2017 IEEE International Conference on Software Quality, Reliability and Security

Widget-Sensitive and Back-Stack-Aware GUI Exploration for Testing Android Apps

Jiwei Yan[1,3], Tianyong Wu[1,3], Jun Yan[1,2,3,†], Jian Zhang[1,3,†]
[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[2] Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
[3] University of Chinese Academy of Sciences
Email: {yanjw, wuty, yanjun, zj}@ios.ac.cn

***Abstract*—GUI exploration is a widely adopted technique to test GUI programs, which traverses the elements of screens during the user interaction and simultaneously constructs the GUI model to describe window transitions. Specific to Android apps, an elaborate GUI model should take Android characteristics into consideration. We propose a GUI exploration approach that dynamically acquires the information of these characteristics, such as the status of widgets and arrangement of the back stack. We attach this information to the window transition graph and form a new model called LATTE (*LAbeled Transition graph with sTack and widgEt*). To balance the accuracy and size of model, we introduce a metric "state similarity" to merge similar states. We perform experiments on 20 real-world apps to test them and construct their LATTE models. The investigation indicates that our systematic exploration approach with regard to the Android characteristics covers more program behaviors, and the generated model can be reused to direct the further testing.**

***Index Terms*—Android Application, GUI Exploration, Dynamic Modeling, Test Generation**

I. Introduction

With the extensive usage of smart phones, mobile application market ushered a high speed developing period, especially the Android application market. Android applications (apps), like other software, need to be adequately tested to eliminate the potential bugs and improve the quality. Android apps are event-driven GUI programs, which can be regarded as a collection of widgets. Each of the widgets is defined in an `Activity` class that is provided by Android system to interact with the user. The user operations on the widgets in the screen trigger the corresponding events to drive the app to transfer from one window to another. Modeling the behaviors of these window transitions accurately and comprehensively is a key step for performing a fine-grained and efficient testing for an Android app.

In recent years, several model-based testing approaches for Android apps have been proposed, which can be categorized into two kinds, including static and dynamic ones. The former one [27, 30] leverages the static analysis techniques on the code of the app to extract the GUI components in each Activity and the transitions between Activities. However, this kind of approaches may fail to describe the changes of GUI widgets during runtime, e.g., some widgets are instantiated under the conditions that should be determined dynamically and the status of the same widget may also change under user operations. Therefore, some researches [10, 16] adopt GUI exploration approaches which can dynamically explore the GUI information and simultaneously construct the GUI model when the app is running. However, this type of model only focuses on the GUI widgets and often omits the information of Android characteristics, such as back stack.

Both the dynamic changes of widgets and back stacks are important characteristics for model construction. On one hand, the dynamic changes of widgets (e.g., the `CheckBox` status for application setting) may influence the execution of programs. Failing to distinguish the minor differences of widget statuses between similar windows will miss some transitions in the model. On the other hand, the back stack is a particular mechanism of Android to store the launched Activities according to complex launch modes. Two GUI windows with different back stacks will perform differently under some user operations (e.g. pressing the back key). Failing to distinguish the back stacks of states will lead to faulty transitions in the model. Therefore, to correctly describe the behavior of Android GUI, the model should carry both the information of dynamic widgets and back stacks.

In this paper, we propose a GUI exploration approach to systematically traverse widgets to test the Android apps. Specifically, we build a window transition graph called LATTE to record the information of GUI widgets in a window and the transitions between windows, as well as the information of Android-specific back stack. Furthermore, to balance the accuracy and size of the model, we introduce a metric "state similarity" to merge similar states into one according to a user-given threshold. Besides GUI information, we also link the transitions in this model to the corresponding executed code snippets via a label mechanism for further testing based on the model. To validate our idea, we implement our proposed techniques into a tool called LAND (*LATTE model generation for Android apps*) and evaluate it experimentally on 20 real-world popular apps. The experimental results consistently show that the proposed exploration approach with the consideration of Android characteristics can cover more program behaviors in most cases than Monkey and Dynodroid, and the model LATTE can be reused to guide the further testing.

†Corresponding authors

978-1-5386-0592-9/17 $31.00 © 2017 IEEE
DOI 10.1109/QRS.2017.14

The main contributions of this work are summarized as follows:

- Provide a dynamic GUI exploration approach for testing Android apps.
- Propose the LATTE model to describe the GUI characteristics of Android apps in detail.
- Implement a model generation tool LAND as well as perform experiments on 20 real-world Android apps to test them and construct their LATTE models.
- Make use of the LATTE model to generate specific test cases in two application scenarios.

The remainder of this paper is organized as follows. In Section II, we discuss some necessary background knowledge about Android features. The LATTE model and exploration technique will be discussed in Section III and IV. Then we present our model generation tool LAND in Section V and evaluate our approach on real-world apps in Section VI. At last, we discuss related works in Section VII and conclude our work in Section VIII.

II. Background

As mentioned before, the dynamic widget and the back stack information are important for an elaborate model of Android apps. In this section, we will present some background knowledge about Android GUI widget and the back stack, and provide a simple Android app as our motivating example.

A. GUI Widget

A widget (also called a view) is an instance of the class `View` in Android, which represents a basic built-in block of component on the GUI window. There are several *basic attributes* of a widget, including the view type, view index and resource id. Besides, there are also several extra attributes during runtime, like the `isChecked` status of `CheckBox` widget, `RadioButton` as well as `ToggleButton`. We call these attributes *status attributes*. The differences of basic attributes between two widgets decide whether they are identical, while the difference of status attributes of one widget may trigger different program behaviors under the same user operation.

A widget is attached with several user events for responding to different user operations. For example, the possible events for the `Button` widget are click, longclick and press. In addition, some events should be triggered by a combination of several user operations. For instance, the typing event on `EditText` widget usually need a text clearing operation followed by text typing and enter key pressing. Besides the events that attach to GUI widgets, there are also some global events that can be executed at any program state, such as the screen rotation, back, home, and enter key pressing events. We consider all these events in our GUI model.

B. Back Stack

The execution of an Android app is composed of a sequence of Activities. Android takes *task* as the collection of Activities that users interact with when performing a certain job and introduces a *back stack* to arrange all these launched Activities by their launched order [2]. When the current Activity starts another one, the new one will be pushed on top of the stack and takes the focus by default. The former Activity remains in the stack but is paused. When the user presses the back key, the Android system will pop and destroy the current Activity, and resume the previous one. Activities in the stack can only be rearranged by push and pop operations according to the system-defined rules.

Android system defines the *Launch Mode* [3] of an Activity to determine the evolution of the back stack when the Activity is launched. The launch mode of an Activity is declared by default in the manifest file or specified locally using `Intent Flags` in the code. We just introduce the launch modes in manifest file since the `Intent Flags` share similar rules. These modes are `Standard`, `SingleTop`, `SingleTask` and `SingleInstance`, of which the last two modes involve complex multitasking interactions that are always used in cross-app circumstances.

