EHBDroid: Beyond GUI Testing for Android Applications

Wei Song
School of Computer Sci. & Eng.
Nanjing University of Sci. & Tech.
Nanjing, China
wsong@njust.edu.cn

Xiangxing Qian
School of Computer Sci. & Eng.
Nanjing University of Sci. & Tech.
Nanjing, China
xiangxingqian@gmail.com

Jeff Huang
Parasol Laboratory
Texas A&M University
College Station, TX, USA
jeff@cse.tamu.edu

***Abstract*—With the prevalence of Android-based mobile devices, automated testing for Android apps has received increasing attention. However, owing to the large variety of events that Android supports, test input generation is a challenging task. In this paper, we present a novel approach and an open source tool called EHBDroid for testing Android apps. In contrast to conventional GUI testing approaches, a key novelty of EHBDroid is that it does not generate events from the GUI, but directly invokes callbacks of event handlers. By doing so, EHBDroid can efficiently simulate a large number of events that are difficult to generate by traditional UI-based approaches. We have evaluated EHBDroid on a collection of 35 real-world large-scale Android apps and compared its performance with two state-of-the-art UI-based approaches, Monkey and Dynodroid. Our experimental results show that EHBDroid is significantly more effective and efficient than Monkey and Dynodroid: in a much shorter time, EHBDroid achieves as much as 22.3% higher statement coverage (11.1% on average) than the other two approaches, and found 12 bugs in these benchmarks, including 5 new bugs that the other two failed to find.**

***Index Terms*—Android, automated testing, event generation, event handlers**

I. Introduction

Despite the popularity of mobile apps, testing them faces significant challenges. The difficulty lies in two main aspects. First, the space of events is often enormous. There could be an infinite number of UI events if the app's state is cyclic, and there are more than a hundred different kinds of system broadcasting events supported by Android currently [1]. It is impractical to generate all possible events and their permutations. Second, many UI events (e.g., drag, hover) and most system and inter-app events, are difficult to generate. For example, generating a drag event requires that the drag should start at a certain position and end at another, and only when the distance between the two positions is larger than a certain value, can the event be regarded as a successful drag. In addition, certain UI events can only be triggered if their preconditions are satisfied (e.g., inputs to all the relevant widgets are provided) [2]. The probability to simulate all combinations of the widget inputs is very small [3]. Thus, it is hard to simulate such UI events thoroughly. For many system events, proper data (e.g., arguments) needs to be constructed and dispatched correctly to the app. For inter-app events, they can only be triggered by external apps under certain conditions.

Although mobile app testing has attracted a large body of active research [1], [2], [4]–[13], existing approaches and tools are still unsatisfactory. According to a recent study [14], most existing approaches are UI-based, and they are either too slow to generate events, or cannot effectively generate certain events. More efficient and effective automated testing techniques are required to ensure correctness, reliability, and security of mobile apps.

In this paper, we present a new approach and an open source tool, called EHBDroid (Event Handler Based), for testing Android apps. EHBDroid is implemented and evaluated for Android. However, the idea of event handler-based testing is not limited to Android, but general to event-driven systems. EHBDroid is based on a simple, but important observation: In event-driven systems, there often exists a correspondence between an event and an event handler. The events that occur on the UI are eventually passed to and are handled by their event handlers, e.g., callback functions in Android [15]. Hence, instead of attempting to generate UI events, we can trigger their event handlers (callbacks) directly. The callback instrumentation can be inserted into the app via either static analysis or code re-writing at class loading time.

There are several advantages of EHBDroid over traditional UI-based approaches. First, it can invoke a set of callback functions quickly because the testing does not need to wait for the latency induced by GUI and the cost of message passing in the system. Second, it can test the callback functions thoroughly even if some of them cannot be easily invoked through the GUI. For many events such as system events and complex UI events, it is much easier to generate calls to their event handlers directly than to generate events from the UI. Consequently, it provides high code coverage in a short testing time. Third, the events for the test are systematically generated and are not redundant, i.e., the events do not invoke the same functions repeatedly.

Although the basic idea of EHBDroid is simple, we are not aware of any previous research or infrastructure that explored this idea. A most related tool is Dynodroid [1]. It instruments the Android framework and relies on the VM to generate a considerable number of UI inputs and system-level events that are relevant for the apps, and uses a pre-configured selection strategy to select an event to execute. However, Dynodroid is less effective and efficient than EHBDroid because of its black-box nature, i.e., it only generates a limited range of events, and for each generated event it relies on the Android framework to pass the event to the event handler, rather than invoking the event handler directly.

To realize EHBDroid, there are several challenges:

1) How to identify the event handlers in Android and construct their invocations?
2) How to construct valid runtime event data passed to the event handlers?
3) How to systematically invoke the event handlers such that the app behavior can be effectively tested without redundant exploration?

These challenges are often specific to the implementation of the event-driven framework. For the first challenge, we identify three registration patterns for event handlers in Android. Based on these patterns, EHBDroid is able to automatically instrument all 58 callbacks in the Android API. For the second challenge, by using default value for data of primitive types and runtime instances of event sources (which are available in the app context) for those of reference types, EHBDroid constructs many meaningful runtime arguments for invoking the callbacks. For the third challenge, instead of invoking those callbacks randomly, EHBDroid performs an activity-directed depth-first search to effectively traverse different activities without redundant exploration.

We implement EHBDroid as an automated testing tool based on the Soot framework [16]. EHBDroid is open source and is publicly available on Github: [1]. EHBDroid has been evaluated on 35 real-world Android apps from F-droid [17] and Google Play store [18], and compared with two state-of-the-art UI-based testing tools, Monkey [19] and Dynodroid [1]. The results suggest that EHBDroid is significantly more efficient and effective than the other two approaches. EHBDroid achieves as much as 22.3% higher statement coverage (11.1% on average) than Monkey and Dynodroid for these apps. For all apps, EHBDroid can quickly cover all activities in ten minutes, whereas for many apps Monkey and Dynodroid either cannot generate events to cover certain activities or get a much lower code coverage even in an hour. Besides, EHBDroid detected 12 bugs in these apps, 5 of which were not found by the other two.

To summarize, we make the following contributions:

- We develop an event handler-based testing approach, EHBDroid, for testing Android apps without the need to generate events. The approach of EHBDroid applies not only to Android apps, but also to general event-driven systems. Though conceptually simple, to the best of our knowledge, EHBDroid is the first event handler-based approach for Android app testing.
- We present a general and systematic testing strategy to simulate UI, system, and inter-app events in Android by instrumenting and automatically invoking their callbacks. We have successfully instrumented all 58 callbacks in Android API with valid arguments.
- We present an open source tool that realizes our approach and we have conducted extensive experiments showing significant performance improvements of our approach over the state-of-the-art.

