AIMDROID: Activity-Insulated Multi-level Automated Testing for Android Applications

*Tianxiao Gu, *‡Chun Cao, *Tianchi Liu, †Chengnian Sun, *Jing Deng, *Xiaoxing Ma, *Jian Lü,
*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
*Department of Computer Science and Technology, Nanjing University, Nanjing, China
†Department of Computer Science, University of California, Davis, USA.
{tianxiao.gu,chengniansun,liutianchi92,dmemoing}@gmail.com, {caochun,xxm,lj}@nju.edu.cn

***Abstract*—Activities are the fundamental components of Android applications (apps). However, existing approaches to automated testing for Android apps cannot effectively manage the transitions between activities, *e.g.*, too rarely or too often. Besides, some techniques need to repeatedly restart from scratch and revisit every intermediate activity to reach a specific one, which leads to unnecessarily long transitions and wasted time. To address these problems, we propose AIMDROID, a practical model-based approach to automated testing for Android apps that aims to manage the exploration of activities and meantime minimize unnecessary transitions between them. Specifically, AIMDROID applies an activity-insulated multi-level strategy during testing and replaying. It systematically discovers unexplored activities and then intensively exploits every discovered individual with a reinforcement learning guided random algorithm. We conduct comprehensive experiments on 50 popular closed-source commercial apps that in total have billions of daily usages in China. The results demonstrate that AIMDROID outperforms both SAPIENZ and Monkey in activity, method and instruction coverage, respectively. In addition, AIMDROID also reports more crashes than the other two.**

***Keywords*-Android Application Testing; Model-based Testing; Reinforcement Learning;**

I. INTRODUCTION

Today's Android devices provide more and more powerful computation capability, which gives rise to the increasingly rapid grow of Android applications (apps). For example, the largest app in our evaluation has more than 180 thousands of method and 3.7 millions of bytecode instructions. Besides, there are over 3.1 million Android apps available from the Google Play but unfortunately 12% of them are low quality apps as of August 2017 [1].

Android apps are GUI-based applications driven by various events. An app usually has a number of activities [2] (*i.e.*, GUI windows) and must adapt to various versions of Android to provide a nearly uniform look-and-feel. These features make it hard to test Android apps [3], [4], [5]. Android app testing still relies heavily on manual testing [6], [7].

Monkey [8] is regarded as the state-of-practice automated testing technique [9], [10], which is released by Google as a built-in tool in Android. Monkey can generate pseudo-random streams of GUI events such as touching and swiping. However, Monkey may spend much time in generating a long sequence of events, which includes many redundant events [11] (*e.g.*, repeatedly jumping between activities) and unfruitful events [12] (*e.g.*, clicking non-interactive area on the screen). Such an exceptionally long event sequence also makes it hard for developers to replay and diagnose [13], [11].

Some techniques leverage GUI layouts to reduce unfruitful GUI events [14]. Moreover, these techniques develop various new strategies to guide the exploration, *e.g.*, using the frequency of actions [14]. However, these techniques lack the management of exploration of activities. For example, Monkey allocates a lopsided distribution of exploration time on activities during testing a commercial app, where the top 4 activities consumes 43.4% time budgets [12]. To mitigate this problem, Zeng *et al.* [12] limit the total time of exploring an activity in order to balance the visiting time of every activity. However, they may still make unnecessary activity transitions before reaching the time limit.

To guide the testing, systematic approaches leverage symbolic execution or evolutionary computation techniques to steer the exploration of the app to some specific part. However, these techniques are less scalable than Monkey [10]. Another popular technique is model-based testing [15], [16], [17], [18], [19]. These approaches usually first build a state machine as the model of the app and then develop a number of exploration strategies such as Depth-First Search (DFS), Breath-First Search (BFS) or hybrid to systematically explore the model [17], [18].

Existing model-based approaches have worse over-all performance than Monkey on some benchmark apps [10]. First, systematic exploration strategies like DFS need to repeatedly restart the app for backtracking [10], [16]. Second, building a precise and complete model is extremely challenging in practice. There do exist non-deterministic transitions/events, which can easily interrupt the steering of the exploration of the app along with the model. Third, model-based approaches also have the *state explosion problem*. For example, a finite state model that takes account of text of widgets may not even exist for real-world apps [18]. In our opinion, the key solution to improve the performance of model-based techniques is to reduce search complexity and restart time.

We propose AIMDROID, the first offering the **A**ctivity-**I**nsulated **M**ulti-level strategy to model-based automated testing for Android apps, which seeks to maximize the cover-

‡Corresponding author: Chun Cao.

age and fault detection while controlling the length of test sequences. AIMDROID models and explores the *apps under test (AUT)* using a multi-level method [18]. The key insight of AIMDROID is twofold. First, AIMDROID systematically discovers every unexplored activity using a BFS algorithm. Second, AIMDROID insulates a discovered activity in a "cage" and intensively exploits the insulated activity with a reinforcement learning guided fuzzing algorithm. Therefore, we can simplify the search complexity, save restart time and manage the time consumed in each activity.

Specifically, AIMDROID divides the overall testing into episodes. In each episode, AIMDROID generates a bounded number of events and focuses on a single activity by disabling activity transitions. In addition, AIMDROID uses a reinforcement learning algorithm, *i.e.*, SARSA [20], to learn the ability of events that can discover new activities. With SARSA, AIMDROID can "looks ahead" and select events that are more likely to trigger new activities and crashes in a greedy manner.

To further save restart time, AIMDROID can start an explored activity for another episode without restarting from scratch. In the normal usage, we need a proper event sequences to reach an activity. Actually, some internal activities can be started directly by providing a proper *intent* [21] only. To this end, we record the intent when starting an activity. Then, we try to start the activity directly first during the remaining test using the saved intents and then restart from the scratch if the intent fails. This method is also used during replaying. Thus, AIMDROID can also provide a short event sequence for replaying.

In this paper, we make the following contributions:

- We propose a novel activity-insulated multi-level strategy for model-based automated testing for Android apps.
- We have implemented our strategy in a practical testing tool named AIMDROID, which is publicly available[1].
- We conduct comprehensive experiments on 50 popular close-source commercials apps to demonstrate the effectiveness of AIMDROID.

The rest of the paper is organized as follows. We first introduce the basic idea of AIMDROID with an illustrative example in Section II. Next, we present the design and implementation of AIMDROID in Section III. Then, we show the experiment results in Section IV. Finally, we discuss related work in Section V and conclude in Section VI.

II. ILLUSTRATIVE EXAMPLE

A. Automated Testing For Android Apps

Existing approaches to automated testing for Android apps can be classified into three categories, *i.e.*, fuzz (random) testing [8], model-based testing [15], and systematic testing [22], [23], [24]. Each kind of approaches tries to efficiently generate a number of events under a particular strategy to drive and explore the apps thoroughly. Though there are many kinds of events that Android supports, the GUI events are the most popular [14], [12], which can be generated by mimicking GUI *actions* (*e.g.*, click). As events are generated by actions, we may use them interchangeably in the following sections.

We observed that almost all existing approaches lack an effective method to manage the exploration time of activities [14], [15], [16], [18], [19]. Since an activity is the entry point to all widgets on it, we should first discover as many as possible activities and then intensively explore every discovered activity for sufficient time to gain confidence (*e.g.*, high coverage) and detect defects. However, existing approaches can trap in certain activities [12] and thus fail to discover new activities in bounded time. Moreover, the intensive exploration of an activity can be interrupted if actions that can jump to other activities are taken too often.