Both in the `Standard` and `SingleTop` modes, one Activity can be instantiated multiple times. The `Standard` mode simply pushes and pops the new launched Activity without considering the existing Activities in the stack. Different with `Standard`, when the Activity is already on top of the back stack, the `SingleTop` mode refuses to create new instance for it. The `singleTask` and `SingleInstance` mode do not allow multiple instances of one Activity. The difference is that `singleTask` mode allows other Activities to be part of its task. If an instance of the Activity with `singleTask` mode to be launched already exists, all Activities above it will be popped until it becomes the top of the back stack. And the `SingleInstance` does not allow other Activities be pushed into its task, i.e., one Activity with `SingleInstance` mode is the only Activity in its task. The diversity of launch modes makes the evolution of the back stack complex so that we need to take effort in correctly modeling the back stack.

TABLE I
RULES FOR BACK STACK CHANGING

Launch Mode	Event	Stack Bef.	Stack Aft.
`Standard`	Launch M	(. . .,a)	(. . .,a,m)
`Standard`	Launch M	(. . .,m)	(. . .,m,m_2)
`SingleTop`	Launch M	(. . .,a)	(. . .,a,m)
`SingleTop`	Launch M	(. . .,m)	(. . .,m)
`SingleTask`	Launch M	(. . .,a)	(. . .,a,m)
`SingleTask`	Launch N	(. . .,a,m)	(. . .,a,m,n)
`SingleTask`	Launch M	(. . .,a,m,n)	(. . .,a,m)
`SingleInstance`	Launch M	([. . .,a])	([. . .,a][m])
`SingleInstance`	Launch N	([. . .,a][m])	([m][. . .,a,n])
`SingleInstance`	Launch M	([m][. . .,a])	([. . .,a][m])

Here we use some concrete examples to show the rules for handling launch modes in Table I, where the letters a, m (m_2) and n denote different instances of Activity A, M and N. We just consider the invocation of Activities in the same app and do not include the complex cases about cross-app interactions. The first column gives the launch mode of Activity M while Activity A and N have the default `Standard` launch mode.

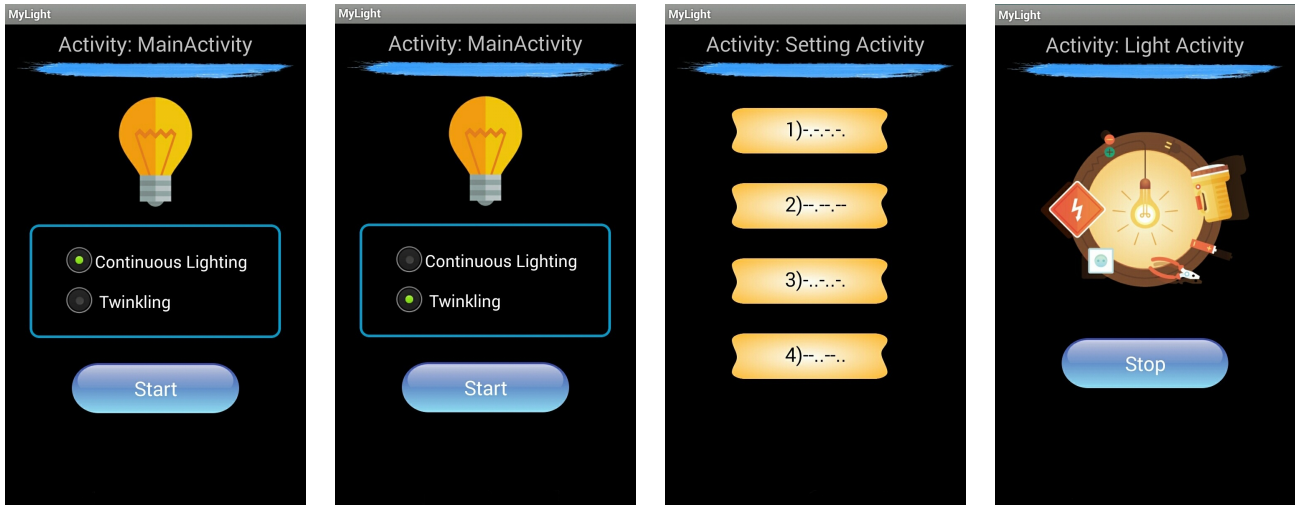


Fig. 1. MyLight Application

The second column gives the event to execute and the last two columns show the arrangement of Activities in the back stack. Note that the Activity instances in same square brackets means that they are in the same task, and the back stack will return to the most recently used task when task switches due to the back operation (in line 8-10).

C. Motivating Example

We use a self-developed app *MyLight* for opening the flashlight hardware to illustrate the motivation of this work. *MyLight* contains three Activities, including `MainActivity`, `SettingActivity` and `LightActivity`. The entry `MainActivity` provides a `RadioButton` with two choices for setting the manner of the flashlight, *Continuous Lighting* or *twinkling*. After the Start button is clicked, the app will transfer to the `LightActivity` if the Continuous Lighting button is selected, or transfer to the `SettingActivity` otherwise. The users can choose a flash pattern in the `Setting Activity` to control the way of the flashlight twinkles and then jump to the `LightActivity`. The launch modes of these Activities are set as `SingleTop` in the manifest file.

The four windows of *MyLight* are displayed in Fig. 1. As we can see, the only difference between the first two windows is the status of the `RadioButton`. In fact, this status will decide the program behavior after the Start button is clicked. Existing works (such as [9, 31, 33]) construct the GUI model without considering the status of widgets, thus they regard these two windows as the same state. In this case, these approaches will create only one state for the first two windows followed by one transition triggered by Start button click and finally get one of the two models displayed in Fig. 2 (a). That is to say, some program behaviors may be omitted in the model.

A model with the widget status information is shown in Fig. 2 (b), in which the `Main1` and `Main2` states represent the first two windows respectively. The dash lines represent the back transitions (triggered by pressing back key) from the `Light` state in the model. Since there are two transitions from different source states (`Main1` and `Setting`) joining into the `Light` state, without the back stack information, we do not know how to go back to the previous state when we press a back key in the `Light` state.

To sum up, with the widget status information, we can construct the model more complete; with the back stack information, we can clearly know the destination of the stack-related transitions to avoid faulty paths. Therefore, both the widget status attributes and the back stack should be regarded as essential elements in model construction. We draw a model that considering both the widget status attributes and the back stack information in Fig. 2 (c), where rectangles attached with the states `Light1` and `Light2` represent their back stack. We split the window with the same widgets into multiple states in the model according to different widget statuses or back stacks to ensure that each event in the same state corresponds to a unique transition.

III. The Proposed Model

In this section, we will describe our LATTE model for describing the GUI information of an Android app. This model extends the finite state machine (FSM) model [14, 19] with the information of the status of widgets and back stack. Besides, we make use of labels to link the transitions in the model to the code snippets that are executed.