[1]https://github.com/wsong-nj/EHBDroid

TABLE I
ANDROID EVENT TYPES AND THEIR REGISTRATION PATTERNS

Event \ **Pattern**		Static (a)	Static (b)	Dynamic	Overridden
UI		√		√	√
System	Receiver		√	√	
	Service		√	√	
Inter-app			√		

The remainder of the paper is organized as follows. Section II introduces the background on Android event registrations. Section III presents our approach. Section IV introduces the implementation of EHBDroid. Section V evaluates EHBDroid. Section VI discusses limitations of EHBDroid. Section VII reviews related work and Section VIII concludes.

II. Android Event Registration

Android apps are event-driven programs with abundant UI events, system events, and inter-app events. Based on our study of the Android API, we identify three event registration patterns (cf. Table I), which are useful for the instrumentation of our approach.

*Pattern 1 (**Static registration pattern**):* A static registration pattern is defined as *s.elm*, where *s* is an event source declared in the XML resource file, and *elm* represents an element (method or field) of *s* declared in the same file:

(a) If *s* is a view, *elm* is the method invoked by the event handler of *s*.
(b) If *s* is a component (i.e., activity, service, or receiver), *elm* is an intent-filter object defined by *s*.

Pattern 1(a) is only applicable to UI events and their handlers, and Pattern 1(b) is only applicable to system and inter-app events and their handlers. Fig. 1(a) shows an example of static registration pattern for UI events, where *Button* is a view and *click* is the method invoked by the event handler of *Button*. Fig. 2(a) presents an example for service event registration, where *service* is a component and *intent-filter* is its attribute.

*Pattern 2 (**Dynamic registration pattern**):* A dynamic registration pattern is defined as *s.rm(h)* in the app code, where *s* is the source of an event, *rm* is the registration method of *s*, and *h* is the event handler that *s* registers.

Pattern 2 applies to both UI and system events. In Fig. 1(b), *btn* is a view, *setOnClickListener* is the registration method, and *listener* is the event handler. In Fig. 2(b), *sm* is a service, *registerListener* is the registration method, and *l* is the event handler.

*Pattern 3 (**Callback overridden pattern**):* A callback overridden pattern is defined as *s.callback*, where *s* is the source of an event **e**, and *callback* corresponds to **e**'s event handler.

Pattern 3 is only applicable to view events. For this pattern, one should override the corresponding callbacks in the app code. The following six callbacks are frequently used in this pattern: *onListItemClick()*, *performClick()*, *onTouchevent()*, *onKeyUp()*, *onKeyDown()*, *performLongClick()*. Fig. 1(c) presents an example, where *MyView* is a view and *performClick* is the overridden method.

```
// View + Callback
<Button android:id="." android:onClick="click">
```
(a)

```
btn . setOnClickListener ( listener ) ;
```
(b)

```
Class MyView extends View{
  void performClick () {
  }
}
```
(c)

Fig. 1. Examples of UI event registration

```
<service name="MusicService">
  <intent− filter >... </intent − filter >
</service>
```
(a)

```
SensorManager sm = getSystemService();
1 = new SensorEventListener () {
     void onSensorChanged();}
sm. registerListener (1) ;
```
(b)

Fig. 2. Examples of service event registration

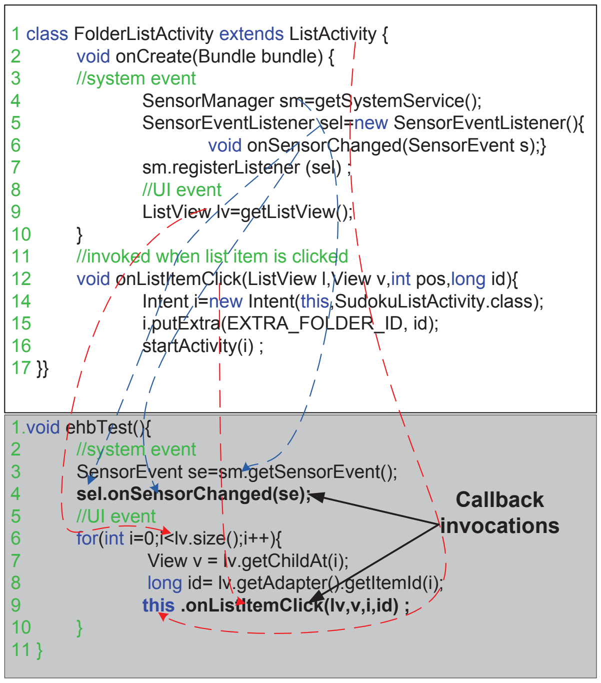
III. EHB Testing

In this section, we propose the event handler-based testing approach for Android apps.

A. Motivation and Challenges

Fig. 3 exhibits a code snippet from the *OpenSudoku* app, which involves one UI event and one system event. For the UI event, the corresponding UI element, event handler, and callback are *listView*, *FolderListActiviy*, and *onListItemClick()*, respectively. For the system event, the corresponding event handler and callback are *sm* and *onSensorChanged()*. To test this app with traditional UI-based approaches, these two events must be triggered as the test inputs. However, generating these two events is not easy, because triggering the UI event needs to click the right list item on the screen, and triggering the system event needs to simulate a sensor change. Moreover, once being triggered, these two events have to be analyzed by the underlying Android OS and then passed to their event handlers through numerous layers of the Android framework, which can slow down the testing.

To test this app, we directly invoke both of the two callbacks *onListItemClick()* and *onSensorChanged()* in the app code (as shown in the grey region), rather than trigger the corresponding events from the UI hierarchy. In this way, not only the two



```
class FolderListActivity extends ListActivity {
    void onCreate(Bundle bundle) {
    //system event
            SensorManager sm=getSystemService();
            SensorEventListener sel=new SensorEventListener(){
                    void onSensorChanged(SensorEvent s);}
            sm.registerListener (sel) ;
            //UI event
            ListView lv=getListView();
    }
    //invoked when list item is clicked
    void onListItemClick(ListView l,View v,int pos,long id){
            Intent i=new Intent(this,SudokuListActivity.class);
            i.putExtra(EXTRA_FOLDER_ID, id);
            startActivity(i) ;
}}
```

```
void ehbTest(){
    //system event
    SensorEvent se=sm.getSensorEvent();
    sel.onSensorChanged(se);
    //UI event
    for(int i=0;i<lv.size();i++){
            View v = lv.getChildAt(i);
            long id= lv.getAdapter().getItemId(i);
            this .onListItemClick(lv,v,i,id) ;
    }
}
```

Fig. 3. A motivating example from the *OpenSudoku* app

events can be simulated easily, but the testing itself can also be performed more efficiently. Considering the large number of events and the difficulty of event generation in many cases, our approach is a promising alternative for app testing.

Challenges. However, there are several challenges in realizing the idea above:

1) We need to identify the event handlers (that is, the callback functions) and construct their invocations. Android provides 58 different callbacks as a part of the framework, and besides, there can be many user-defined or overridden callbacks. It is challenging to correctly identify them and construct their invocations.
2) We have to construct valid arguments for the callback invocations. For example, the callback *onListItemClick()* has four different types of parameters: *listView*, *view*, *position* and *id*. It is uneasy to obtain valid values for the parameters from the app.
3) We should insert these callbacks at proper locations in the app and to control their invocations such that the app can be systematically tested. It is challenging to insert these invocations without breaking the lifecycle of activities and to invoke them to effectively cover different activities.