B. Activity and Its Lifecycle

Activities are the fundamental components of Android apps. An activity is a collection of widgets that are organized into a tree-like structure. We can easily obtain this tree using a black-box tool, *i.e.*, UI Automator Viewer [25], which is also released within the Android platform. UI Automator Viewer also provides various attributes of each widget. Thereby, we can determine the appearance, *e.g.*, bounds or text, and the functionality, *e.g.*, clickable or scrollable, for each widget, which can be leveraged to determine available actions.

The lifecycle of an activity is tightly coupled with the Android framework [26]. An activity may be interrupted and change its lifecycle status unexpectedly to respond user interactions or environment changes, *e.g.*, from foreground to background by phone calls. The hiding of a top activity actually gives rise to the transition to a new top activity. An Android app, particularly a commercial app, usually has hundreds of activities (See Table I). As a result, the transitions between activities are common during both daily usages and testing.

The lifecycle of an activity is managed by an essential service in Android called *Activity Manager Service (AMS)*. AMS provides various methods for an activity to start another activity. For example, to start another activity, even of another app, an activity needs to invoke a method like `startActivity` to notify the AMS of the request. Method `startActivity` requires as parameter an *intent* that contains the *URI* [27] of the request activity. In addition to the intent, an activity may also rely on persistent data during changing its lifecycle.

C. Illustrative Example

AIMDROID is a model-based approach that aims to manage the exploration of each activity and meanwhile reduce unnecessary transitions between activities. Figure 1 is an example with three activities, *i.e.*, A_1, A_2 and A_3, which describes the flow of a shopping app. We use the large circle to represent the activity and the small circle to represent the *intra-activity states* inside an activity. There is no gold rule to identify states. In practice, most existing techniques make use of the widgets inside an activity [16], [14], [18], [19].

[1]https://icsnju.github.io/AimDroid-ICSME-2017/

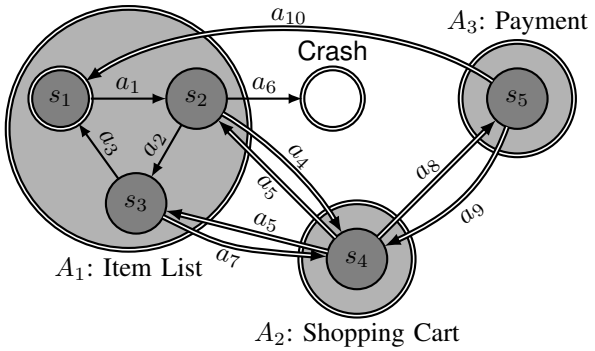


Fig. 1: Illustrative Example

Specifically, A_1 lists the items for shopping, while A_2 is the shopping cart and A_3 is for the payment. An end user may start from A_1 and search for interested items by action a_1. The user may alter the search keywords or filter conditions by action a_2 or even clear the search keyword by action a_3. Besides, the user may also choose an item and jump to the shopping cart to preview the shopping list by action a_4 or a_7. The state then becomes s_4, which belongs to activity A_2. The user may continue to add items and go back to s_2 or s_3 by action s_5, or checkout to complete the payment by action a_8, where the states become s_5. A good app with well designed user experience usually provides an opportunity to cancel the payment. Hence, the state may become s_4 again by action a_9. Finally, the state becomes s_1 again if the payment is confirmed by action a_{10}.

Actions have four kinds of results, *i.e.*, *state transitions*, *activity transitions*, *app transitions* and *crash*. The most common transitions are between states of a same activity. Note that the transition may be a cycle if the source state and the target are the same. The transition is treated as an activity transition if the target state belongs to another activity and an app transition if the target state or the target activity belongs to another app. We will discuss why to treat these kinds of transitions differently in Section III-D.

We have the following observations on the results of actions.

1) Most transitions between activities are bi-directional, because there is a BACK button for Android devices, either physical or virtual [28]. For example, a_5 and a_9 are actions made by the BACK button.
2) There also do exist uni-directional transitions between activities. In this situation, we need to restart from the initial activity to reach the target activity again. For example, we need start from s_1 and follow a_1, a_4 or a_7, and a_8 to visit s_5 or A_3 again.
3) Activity transitions are prevalent, particularly in commercial apps. Many actions on a same state or activity can even make the same transitions. For example, both a_7 and a_4 can also jump to A_2 from A_1. These actions can interrupt the intensive exploration of A_2.
4) Every activity has a different number of states and actions. Some are worthy of more explorations. For example, intuitively we should stay in A_1 for longer time than the other two activities because it has more internal states and actions, and much more complex interactions between internal states and actions.

Unfortunately, we observed that many actions that are made by existing approaches result in unnecessary transitions between activities. For example, a random based approach may generate equal number of actions. But actually, we should generate more actions like a_2 or a_3. Actions such as a_4, a_7, and a_8 are also important because they can discover new activities but at the same time must not be taken too frequently, because they can interrupt the exploration inside an activity. Actions like a_5 or a_9 should be considered less important because they sometimes just go back to a previously visited activity. Besides, a systematic approach that aims to cover every action may take actions a_1, a_4, and a_8 again to visit a_9 on s_5 if the first visited action on s_5 is not a_9 but a_{10}.

To mitigate these problems, we first introduce the activity-insulated strategy to block unnecessary transitions between activities during exploiting an activity. This strategy can balance the time of each activity according to its available actions. For example, we may stay in A_1 for a long time to exploit thoroughly by blocking a_4 and a_7. Besides, we record every intent that is used for starting an activity. By this way, we can avoid replaying the previous actions to reach the final state again and only replay the entire action sequence if the intent fails. Thus, if we want to explore A_3, we could directly start A_3 without restarting from A_1 and A_2.

Within our experience, a certain number of activities can be started directly by providing a proper intent only (See Table I). This may be because first different activities may be developed and tested by different groups following the principle of *separation of concerns*. Second, a good mobile app usually consider the state persistence for unexpected interruptions. For example, the battery may run out of charge and a call may arrive at any second. Thus, an activity must support saving and restoring from persistent data directly.

Apparently, an action or a transition that can start a new activity is worthy of revisiting multiple times, which helps to discover new intents as well. To learn such ability of an action, we apply a *reinforcement learning (RL)* enhanced fuzzing algorithm during exploring an activity. With this enhancement, AIMDROID can select actions that are likely to trigger new crashes or new activities.

III. THE AIMDROID APPROACH

We first give an overview of AIMDROID, and then discuss its design and implementation in detail in this section.

A. Overview

Figure 2 depicts an overview of AIMDROID. AIMDROID has various components and is implemented in a client-server style. The server is a controller that actually guides the exploration. The client is an Android device, which is either an emulator or a real device that is connected to the controller via networking. AIMDROID takes the APK file of the AUT as input and produces various reports for problem diagnosing

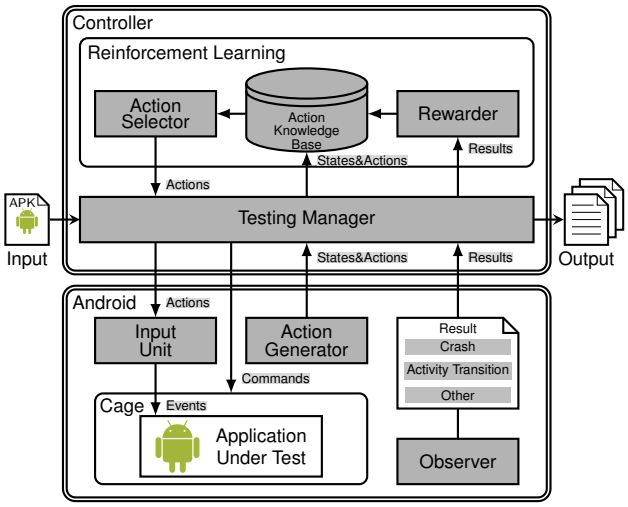


Fig. 2: Overview of AIMDROID.

as output. The overall testing is divided into episodes. Each episode focuses on exploring a single *target activity*. The target activity is insulated in a *cage*, which blocks any transitions to any other activity of the AUT from the target activity.