A. Definition of LATTE Model

As discussed before, an Activity may contain multiple suites of widgets and events in runtime, which may lead to the loss of some program information when we simply regard an Activity as one state. To solve this problem, we propose a LATTE model, whose main idea is to split each Activity into one or more states, depending on both the widget and back stack information. The LATTE model can be formally described as a 5-tuple $\mathcal{M} = \langle S, La, T, s_0, R \rangle$, where

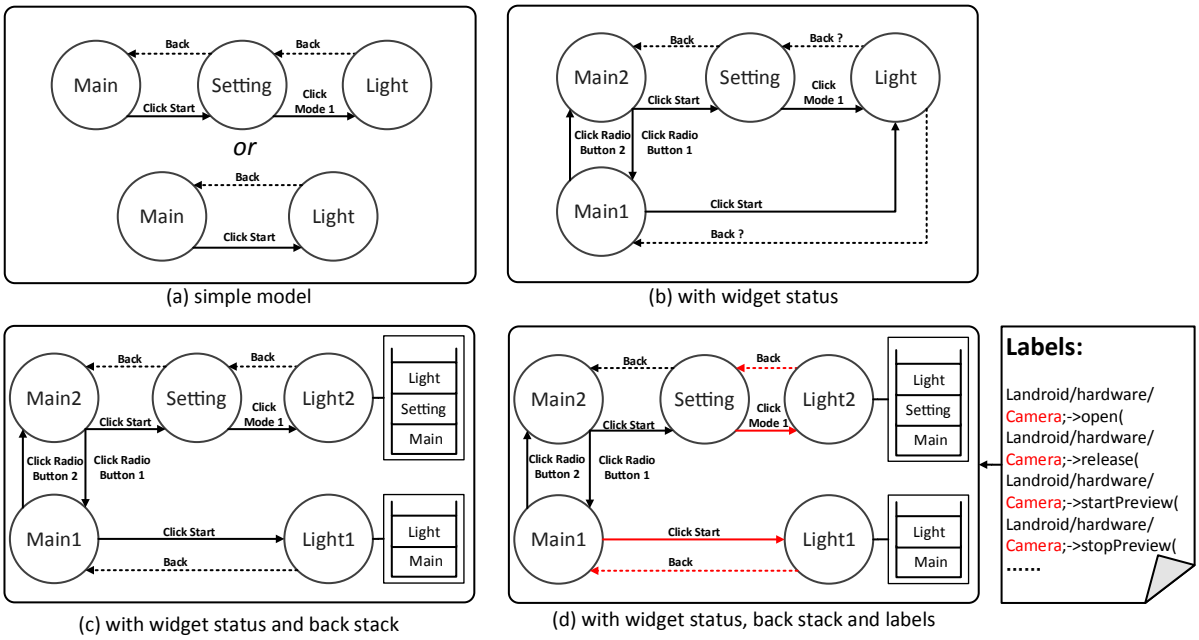


Fig. 2. Models of MyLight

- S is a set of application's runtime states. An element $s \in S$ is a triple $\langle a, W_s, L_s \rangle$ where a denotes the Activity that s corresponds to, and W_s indicates the set of widgets that belong to the corresponding window, and L_s is a list of Activities in the back stack.
- La is the set of labels.
- T denotes the set of transitions. Each element $t \in T$ is a 4-tuple $\langle src, e, la, des \rangle$ representing the transition from the source state src to the destination state des caused by event e bound to the state src, and the label set $la \subset La$ denotes the labels assigned to this transition.
- $s_0 \in S$ is the entry state that represents the initial state of the app.
- R is a set of terminal states denoting that the current application quits or jumps to another application, where set $R \subset S$ and each element $r \in R$ can not be s_0.

B. Label of LATTE Model

To embed the code information into our model, we introduce a label set La. In general, each element in La corresponds to a part of specific code. For instance, we can set each label to represent a distinct method of the app (method label), all the methods in the same class (class label), or instructions with a specific keyword (keyword label). The mapping rule for the labels and the code snippets is designed according to the actual testing or analysis requirements. A fine-grained rule can make the model contain more accurate code information while it increases the model size and the cost of the model construction. With the mapping rule, the labeling procedure is implemented via code instrumentation in our approach.

Method Label:
startActivity
Code Snippets:
Lcom/light/MainActivity;−>startActivity (
Lcom/light/ SettingActivity ;−>startActivity (
Lcom/light/ LightActivity ;−>startActivity (
Class Label:
Lcom/light/MainActivity
Code Snippets:
Lcom/light/MainActivity;−>onCreate(
Lcom/light/MainActivity;−>openContiLighting(
Keyword Label:
Camera
Code Snippets:
LAndroid/hardware/Camera;−>open(
LAndroid/hardware/Camera;−>release(
LAndroid/hardware/Camera;−>startPreview(
LAndroid/hardware/Camera;−>stopPreview(

Fig. 3. Labels and `Smali` Code Snippets

Fig. 3 shows an example of the mapping between labels and corresponding code snippets in the app *MyLight*. The method label `startActivity` represents a distinct method that can be invoked in any class, the class label `com/light/MainActivity` represents all the methods in the given class and the keyword label `Camera` represents all code instructions containing the keyword "Camera". Fig. 2 (d) illustrates the program model with user set keyword label `Camera` in *MyLight*. The label-related transitions are detected and marked with red line in the model.

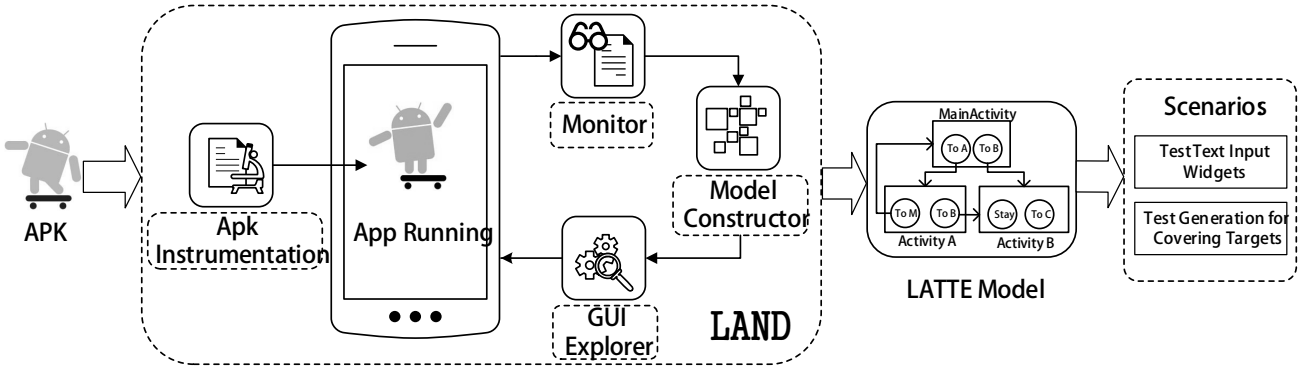


Fig. 4. Approach Overview

IV. GUI Exploration

In this section, we propose an approach that dynamically explores the GUI and automatically constructs the LATTE model for an Android app. The overview of the approach and several key elements will be introduced.