B. Invoking Event Handlers

We first identify all event registration statements in the app based on the three patterns presented in Section II. Algorithm 1 performs a linear scan of the app to find all events and their handlers (declared either in the XML resource files or in the app code). Then, for each event registration statement, we construct an instrumented invocation statement (which will be

Algorithm 1: Search for events and their handlers

```
Input: The APK file of an Android app
Output: $E$: XML elements matching with Pattern 1, $S$: Statements
        matching with Pattern 2, $M$: Methods matching with Pattern 3
for each XML resource file $f \in app$ do
    for each XML element $e \in f$ do
        if $e$ matches with Pattern 1 then
            $E \leftarrow E \cup \{e\}$
for each class $c \in app$ do
    for each $m \in c$ do
        if $m$ matches with Pattern 3 then
            $M \leftarrow M \cup \{m\}$
        for each statement $s \in m$ do
            if $s$ matches with Pattern 2 then
                $S \leftarrow S \cup \{s\}$
```

inserted into the app), according to the invocation pattern of event handlers defined below.

*Pattern 4 (**Invocation Pattern**):* The invocation pattern for event handlers used for the instrumentation is defined as *h.callback(s)*, where *s* is the event source, *h* the registered event handler, and *callback()* the callback method of *h*.

To construct the callback invocation, we need to identify the three elements *s*, *h*, and *callback* for each event registration pattern. For Pattern 1(a) and Pattern 3, view *s* is the event source, *elm* is the callback, and the object that defines *elm* is the event handler *h*. For Pattern 1(b), component *s* is the event source and *elm* is the intent-filter. In this case, the life-cycle method of *s* is the callback, and the class that defines this method is the corresponding event handler. For this pattern, besides *s*, the intent-filter *elm* is also the parameter of the callback. For Pattern 2, we can obtain the callback from the event handler registered by the registration method *rm*. Figs. 4a-4c present the instrumented invocation statements for the UI event registration examples in Figs. 1a-1c, respectively. Fig. 4d shows the instrumented invocation statements for the service event registration examples in Fig. 2.

```
//m is click method
Object c = m.getClass().newInstance();
c.click(btn);
```
(a)

```
listener.onClick(btn);
```
(b)

```
MyView.newInstance().performClick();
```
(c)

```
Activity mainActivity;
Receiver ma = BootReceiver.newInstance();
Intent intent = createIntent( intentfilter );
ma.onReceiver(mainActivity, intent);
```
(d)

Fig. 4. Examples of invocation patterns for instrumentation

C. Constructing Callback Arguments

For all the 58 callbacks provided in the Android API, we have successfully instrumented their invocations in apps. In this subsection, we describe how to construct valid arguments for these callback invocations.

One way to construct the callback arguments is to collect the runtime data for a few test runs (e.g., with existing test cases) and use the collected data as the arguments. Nevertheless, this method requires the existence of good test inputs. After analyzing the 58 callbacks, we propose a simple yet effective method to construct valid arguments based on the default values and from the app context. This method is complementary to the first method, and it does not require extra test runs.

- **Using default values**. Data types in Java can be divided into primitive types and reference types. For primitive data types, Android API usually provides default valid values. For this kind of parameters, we use their default values as arguments for the corresponding callback invocations. If a callback involves several parameters, we enumerate all combinations of their default values.
- **Deriving from app context**. For parameters of the reference data types, the valid values (objects) can be obtained from the app context. These parameters are usually connected with the event source, and thus can be obtained by invoking relevant APIs of the event source. For arguments that cannot be directly obtained from the app context, we can get them through Java reflection. For instance, to construct the arguments for the *onDrag()* callback, we cannot get a *DragEvent* instance via *new DragEvent()* directly because only a private construction method is provided. Instead, we utilize Java reflection to get the private constructor, set its accessibility to true, and finally get an instance via invoking *class.newInstance()*.

Example. The grey region in Fig. 3 illustrates how we construct the callback arguments for the motivating example. Take the callback *onListItemClick(ListView lv, View v, int pos, long id)* as an example. Its first argument is a ListView instance which can be obtained from the app context. The second is the view that is clicked within the ListView. The third is the position of the view in the ListView. The last is the row id of the view. The last three parameters are all correlated with the first parameter *lv*, and thus can be constructed from it. Lines 6-8 show how to obtain the second, third and fourth arguments. Line 9 is the instrumented callback invocation. The reason why we do not use default values for the parameters of primitive types is to enumerate all possible values. Note that our approach can automatically construct such arguments from the correlated objects using Android APIs.

D. Exploration Strategy

The logic of an app can often be described as a state machine (or a directed graph) where nodes represent activities and edges the activity transitions caused by events (cf. Definition 1). Fig. 5 depicts a part of the state machine of the *OpenSudoku* app. Note that the state machine is just to help readers understand our approach; it is unnecessary to explicitly construct it in our approach.

*Definition 1 (**App abstraction**):* An app can be abstracted as a state machine ($\mathcal{A}$, $\mathcal{E}$, $\rightarrow$) where

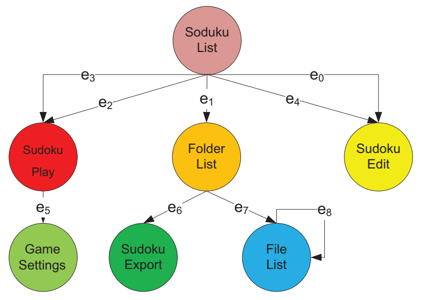


Fig. 5. Part of the state machine of the *OpenSudoku* app.

Algorithm 2: EHBDroid Test exploration

```
Input: The instrumented APK file of an Android app
1 $S \leftarrow S$.push($MainActivity$)
2 while $S$ is non-empty do
3     $currentActivity \leftarrow S$.pop()
4     if $currentActivity \notin L$ then
5         $L = L \cup \{currentActivity\}$ trigger new event "test"
6         for each activity $A_i$ generated by "test" do
7             $S$.push($A_i$)
8     else
9         click navigation "Back"
```

- $\mathcal{A}$ is an activity set; $A_0 \in \mathcal{A}$ is the main activity, and $A_i \in \mathcal{A}$ is called the current activity if it is active on the screen of the user device.
- $\mathcal{E}$ is the set of all possible events in the app, and $E(A_i) \subset \mathcal{E}$ is the set of events that can be triggered in A_i.
- $\rightarrow \subset \mathcal{A} \times \mathcal{E} \times \mathcal{A}$ is a ternary transition relation.

Without loss of generality, our testing focuses on event coverage, i.e., aiming to cover all events in an app. To this end, all nodes (activities) where events can be triggered should be visited. Android provides an *activity stack* managing all activities, which allows us to use a depth-first-search strategy for activity (event) traversal.