Each episode is further divided into a number of iterations by a fixed time interval. During each iteration, AIMDROID first rips the current GUI to build the current state and actions. Next, AIMDROID selects an action based on the knowledge maintained by the reinforcement learning module. On the other hand, AIMDROID captures the results caused by the previous action, which are used to update the knowledge about the action in the reinforcement learning module. Finally, the selected action is used to generate events to feed the AUT.

B. Multi-level State Representation

AIMDROID uses two levels of states to guide the testing. In addition to the intra-activity states (*i.e.*, inner-states), AIMDROID considers an activity as an outer-state. An illustrative example is shown in Figure 1. In the following sections, we may also use "activity" to refer to the outer-state, and "state" to refer to the intra-activity state or the inner-state interchangeably.

Baek *et al.* [18] first proposed the idea of applying multi-level states for Android app testing. However, their approach picks a single fixed level of state at a time and applies the same exploration strategy to them. In contrast, AIMDROID uses two levels of states at the same time and applies different exploration strategies to them. More details about the exploration strategy can be found in Section III-C.

The outer-state space has no state explosion problem. The number of activities of an app is usually not significant. For example, commercial apps used in our evaluation usually have hundreds of activities. Consequently, AIMDROID can *systematically explore activities*. Apparently, this coarse-grained state representation cannot be used in intra-activity exploration. Thus, we need a fine-grained inner-state representation to help generate actions for intra-activity exploration.

An inner-state is derived from the widgets inside an activity. In a nutshell, we identify a widget by its bounds on the screen and use the set of actions on *interactive* widgets as an inner-state [16]. Note that there is only a limited number of widgets visible on the screen. For example, a list has 100 items but maybe only 10 of them are visible at a time. We need to swipe down to show another 10 items with different but similar contents. The new 10 items have the same bounds and the new GUI layout is treated as the same state. Hence, using bounds only can help to reduce the inner-state space.

C. Activity-Insulated Multi-level Strategy

We apply different strategies to each level of states. At the activity level, we apply model-based methods to systematically explore activities. Inside an activity, we use a reinforcement learning guided fuzzing algorithm to mitigate the problem of state explosion in the inner-state space. We present the overall exploration strategy of AIMDROID in Algorithm 1.

Algorithm 1 systematically traverses activities using a BFS strategy and dynamically constructs a BFS tree (*i.e.*, $\mathbb{T}$) to represent the simplified transitions of activities, *i.e.*, a tuple (A, I', A') of the source activity A, the intent to start the target activity I', and the target activity A' (at line 21). The BFS tree is used by function `quickLaunch` to start activities directly.

The exploration starts from the main activity as the target activity at line 3. In every iteration of BFS, a new activity is removed from the queue as the target activity. Once the target activity has been launched, AIMDROID insulates the activity in the cage and starts to explore the inner-states of the activity (in the function `exploreInCage`). During the in-cage exploration, AIMDROID enqueues *newly discovered* activities into the queue, and saves the simplified transition in the BFS tree at line 22. The BFS strategy ensures that each activity is in theory explored once by the function `exploreInCage`, Hence, AIMDROID can reduce the restart and backtracking frequency, which wastes much time in traditional model-based approaches [16].

In fact, activity transitions are a common behavior in Android apps. It has some technical difficulties to insulate activities. Consequently, previous work usually pays little attention to the impact of activity transitions on their effectiveness [8], [16], [13], which causes two aforementioned problems, *i.e.*, trapping in certain activities and wasting time on restarting apps for backtracking to particular states/activities. To address these problems, we propose the activity-insulated strategy to explore activities and their intra-activity states with different strategies respectively to reduce the searching complexity.

1) Insulate Activity: Actions have four types of results.

- R_1*: Activity Transitions.* An action starts another activity of the AUT or just finishes itself and goes back to the parent activity. We use $\hat{R_1}$ to denote a fresh activity transition, where the target activity is firstly discovered.
- R_2*: App Transitions.* An action invokes another app or just closes the AUT.
- R_3*: Crash.* An action exposes a crash in the activity. As many crashes may be manifested by the same error, we

Algorithm 1: AIMDROID.

Input: $\mathbb{A}$, the AUT

Output: $\mathbb{T}$, the BFS tree of activities represented by a set of activity transitions, $\mathbb{S}$, the set of action sequences, $\mathbb{C}$, the set of crash reports.

```
1  (Q, T, S, C) ← (∅, ∅, ∅, ∅)
2  launch(A)
3  A ← getActivity()
4  Q ← enqueue(Q, A)
5  T ← T{(∅, ∅, A)}
6  while Q ≠ ∅ do
7      (Q, A) ← dequeue(Q)
8      (Q, T, S, C) ← exploreInCage(A, A, Q, T, S, C)
9  Function exploreInCage(A, A, Q, T, S, C)
10     while true do
11         quickLaunch(A, A, T, S)
12         (s, a, S) ← (∅, ∅, ∅)
13         stop ← true
14         while true do
15             s' ← getState()
16             a' ← getAction(s')
17             SARSA(s, a, s', a')
18             R ← getResult(s, a, s')
19             if R = R̂1 then
20                 A', I' ← getActivityAndIntent()
21                 T ← T ∪ {(A, I', A')}
22                 Q ← enqueue(Q, A')
23                 stop ← false
24             if R = R̂3 then
25                 C ← C ∪ getCrash(R)
26                 stop ← false
27             S ← append(S, {(a, R)})
28             if R ∈ {R2, R3} ∨ |S| ≥ bound(A) then
29                 S ← S ∪ S
30                 break
31             s ← s', a ← a'
32             execute(a')
33         if stop then
34             return (Q, T, S, C)
```

use $\hat{R_3}$ to denote a fresh crash, where the corresponding error is firstly exposed.

- R_4*: State Transitions.* An action leads to a transition to another state of the activity or even the same state.

Activity transitions are concerned mostly in our approach. AIMDROID blocks this kind of transitions and insulates the currently visiting target activity from other activities by putting it in the cage. The cage doesn't block transitions between states of the target activity. AIMDROID doesn't prevent app transitions as well. In this way, AIMDROID has the opportunity to test whether the AUT can interact with other apps properly. In the meantime, we also don't expect this kind of transitions to occur often. Crashes are unpredictable and unstoppable. Thereby, AIMDROID doesn't deter this kind of behaviors and just records the stack information of every crash.

To block activity transitions, we intercept the corresponding transition request. AIMDROID embodies the black-box testing methodology. Thus, we choose to instrument the Android framework instead of the AUT. Recall that the lifecycle [26] of an activity is managed by AMS. For example, to start another activity, an activity needs to invoke the method `startActivity`, which requires an intent that contains the *URI* [27] of the request activity. We instrument the method and all its variants in the Android framework to implement the barrier for the cage. The barrier is closed and opened by the testing manager when necessary. Specifically, the testing manager opens the barrier and starts the activity normally when it attempts to launch the target activity, and closes the barrier once the target activity has been launched. When blocking an activity transition, AIMDROID records the corresponding intents to implement the function `getActivityAndIntent` and `quickLaunch` in Algorithm 1.