A. Overview

Our GUI exploration approach takes an Android apk file as input, and outputs the corresponding LATTE model. The basic work-flow of this approach is an iterative operation of app GUI exploration and model construction. We first initialize a LATTE model with an empty state and a transition set, and then launch the app for GUI exploration. When the GUI exploration procedure drives the app to some windows, we collect the information of this window for constructing the model, like information about widgets and the back stack, and update the model with this information. Then we trigger each event attached with the widgets in the current window to traverse all the widgets. The app may be driven to a new window after an event is triggered, and then we repeat the above procedure until all states are traversed. We can use some existing testing frameworks (like Robotium [8]) to obtain the widgets of a window, traverse each widget and trigger the events. However, the back stack information can not be directly obtained (except the top Activity of back stack) by all of these frameworks. In addition, we also want to link code snippets to the transitions in the model for the model reuse. To address these issues, we leverage the byte-code instrumentation technique to record runtime information which is related to the back stack and executed code.

Fig. 4 shows the overview of our GUI exploration approach. It is composed of four modules, including *Apk Instrumentation*, *Monitor*, *Model Constructor* and *GUI Explorer*. The Apk Instrumentation module analyzes the apk byte-code and inserts some extra instructions into it for recording the app's runtime information, while the Monitor module records the runtime information, like the widget information in the current window and the executed code of the app. The Model Constructor initializes an empty LATTE model, and analyzes the information from the Monitor module to update the LATTE model. The GUI Explorer module determines the next event based on the model and triggers this event (e.g., click a button) to drive the app execution. With the help of the generated LATTE model, we can perform further testing under different application scenarios, such as text input widgets testing and target directed test generation.

We only introduce the Model Constructor and the GUI Explorer module in this section, while the other modules will be discussed in Section V. There are three key issues that must be considered in these two modules: how to obtain the accurate back stack information during runtime; how to abstract the GUI windows to the states in the LATTE model; and how to systematically traverse the GUI widgets and events of the app for exploration. We will discuss the three issues in the following subsections, respectively.

B. Obtain Back Stack Information

The dynamic monitoring technique can be used to directly obtain the top of the back stack (current Activity displayed on window), but not the complete back stack information. However, developers may finish current Activity before starting a new Activity, or start two different new Activity under one operation, so that the change of the current Activity detected and the back stack in real is inconsistent. We need to obtain more back stack information to find out how it evolves.

By further investigation of the back stack mechanism, we observe that there are three system APIs `startActivity`, `startActivityForResult` and `finish` that can influence the operations of the back stack, of which the first two will create a new Activity and the last one destroys the Activity. Besides, the implicit callback `onBackPressed` determines how the back stack acts when back key is pressed. For an Activity, the default back operation is invoking API `finish`, but for a dialog or menu, its default back operation is just closing the current window and focusing on the Activity they belong to. Developers can also overwrite `onBackPressed` to meet specific requirements. In addition, the launch modes defined in the manifest file decide how the back stack evolves.

The details of how the back stack updated are shown in Algorithm 1. We first get the mapping of all the Activities and their launch modes by analyzing the manifest file (in line 1). The detailed information of launch mode can refer to Section II-B. Then we maintain a global stack to simulate the Android back stack of the app. For any event operation except the back event, we dynamically monitor the calling of the above three system APIs (in line 6). Specifically, when the `startActivity` or `startActivityForResult` is invoked, we update the global stack according to the corresponding back stack updating rules of its launch mode. When the `finish` is invoked, we pop the top Activity in the global stack. When a back event is triggered, we detect the invocation of `onBackPressed` (in line 8-11). If it is invoked, the back stack will evolve according to the calling of the related system APIs; otherwise, callback function `onBackPressed` will be called implicitly. In the latter case, we obtain and compare the Activity name before and after operation. The changing of Activity indicates the calling of `finish`, and the invariability of Activity indicates the closing of a dialog or menu, which can be taken as a variant window with different widgets and can not cause the stack change when opening and closing. Finally, we assign the information of the global stack to the back stack of the state in the model.

Algorithm 1 Back Stack Updating

Input: Stack<Activity> $stack$, runtime info, manifest file
Output: updated Stack<Activity> $stack$

1: Get the launch mode mapping act_mode
2: Get current Activity a_1 and perform an event e
3: Get new current Activity a_2
4: Monitor the calling of related APIs api
5: **if** e is not a back event **then**
6: Update $stack$ according to api and act_mode
7: **else**
8: **if** $onBackPressed()$ is invoked **then**
9: Update $stack$ according to api and act_mode
10: **else if** $a_1 \neq a_2$ **then**
11: $stack$.pop()
12: **end if**
13: **end if**

C. Calculate State Similarity

In our LATTE model, we treat a window during runtime as a state with some attributes (package, Activity, widget and back stack information). When we meet a window, we should not always create a new state in the model for this window since it may lead to the explosion of the number of the states. Instead, we need to define a metric to measure the similarity of two states and then merge them into one when they are the same or have high similarity. Existing works [10, 14, 15] have taken the information of package and Activity as well as the basic attributes of widgets (see Section II-A) to calculate the similarity. We pick this information as the state similarity rule as well. Besides, we also consider some additional factors that influence the similarity of states in our work, including the widget status attributes and back stack information.

Let s_1 and s_2 be two states, W_1 and W_2 represent the widget sets in s_1 and s_2 respectively where each element of them has two attributes *basic* and *status*. Let $Sim(s_1, s_2)$ denote the similarity of the states s_1 and s_2. If two states s_1 and s_2 have different package, Activity or back stack information, then we have $Sim(s_1, s_2) = 0$, otherwise $Sim(s_1, s_2)$ can be calculated by

$$Sim(s_1, s_2) = \begin{cases} \frac{(1-\eta)|C_1|+\eta|C_2|}{|C_0|} & |C_0| \neq 0 \\ 1 & |C_0| = 0 \end{cases}$$

where the set $C_0 = \{a|\exists\alpha \in W_1, \beta \in W_2(\alpha.basic = a \vee \beta.basic = a)\}$ represents the collection of distinct *basic* attributes in W_1 and W_2; $C_1 = \{b|\exists\alpha \in W_1, \beta \in W_2(\alpha.basic = b \wedge \beta.basic = b)\}$ represents the collection of common *basic* attributes in W_1 and W_2 and set $C_2 = \{c|\exists\alpha \in W_1, \beta \in W_2(\alpha.status = c \wedge \beta.status = c)\}$ represents the collection of common *status* attributes in W_1 and W_2. We define the configurable parameter η ($0 \leq \eta \leq 1.0$) to adjust the weight of the widget status for the state similarity where $\eta = 0$ implies that the similarity is calculated by the *basic* attribute and $\eta = 1$ implies that it mainly depends on the *status* attribute. We also introduce a similarity threshold S_T. If the similarity of two states exceeds S_T, we will merge these two states into one to reduce the model size.