Beginning from the main activity A_0, we first identify the event handlers (callbacks) of events in $E(A_0)$. Then, we invoke all these callbacks in a random but valid order CS. That is, each callback can only be invoked if the corresponding event is valid at the current state of A_0; if there is more than one valid event, the invocation order of these callbacks is random. Notably, the invocation of some event handlers in CS may cause the transitions from A_0 to other activities. These activities are automatically pushed into the Android activity stack. After all event handlers in A_0 are invoked, the activity A_i at the top of the stack is popped. The same strategy is used to explore all event handlers in A_i. The above procedure iterates until all activities of the app are explored.

To ensure that our instrumentation is valid, the callbacks should be invoked in a similar way as the user's click on the UI elements. With this in mind, for each activity, we build a specific menu-item "test" with its event handler encapsulating a valid callback sequence of events in $E(A_i)$. In the following example, the menu-item "test" registers a listener *onMenuItemClickListener* whose callback is *onMenuItemClick()*. In the callback, the invocations of event handlers are instrumented.

```
MenuItem menuItem = new MenuItem("test");
OnMenuItemClickListener oml;
oml = new OnMenuItemClickListener({
  void onMenuItemClick(){
  // inserting invoking statements
  }};
menuItem.setOnClickListener(oml);
```

Algorithm 2 presents our test exploration strategy, which accepts the instrumented APK file as input. In the algorithm, besides the Android activity stack, a list L is utilized to denote the activities that have been explored. Algorithm 2 works as follows: If $currentActivity$ has not been explored, all event handlers in $currentActivity$ are triggered (by clicking "test") and the generated activities are all pushed into S. Otherwise, the testing continues to explore next activity on the top of the stack (by pressing "back"). The algorithm terminates when the activity stack S becomes empty. Since each activity is explored only once, time complexity of Algorithm 2 is linear in the number of activities.

Example. Consider the example in Fig. 5 where $SodukuList$ is the main activity. When the instrumented "test" menu-item in $SodukuList$ is "clicked", event handlers of the five events $e_0 \sim e_4$ in $SodukuList$ are triggered and the generated activities are pushed into the stack S following a random order, e.g., $SodukuEdit$, $SodukuPlay$, $SodukuPlay$, $FolderList$, and $SodukuEdit$. Although there are repeated activities in S, their exploration is not redundant according to Algorithm 2. Next, $SodukuEdit$ is explored as it is at the top of S. The other activities are all explored similarly.

IV. Implementation

We have implemented EHBDroid as a fully automated tool based on Soot [16]. Fig. 6 depicts its architecture which consists of two main parts: an **Instrumentor** that instruments the target app, and a testing **Explorer** that tests the app. Important modules of the two parts are explained below.

XML Parser and **Recognizer**. The first step of EHBDroid is to search different event registration patterns in both the XML resource files and the app code. The XML Parser utilizes XMLPrinter [20] to parse the input XML resource files. Based on Pattern 1, it searches for two kinds of tuples. The first kind consists of a view and a method, while the second kind consists of a component and one or more intent-filters. All these tuples are stored in a map. The Recognizer then searches for all statements that match with Pattern 2 and Pattern 3. All qualified statements are summarized in a set of Java objects. In each of the Java objects, there are three different fields: event source, registered methods, event handlers.

Dispatcher. The Dispatcher determines the activities that contain the event sources identified by the XML Parser and the Recognizer. It constructs in each activity a "test" menu-item that manages the invocation statements for the corresponding

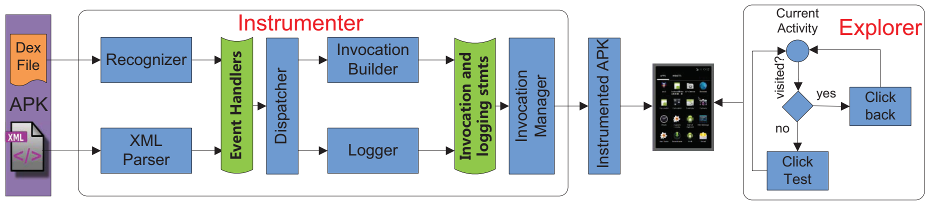


Fig. 6. Architecture of EHBDroid.

callbacks. The target activity of the invocation statement is determined as follows. If the event source is a view, the target activity is obtained by *view.getContext()*. If the event source is a component (i,e., activity, service, receiver), the corresponding event can be triggered in any activity. Without loss of generality, in this case, the main activity is considered as the target activity.

Invocation Builder. This module constructs the invocation statements according to Pattern 4. The arguments of callbacks are constructed according to the two means presented in Section III-C. For inter-app events, since the corresponding activities are started by third-party apps, we simulate these apps by directly instrumenting the statements that start new activities (like *startActivity(intentFromIntentFilter)*) in the main activity of the app to be tested.

Invocation Manager. An Invocation Manager corresponds to our instrumented event "test", and is realized by a "test" menu-item in each activity. A valid sequence of callback invocations in an activity is gathered in the event handler of "test". Once the "test" menu-item is clicked, all instrumented statements in the activity will be triggered.

Explorer. By maintaining a list of visited activities, the Explorer can determine whether the current activity has been explored. If an activity has not been explored, the Explorer will automatically click the "test" menu-item in the activity to explore it. Otherwise, the Explorer clicks "back", and the another activity on the top of the Android activity stack becomes the current activity. The above step iterates until the activity stack becomes empty. We leverage MonkeyRunner in the Android automated testing framework to send the device two UI events: click "test" and press "back". The current activity is obtained by the command: `adb shell dumpsys activity` of Android Debugging Bridge [21].

After introducing the implementation of EHBDroid, we discuss how it handles crashes and how it is used for debugging.

Handling crashes. In Android, when an uncaught exception occurs, the app will crash and terminate. To ensure that the testing continues until all event handlers are explored, we instrument every invocation statement in a try-catch block. Thus, when an exception (bug) is found, we can record the exception information and go on testing.

Enhancing debugging. EHBDroid also allows to piggyback the instrumentation to generate execution logs (**Logger** is in charge of this.). The log records the runtime state of each event (e.g., activity, event source, event handler, and callback), which is useful for debugging. For example, when a failure occurs, the recorded events can be used by Robotium [22] or other UI-based tools to reproduce the failure. Moreover, the corresponding bug can be easily located by inspecting the failing event handler.

V. Evaluation

We have evaluated EHBDroid on a collection of 35 real-world Android apps from F-droid and Google Play, and compared it with two popular UI-based app testing approaches: Monkey and Dynodroid, both of which have proven fault detection ability [14]. Through the experimental evaluation, we aim at answering the following three research questions:

- **RQ1** - Code coverage: Can EHBDroid achieve a higher code coverage than the other two approaches?
- **RQ2** - Testing efficiency: How efficient is EHBDroid in terms of event handlers triggered per minute?
- **RQ3** - Fault detection ability: Compared with the other two approaches, can EHBDroid find more bugs?