2) Quick Launch: In order to reach a certain activity or state, traditional model-based techniques need to restart the AUT and replay the entire event sequence that can drive the AUT from the initial state to the target state. Even though AIMDROID has reduced the frequency of restart with the activity-insulated strategy, we expect to further expedite the process of activity launching by the *quick launch* method.

As previously described, an intent contains the URI and necessary data for starting an activity. Hence, AIMDROID tries to start the target activity directly with the intent. As shown in Figure 1, suppose that A_3 is the target activity. Traditional approaches need to restart the AUT to reach A_1 first and then replay the action sequence $\langle a_1, a_4, a_8 \rangle$ to reach A_3. By the quick launch method, AIMDROID only needs to directly start A_3 with the intent, which avoids replaying the action sequence.

Obviously, the quick launch approach is able to save time of replaying events, but there do exist activities that strongly depend on other activities. It may result in false crashes to start activities directly. In this situation, AIMDROID tries to directly start the parent activity A_2, and then replay the action a_8 to reach A_3. Then, the launching process of AIMDROID contains only a single action, *i.e.*, a_8.

Although these crashes are caused by launching with the intent directly and thus may not happen in normal use, AIMDROID still records the runtime context information of these crashes. In the worst case, AIMDROID falls back to the traditional approaches, which needs to replay the action sequence $\langle a_1, a_4, a_8 \rangle$.

D. Exploration in the Cage

Based on *SARSA* [20], we propose our exploration strategy for exploring the target activity in the cage. Function `exploreInCage` in Algorithm 1 illustrates the basic idea of the in-cage exploration. In trivial cases, the target activity is

explored only one episode to avoid unnecessary explorations. If AIMDROID can discover new activities or new crashes in the activity, the target activity is explored more than one episodes.

During an episode (*i.e.*, the outer loop of `exploreInCage` in Algorithm 1), AIMDROID limits the total number of iterations/actions based on the total available actions on an activity (by function `bound` at line 28 in Algorithm 1). We provide two parameters, *i.e.*, $minL$ and $maxL$, to further control the number of total iterations. Specifically, the function `bounds` simply returns the value of `min(max(`$minL$`,` $|\mathcal{A}_A|$`),` $maxL$`)`, where $\mathcal{A}_A$ is the union of all actions of all states of the activity A.

All actions during an episode are recorded into a new sequence $\mathcal{S}$ and saved into $\mathbb{S}$ for output. In addition to controlling the exploration by the length of the sequence, AIMDROID stops the in-cage exploration immediately if a crash occurs (*i.e.*, $R = R_3$) or the current state is out of the AUT (*i.e.*, $R = R_2$).

We assume that the number of new activities and new crashes discovered are finite in each activity. Therefore, AIMDROID can finish the exploration for each activity within a limited time. Traditional model-based approaches usually terminate their exploration if they cannot detect new states. Our strategy can stop the exploration if it cannot find new activities and crashes. Note that users can still make AIMDROID to re-explore the BFS tree again if testing time allows.

The action generator provides a function `getState` to determine the state and its available actions from the current GUI layout.

1) States and Actions: App behaviors can be suitably exercised by generating GUI events only [10]. The research [29] found that the time needed to trigger a crash, can be significantly reduced if the tools for GUI testing is aware of the visible widgets and specifically targets them. Therefore, AIMDROID obtains the visible widget tree of the current GUI first. Then action generator traverses the widgets in the tree to find interactive widgets and analyzes the attributes of those interactive widgets to collect actions. We use the bounds of a widget to refer to it. For simplifying the state representation, we use the set of actions on all interactive widgets to represent the current state, which is widely used in previous GUI testing techniques [16], [30].

2) Observer and Rewarder: As mentioned in Section III-C, an action in an activity can result in four kinds of transitions. The responsibility of the observer is to monitor the results. The rewarder then calculates rewards for actions based on the results. To maximize coverage and fault detection, we should give high rewards to actions that can discover new activities and new crashes. Therefore, we use the function in Equation 1 to calculate the reward of a result R, where $\hat{R_1}$/$\hat{R_3}$ denotes detecting a new activity/crash.

$$reward(R) = \begin{cases} 1 & R = \hat{R_1} \ or \ \hat{R_3} \\ -1 & R = R_2 \\ 0 & otherwise \end{cases} \quad (1)$$

3) Action Selector: We expect to use a fuzz testing strategy for exploring states to avoid systematically handling the mazy GUI layouts in each activity. However, it is unlikely that a simple random algorithm without any comprehension of the AUT can explore an activity sufficiently within a reasonable amount of time. Some techniques expect to improve random strategies by obtaining information of GUI widgets to generate actions and selecting the actions based on profiling information like frequency [14]. We think that the historical results and the inner relations of actions, which are neglected by previous fuzzing techniques, should also be considered [12].

Actions that can result in discovering new activities and new crashes, together with their prior actions, should have a high priority to be selected. As a result, AIMDROID should have the ability to learn from action results and predict which action can bring more benefits. Finally, we adopt the SARSA algorithm that conforms with our idea, and choose ϵ-greedy [31] as the learning policy. The function `getAction` is implemented as follows.

$$\texttt{getAction}(s) = \begin{cases} \arg\max_{a \in \mathcal{A}(s)} Q(s,a) & 1-\epsilon \\ \texttt{random}(\mathcal{A}(s)) & \epsilon \end{cases} \quad (2)$$

Equation 2 depicts the basic idea of the ϵ-greedy strategy, where $\mathcal{A}(s)$ is the action set of the state s. The value $Q(s,a)$ (*i.e.*, Q-value) represents the possible reward received for taking action a in state s in the following iterations. In the ϵ-greedy strategy, the action with the highest Q-value is selected with the probability $1-\epsilon$, while the action is selected at random from all available actions with the probability ϵ.

SARSA uses the following equation to update the Q-value of states and actions.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a)) \quad (3)$$

where r is the reward of action a. This equation takes currently learned information, newly acquired reward and future rewards into consideration. The importance of newly acquired reward and future rewards are determined by α and γ separately. The initial Q-value of an action on a new state is 1.

With the Q-value learned by SARSA, AIMDROID can use the ϵ-greedy strategy to select each action cleverly, which naturally balances the exploration and the exploitation of the AUT via a random and a greedy selection criterion, respectively. From Equation 3, we can know that the action with the largest Q-value is more likely to be selected repeatedly. But the largest Q-value will decrease if the corresponding action cannot receive positive rewards, *i.e.*, when they cannot trigger new crashes or discover new activities. Finally, the greedy selection criterion (*i.e.*, $\arg\max_{a \in \mathcal{A}(s)} Q(s,a)$) will guide the testing tool to determine another action instead.

E. Implementation

As shown in Fig. 2, AIMDROID is implemented in a client-server style. This design choice enables us to easily parallelize AIMDROID in the future work. At the client side, we employ *Xposed* [32] to implement the observer and the cage as two Xposed modules, which can be easily activated

and deactivated when necessary. The input unit is implemented on top of Monkey and supports six types of actions, including *touch*, *flip*, *trackball*, *drag*, *keyboard* and *type*. The action generator is implemented on top of UI Automator Viewer [25]. The latest UI Automator Viewer can dump layouts inside a `WebView`. This feature enables AIMDROID to test hybrid apps. The server side of AIMDROID is implemented in the *Go* programming language. Thereby, we can deploy AIMDROID cross different platforms.