To get a deterministic model of app, threshold S_T should be set equal to 1. Note that, the threshold could also be set less that 1. However, in such a setting, the similarity relation formulated could lead to different results depending on the order in which Activities are visited. And with a lower threshold, we can get less model states and consumption of model construction. The self-defined threshold will provide a trade-off between the accuracy and efficiency.

D. Traverse GUI Widget and Construct Model

Algorithm 2 shows the details of our approach to traverse the GUI widgets and construct the LATTE model, which is based on the expanded breadth-first search (BFS) traversal algorithm. In this algorithm, we maintain the model $\mathcal{M}$ of the detected part of app and a queue q storing unvisited states. The exploration starts from the entry state s_0 (corresponds to the entry window when the app is launched) and ends when all states have been visited. Here, we call a state that is visited when all the corresponding events of that state are visited. In each iteration, we first get the front state s_h from the queue q and automatically drive the app to the state s_h according to the event sequence from the entry state to the state s_h that we record when s_h is detected. Next, we select an unvisited event e of s_h and execute it with this event to a new state s_n. Then we collect the runtime information of s_n and create a transition from s_h to s_n. The event e and the labels la related to the executed code are also assigned to this transition. Then we will calculate the similarity values of s_n and existing states in the model $\mathcal{M}$. If the maximum similarity value exceeds the

threshold S_T, we will merge the new state to the existing state that is most similar to it; otherwise, we will append s_n to q and $\mathcal{M}$ as a new state. If all events of s_h have been visited, we mark s_h as a visited state and remove it from q.

Algorithm 2 LATTE Model Construction

Input: Instrumented Apk File, Label Set
Output: LATTE model $\mathcal{M}$

```
1: var: Queue<State> q
2: initialize q with the entry state s_0
3: while q is not empty do
4:    Drive the app to the front state s_h in q
5:    Perform an unvisited event e of s_h
6:    Obtain the stack information and label set la of e
7:    Create a new state s_n
8:    Create a transition from s_h to s_n with e and la
9:    Find state s_m that has the maximum similarity with s_n
10:   if Sim(s_m, s_n) > S_T then
11:      Merge s_n to s_m in model
12:   else
13:      Append s_n to queue q and model M
14:   end if
15:   if all events of s_h have been visited then
16:      Remove s_h from q
17:   end if
18: end while
```

V. Tool Implementation

We have implemented our proposed techniques into a tool called LAND in Java language. In this section, we will briefly introduce the implementation of its major modules.

To obtain the back stack information and executed code snippets during runtime, we leverage the instrumentation technique to insert a few statements into the original app. The statements inserted are called program probes. They are usually added for examining the execution of program statements and runtime change of variables. These probes will work during the dynamic running of instrumented application, and provide runtime information of program statements. The instrumentation technique is often implemented on the source code, but the source code is not always available especially for apps in Android markets. Hence, we perform instrumentation on the "smali" byte-code, a readable format of the byte-code of Android app. The smali format provides multiple keywords, e.g., the `.method` keyword denotes the beginning of a method body and `invoke` keyword is used to invoke the indicated method. We rely on ApkTool [5] to decompile the apk file and generate the corresponding smali code. Then we take a light-weight byte-code analysis, which helps to find the proper location for instrumentation according to the grammar and semantic context of original program. We make use of our tool InsDal [24] to instrument additional code to capture the runtime information on Android phones.

In the GUI Explorer and Monitor module, we implemented a script on top of Robotium to trigger the specific event, drive the app to run, and obtain the widget information of a window during runtime. Robotium is an open source Android test framework that drives the app to execute under manually designed test cases. It is based on the test instrumentation mechanism [1] provided by Android system, that can monitor the interaction of application and Android system, and control the execution of the app. Robotium also provides a series of APIs to capture and access the widgets and send instructions to simulate user events.

VI. Evaluation

In this section, we will present our experimental results of GUI exploration and model construction. All of our experiments are done on a mobile smartphone with 1.82GHz CPU and 3GB RAM.

A. Experimental Setup

To evaluate the effectiveness of our approach, we raise several research questions as follows.

- RQ1. Can the GUI exploration approach achieve a high code coverage of apps?
- RQ2. How does the similarity impact the model size and coverage?
- RQ3. How can we reuse the LATTE model for further testing?

To answer these questions, we collect 20 real-world apps as experimental instances. Tool LAND is applied to construct LATTE model for them and a series of experiments are conducted on them. Table II lists the detailed information of these experimental instances, of which the first ten apps are from F-Droid [7] (with source code) and the rest ones are from the commercial market (without source code). The first column denotes the name of an app. The following four columns show the size (MB) of the app, the numbers of its classes (#C), methods (#M), and Activities (#A). The last three columns give the numbers of Activities whose launch mode is not `Standard` (#NS), back stack related API calls (#B) and widgets that have dynamic status attributes (#W).

For RQ1, we measure the effectiveness of GUI exploration approach with the code coverage on the behavior of apps. The code coverage can be calculated by analyzing the basic information of byte-code and collecting the runtime information of executed code. We pick two popular automatic testing tools Monkey [4] and Dynodroid [25] for comparison, since a recent research [17] shows that Monkey and Dynodroid achieve higher coverage than other existing testing tools for Android apps. The number of generated events for Monkey is 10000 and for Dynodroid is 2000 (same with what Machiry et al. suggested in their work). The similarity threshold S_T of LAND is set to an experimental value 0.8 (refer to Section VI-C).

For RQ2, we design experiments to show that how the similarity setting of states influences the size of LATTE model. In these experiments, we set the value of threshold S_T from 0 to 1.0 and compare the number of transitions in the generated LATTE model and the code coverage by traversing this model

TABLE II
EXPERIMENTAL APPLICATIONS

App Name	Size	#C	#M	#A	#NS	#B	#W
aGrep	0.34	46	174	6	2	21	3
aLogcat	0.14	35	185	2	1	15	3
BookCl	2.73	877	4361	35	2	174	15
Budget	0.19	63	272	8	0	27	1
HotDeath	7.93	28	355	3	0	4	0
PassWordMP	1.67	89	452	8	0	43	13
TippyTipper	0.09	44	226	5	0	7	3
TomDroid	1.08	154	834	8	0	54	8
WebSearch	1.90	45	176	3	0	46	2
WhoHasMS	0.79	24	139	2	0	32	3
Btime	14.86	3752	25641	217	12	1037	21
BubeiListen	3.84	902	4637	91	8	491	0
Compass	1.38	29	316	2	0	69	0
Cradio	1.57	43	486	6	0	72	3
Flashlight	5.44	91	479	11	4	72	4
FreshBowser	2.29	164	463	7	1	29	3
QiuShiBaiKe	14.31	2803	15482	146	9	729	21
SaoleiGame	0.34	23	111	3	0	13	4
SgSearch	8.42	272	1083	66	8	305	26
Terminal	11.67	6	24	4	0	88	2

under different values of S_T. A higher threshold S_T may cause more state splitting and lead to a more accurate model, and further lead to a higher code coverage.