Benchmarks. Table II lists the apps used in our evaluation. These apps are randomly chosen from F-droid and Google Play with no prior knowledge about their event handlers. The first 25 (from F-droid) are popular benchmarks with a wide variety of functionalities, with 21K lines of code (Jimple), 2,000 methods, and 9 activities on average for each app. The other 10 apps (from Google Play) are all large and complex apps among the top 1,000 in the Android market, with 96K lines of code, 6.7K methods, and 18 activities on average.

Experimental setup. The Monkey [19] tool is a part of the Android SDK and is widely used by developers. It regards the app under test as a black-box and randomly generates UI (*touch(x,y)*) events. Since the number of generated events in Monkey can be configured by users, in our experiment, we set a sufficiently high bound (1M events) to ensure that Monkey does not stop too early, in order to make a fair comparison. Dynodroid [1] enhances Monkey by reducing the possibility of redundant event generation. We employ the default setting (BiasedRandom strategy) of Dynodroid in our experiment.

All experiments were performed on a PC with a 4GB memory and 2.4GHz processor, running Linux and Android 4.4. All experimental data were averaged over three runs.

A. Code Coverage

We use statement coverage as the criterion to compare the testing effectiveness of different approaches. Because existing tools for calculating statement coverage such as JaCoCo [23] and Emma [24] require Java source code, they cannot be

TABLE II
BENCHMARKS USED IN OUR EXPERIMENTS

App name	LOC	#Class	#Method	#Act
AGrep	2,106	39	129	6
AndroidomaticK	1,859	26	111	4
audiobook	54,752	892	3,648	7
browser	9,592	128	681	5
dalvikexplorer	2,081	43	212	16
kolen	1,803	25	95	4
vector	77,994	890	4,551	21
atomic	1,078	157	1,9137	10
podax	20,130	222	1,097	4
Nectroid	5,185	88	516	6
Notepad	549	10	32	4
OpenSudoku	7,624	104	561	10
ringdroid	7,860	64	332	3
Sanity	13,046	204	1,048	28
TippyTipper	2,998	47	238	5
VirtualDataLine	2,489	41	256	4
Vudroid	2,762	47	270	3
dashclock	3,059	15	129	7
pedometer	2,502	28	173	2
Apollo	19,446	156	1,391	9
SipUA	41,127	284	2,488	12
Anima	4,840	86	404	8
Editor	8,864	121	832	5
QKSMS	119,005	1,100	6,279	9
K9	127,012	931	6,532	28
Average	**2,1591**	**230**	**2,046**	**9**
TinyFlashLightED	75,459	756	4,666	6
AdobeReader	61,734	709	5,445	15
fightpic	59,210	679	7,689	9
ColorNote	57,825	748	3,249	20
podcast	165,850	2,210	12,205	17
AdobeAir	194,275	1,841	12,371	15
WordPress	11,2941	1,270	7,594	54
pokegowallpaper	22,131	239	1,482	9
BeautyPlus	167,516	1,493	9,293	24
WikiPedia	48,711	406	3,239	12
Average	**96,565**	**1,035**	**6,723**	**18**

used in our scenario where only the APK file is available. We hence implement a new tool called Asc (Android statement coverage) for calculating the statement coverage for APK files. Asc first uses static analysis to extract the length of each basic block of each method. Then, it instruments a statement at the end of each block to print the length of the executed block. The total lines and visited lines of statements can be calculated by $L = \sum_{i=1}^{n} l_i$ and $L_v = \sum_{i=1}^{n} b_i \times l_i$, respectively, where n is the number of blocks in the app, l_i the length of a block, and b_i (= 1 or 0) indicates whether the block is executed.

Overall results. We test each benchmark with the three tools and collect the coverage statistics for both the first 10 minutes and the end of an hour. Table III reports the results. For all the benchmarks, EHBDroid was able to finish testing (i.e., no more activities to explore) in 10 minutes, and for most benchmarks (20/25 in F-droid and 10/10 in Google Play), EHBDroid achieved a much higher coverage than the other two tools in 10 minutes. For the 25 F-droid benchmarks, on average, the coverage of EHBDroid is 7.6% and 9.39% higher than Monkey and Dynodroid, respectively. For the 10 Google Play apps, the difference is even larger. EHBDroid achieves as much as 26% more coverage than Monkey(on *ColorNote*) and 15.1% on average. Dynodroid failed to run on all of these apps, because it requires instrumenting the Manifest file before testing, which throws exceptions in these large apps.

Given more testing time, both Monkey and Dynodroid were able to increase coverage for many apps. However, after one hour, their achieved statement coverage is still smaller than that by EHBDroid. For apps from F-droid, the average coverage by Monkey and Dynodroid is close to (1%-2% smaller than) EHBDroid. Nevertheless, for those from Google-Play, the difference is still significant. Compared to Monkey, EHBDroid achieved as much as 22.3% higher statement coverage and 11.1% higher on average in an hour.

Result analysis. Our empirical results suggest that EHB-Droid is more effective and efficient than the other two UI-based testing approaches. This confirms the advantage of EHBDroid over Monkey and Dynodroid by directly triggering event handlers of UI, system, and inter-app events, whereas Monkey and Dynodroid can only trigger a part of UI events and some system events (Monkey cannot). For instance, the *OpenSudoku* app contains a few context-menu events which require two steps to trigger: long pressing and clicking on the pop-up menu item. Monkey and Dynodroid hardly trigger these events, as they do not consider long-press events, let alone their combinations with a successive click.

We also found that it is difficult for Monkey and Dynodroid to generate events associated with navigation drawers, list items, preferences, etc. For instance, the app *ColorNote* has a navigation drawer containing a few items that drive users to different activities. Usually, these items are invisible until the navigation view is clicked. However, when one of these items is clicked, other items are invisible again. Hence, given a limited time, Monkey and Dynodroid can only trigger a few items but not all of them. In contrast, EHBDroid guarantees to trigger callbacks for all items, because those events correspond to the same callback and their event handlers can be invoked easily by EHBDroid with different parameters.

Another reason why EHBDroid achieves a higher statement coverage than the other two tools is that when most events in an activity do not cause activity jump, the UI-based approaches often fall into a loop, which is hard to jump out. The loop consumes time but does not explore more app behavior. For example, in an activity of the *podcast* app (also *ColorNote*), there are many enabled events that do not cause activity jump; Monkey stayed in this activity for a long time until the sole event that can cause activity jump was triggered.

