EHBDroid: An Event Handler Based Android Testing Tool

Wei Song
School of Comp. Sci. & Eng.
Nanjing Univ. of Sci. & Tech.
wsong@njust.edu.cn

Xiangxing Qian
School of Comp. Sci. & Eng.
Nanjing Univ. of Sci. & Tech.
xiangxingqian@gmail.com

Jeff Huang
Parasol Laboratory
Texas A&M University
jeff@cse.tamu.edu

ABSTRACT

We present EHBDroid, a tool for testing Android apps, which is an event-handler based approach. Different from traditional GUI testing approaches, EHBDroid does not need to generate events, but directly invokes the callbacks of event handlers via code instrumentation. We have evaluated EHBDroid on a set of real-world Android apps. The experimental results show that EHBDroid significantly outperforms the state-of-the-art UI-based approaches in terms of both code coverage and testing efficiency. EHBDroid can quickly reach higher code coverage than two popular GUI testing tools, Monkey and Dynodroid, and has revealed several bugs that cannot be detected by the others.

1. INTRODUCTION

With the growing popularity of mobile applications (or simply, "*apps*") today, app testing is crucial not only to the business of vendors, but also to the quality of our lives. However, despite of intensive research, app testing still faces significant challenges due to the asynchronous nature and complex events of mobile systems. A key challenge is how to effectively and efficiently generate events (i.e., test inputs) that can simulate user or system behaviours. For real-world apps, the space of events is often enormous. In addition, it is difficult to simulate most system events and inter-app events, as well as a number of UI events (e.g., drag, hover). Although this problem has attracted a great deal of recent research [6, 10, 5, 12, 7, 8], the code coverage rate and testing efficiency are still unsatisfactory. According to a recent study [9], most existing approaches are UI-based, and they are either too slow to generate events, or cannot effectively generate certain events.

In this paper, we present a new Android testing tool, EHBDroid, powered by the *Event Handler Based* (EHB) testing. The key observation of EHB is that, in event-driven systems, there usually exist correspondences between events and event handlers. To automatically explore the behaviour of an app, events are not necessarily needed, instead, their event handlers can be invoked directly. For example, consider a statement `view.setOnClickListener(myListener)`, in which `view` is a UI element, `click` is an event on `view`, and `myListener` is the event handler of `click`. Instead of attempting to generate a `click` event on the `view`, we can directly invoke the `click` event handler `myListener` in the app. In this way, testing becomes both more effective and more efficient, because all event handlers can be tested, and there is no need to generate the events that are hard to simulate.

EHBDroid is implemented on top of Soot [11], a popular Java and Android static analysis framework. EHBDroid is a fully automated app testing tool which works as follows. For an input APK file, it instruments all event handling statements in each activity of the app. When the app is started, it directly triggers the event handling statements instrumented before. EHBDroid terminates when there is no new activity to explore or a pre-set time budget has reached.

We have evaluated EHBDroid on 35 real-world Android apps from F-droid [1] and Google Play Store [2], and compared the performance of EHBDroid with two state-of-the-art UI-based app testing tools, Monkey [4] and Dynodroid [10], based on three metrics: code (method) coverage, testing speed (the number of events or event handlers triggered per minute), and the ability to detect faults. The experimental results show that EHBDroid significantly outperforms Monkey and Dynodroid in terms of testing coverage and testing speed. Moreover, it finds several new bugs that cannot be detected by the other two tools.

2. ANDROID EVENTS AND REGISTRATION MECHANISMS

In this section, we briefly summarize Android events and explain their registration mechanisms, which are important for realizing EHBDroid.

Events. There are three categories of events in Android: UI events, system events, and inter-app events. UI events are usually associated with UI components, such as views and menus. System events are generated by the Android system, such as incoming phone-calls, SMS messages, notification of low battery, etc. An inter-app event is used for communication between applications, which is published by one app and subscribed by others. For example, when a user clicks a mp3 file received in an IM app (e.g., WeChat), an inter-app event will be sent from the app to a music player.