The generated test cases based on LATTE can be manually adjusted for specific testing goals. For RQ3, we utilize LATTE model for test generation in two application scenarios. The first scenario is testing text input widgets. Most apps make use of the text input widget `EditText` to accept a string from users, therefore, we need to reach these widgets and then use various strings to test them thoroughly. With the help of LATTE model, we can easily generate test sequences to cover these widgets. Then, for each widget in the sequence, we design ten test inputs based on black-box testing techniques (e.g., boundary value analysis) to replace the original random generated strings, and retest the apps.

Another scenario is test generation for covering targets. In practice, the testers often concern some specific part of codes, for example, when the testers want to analyze a functionality of the app, they only focus on the methods related to this functionality. We call the set of these specific code the "target", which is a subset of labels in the label set La. We consider two types of targets in this paper, including a set of specific user-developed methods and a set of system APIs related to resource and privacy. The misuse of the latter will cause performance and security problems [13, 22, 29]. Experiments are done between LAND and Monkey to compare the minimal sequence length they need to cover the given target. To get the minimal sequence length of Monkey, we implement a script to repeatedly run Monkey with the event limits increased by 1000 in each iteration, until the given target is covered.

B. The Code Coverage of GUI Exploration

In this section, we first generate the test suites for each experimental instance by Monkey, Dynodroid, and our approach respectively, and then calculate the coverage of these three test suites. For the apps with the source code, we calculate the method coverage (#MC) and line coverage (#LC) by *EMMA* [6], a code coverage measurement tool for Java programs. For the commercial apps without source code, there is no publicly available code coverage measurement tool. So we make use of the tool InsDal to record the executed code information during runtime and calculate the class coverage (#CC) and method coverage on byte-code.

TABLE III
COVERAGE COMPARISON ON DALVIK BYTE-CODE

App Name	LAND		Monkey		Dynodroid	
	CC	MC	CC	MC	CC	MC
aGrep	83	52	58	33	76	58
aLogcat	74	67	65	58	72	64
BookCl	51	41	27	24	30	26
Budget	76	65	59	52	–	–
HotDeath	86	79	68	54	85	72
PassWordMP	74	57	67	52	–	–
TippyTipper	93	77	55	58	–	–
TomDroid	60	42	38	32	58	40
WebSearch	69	58	64	49	62	57
WhoHasMS	91	57	75	44	–	–
Btime	37	22	10	6	–	–
BubeiListen	55	53	35	32	19	11
Compass	53	21	55	24	48	14
Cradio	81	57	77	54	–	–
Flashlight	67	53	60	49	–	–
FreshBowser	90	64	52	31	65	41
QiuShiBaiKe	40	30	20	14	–	–
SaoleiGame	78	58	78	56	82	64
SgSearch	46	38	36	27	29	18
Terminal	100	80	100	80	100	76

TABLE IV
COVERAGE COMPARISON ON SOURCE CODE

App Name	LAND		Monkey		Dynodroid	
	MC	LC	MC	LC	MC	LC
aLogcat	69	62	57	51	68	60
Budget	64	56	50	45	–	–
HotDeath	71	55	54	43	69	53
TippyTipper	70	64	67	59	–	–
WhoHasMS	65	53	60	47	–	–

We compare the coverage of the test suites generated by different testing tools and the detailed information about coverage results are in Table III and IV [1]. The first table shows that the #CC (%) and #MC (%) of all apps and the second one shows the #MC (%) and #LC (%) of the apps with source code. Besides, for some of the instances, Dynodroid fails to report the test results and we use "–" to represent them in the tables. As shown in the tables, LAND can reach higher coverage (about 20% improvement on average) in most cases than Monkey and Dynodroid.

Let us use a specific app, i.e., *Tippy Tipper* to demonstrate how the widget status and the back stack influence the GUI model of the app. The app *Tippy Tipper* is a popular open source calculator app, which can be used for calculating the tip amount for a meal. The user can enter the meal amount on

[1]Table IV only gives the results of five apps of the ten open-source apps, as *EMMA* crashes due to engineering reasons and fails to measure the coverage of the rest five apps in our experiments.

the entry screen and get the result by clicking the `Calculate` button. Both the entry and result screen are attached with a menu. If the menu item `Setting` is selected, the app will take the user to the Setting window. The paths that can reach the Setting window are ⟨Entry, Setting⟩ and ⟨Entry, Result, Setting⟩. Although the final GUI window "Setting" they reach is the same, their back stacks are different. If we send a Back event to the app at these two windows, their behavior will be different. Therefore, they should not be merged as one state. Besides, there is a `CheckBox` called `Enable Exclude Tax Rate` on the Setting window. If its status is "checked", the button `Tax Rate to Exclude` below it will be enabled on current Activity, or else disabled. Clicking the button `Tax Rate to Exclude` will drive the app to a new dialog window. In this occasion, the status change of a widget influences other widgets related to it, furthermore, it influences the corresponding events of current state. We measured the size of model influenced by widget status and the back stack in this case. Without considering these characteristics , the model contains 10 states and 172 transitions. And it will grow to 23 states and 450 transitions if these details are considered.

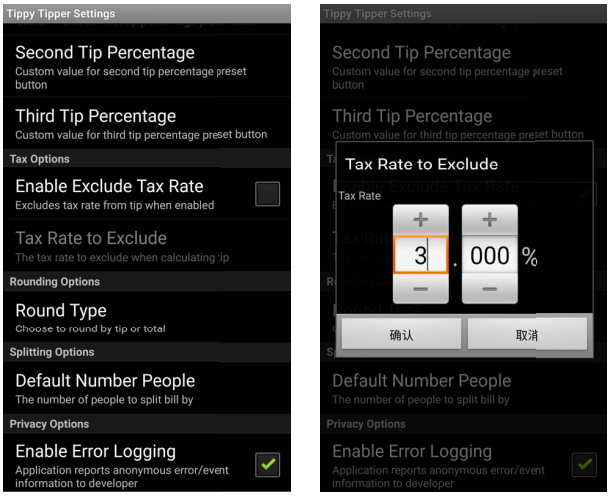
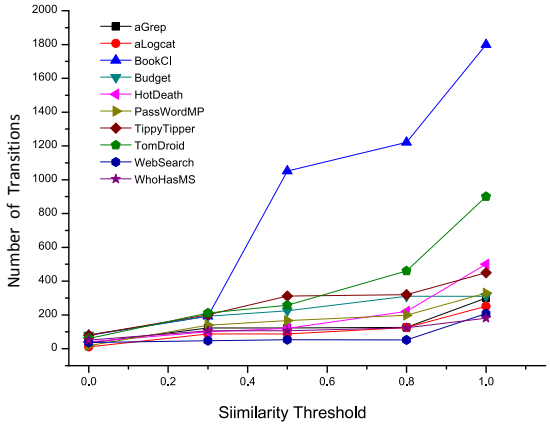