AIMDROID generates a set of artefacts for crash diagnose, including *crash reports*, *activity transition reports* and *action sequences*. Activity transition reports are generated from the BFS tree (*i.e.*, $\mathbb{T}$ in Algorithm 1) to record the simplified transitions of activities in the AUT. Note that every action in the execution sequences (*i.e.*, $\mathbb{S}$ in Algorithm 1) are tagged with its actual result. Hence, these transitions and action sequences can be used together by the function `quickLaunch` during either testing or replaying.

Most closed-source commercial apps cannot be repacked to enable collecting method and instruction coverage. To this end, we implement a tracing tool by modifying the davlik runtime on Android 4.4 for SAPIENZ, and the ART runtime [33] for Monkey and AIMDROID on Android 6.0.1, respectively. The tracing tool traces every method except those in the Android framework. These method are forced to be interpreted by the interpreter. Then, we modify the bytecode interpreter to record every executed instruction in traced methods. During our evaluation, the performance degradation caused by disabling compilation is negligible on real devices. We also provide a command line tool to harvest the coverage data and a parser to generate the coverage report.

AIMDROID aims to be a practical black-box testing tool, though the current implementation relies on the Xposed, which needs to root the devices. Actually, we have figured out a new implementation approach that can be purely built on top of the Android framework. That means in the future we could deploy AIMDROID on any non-modified Android platform since Android 6. We briefly explain the mechanism here. To start an activity, we can use the command `am` [34] or the API it called to start an activity. Here, the activity must be implicitly or explicitly exported [35]. Though we cannot use the quick launch method for every activity, the false crashes caused by the quick launch method can be reduced. To block activity transition, we can use a hidden framework API used by Monkey. Monkey has already used this hidden API to block inter-app activity transitions (*i.e.*, the option `-p`). A detailed technique explanation can be found at the web site of AIMDROID.

IV. EMPIRICAL EVALUATION

We regard SAPIENZ and Monkey as the state-of-the-art and state-of-the-practice techniques, respectively. Hence, we conducted an empirical study on 50 real-world apps on real Android devices to compare the performance of AIMDROID with them. We want to answer the following research questions:

- **RQ1: (Code Coverage)** How does the coverage achieved by AIMDROID compare to SAPIENZ and Monkey? Coverage is an important indicator to reflect the effectiveness. We use the activity, method, instruction as the level of coverage measure in our experiment.
- **RQ2: (Crashes)** How do the crashes triggered by AIMDROID compare to those triggered by SAPIENZ and Monkey? We believe that the ability to discover faults is inevitably essential in assessing the effectiveness of any testing tool.

A. Experimental Setup

We downloaded 50 popular closed-source commercial apps from a third-party market that is maintained by a mobile phone manufacturer[2]. These apps in total have billions of daily usages in China. We excluded a certain number of popular apps that require login first. The latency of networking has an impact on the time interval for GUI updating, which is critical for GUI-ripping based testing tools. Thereby, we have not downloaded apps from the Google Play market because most apps there may not have a server in China. The details of the selected apps, including the number of activities, methods, and bytecode instructions, are shown in Table I.

We conducted our experiment on a PC, which has eight-cores 3.40GHz CPU and 20GB RAM on Ubuntu 14.04, and a real-world Android device Nexus 7 (2013 wifi) tablet. For Monkey and AIMDROID, we prepared Android Marshmallow (6.0.1) version (API 23) in the Nexus 7 tablet because it was the latest version when we started our study. However, SAPIENZ only supports Android KitKat version (API 19), because a crucial component called *MotifCore* of SAPIENZ only works on Android KitKat and this component is not open-source. Therefore, we prepared Android KitKat (4.4.2) in another Nexus 7 tablet (with the same configuration) for SAPIENZ separately. We also made some efforts to make SAPIENZ work on real devices, since SAPIENZ supports Android emulators only by default.

We configured three testing tools to wait 700 milliseconds for the finish of GUI updating. This kind of delay was also set in [10], [16], [11]. Every tool was given one hour to test each app on one Android device, which was also set in [13]. If a testing tool finished its testing process before timeout, our experiment script restarted the tool to test the app again. For overtime testing processes, only activities and crashes discovered in one hour were counted. We made a coverage harvest every minute since the testing process started.

We ran Monkey and SAPIENZ with their default configurations (except the wait interval). For AIMDROID, we set $minL = 20$ and $maxL = 50$ (which is smaller than the default max sequence length of SAPIENZ). These two numbers were estimated based on the average number of actions per state we observed. We configured ϵ, α and γ with 0.1, 0.8 and 0.8 respectively. We plan to study the effect of these parameters in the future.

[2]http://www.smartisan.com/.

TABLE I: Results on 50 real world commercial apps.