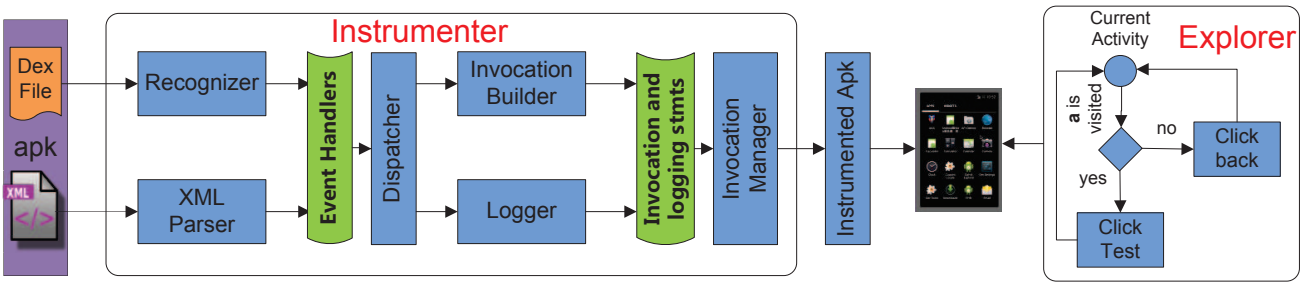


Figure 1: Architecture of EHBDroid.

Registration. UI Events. Android SDK provides two ways for UI elements to register UI events: specifying attribute `onClick="click"` in XML resources files or using `view.setXXListener(l)` in the Java code.

System Events. System events can be categorized into two types: broadcast receiver events and system service events. For broadcast receiver events, similar to UI events, there are two ways to register receiver events: declaring intent-filter in the Manifest file or using `context.registerReceiver()` in the code. In either case, an intent-filter should be declared. A receiver event can be accepted by the app whose intent-filters match the sent intent. For system service events, system service is managed by the `Service Manager`. Android provides one method to register service event via using `serviceManager.addXXListener(l)` in the code.

Inter-app events. Inter-app events are used when apps communicate with each other. Similar to broadcast receiver events, apps need to declare intent-filters to filter intents in the Manifest file. The events are handled by the activity whose intent-filters match the sent intent [3].

3. EHBDROID

Figure 1 shows the architecture of EHBDroid, which consists of two major components: Instrumenter and Explorer. Taking an app as input, the Instrumenter instruments the app by inserting invocation statements for event handlers. The Explorer systematically tests the app by exploring the instrumented invocation statement. In the following, we explain the elements of these two components.

XML Parser and **Recognizer.** EHBDroid first locates all event handlers in the app by searching both the XML resource files and the app source code. The XML Parser utilizes XMLPrinter to parse the input XML resource files and searches for two types of event handlers: one for UI events, and the other for system and inter-app events. The first type consists of view and method, and the second type consists of component(i,e., activity, service, receiver) and intent-filter objects. Both of them are stored in a map called *eventmap*. Recognizer uses Soot [11] to transform the dalvik code to Jimple, and then analyzes the event handler registration statements (see Section 2). From all such statements, three different attributes (event source, registered method, and event handler) are extracted and stored in a Java class called *eventtriple*.

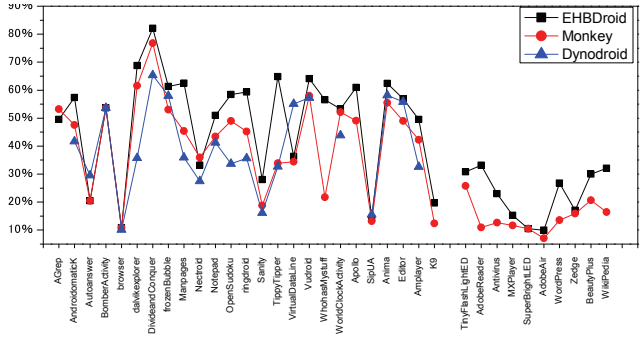
Dispatcher. This element is used to dispatch the collected objects (i.e., *eventmap* and *eventtriple*) to activities. In real-world apps, because most of the identified statements are not directly defined in *classes* of activities, we need to first locate the activity that the event source belongs to, and then instrument the invocation statements into the *classes* of those activities. For *eventmap*, if the event source is a component (i,e., activity, service, receiver), we use the `MainActivity` as the target activity by default because the corresponding event does not belong to any specific activity. If the event source is a view, we use the activity which initializes the view as the target activity. For *eventtriple*, if the event source is a view, the target activity can be acquired by instrumenting the statement `view.getContext()`.