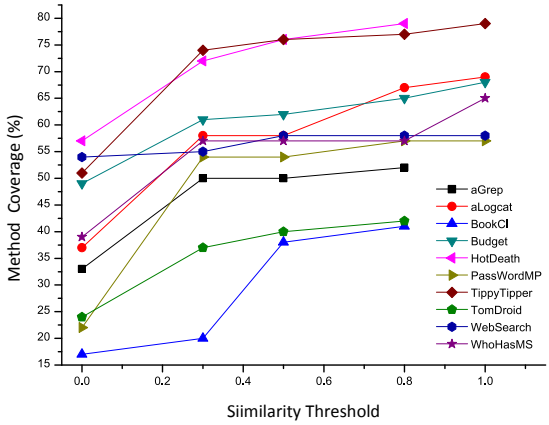
Fig. 5. Tippy Tipper Application

C. State Similarity and Model Size

In this subsection, we will discuss the impact of S_T on the size of the model and the benefit to the coverage from high S_T. The value of threshold S_T is set from 0 to 1. We apply our tool to ten experimental instances to generate the LATTE model, and compare the numbers of states and transitions in the generated LATTE model and the method coverage of this model under different values of S_T. Here, the method coverage is calculated as the total number of methods in the app divided by the number of methods executed during the model construction, which is an indicator about the code coverage of the model. A high threshold S_T may cause an extremely large even infinite model size. For example, apps with the file picking functionality will have an extremely large model size that causes its window to dynamically change a lot. Therefore, we set 3 hours as the upper bound of the execution time, record the final number of transitions and do not show the coverage result if the model construction is not finished within this bound.



(a) S_T and Model Size



(b) S_T and Coverage

Fig. 6. Impact of Similarity Threshold

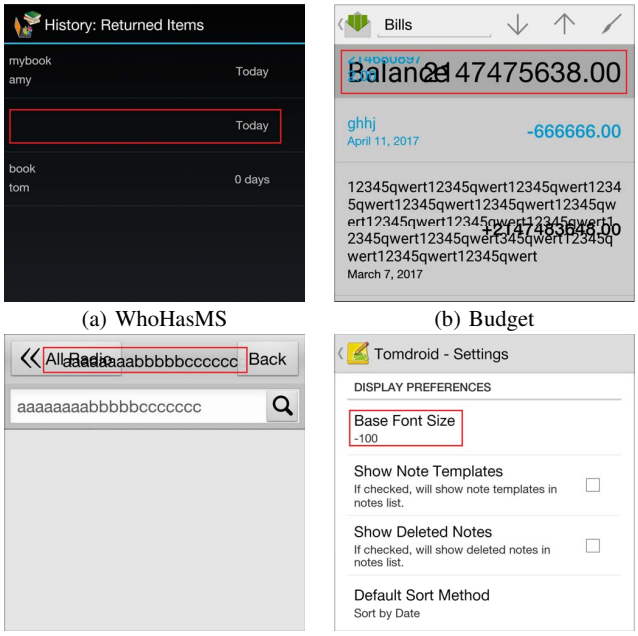
Fig. 6 demonstrates the tendency of the number of transitions in model and the method coverage on byte-code of generated test cases under different S_T value for the ten open-source apps. As we can see, with the increase of similarity threshold S_T, the size of the model increases dramatically. Obviously, the cost of model construction will increase accordingly. However, the coverage of test suites generated from the model will not increase significantly when S_T reaches a certain level. Therefore, S_T is a proper control variable to make a trade-off between the accuracy and efficiency. We find that 0.8 is a reasonable choice according to the result and use it as the default value in the following parts.

D. Application Scenarios

In this section, we will briefly introduce two application scenarios based on our LATTE model.

1) Test Text Input Widgets: For the 20 apps in our benchmark, 18 apps have at least one `EditText` widget. We

observe that the app *Budget* crashes by throwing the exception `NumberFormatException` under our generated string as input. Besides, four apps display unfriendly interfaces, including empty item name (*WhoHasMS*), messy long string displaying (*Budget*, *Cradio*) and blank window caused by abnormal font size setting (*Tomdroid*). Some of these abnormal displays are shown in Fig. 7. These experimental results indicate that our model can be effectively reused for the in-depth testing of `EditText`.



(a) WhoHasMS (b) Budget

(c) Cradio (d) Tomdroid

Fig. 7. Abnormal Displays

2) Test Generation for Covering Targets: In this part, we set different targets for the randomly picked experimental instances, and compare the targeted event sequences generated using LATTE model and random event sequences generated by Monkey, which is shown in Table V. The first column shows the application name, while the second column gives the targets in the apps. We select several user-defined methods (e.g., `showCardHelp`) as well as the system APIs that are related to either resources (e.g., `AudioRecord.release`) or privacy (e.g., `ContactsContract.content_uri`) information. The third column (#MK) gives the minimal number of events for Monkey to cover the target. Note that the number of events n denotes Monkey can cover the target by an event sequence whose length is in the range of $(n - 1000, n]$. The fourth column (#EM) gives the number of generated events in the model construction and the fifth column (#ET) gives the minimal length of the event sequences generated using LATTE model that involve the target.

TABLE V
TEST GENERATION FOR COVERING TARGETS

App Name	user-defined method/ system API	#MK	#EM	#ET
Cradio	search,add,addDel	4000	209	14
HotDeath	showCardHelp	3000	221	3
PassWordMP	onExportClick	12000	199	2
SgSearch	addCard	15000	869	4
TomDroid	delNote,undelNote	42000	460	12
BubeiListen	Url.openConnection	13000	566	5
Budget	VelocityTracker.obtain	6000	310	2
SaoleiGame	Chronometer.stop	3000	81	2
QiuShiBaiKe	Camera.open	4000	1814	2
WhoHasMS	ContactsContract.content_uri	8000	124	2

VII. RELATED WORK

There are many kinds of GUI testing approaches for Android apps, including model-based testing, random testing and systematic testing techniques. In this section, we will introduce several representative works based on these approaches and highlight the differences between our work and theirs.

Model-Based Testing. Several researches construct the model by static analysis for testing. W. Yang et al. [31] model the GUI behavior of application as an FSM. They proposed an approach that uses static analysis on Java source code of Android to extract actions associated with view components on a GUI state. S. Yang et al. [30] provided a similar model called Window Transition Graph (WTG), with a more accurate static callback analysis. It gives a more careful model containing the stack and window transition information. There are some differences between them and our work. The first one is that their model construction relies on the source code of the app, while we can handle the apk file directly. The second one is that they build a model statically that might miss the changes of GUI screen during runtime. Mirzaei et al. [27] aim at extracting the dependencies of GUI elements to reduce the number of test cases. The reduction is under the help of Interface Model and Activity Transition Model (ATM), which describe the GUI inputs of widgets and transition relationship of Activities respectively. However, the ATM omits the back stack attribute and the dynamic window changes.