For a few apps such as *AGrep* and *kolen*, the statement coverage of EHBDroid is lower than that of Monkey or Dynodroid. The reasons are two-fold. First, EHBDroid currently does not support text input [25]. In *AGrep*, a “submit” button is unavailable when the edit texts for userID and password are empty. Hence, EHBDroid cannot reach the successor activities. Second, for performance reason, EHBDroid currently does not handle all items in a list, because such

TABLE III
EXPERIMENTAL RESULTS - RQ1: CODE COVERAGE AND RQ2: TESTING EFFICIENCY

App name	**Statement coverage(10min)(%)**			**Statement coverage(1h)(%)**			**#Events per minute**		
	Monkey	**Dynodroid**	**EHB**	**Monkey**	**Dynodroid**	**EHB**	**Monkey**	**Dynodroid**	**EHB**
AGrep	**74.28**	70.23	69.34	**74.28**	70.23	69.34	1,500	12	18
AndroidomaticK	66.39	55.98	**79.80**	83.11	**88.78**	79.80	1,818	12	28
audiobook	24.03	32.94	**35.02**	35.09	**41.67**	35.02	1,340	12	21
browser	**58.89**	57.49	**59.38**	**59.38**	57.49	**59.38**	1,412	12	36
dalvikexplorer	50.25	29.21	**56.14**	**57.02**	40.56	56.14	1,500	12	24
kolen	**72.89**	67.86	67.86	**72.89**	67.86	67.86	1101	12	44
vector	36.89	37.90	**47.44**	**48.61**	45.79	47.44	2,143	12	20
atomic	54.67	58.89	**66.47**	**66.47**	**66.47**	**66.47**	1,500	12	26
podax	59.27	**65.66**	46.77	**67.99**	65.66	46.77	1,519	12	25
Nectroid	55.27	65.89	**76.63**	76.63	**85.72**	81.63	1,519	12	25
Notepad	69.88	66.37	**82.37**	72.88	73.00	**82.37**	1,463	12	15
OpenSudoku	60.03	40.63	**71.97**	**71.97**	64.57	**71.97**	1,846	12	35
ringdroid	55.65	43.19	**73.49**	**78.35**	73.97	73.49	1,579	12	22
Sanity	**50.18**	**50.18**	**50.18**	**50.18**	**50.18**	**50.18**	1,000	12	29
TippyTipper	42.86	41.41	**78.29**	60.57	53.78	**78.29**	1,714	12	78
VirtualDataLine	32.39	**53.08**	36.33	42.15	**53.08**	36.33	1,538	12	25
Vudroid	64.01	58.10	**72.41**	72.41	68.35	**77.41**	952	12	30
dashclock	**82.18**	**82.18**	**82.18**	**82.18**	**82.18**	**82.18**	1,538	12	42
pedometer	**88.45**	73.89	**88.45**	**88.45**	84.23	**88.45**	1,846	12	38
Apollo	56.73	55.76	**70.46**	56.73	55.76	**70.46**	2,297	12	29
SipUA	18.74	**22.35**	17.98	18.74	**22.35**	17.98	1,846	12	31
Anima	66.08	69.28	**74.26**	66.08	**75.50**	74.26	469	12	32
Editor	54.44	62.02	**63.22**	**65.57**	62.02	63.22	543	12	32
QKSMS	38.98	32.57	**39.51**	**39.51**	32.57	**39.51**	403	12	26
K9	43.66	40.18	**62.88**	51.91	54.56	**62.88**	3,333	12	41
Mean	55.08	53.36	**62.75**	62.37	61.45	**62.75**	1,488	12	31
animeradio	28.67	-	**47**	32.75	-	**47**	2,595	-	40
AdobeReader	27.88	-	**46.78**	29.36	-	**46.78**	3,141	-	27
fightpic	33.56	-	**49.99**	35.83	-	**49.99**	3,750	-	23
ColorNote	24.57	-	**50.21**	27.88	-	**50.21**	3,380	-	18
podcast	27.04	-	**38.92**	34.39	-	**38.91**	3,692	-	35
AdobeAir	14.84	-	**20.09**	15.43	-	**20.09**	2,390	-	23
WordPress	26.57	-	**35.47**	**36.67**	-	35.47	2,521	-	31
pokegowallpaper	36.06	-	**48.97**	41.27	-	**48.97**	2,312	-	55
BeautyPlus	27.89	-	**45.97**	32.98	-	**45.97**	3,774	-	19
WikiPedia	32.63	-	**47.24**	32.63	-	**47.24**	3,015	-	46
Mean	27.97	-	**43.06**	31.92	-	**43.06**	3,057	-	32

item events often trigger the same behavior. We set a fixed number (e.g., 10) to limit the invocation times of the callback *onItemClicked()*, with the assumption that most of the items are handled in the same way. Thus, if a *ListItem* contains more than 10 items each of which is handled differently, some app behavior may be missed.

Low code coverage. For most apps, the achieved statement coverage by all the three tools is low (less than 50%). There are several reasons. First, in many apps there exists a large portion of code that is not relevant to the business logic of the app. Such code can only be reached under specific scenarios such as bug reporting, version updating, exception handling, etc. Second, some apps have dead code that can never be explored. For example, *SipUA* contains a large number of branches and methods that can never be reached. Third, some activities fail to be explored due to the missing of certain events for activity jumping. When such an event is missed, the target activity and its successor activities may not be explored. Fourth, certain code requires special permissions and is not accessible to ordinary users. For example, commercial apps usually provide both a free version and a paid version. The paid functionalities can only be explored via a paid account.

B. Testing Efficiency

The last three columns in Table III report the number of events (for Monkey and Dynodroid) or event handlers (for EHBDroid) generated per minute by the three tools. We use this metric to further show the testing efficiency of the three approaches. Monkey is the fastest approach among the three. It generates 1.5K events and 3K events per minute for the F-droid and Google Play apps, respectively. However, these events contain a large number of redundant (i.e., repeated) or invalid events that are not useful in exploring new app behavior. Dynodroidcollects events and sends them to the app at a fixed frequency (once every 5s) via the Android Debug Bridge. Accordingly, the number of events generated per minute by Dynodroid is a constant 12. Among these events, there may also exist redundant or invalid events. In contrast, each event

TABLE IV
EXPERIMENTAL RESULTS - RQ3: FAULT DETECTION ABILITY

App name	Monkey	Dynodroid	EHBDroid
AGrep	1	1*	1
Notepad			1
OpenSudoku	1	1	2
ringdroid	1	1	1
Sanity	1*	1*	1
padometer	1	1*	1
Apollo	1*	1*	1
K9	1	1	2
ColorNote			1
AdobeReader			1
Total	7*	7*	**12**

handler invocation by EHBDroid is unique and non-redundant. Moreover, the speed of EHBDroid is more than twice as that of Dynodroid, generating 32 events per minute. Since in our experiment the average time to instrument one app is one minute, EHBDroid is still more efficient even if the instrumentation time is included.

Another reason why EHBDroid is more efficient than Monkey and Dynodroid lies in the fact that with one click on “test”, all event handlers in the current activity are invoked together by EHBDroid, which avoids the latency caused by GUI and the cost of message passing in the system. In particular, if an activity contains many events that cause activity jumping, the testing time can be improved significantly by Dynodroid, because activity jumping is expensive and the UI-based approaches must jump back and forth repeatedly to trigger all events in that activity. For example, *TippyTipper* is a rate calculator containing 20 buttons. To test all the buttons, Dynodroid needs 20×5 = 100s, while EHBDroid takes only 2s to invoke the callbacks 20 times.

