testing and *SwiftHand*. This indicates that $\mathcal{L}^*$ executes the same prefix more often than random testing and *SwiftHand* do.

5.3 What Restricts Test Coverage?

We manually analyzed apps with a coverage lower than 40% to understand the cause behind low branch coverage. We discovered the following three key reasons behind low branch coverage.

Combinatorial State Explosion. `mileage` is a vehicle management app. The app has a moderately complex GUI. The app provides tab bar for easy context switching between different tasks. In this app, tab bar creates a combinatorial state-explosion problem. Conceptually, each tab view is meant for a different independent task. Most actions on a tab view affect only the local state of the tab and few actions may affect the app's global state. *SwiftHand* does not understand this difference. As a result a slight change on one tab view makes all other tabs to treat the change as a new state in the model. The following diagram illustrates this situation:

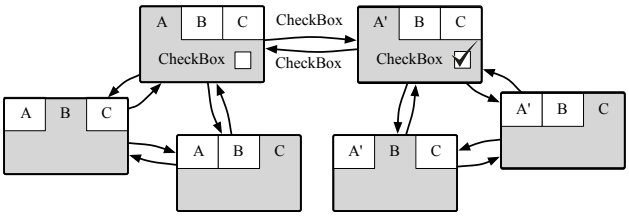


Figure 8: Model state explosion with tab view

The app `sanity` has a similar problem. This is an inherent limitation of our current model. We believe that this limitation can be overcome by considering a framework where each model is a cross-product of several independent models. We can then perform compositional machine learning to learn the sub models independent of each other. However, at this point it is not clear how this compositional learning can be made online.

Network Connection. `anymemo` is a flash card app with dictionary support. The user can access a number of repository to download dictionaries and wordlists. We cannot test this part of the app, because we disable internet connections to ensure clear restart.

Inter-app Communication. `mininote` uses intent to open a text file from file system navigation. When an intent is captured, the Android system pops a system dialog for confirmation. The confirmation dialog box is not a part of the app. *SwiftHand* simply waits for the app to wake up and terminates the app when it fails to respond. As a result the viewer part of the app is never tested. 35% code coverage solely comes from testing file system navigation part.

6. Related Work

Automated GUI Testing. Model-based testing approaches are often applied to testing graphical user interfaces [8, 11, 28, 37, 39, 43]. Such approaches model the behavior of the GUI abstractly using a suitable formalism such as event-flow graphs [28], finite state machines [32, 39], or Petri nets [37]. The models are then used to automatically generate a set of sequences of inputs (or events), called a test-suite.

Memon et al. [28, 46] have proposed two-staged automatic model-based GUI testing idea. In the first stage, a model of the target application is extracted by dynamically analyzing the target program. In the second stage, test cases are created from the inferred model. This approach differs from our work in two ways: (a) in our work, model creation and testing forms a feedback loop, (b) we use *ELTS*, a more richer model than *Event Flow Graph* (EFG). Yuan and Memon [47] also investigated a way to reflect runtime information on top of EFG model to guide testing.

Crawljax [29] is an online model-leaning based testing technique for *Javascript* programs. Crawljax tries to learn a finite state model of the program under test using state-merging. State-merging is based on the edit distance between DOMs. Crawljax can be seen as a learning-based GUI testing technique that uses ad-hoc criteria for state-merging. Recently, a number of model-learning based testing techniques targeting Android application [6, 33, 36, 40, 45] have been proposed. Nguyen et al.'s approach [33] combines offline model-based testing and combinatorial testing to refine created test cases. Yang et al. [45] take online model-based testing approach similar to *Crawljax* [29]. They use static pre-analysis to refine the search space. Takala et al. [40] report a case study using online model-based testing. Rastogi et al. [36] proposed a testing technique based on learning a behavioral model of target application. All of these techniques use some form of heuristic state-merging and are not based on formal automata learning. The main difference is in the treatment of counter-examples. Online automata learning techniques try to learn an exact model reflecting counter-examples. On the contrary, none of the above approaches learn from counter-examples. Therefore, these techniques may fail to learn the target model even for a finite state system, unlike online automata learning techniques. As far as we know, *SwiftHand* is the first GUI testing technique based on automata learning.

The Android development kit provides two testing tools, *Monkey* and *MonkeyRunner*. *Monkey* is an automated fuzz testing tool creates random inputs without considering application's state. Hu and Neamtiu [20] developed an useful bug finding and tracing tool based on *Monkey*. *MonkeyRunner* is a remote testing framework. A user can control application running on the phone from the computer through USB connection.