App	Act.$^{\alpha}$	Method	Instruction	AIMDROID				SAPIENZ				Monkey			
	(#)	(#)	(#)	C (#)	A (%)	M (%)	I (%)	C (#)	A (%)	M (%)	I (%)	C (#)	A (%)	M (%)	I (%)
cn.amazon.mShop.android	130	84,107	1,087,308	2	20.8	19.1	16.5	0	11.5	9.5	8.6	0	6.2	16.7	14.4
cn.dxy.android.aspirin	83	58,084	838,064	4	45.8	30.4	26.3	2	3.6	21.3	18.1	0	7.2	22.9	18.5
cn.kuwo.player	50	82,457	1,362,898	0	14.0	11.8	10.2	0	14.0	6.0	5.1	0	6.0	12.2	11.2
cn.wps.moffice_eng	232	181,693	3,702,610	1	8.2	8.8	6.4	0	7.3	7.0	4.9	0	3.0	8.5	6.4
com.achievo.vipshop	293	121,488	1,982,531	1	13.0	11.2	9.5	0	9.2	12.0	10.6	0	2.0	8.4	6.9
com.autonavi.minimap	110	91,215	1,836,208	1	3.6	24.6	19.8	0	4.5	20.5	16.6	1	3.6	13.2	10.3
com.baidu.BaiduMap	107	109,719	1,784,620	0	9.3	16.0	14.1	0	4.7	11.0	9.6	0	1.9	13.4	12.0
com.baidu.lbs.waimai	127	64,586	1,014,716	1	23.6	8.0	4.6	0	11.0	15.1	12.4	0	5.5	13.5	11.1
com.baidu.searchbox	277	106,291	1,683,206	0	24.5	15.9	12.1	0	8.7	16.4	12.7	0	3.6	16.0	13.1
com.baidu.tieba	398	51,616	818,032	0	5.3	19.5	14.7	0	3.0	19.4	15.6	0	1.8	22.6	18.4
com.chaozh.iReaderFree	105	42	578	0	8.6	42.9	39.4	0	11.4	40.5	30.6	0	7.6	42.9	39.4
com.cubic.autohome	201	55,977	907,121	4	15.9	20.7	17.8	0	9.0	20.5	17.5	1	3.0	14.8	12.9
com.dianping.v1	535	153,582	2,660,563	0	3.2	12.8	10.3	0	0.4	13.0	10.6	0	2.4	8.5	6.6
com.douban.frodo	202	72,259	1,116,161	2	23.8	16.6	14.6	1	5.0	9.4	8.1	0	6.9	14.4	12.5
com.estrongs.android.pop	79	75,745	1,675,939	0	29.1	18.6	17.8	0	31.6	10.2	7.1	0	20.3	15.2	11.4
com.hexin.plat.android	63	96,297	1,616,479	0	9.5	11.9	11.0	0	11.1	11.6	10.8	0	3.2	7.5	7.0
com.hunantv.imgo.activity	75	75,502	1,167,702	0	29.3	23.0	19.8	0	14.7	21.2	18.5	0	4.0	17.7	15.2
com.kugou.android	258	122,571	2,201,108	1	16.3	20.6	18.1	0	6.2	15.2	13.3	0	1.9	7.7	7.2
com.letv.android.client	173	130,096	2,216,365	3	22.5	15.3	13.6	0	6.4	13.0	11.7	0	1.7	12.0	10.7
com.meitu.meiyancamera	54	51,131	869,366	1	50.0	26.6	30.3	2	53.7	26.7	30.5	0	22.2	22.2	26.7
com.mfw.roadbook	225	113,807	1,726,640	0	28.0	18.4	16.4	0	17.3	19.2	17.4	0	4.0	9.2	8.1
com.MobileTicket	9	27,718	408,275	0	44.4	6.4	5.7	2	11.1	5.8	4.8	0	11.1	6.5	5.7
com.moji.mjweather	167	75,593	1,281,964	3	34.1	26.0	23.1	0	8.4	18.4	16.6	0	5.4	21.4	18.6
com.mt.mtxx.mtxx	111	58,125	1,080,410	0	25.2	22.4	24.0	0	18.9	16.1	19.7	0	24.3	21.2	23.9
com.netease.newsreader.activity	249	145,177	2,437,901	0	3.2	19.2	18.2	0	2.4	24.4	23.7	0	1.2	18.9	18.4
com.nuomi	137	112,840	1,868,688	0	6.6	10.4	8.8	2	6.6	11.6	10.3	0	3.6	9.6	8.0
com.panda.videoliveplatform	48	48,485	756,966	2	20.8	17.6	13.8	0	16.7	22.2	17.8	0	8.3	21.2	17.0
com.qiyi.video	279	138,788	2,485,083	12	10.0	19.8	16.8	1	0.7	5.1	4.4	0	3.9	17.1	14.2
com.Qunar	887	35,815	536,055	0	4.2	14.2	14.1	0	6.5	17.5	17.8	0	1.4	13.8	13.9
com.sankuai.meituan	613	157,479	4,187,341	0	5.5	15.2	9.5	0	0.7	9.9	5.4	0	0.2	7.1	3.8
com.sankuai.meituan.takeoutnew	134	66,888	1,429,411	1	17.2	20.6	15.2	0	3.7	11.4	7.1	0	6.0	19.5	13.9
com.sdu.didi.psnger	237	141,488	2,233,520	8	6.3	15.9	14.3	0	2.5	10.7	9.8	0	1.7	10.9	10.0
com.shoujiduoduo.ringtone	40	36,769	630,841	2	45.0	24.6	21.7	0	35.0	21.1	18.9	1	35.0	20.0	18.0
com.sina.weibo	531	130,347	2,073,948	1	6.6	10.2	12.3	1	1.3	7.6	9.8	0	1.7	8.6	10.4
com.smile.gifmaker	152	74,579	1,397,362	0	9.9	15.7	12.2	0	9.2	16.8	13.2	0	1.3	13.5	10.3
com.ss.android.article.news	249	91,416	1,652,016	1	5.6	21.3	17.3	0	10.0	22.8	18.5	0	2.8	20.0	16.0
com.taobao.taobao	469	49,281	957,170	2	12.2	30.4	26.7	2	6.8	27.1	24.0	0	3.0	27.3	24.3
com.taobao.trip	127	88,455	1,483,885	0	16.5	15.3	13.6	1	11.8	13.2	6.5	0	5.5	10.5	5.2
com.tencent.mtt	64	40,524	867,533	0	7.8	34.6	25.8	0	4.7	39.3	30.8	0	9.4	39.8	30.9
com.tencent.news	149	84,802	1,517,873	7	28.9	20.0	14.7	0	16.1	26.6	22.0	0	11.4	23.6	19.5
com.tencent.qqlive	145	92,155	1,743,390	4	17.2	24.4	19.4	0	15.2	28.5	23.6	0	4.1	21.3	17.1
com.tencent.qqmusic	120	97,289	2,061,809	2	20.8	24.7	19.0	0	10.8	20.0	15.1	1	5.8	19.8	15.3
com.wuba	147	111,834	1,549,681	1	38.8	14.7	13.5	0	24.5	15.3	14.4	0	6.8	13.0	11.8
com.xiachufang	193	27	702	3	31.6	51.9	43.3	0	18.7	51.9	46.4	0	5.2	51.9	43.3
com.xunlei.downloadprovider	145	73,517	1,341,254	2	28.3	20.4	18.3	0	13.1	17.3	15.2	0	6.9	12.1	10.7
com.youdao.dict	107	108,212	1,640,566	1	32.7	17.0	13.7	0	0.9	1.1	1.1	0	6.5	12.8	10.4
ctrip.android.view	331	48,378	813,809	0	11.5	19.3	16.9	1	7.6	17.9	16.0	0	3.9	18.1	15.9
fm.xiami.main	74	83,631	1,428,379	2	28.4	28.4	25.3	1	12.2	26.7	23.8	0	4.1	16.5	15.1
me.ele	92	87,804	1,229,535	3	42.4	28.0	23.7	0	19.6	28.2	24.0	0	13.0	21.8	18.4
tv.danmaku.bili	117	86,472	1,238,251	2	16.2	19.2	16.4	0	14.5	10.3	8.7	0	10.3	19.9	17.0
Total/Average$^{\beta}$	9,900	4,292,153	74,601,833	**80**	**19.6**	**20.0**	**17.2**	16	11.1	17.7	15.2	4	6.4	17.1	14.7

$^{\alpha}$ **Act.** and **C** show the number of activities and crashes, respectively. **A**, **M**, and **I** show the coverage of activity, method, and instruction, respectively.
$^{\beta}$ In this row all coverage data are the average and other data are the total.

B. Results

For each testing tool, we collected the activity, method and instruction coverage of each app. In addition, we also collected the number of unique crashes of each app. The detailed results of each app are shown in Table I.

RQ1: (Code Coverage) AIMDROID significantly outperformed SAPIENZ and Monkey in the activity coverage. This result helps to show the effectiveness of the activity-insulated exploration strategy. For method and instruction coverage, AIMDROID also has better results than the other two on some apps but are not as significant as the activity coverage. This may be because the function `bound`, which controls the total iteration of an episode, was not properly implemented. Actually, AIMDROID may apply the in-cage exploration to every discovered activity again if it still has time.

Monkey obtained better results on some apps, which is also observed by previous work [14], [13], [10]. SAPIENZ didn't get a comparable results on our subject apps in comparison with those on open-source apps in [13]. This is because

TABLE II: Exception Types of 80 Crashes

Exception	#
java.lang.NullPointerException	62
java.lang.SecurityException	10
java.lang.ClassNotFoundException	2
android.content.ActivityNotFoundException	1
android.util.AndroidRuntimeException	1
android.util.SuperNotCalledException	1
java.lang.OutOfMemoryError	1
Native crash	2

we counted all methods and instructions in the installation file (*i.e.*, APK) of every app, including those from third-party libraries. Besides, we used the default configuration of SAPIENZ. In addition, SAPIENZ needs to reset the app to the clean state before evaluating every testing script. Therefore, SAPIENZ generated much fewer events than AIMDROID and Monkey during our evaluation. We may need more hours to show the effectiveness of SAPIENZ on testing the subject apps.