C. Fault Detection Ability

Table IV summarizes all bugs found by the three tools in our experiments, where $*$ represents bugs found after an hour. In ten minutes, EHBDroid found a total of 12 bugs (manifested as crashes or runtime exceptions) in these 35 apps, whereas Monkey and Dynodroid only found five and four, respectively. After running for an hour, Monkey and Dynodroid found two and three more bugs, respectively. Overall, EHBDroid found five new bugs that could not be found by Monkey and Dynodroid, and all bugs that found by Monkey and Dynodroid were also found by EHBDroid. These 12 bugs fall into three classes: UI bugs, inter-app bugs, and special bugs. We have also manually inspected these bugs and confirmed their validity. We next describe them in detail.

Eight UI bugs. These bugs were found in apps *AGrep*, *OpenSudoku*, *ringdroid*, *Sanity*, *padometer*, *Apollo*, *K9*, and *ColorNote* by EHBDroid. Except *ColorNote*, the other seven were also found by Monkey and Dynodroid. When the corresponding events are triggered, these apps crash. For example, in the *K9* Mail client, when an option button in the *Setting* activity is clicked, the app terminates abnormally. The error is due to the fact that in the corresponding event handler, *K9* uses an implicit intent to start a new activity, but neither the app nor the device can handle the intent. The UI bug of the app *ColorNote* was found in activity *PreferenceActivity* by EHBDroid. The other two tools failed to find this bug because they did not reach activity *PreferenceActivity*.

Three inter-app bugs. These bugs are found in apps *NotePad*, *OpenSudoku*, and *K9* by EHBDroid. These bugs are caused by the mismatches between some intent-filters and the received intents. The code below shows such a bug detected in *K9*. The activity *MessageList* defines an intent-filter in the Manifest file to filter the incoming intents. *MessageList* employs the method *decodeExtra* to resolve intents, requiring that the attribute *action* of the intent should be “android.view.action.View” (Line 2) and the attribute *data* of the intent should contain a non-null *path* (Line 5). If *data* does not contain a *path* attribute, when Line 5 is executed, an exception is thrown and the app crashes. Monkey and Dynodroid failed to find this kind of bugs, because they do not consider inter-app events.

```
1   class MessageList extends Activity{
2  boolean decodeExtras( Intent intent ){
3    if (("android.intent.action.VIEW".equals(action))
4    &&(intent.getData()!=null)){
5      List localList = intent.getData().getPathSegments();
6      String path = (String) localList.get(0);
7    }
8  }}
9  <activity name="MessageList">
10 <intent-filter>
11 <action name="android.intent.action.VIEW"/>
12 <data host="messages" scheme="email"/>
13 <category name="android.intent.category.DEFAULT"/>
14 </intent-filter>
15 </activity>
```

One special “bug”. This bug was found in *AdobeReader* by EHBDroid. It is special because it is in an event handler of a view that is invisible from the screen. Thus, from the perspective of end users, it is not a bug, but from the perspective of programmers, it is. The code below illustrates the bug. Note that *Automationt.class* is not declared in the Manifest file. Hence, the app violates the rule that an activity is valid only if it is declared in the Manifest file. When the non-visible menu item *2131231108* is triggered, the app crashes. Neither Monkey nor Dynodroid can find this bug, whereas EHBDroid can because it can obtain all the menu items via *menu.getMenuItems()* and directly trigger the events through *onOptionsItemSelected(item)*.

```
boolean onOptionsItemSelected(MenuItem item){
swtich(item.getId())
case 2131231108:
 Intent i=new Intent(this, Automationt.class)
  startActivity(i);
}
```

VI. LIMITATIONS

Except the 58 callbacks provided by Android, currently EHBDroid does not specially consider user-defined callbacks.

Fortunately, we find that many user-defined callback functions are invoked by the callbacks in Android, and thus these user-defined callback functions can also be explored by EHBDroid. Nevertheless, since an event handler may be called in a partial invocation context that differs from the real scenario, EHBDroid may yield false alarms (false positives) that are never triggered by end users, though some of the false alarms are still helpful for programmers.

Since the mechanism of EHBDroid is parallel to the event generation, it is orthogonal to the coverage criteria and exploration strategy. Currently, EHBDroid is at an early stage. Although a depth-first-search strategy is used for activity exploration, the event handlers belonging to an activity are invoked in a random (though valid) order and only event coverage is considered. Thereby, EHBDroid is not compared with the model-based and more advanced app testing tools that are based on event (event sequence) generation, such as GUIRipper [6], SwiftHand [9], Sapienz [11], etc.

EHBDroid relies on Soot for app instrumentation. However, Soot fails to instrument certain large apps with more than two DEX files. Besides, some apps prevent from being instrumented by hiding DEX files, checking signatures, etc. Thus, EHBDroid may fail to instrument such apps.

VII. Related Work

For Android app testing, a large body of work focuses on test input generation, i.e., event generation. According to their exploration strategies, existing approaches fall into three categories: *random testing* (or *fuzzing*), which generates random events to apps; *model-based testing*, which generates events according to certain models (such as finite state machines) of apps; and *advanced testing*, which uses more sophisticated techniques such as symbolic execution to generate events.

Compared to existing work, EHBDroid is distinguished by not generating events but directly invoking callbacks of the relevant event handlers, which is more efficient and effective. Considering the correlation between events and event handlers, our testing approach applies to general event-driven systems.

Random testing. Besides Monkey and Dynodroid, most early work focuses on random testing [6], [26]–[31]. Amalfitano et al. [6], [27] present a crawling-based approach to generate random but unique test inputs. There exist many tools for generating inter-app events by random generation of intent values [1], [19], [28]–[31]. Null intent fuzzer [28] concentrates on revealing crashes of activities that fail to properly check input intents. Intent Fuzzer [29] focuses on generating invalid test events with the goal of testing the robustness of apps. The primary limitation of random testing is that it often generates redundant events that are not useful for exploring new app behavior.

Model-based testing. This line of research often requires a GUI model of the application and has been intensively studied in GUI testing [6]–[10], [32]. The GUI model can be obtained manually [33] or via static/dynamic analysis [8]. Hierarchy Viewer [34] is a tool for generating GUI models for Android apps. Based on the GUI model, events can be generated to systematically explore the behavior of the app. For instance, GUIRipper [6] dynamically constructs an event flow graph for an Android app and follows a depth-first exploration strategy to test the app. ORBIT [7] adopts the same exploration strategy, but it is a white-box approach that utilizes the source code of the app to determine the events that can be triggered in each activity. Similar approaches and tools can be found in [8], [9], [35]. However, the effectiveness of these approaches heavily relies on the quality of the GUI model. In practice, the models are often very abstract and may not capture the complete behavior of the apps.

Advanced testing. Jensen et al. use symbolic testing to generate event sequences that reach specified target locations [33]. ACTEve [4] utilizes concolic execution to track events from their generation to their processing. These approaches instrument both the Android framework and the app, and can generate more complex event sequences that the other tools cannot. However, the scalability of symbolic testing is often limited. Similar approaches can be found in [8], [36]–[38].