MacHiry et al.[25] suggest a technique to infer representative set of events and perform event aware random testing. The idea is similar to the random strategy used in our experiment. Their event inference technique targets a larger event space including tilting and long-touching while our technique only considers touching and scrolling events. Also, their tool acquires more runtime information by modifying Android framework to prune out more events at the expense of being less portable. On the contrary, *SwiftHand* modifies only the target app binary. Finally, they provide comparison with *Monkey*. The result shows that *Monkey* needs signifi-

cantly more time to tie the branch coverage of event aware random testings.

Anand et al. [7] applied concolic execution [16] to guide Android app testing. They use state subsumption check to avoid repeatedly visiting equivalent program states. Their technique is relatively sound than our approach, since concolic execution engine creates an exact input to find new branches in each iteration. In Mirzaei et al. [30]'s compiles Android apps to Java byte code and applies *JPF* with a mock Android framework to explore the state-space.

Learning Algorithms. Angluin's $\mathcal{L}^*$ [10] is the most well-known active learning algorithm. The technique has been successfully applied to modular verification [13] and API specification inference [5]. These techniques try to learn a precise model of the target application.

Passive learning techniques [15, 23] do not assume the presence of a teacher and uses positive and negative examples to learn a model. François et al. [14] have introduced ideas of exploiting domain knowledge to guide passive learning. The technique was subsequently improved by Lambeau et al. [21].

Learning based Testing. Machine learning has previously been used to make software testing effective and efficient [12, 17–19, 26, 34, 35, 41, 42, 44]. Meinke and Walkinshaw's survey [27] provides a convenient introduction to the topic. In general, classic learning based testing techniques aim to check whether a program module satisfies a predefined functional specification. Also, they use a specification and a model checker to resolve equivalence queries. On the contrary, *SwiftHand* tries to maximize a test coverage and actually executes a target program to resolve equivalence queries.

Meinke and Sindhu [26] reported that learning algorithms similar to $\mathcal{L}^*$ are inadequate as a testing guide. They proposed an efficient active learning algorithm for testing reactive systems, which uses a small number of membership queries.

7. Conclusion

We showed that a straight-forward $\mathcal{L}^*$-based testing algorithm is not effective for testing GUI applications. This is because $\mathcal{L}^*$-based testing requires a lot of expensive restarts and it spends a lot of time in re-exploring the same execution prefixes. We proposed a novel learning-based testing algorithm for Android GUI apps, which tries to maximize branch coverage quickly. The algorithm avoids restarts and aggressively merges states in order to quickly prune the state space. Aggressive state-merging could over-generalize a model and lead to inconsistency between the app and the learned model. Whenever the algorithm discovers such an inconsistency, it applies passive learning to rectify the inconsistency. Our experiments show that for complex apps, our algorithm outperforms both random and $\mathcal{L}^*$-based testing. Our algorithm also achieves branch coverage at a much faster rate than random and $\mathcal{L}^*$-based testing.

Acknowledgement

This research is supported in part by NSF Grants CCF-1017810, CCF-0747390, CCF-1018729, and CCF-1018730, and a gift from Samsung. The last author is supported in part by a Sloan Foundation Fellowship. We would like to thank Cindy Rubio Gonzalez and Wonchan Lee for insightful comments on a previous draft of the paper.

References

[1] Managing the Activity Lifecycle. `http://developer.android.com/training/basics/activity-lifecycle/index.html`.

[2] MonkeyRunner. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`.

[3] UI/Application Exerciser Monkey. `http://developer.android.com/tools/help/monkey.html`.

[4] axml, read and write Android binary xml files. `http://code.google.com/p/axml/`, 2012.

[5] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109, 2005.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *ASE*, pages 258–261, 2012.

[7] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *SIGSOFT FSE*, page 59, 2012.

[8] A. A. Andrews, J. Offutt, and R. T. Alexander. Testing web applications by modeling with FSMs. *Software and System Modeling*, 4(3):326–345, 2005.

[9] D. Angluin. Inference of reversible languages. *J. ACM*, 29(3):741–765, July 1982.

[10] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[11] F. Belli. Finite-state testing and analysis of graphical user interfaces. In *12th International Symposium on Software Reliability Engineering (ISSRE'01)*, page 34. IEEE Computer Society, 2001.