Invocation Builder. Invocation Builder constructs the invocation statements for the collected list *eventmap* and *eventtriple*. The order of building invocation statements is the same with the order of collecting event handlers. Given an event source and its event handler, we directly invoke the callback of event handler. The key challenge is how to construct the parameters of callbacks. We use the arguments extracted from the event sources for this purpose. For example, suppose that we find an event registration statement `adapterView.setOnItemClickListener(listener)`. For all methods of the event handler `listener`, we can determine the callback `onItemClick` whose name matches the registration method `setOnItemClickListener`. By using Java Reflection, the signature of `onItemClick` can be obtained: `void onItemClick(AdapterView av,View v,int i,long l)`. The second parameter is the view where a click event can be triggered; the first parameter is the view list to which the view belongs; the third parameter indicates the position of the view in the view list; the forth parameter is the row id of the view. The value of the first parameter AdapterView can be obtained directly from the code, while the last three are associated with AdapterView and can be obtained through the following instrumented statements:

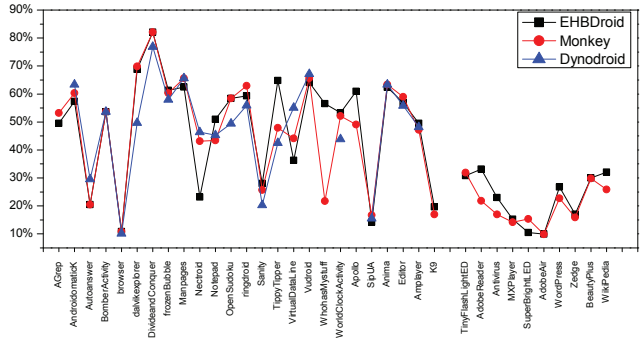
```
1 for(int i=0; i<av.size();i++){
2   View v = av.getChildAt(i);
3   int p = i;
4   long id = av.getAdapter.getItemId(j);
5   listener .onItemClick(av,v,p,id);
6 }
```

For different callbacks, however, the code to construct the arguments is different. Currently, we support constructing arguments for a total of 43 different callbacks.

Logger. This element aims to generate reproducible test cases and provide debugging support. It extracts two extra elements from *eventmap* and *eventtriple*: the activity that contains the event source, and the callback of the event handler. The outputs are four elements: event source, activity that contains the event source, event handler and its callback. These four elements constitute the test inputs. The instrumented statements built by Logger are called logging



(a) in 10min



(b) in 1h

Figure 2: Method Coverage

statements.

Invocation Manager. This element is realized by a "test" menu-item in each activity. All instrumented statements (invocations of callbacks and the corresponding logging statements) are summarized in the event handler of "test". Once the "test" is clicked, all instrumented statements will be invoked one-by-one. Take the following code snippet for example. The menu-item "test" registers a listener `oml` whose callback is `onMenuItemClick` in which Invocation Manager instruments all the invoking statements.

```
1 MenuItem menuItem = new MenuItem("test");
2 OnMenuItemClickListener oml;
3 oml = new OnMenuItemClickListener({
4   void onMenuItemClick(){
5   //invoking statements
6   }};
7 menuItem.setOnClickListener(oml);
```

Explorer. The Explorer is a separate tool written in Python. It tests an app with a DFS exploration strategy. When an activity has not been explored yet, Explorer automatically clicks the menu-item "test" to invoke all the event handlers instrumented in the activity. If there exist event handlers that cause activities jumping, new activities will be pushed into the *Android Activity Stack* provided by Android in turn. When the new activity has been explored, Explorer automatically clicks the navigation button "back", which pops up an activity in the stack. The above step is repeated until no new activity can be explored.

4. TOOL USAGE

To use an app with EHBDroid, users simply follow three steps: instrumentation, installation, and testing.

Instrumentation. Put the target app (e.g., apk) under the `srcApk` directory, and run `instrument.py`:

- `cp app.apk srcApk`
- `python instrument.py android-platforms APKNAME`

Installation. Run `install.py` to create and start an emulator named "Test", and install the instrumented app:

- `python install.py APKNAME`

Exploration. After the app is successfully installed and started, run `exploration.py`:

- `python exploration.py APKNAME`

5. EVALUATION

In this section, we present our evaluation results of EHBDroid. We first describe the benchmarks, then the experimental setup, and finally the experimental results on three metrics: method coverage, number of triggered events per min, and fault detection ability.

We compare EHBDroid to two popular UI-based app testing tools: Monkey [4] and Dynodroid[10]. Monkey is popular testing tool integrated in the Android SDK. It employs a fuzzing black-box testing approach that generates events randomly. Compared to Monkey, Dynodroid improves the random exploration efficiency by more directed exploration. For example, Dynodroid can select the events that have been least frequently selected (the frequency strategy) and prioritize events that are relevant in more contexts. Besides, by instrumenting the Android framework, Dynodroid can monitor the app state so that more relevant events can be generated.