By analyzing the interactions between widgets, TrimDroid [2] employs the constraint solver to generate a subset of event sequences, which can achieve a comparable coverage as the exhaustive combinatorial testing. EvoDroid [37] employs evolutionary algorithms to generate complex event sequences. Sapienz [11] uses search-based testing to automatically explore test sequences to minimize the length of event sequences, and simultaneously maximize code coverage and fault revelation. The tool AppDoctor [39] also triggers event handlers to simulate events. However, it only considers 20 types of events and merely focuses on specific bugs that cause apps to crash. In addition, it uses Java reflection to trigger event handlers, which is not so efficient. In contrast to AppDoctor, EHBDroid directly invokes event handlers based on instrumentation, which is more general and more efficient.

VIII. Conclusions

We have presented a new approach called EHBDroid for testing event-driven systems and specifically discussed its concretization for testing Android apps. The key advantage of EHBDroid is that by directly invoking the event handlers, it avoids the difficulty of generating complex events that are hard to trigger by traditional UI-based approaches, and it avoids the latency induced by the GUI and the cost of message passing in the system. We have presented an open source tool and evaluated its performance on a collection of 35 real-world Android apps. Experimental results demonstrate that EHBDroid can quickly reach higher statement coverage than the state-of-the-art UI-based testing approaches, and it is more powerful than the other approaches for finding bugs.

Acknowledgments

This work was supported in part by the National Key R&D Program of China under Grant No. 2017YFB1001801, the Natural Science Foundation of Jiangsu Province under Grant No. BK20171427, and the Fundamental Research Funds for the Central Universities under Grant No. 30917011322.

References

[1] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for Android apps," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26*, 2013, pp. 224–234.

[2] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering, ICSE'16, Austin, TX, USA, May 14-22*, 2016, pp. 559–570.

[3] 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, ISSTA'16, Saarbrücken, Germany, July 18-20*, 2016, pp. 82–93.

[4] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16*, 2012, p. 59.

[5] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, "Using GUI ripping for automated testing of android applications," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7*, 2012, pp. 258–261.

[7] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Fundamental Approaches to Software Engineering - 16th International Conference, FASE'13, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS'13, Rome, Italy, March 16-24. Proceedings*, 2013, pp. 250–265.

[8] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13, part of SPLASH'13, Indianapolis, IN, USA, October 26-31*, 2013, pp. 641–660.

[9] W. Choi, G. C. Necula, and K. Sen, "Guided GUI testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13, part of SPLASH'13, Indianapolis, IN, USA, October 26-31*, 2013, pp. 623–640.

[10] Y. M. Baek and D. 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, ASE'16, Singapore, September 3-7*, 2016, pp. 238–249.

[11] 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, ISSTA'16, Saarbrücken, Germany, July 18-20*, 2016, pp. 94–105.

[12] Z. Shan, T. Azim, and I. Neamtiu, "Finding resume and restart errors in android applications," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'16, part of SPLASH'16, Amsterdam, The Netherlands, October 30 - November 4*, 2016, pp. 864–880.

[13] H. Zhang and A. Rountev, "Analysis and testing of notifications in android wear applications," in *Proceedings of the 39th International Conference on Software Engineering, ICSE'17, Buenos Aires, Argentina, May 20-28*, 2017, pp. 347–357.

[14] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE'15, Lincoln, NE, USA, November 9-13*, 2015, pp. 429–440.

[15] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *37th IEEE/ACM International Conference on Software Engineering, ICSE'15, Florence, Italy, May 16-24*, 2015, pp. 89–99.

[16] R. Vallée-Rai, P. Lam, C. Verbrugge, P. Pominville, and F. Qian, "Soot (poster session): A java bytecode optimization and annotation framework," in *Addendum to the Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (Addendum)*, ser. OOPSLA'00, 2000, pp. 113–114.

[17] F. Limited, "Free and open source app repository." https://f-droid.org/, 2010.

[18] Google, "Google play store," https://www.androidcentral.com/google-play-store, 2016.

[19] ——, "The monkey ui android testing tool," http://developer.android.com/tools/help/monkey.html, 2015.

[20] XMLPrinterGroup, "Xmlprinter," http://www.xmlprinter.com, 2005.

[21] Google, "Android debug bridge," http://developer.android.com/tools/help/adb.html, 2015.

[22] RobotiumTech, "Robotium," https://code.google.com/p/robotium, 2010.

[23] EclEmmaTeam, "Jacoco: Java code coverage library," http://www.eclemma.org/jacoco/, 2009.

[24] SourceForge, "Emma: a free java code coverage tool," http://emma.sourceforge.net/, 2006.

[25] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *Proceedings of the 39th International Conference on Software Engineering, ICSE'17, Buenos Aires, Argentina, May 20-28*, 2017, pp. 643–653.

[26] C. Hu and I. Neamtiu, "Automating GUI testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test, AST'11, Waikiki, Honolulu, HI, USA, May 23-24*, 2011, pp. 77–83.

[27] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A GUI crawling-based technique for android mobile application testing," in *the Fourth IEEE International Conference on Software Testing, Verification and Validation, Berlin, Germany, 21-25 March, Workshop Proceedings*, 2011, pp. 252–261.

[28] N. Group, "Intent fuzzer," http://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx, 2009.

[29] R. Sasnauskas and J. Regehr, "Intent fuzzer: crafting intents of death," in *Proceedings of the Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), WODA+PERTEA'14, San Jose, CA, USA, July 22,*, 2014, pp. 1–5.

[30] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *The 11th International Conference on Advances in Mobile Computing & Multimedia, MoMM'13, Vienna, Austria, December 2-4*, 2013, p. 68.

[31] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11*, 2014, p. 29.

[32] F. Gross, G. Fraser, and A. Zeller, "EXSYST: search-based GUI testing," in *34th International Conference on Software Engineering, ICSE'12, June 2-9, Zurich, Switzerland*, 2012, pp. 1423–1426.

[33] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20*, 2013, pp. 67–77.

[34] Google, "Hierarchy viewer," http://developer.android.com/tools/help/hierarchy-viewer.html, 2012.

[35] 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 *The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19*, 2014, pp. 204–217.

[36] H. van der Merwe, B. van der Merwe, and W. Visser, "Execution and property specifications for jpf-android," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–5, 2014.

[37] 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, ESEC/FSE'14, Hong Kong, China, November 16 - 22*, 2014, pp. 599–609.

[38] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "SIG-Droid: Automated system input generation for android applications," in *the 26th IEEE International Symposium on Software Reliability Engineering, ISSRE'15, Gaithersbury, Maryland, USA, November 2-5*, 2015, pp. 461–471.

[39] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with appdoctor," in *Ninth Eurosys Conference, EuroSys'14, Amsterdam, The Netherlands, April 13-16*, 2014, pp. 18:1–18:15.