Figure 3 depicts the progressive average coverage on 50 apps. The activity coverage of AIMDROID increases more rapidly than SAPIENZ and Monkey. For the method and instruction coverage, AIMDROID and SAPIENZ almost have the same result and both have a better result than Monkey.

RQ2: (Crashes) As shown in Table I, AIMDROID triggered 80 unique crashes, while SAPIENZ and Monkey triggered 16 and 4 unique crashes, respectively. A crash is uniquely identified by the error message and the crashing activity. Hence, at most 80 activity discovered by AIMDROID cannot be properly started by the quick launch method. We do not compare the sequence length in our experiment, since most activities can be launched directly and AIMDROID limited the longest length of each testing sequence in an episode with a small constant $maxL = 50$.

Table II shows the distribution of different errors. Most errors were caused by accessing null references. AIMDROID may generate two kinds of false crashes. First, the quick launch method may fail to create the target activity. Second, blocking the start of another activity may fail to resume the current activity. We identify these two kinds of potential false crashes by analyzing the stack traces. Specifically, if the stack trace contains the method `android.app.Activity.performCreate()` or `android.app.Activity.performResume()`, then we treat the crash as a false crash. Finally, we found 26 false crashes during creating an activity and 4 false crashes during resuming an activity. Note that 20 of 50 apps have neither true crash nor false crash.

C. Threats to Validity

Subjects We haven't evaluated AIMDROID with benchmark apps used by previous work [10], [13]. This is because most of those apps are open-source and only have a few activities. The improvement of activity-insulated exploration may not be significant on these apps.

Emulator We haven't evaluated AIMDROID on emulators or any other phones. All experiments were conducted on Nexus 7. Nexus series are products of Google and used as the reference for device vendors. Hence, we can avoid problems caused by emulator bugs or device fragmentation [36].

Wait Interval A proper wait interval is necessary for testing real-world Androids apps [16], [11]. Existing work sets different wait intervals for different apps on different platforms, *e.g.*, 200 milliseconds in [10], 5 seconds in [16], and 4 seconds in [11]. With our knowledge of subject apps, we set a 700 milliseconds wait interval for every testing tool.

False Crashes AIMDROID reported many crashes that cannot be triggered during normal usage. However, it may not be annoying to manually inspect these errors: the number of crashes for each app ranges from 0 to 12, which is not significant. We also found some patterns to triage these crashes, which further helps to reduce human efforts. AIMDROID can also help to reveal bad code smells, *e.g.*, accessing a potential null pointer without check in a method.

SAPIENZ The currently available version of SAPIENZ must be modified to test commercial apps on real devices. To install SAPIENZ on real devices, we manually resolved the permission problems. To mitigate the problem of GUI updating, we set a wait interval for all testing tools. The wait interval may not be necessary for SAPIENZ, though many previous work has set it [10], [16], [11]. Therefore, we need more hours to evaluate SAPIENZ.

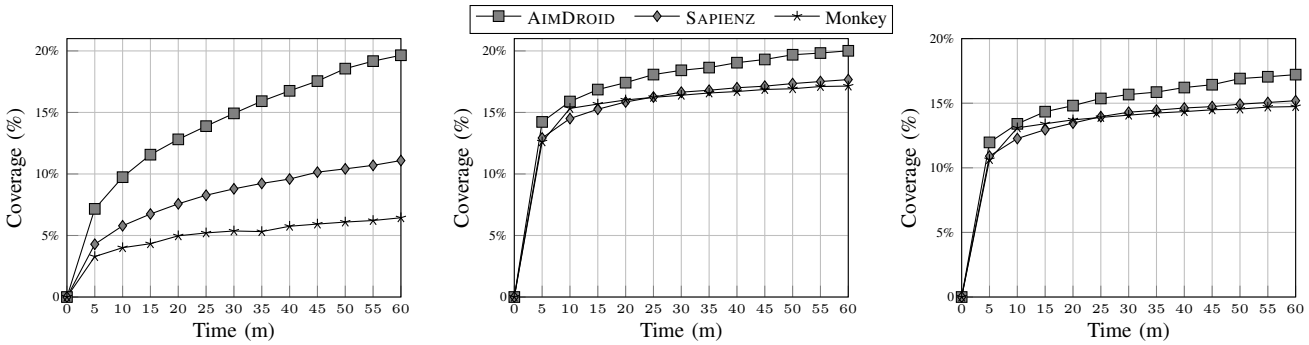
V. RELATED WORK

A. Mobile Application Testing

Mobile application development has many challenges, such as divergent context [37], [38], performance [39], [40], resource and energy [41], [42], [43], [44], [45]. For Android apps, an addition difficulty is the fragmentation [5], [36]. In the following sections, we mainly discuss techniques addressing a more fundamental problem, *i.e.*, how to effectively generate events to exercise apps thoroughly. We classify these techniques into three types following previous work [10], *i.e.*, *fuzz testing*, *model-based testing*, and *systematic testing*. These techniques are mostly based on primitive actions/events but can piggy-back on macro actions/events that are inferred from execution traces from human users [46], [47].

B. Fuzz Testing

Monkey [8] adopts a random strategy to generate a tremendous number of irrelevant input events. In comparison with techniques in academy [10], Monkey actually has a better usability and also better performance on some benchmark apps [10]. The main limitation of Monkey is that the long event sequences have many redundant and unfruitful events [13]. To this end, many techniques focus on reducing the length of testings [13], [11], which we believe is essential to make automated testing tools practical.



(a) Average activity coverage. (b) Average method coverage. (c) Average instruction coverage.

Fig. 3: Progressive average coverage on 50 real world commercial apps.

To avoid unfruitful events, Dynodroid [14] rips the GUI to generate events on interactive widgets. To avoid being trapped in certain activities, Zeng *et al.* [12] additionally limit the total exploration time of an activity and prohibit the transition to an over-explored activity. AIMDROID manages the exploration of an activity in a systematic way, which also can help to reduce necessary activity transitions.

Android apps also need to behave well in adverse conditions [48], [49], [50], [51]. Some techniques manufacture intents as input for Android components (*e.g.*, activities and services) by static analysis [52], [53]. Besides, fuzzing intents can also be used with model-based testing [19]. In contrast, AIMDROID collects intents dynamically. In the future work, we plan to use these intents as feeds for intents fuzzing and use these generated intents for quick launch.

C. Model-based Testing

Model-based testing techniques have been widely studied in testing Android apps [54], [15], [17], [16], [18], [55]. The exploration can be guided to specific unexplored parts using a systematic strategy such as DFS, BFS or hybrid [54], [17], [18], or a stochastic model [19].

However, a critical problem of model-based testing for Android apps is the granularity of state abstraction. Baek *et al.* [18] conducted a study of multi-level state representations to show that different levels of abstraction actually have an impact on the effectiveness of a modeling based tool. While Baek *et al.* adopt a single level abstraction a time, AIMDROID integrates two-level abstractions and two different strategies.

In addition, a systematic strategy like DFS needs to restart the app from the initial state to backtrack to previous states. Restarting and backtracking badly slow the testing process, although SwiftHand [16] tries to solve this problem by using actions such as BACK. AIMDROID insulates each activity in the cage and explores each activity independently. By using the quick launch method, AIMDROID can significant reduce unnecessary transition between activities.