[12] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In *FASE*, pages 175–189, 2005.

[13] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, pages 331–346, 2003.

[14] F. Coste, D. Fredouille, C. Kermorvant, and C. de la Higuera. Introducing domain and typing bias in automata inference. In *ICGI*, pages 115–126, 2004.

[15] J. N. Departarnento and P. Garcia. Identifying regular languages in polynomial. In *Advances in Structural and Syntactic Pattern Recognition, volume 5 of Series in Machine Per-*

ception and Artificial Intelligence, pages 99–108. World Scientific, 1992.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[17] A. Groce, D. Peled, and M. Yannakakis. AMC: An adaptive model checker. In *CAV*, pages 521–525, 2002.

[18] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of the IGPL*, 14(5):729–744, 2006.

[19] R. Groz, M.-N. Irfan, and C. Oriat. Algorithmic improvements on regular inference of software models and perspectives for security testing. In *ISoLA (1)*, pages 444–457, 2012.

[20] C. Hu and I. Neamtiu. A GUI bug finding framework for Android applications. In *SAC*, pages 1490–1491, 2011.

[21] B. Lambeau, C. Damas, and P. Dupont. State-merging DFA induction algorithms with mandatory merge constraints. In *ICGI*, pages 139–153, 2008.

[22] K. Lang, B. Pearlmutter, and R. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm, 1998.

[23] K. J. Lang. Random DFA's can be approximately learned from sparse uniform examples. In *COLT*, pages 45–52, 1992.

[24] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[25] A. MacHiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *SIGSOFT FSE*, pages 224–235, 2013.

[26] K. Meinke and M. A. Sindhu. Incremental learning-based testing for reactive systems. In *TAP*, pages 134–151, 2011.

[27] K. Meinke and N. Walkinshaw. Model-based testing and model inference. In *ISoLA (1)*, pages 440–443, 2012.

[28] A. M. Memon. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.*, 17(3):137–157, 2007.

[29] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *TWEB*, 6(1):3, 2012.

[30] N. Mirzaei, S. Malek, C. S. Pasareanu, N. Esfahani, and R. Mahmood. Testing Android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[31] J. Nevo and P. Crégut. ASMDEX. `http://asm.ow2.org/asmdex-index.html`, 2012.

[32] W. M. Newman. A system for interactive graphical programming. In *Proc. of the spring joint computer conference (AFIPS '68 (Spring))*, pages 47–54. ACM, 1968.

[33] C. D. Nguyen, A. Marchetto, and P. Tonella. Combining model-based and combinatorial testing for effective test case generation. In *ISSTA*, pages 100–110, 2012.

[34] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE*, pages 225–240, 1999.

[35] H. Raffelt, B. Steffen, and T. Margaria. Dynamic testing via automata learning. In *Haifa Verification Conference*, pages 136–152, 2007.

[36] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *CODASPY*, pages 209–220, 2013.

[37] H. Reza, S. Endapally, and E. Grant. A model-based approach for testing GUI using hierarchical predicate transition nets. In *International Conference on Information Technology (ITNG'07)*, pages 366–370. IEEE Computer Society, 2007.

[38] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences (extended abstract). In *STOC*, pages 411–420, 1989.

[39] R. K. Shehady and D. P. Siewiorek. A methodology to automate user interface testing using variable finite state machines. In *FTCS*, pages 80–88, 1997.

[40] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an Android application. In *ICST*, pages 377–386, 2011.

[41] N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris. Increasing functional coverage by inductive testing: A case study. In *ICTSS*, pages 126–141, 2010.

[42] N. Walkinshaw, J. Derrick, and Q. Guo. Iterative refinement of reverse-engineered models by model-based testing. In *FM*, pages 305–320, 2009.

[43] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *11th International Symposium on Software Reliability Engineering (ISSRE'00)*, page 110. IEEE Computer Society, 2000.

[44] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *FATES*, pages 60–69, 2003.

[45] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *FASE*, pages 250–265, 2013.

[46] X. Yuan, M. Cohen, and A. M. Memon. Covering array sampling of input event sequences for automated GUI testing. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 405–408, New York, NY, USA, 2007. ACM.

[47] X. Yuan and A. M. Memon. Iterative execution-feedback model-directed GUI testing. *Information & Software Technology*, 52(5):559–575, 2010.