Some researchers [16, 19] leverage dynamic techniques to construct the model. Amalfitano et al. [10, 11] implemented tool *AnroidRipper* to explore the GUI widgets of the app. However, the approach always creates a new state in the model after an event is triggered without calculating the similarity between windows. As a result, each GUI window may be included multiple times, which makes the size of the model too large. Azim et al. [14] design a static analysis algorithm on the app to extract the Static Activity Transfer Graph (SATG), and use dynamic GUI exploration to handle dynamic Activities layouts to complement the SATG. They also present a tool A^3E for systematically exploring real-world apps. However, they regard the Activity as the minimum unit in ATG and also do not consider the different statuses of widgets and the back stack information of the same Activity. Because of automated testing mostly falls short of handling mobile apps' complex interactions, Li et al. [23] present a user-guided technique which exploits user insights to complement automated testing. This approach can work with other GUI-testing approaches.

There are many factors that affect the precision of the model for testing. For example, our work has not taken into consideration of the diversity of different mobile platform features. Galindo et al. [18] proposed an approach that uses automated analysis of feature models to optimize the testing of variability-intensive systems. Their approach can help to maximize the benefit while meeting a budgetary constraint of testing cost.

We have tried several model-based testing tools, such as Android Ripper and A^{3}E. These tools were developed years ago and are incompatible for newly released Android apps, including most of our benchmarks, which also has been mentioned by Mirzaei et al. [27]. Therefore, the experimental comparison between these tools and LAND is not included in this paper.

Random Testing. In random testing, the test events will be generated randomly with less care of current state of the application under test. Monkey [4] is a widely used black-box testing tool, which can send sequences of random events to Android apps. It is a simple and fully automatic tool that can generate a great deal of test events within a short time. Zent et al. [32] use Monkey to conduct an industrial case study and report Monkey's limitations in an industrial setting. There are also works based on Monkey for detecting GUI bugs [20] and security bugs [26]. However, Monkey is not suitable for generating highly specific event sequences. Dynodroid [25] proposed by Machiry et al. provides a more efficient random GUI exploration approach compared with Monkey. They define several strategies for selecting events to guide the test generation procedure and support system event generation by instrumenting the Android framework.

Systematic testing. Systematic testing is another testing approach that will be applied in more complicated circumstances. For example, we must generate specific testing data to reveal the specific application behavior. ACTEve [12] is a concolic-testing tool that generates sequences of events automatically and systematically. It symbolically tracks events from the generated point in the framework to the handled point in the app, thus both the framework and the application under test need to be instrumented. Jensen et al. [21] provides another concolic-testing approach that aims at automatically finding event sequences that reach a given target line in the application code. This approach improves automated testing for Android applications that are not computationally heavy but may have complex user interaction patterns. However, their work using symbolic execution can only process `Integer` but not `String` or other complex data types. In addition, this approach also needs a model for test case generation and they build it manually.

VIII. Conclusion

We proposed a widget-sensitive and back-stack-aware exploration approach for testing Android apps. During the exploration, we build the LATTE model which takes into consideration the changes of GUI widgets during runtime omitted by static analysis as well as the back stack information using a dynamic construction approach. To balance the accuracy and size of model, we introduce a metric "state similarity" to merge similar states. Our model also represents some of the code information via the label mechanism. The experimental results show that with the help of comprehensive information of widget and back stack, the GUI exploration approach can describe the behavior of apps more accurately and completely and thus achieve a higher coverage compared to the state-of-art exploration tools. It also indicates that the generated LATTE model is well-designed to be reused for further testing.

Currently the functionalities of our GUI exploration tool rely on the testing framework for generating events to drive the app. In the future, we will manage to enhance a testing framework by supporting the cross-app testing and system events, etc.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. This work is supported by National Natural Science Foundation of China (Grant No. 61672505) and the National Key Basic Research (973) Program of China (Grant No. 2014CB340701) as well as the CAS/SAFEA International Partnership Program for Creative Research Teams.

References

[1] Android developers. ActivityInstrumentationTestCase2. http://developer.Android.com/reference/Android/test/ActivityInstrumentationTestCase2.html.

[2] Android developers. Back Stack. http://developer.Android.com/guide/components/tasks-and-back-stack.html.

[3] Android developers. Launch Mode. http://developer.Android.com/guide/topics/manifest/activity-element.html#lmode.

[4] Android developers. ui/application exerciser monkey. http://developer.Android.com/tools/help/monkey.html.

[5] Apktool. ibotpeaches.github.io/Apktool/.

[6] Emma. http://emma.sourceforge.net.

[7] F-Droid. https://f-droid.org.

[8] Google code. Robotium. http://code.google.com/p/robotium/.

[9] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A GUI crawling-based technique for Android mobile application testing. In *ICST 2011*, pages 252–261, 2011.

[10] 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 2012*, pages 258–261, 2012.

[11] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

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

[13] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI 2014*, page 29, 2014.

[14] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA 2013, part of SPLASH 2013*, pages 641–660, 2013.

[15] Y. M. Baek and D. Bae. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *ASE 2016*, pages 238–249, 2016.

[16] W. Choi, G. C. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA 2013, part of SPLASH 2013*, pages 623–640, 2013.

[17] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? In *ASE 2015*, pages 429–440, 2015.

[18] J. A. Galindo, H. A. Turner, D. Benavides, and J. White. Testing variability-intensive systems using automated analysis: an application to android. *Software Quality Journal*, 24(2):365–405, 2016.

[19] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan. PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *MobiSys 2014*, pages 204–217, 2014.

[20] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *AST 2011*, pages 77–83, 2011.

[21] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *2013*, pages 67–77, 2013.

[22] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *ICSE 2015*, pages 280–291, 2015.

[23] X. Li, Y. Jiang, Y. Liu, C. Xu, X. Ma, and J. Lu. User guided automation for testing mobile apps. In *APSEC 2014*, pages 27–34, 2014.

[24] J. Liu, T. Wu, X. Deng, J. Yan, and J. Zhang. Insdal: A safe and extensible instrumentation tool on dalvik bytecode for Android applications. In *SANER 2017*, pages 502–506, 2017.

[25] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for Android apps. In *ESEC/FSE 2013*, pages 224–234, 2013.

[26] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A whitebox approach for automated security testing of Android applications on the cloud. In *AST 2012*, pages 22–28, 2012.

[27] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing combinatorics in GUI testing of Android applications. In *ICSE 2016*, pages 559–570, 2016.

[28] F. Song and T. Touili. Model-checking for Android malware detection. In *APLAS 2014*, pages 216–235, 2014.

[29] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. *IEEE Trans. Software Eng.*, 42(11):1054–1076, 2016.

[30] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for Android. In *ASE 2015*, pages 658–668, 2015.

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

[32] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie. Automated test input generation for Android: are we really there yet in an industrial case? In *FSE 2016*, pages 987–992, 2016.

[33] H. Zhu, X. Ye, X. Zhang, and K. Shen. A context-aware approach for dynamic GUI testing of Android applications. In *COMPSAC 2015*, pages 248–253, 2015.