D. Systematic Testing

Systematic testing approaches [22], [23], [24], [13], [9] apply symbolic execution or evolutionary algorithms to guide the input generation. However, these tools are considered less scalable than black-box techniques like Monkey [10].

SAPIENZ [13], which we consider as the state-of-the-art in this paper, introduces multi-objective approach to test Android apps. SAPIENZ applies evolutionary algorithms to optimize the event sequences for three objectives: code coverage, sequence length and the number of crashes found. However, SAPIENZ has to iteratively evaluate new generated event sequences, which consumes much time and resources. Besides, the app should be reset and restarted to obtain a clean state for replaying the testing scripts. Restart is time-consuming and replaying can also be interrupted by non-determinism [16].

AIMDROID can adopt search-based techniques such as EvoDroid [9] and SAPIENZ [13] for the intra-activity exploration. By using the quick launch method, we can avoid replaying events in the test scripts to start the activity. In addition, non-determinism caused by intents created from dynamic data can also be mitigated if we start the activity with a chosen fixed intent.

VI. CONCLUSIONS

In this paper, we propose AIMDROID, an approach to model-based automated testing for Android apps. AIMDROID implements the activity-insulated multi-level strategy, which can manage the exploration of activities and avoid unnecessary activity transitions between them. We believe that AIMDROID is a practical black-box testing tool. We have evaluated AIMDROID on 50 popular real world closed-source commercial apps in China. Our evaluation has shown that AIMDROID outperforms the state-of-the-art and the state-of-the-practice in activity, method, and instruction coverage, respectively. In addition, AIMDROID reveals more crashes than the other two.

ACKNOWLEDGMENT

This work was supported in part by High-Tech Research and Development Program of China under Grant No.2015AA01A203, National Natural Science Foundation (Grant Nos. 61690204, 61472177) of China, and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

REFERENCES

[1] "Number of Android applications," http://www.appbrain.com/stats/number-of-android-apps, accessed: 2017-08-01.

[2] "Activity," https://developer.android.com/reference/android/app/Activity.html, accessed: 2017-08-01.

[3] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan, "Prioritizing the devices to test your app on: A case study of Android game apps," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 610–620.

[4] K. Moran, M. Linares-Vsquez, and D. Poshyvanyk, "Automated GUI testing of Android apps: From research to practice," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 648–648.

[5] M. Linares-Vsquez, G. Bavota, C. Bernal-Crdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "API change and fault proneness: A threat to the success of Android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 477–487.

[6] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 15–24.

[7] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.

[8] "UI/Application Exerciser Monkey," https://developer.android.com/studio/test/monkey.html, accessed: 2017-08-01.

[9] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of Android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 599–609.

[10] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?(e)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 429–440.

[11] L. Clapp, O. Bastani, S. Anand, and A. Aiken, "Minimizing GUI event traces," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 422–434.

[12] 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 *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 987–992.

[13] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 94–105.

[14] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 224–234.

[15] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.

[16] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 623–640.

[17] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 641–660.

[18] Y.-M. Baek and D.-H. Bae, "Automated model-based Android GUI testing using multi-level GUI comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 238–249.

[19] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, p. to appear.

[20] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

[21] "Intent," https://developer.android.com/reference/android/content/Intent.html, accessed: 2017-08-01.

[22] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 59.

[23] J. Jeon, K. K. Micinski, and J. S. Foster, "Symdroid: Symbolic execution for dalvik bytecode," 2012.

[24] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[25] "UI Automator Viewer," https://developer.android.com/training/testing/ui-automator.html#ui-automator-viewer, accessed: 2017-08-01.

[26] "Activity Lifecycle," https://developer.android.com/training/basics/activity-lifecycle/index.html, accessed: 2017-08-01.

[27] "URI," https://developer.android.com/reference/android/net/Uri.html, accessed: 2017-08-01.

[28] "Tasks and back stack," https://developer.android.com/guide/components/activities/tasks-and-back-stack.html, accessed: 2017-08-01.

[29] C. Bertolini, G. Peres, M. d'Amorim, and A. Mota, "An empirical evaluation of automated black box testing techniques for crashing GUIs," in *2009 International Conference on Software Testing Verification and Validation*. IEEE, 2009, pp. 21–30.

[30] S. Bauersfeld and T. E. Vos, "User interface level testing with TESTAR; what about more sophisticated action specification and selection?" in *SATToSE*, 2014, pp. 60–78.

[31] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *arXiv preprint arXiv:1402.6028*, 2014.

[32] "Xposed module repository," http://repo.xposed.info/, accessed: 2017-08-01.

[33] "ART and Dalvik," https://source.android.com/devices/tech/dalvik/, accessed: 2017-08-01.

[34] "Activity manager command," https://developer.android.com/guide/topics/manifest/activity-element.html#exported, accessed: 2017-08-01.

[35] "App manifest," https://developer.android.com/guide/topics/manifest/activity-element.html#exported, accessed: 2017-08-01.

[36] L. Wei, Y. Liu, and S. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, 2016, pp. 226–237.

[37] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao, "Caiipa: Automated large-scale mobile app testing through contextual fuzzing," in *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, 2014, pp. 519–530.

[38] K. Moran, M. Linares-Vsquez, C. Bernal-Crdenas, C. Vendome, and D. Poshyvanyk, "CrashScope: A practical tool for automated testing of Android applications," in *Proceedings of the 39th International Conference on Software Engineering Companion*, 2017, pp. 15–18.

[39] Y. Liu, C. Xu, and S. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1013–1024.

[40] M. Linares-Vsquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in Android apps," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 352–361.

[41] Y. Liu, C. Xu, S. Cheung, and J. Lu, "Greendroid: Automated diagnosis of energy inefficiency for smartphone applications," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 911–940, Sept 2014.

[42] M. Wan, Y. Jin, D. Li, and W. G. J. Halfond, "Detecting display energy hotspots in Android apps," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.

[43] D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond, "Automated energy optimization of HTTP requests for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 249–260.

[44] M. Linares-Vsquez, G. Bavota, C. E. B. Crdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing energy consumption of GUIs in Android apps: A multi-objective approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 143–154.

[45] Q. Li, C. Xu, Y. Liu, C. Cao, X. Ma, and J. L, "CyanDroid: Stable and effective energy inefficiency diagnosis for Android apps," *SCIENCE CHINA Information Sciences*, vol. 60, no. 1, p. 12104, 2017.

[46] M. Linares-Vsquez, M. White, C. Bernal-Crdenas, K. Moran, and D. Poshyvanyk, "Mining Android app usages for generating actionable GUI-based execution scenarios," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015, pp. 111–122.

[47] M. Ermuth and M. Pradel, "Monkey See, Monkey Do: Effective generation of GUI tests with inferred macro events," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 82–93.

[48] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with AppDoctor," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 18:1–18:15.

[49] C. Q. Adamsen, G. Mezzetti, and A. Mller, "Systematic execution of Android test suites in adverse conditions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 83–93.

[50] P. Huang, T. Xu, X. Jin, and Y. Zhou, "DefDroid: Towards a more defensive mobile OS against disruptive app behavior," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, 2016, pp. 221–234.

[51] Z. Shan, T. Azim, and I. Neamtiu, "Finding resume and restart errors in Android applications," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 864–880.

[52] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, 2013, pp. 68:68–68:74.

[53] R. Sasnauskas and J. Regehr, "Intent Fuzzer: Crafting intents of death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, 2014, pp. 1–5.

[54] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android pplications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.

[55] T. Su, "FSMdroid: Guided GUI testing of Android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 689–691.