5.1 Benchmarks

We evaluate EHBDroid on 35 real-world Android apps, including 25 from F-droid [1] and 10 from Google Play store [2]. The apps in F-droid are popular benchmarks with a wide variety of functionalities, with 19K lines of code and 1,500 methods on average for each app. The 10 apps from Google Play are all large complex apps among the top 100 in Android market, with 300K lines of code and 19K methods on average for each app. We use small apps to verify the effectiveness of EHBDroid, and large apps to assess the scalability of EHBDroid.

5.2 Setup

All our experiments were performed on a Linux PC with 2.4GHz processors and 4GB memory, running a KitKat version of Android. Before starting an app, we first need to configure the emulator through uploading some extra files: 2 pdf, 2 image, 1 mp3 and 1 mp4. Those files are essential to test some apps. In addition, for some specific apps, there are two manual work left to users: providing text input and scrolling the screen. For example, some apps require the account number and the password which should be provided by users. When an app releases a new version, it often shows the new features on its first activity. In this case, users should swipe left or right to access the app. These two efforts should be done before all the three tools Monkey, Dynodroid, and EHBDroid start.

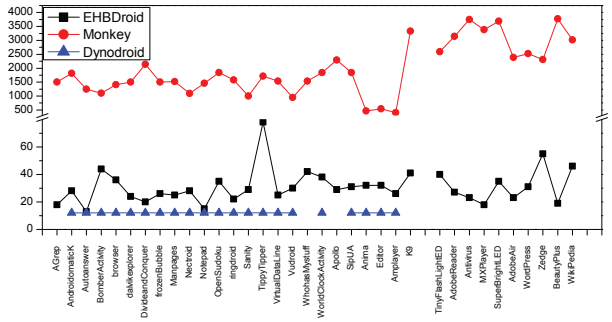


Figure 3: Numbers Per Min

5.3 Results

Method Coverage. We tested each benchmark with one of the three tools and collected the coverage data for the first 10 minutes and by the end of one hour (Figure 2). For all the benchmarks (including both F-droid and Google Play), EHBDroid was able to finish testing in 10 minutes, while the other two tools were not. In the experiment of 10 minutes, EHBDroid achieved a higher coverage than the other two tools on most of apps. Dynodroid failed on four apps in F-droid and all apps in Google Play because it could not regenerate the instrumented apk files. Given more testing time (one hour), both Monkey and Dynodroid were able to increase method coverage for many apps. The three approaches achieved almost an equally matched coverage. However, the average coverage achieved by EHBDroid was still a bit higher than Monkeyand Dynodroid. In summary, the results indicate that EHBDroid is more effective (higher coverage in less time) than UI-based testing approaches.

Numbers of per min. Figure 3 reports the number of events or event handlers generated per minute by the three tools. We use this metric to show the testing efficiency of the three approaches. Monkey is the fastest approach, EHBDroid comes next, and Dynodroid is the last. Monkey generated 1.5K or higher events per minute for all the apps. However, these events contain a large number of redundant or invalid events that are not useful in exploring new app behaviours. For Dynodroid and EHBDroid, the events are all unique and non-redundant. Dynodroid collects events and sends them to the app at a fixed frequency (once every 5s). The number of events generated per minute by Dynodroid is a constant 12. For EHBDroid, it achieved 20 or higher events per min for most apps, which is nearly twice that of Dynodroid.

Fault Detection Ability. Table 1 summarizes the bugs found by the three tools in our experiments. In ten minutes, EHBDroid found a total of 10 bugs (manifested as crashes or runtime exceptions) in these 35 apps, whereas Monkey and Dynodroid only found 3 and 2, respectively. After running for more than an hour, Monkey and Dynodroid found two and one more bugs, respectively. Among the 10 bugs, there are five UI bugs, four inter-app bugs and one special bug. UI bugs can be detected by the three tools, while the four inter-app bugs and the special bug can only be found by EHBDroid.

Table 1: Bug Detection Results. * – found after an hour.

AppName	Monkey	Dynodroid	EHBDroid
AGrep	1		1
Notepad			**1**
OpenSudoku			**1**
ringdroid	1	1	1
Sanity	1*	1*	1
Apollo	1*		1
K9	1		**2**
AdobeReader			**1**
Total	5*	2*	**10**

6. CONCLUSION

In this paper, we present EHBDroid, an novel tool for app testing, which directly explores the event handlers for the testing purpose. We implement EHBDroid via the instrumentation on Android APK files. Our evaluation on a set of real-world Android apps shows that the tool can quickly reach higher code coverage and find more bugs than the state-of-the-art UI testing tools do.

7. REFERENCES

[1] Free and open source app repository. `https://f-droid.org/`.

[2] Google play store. `https://www.androidcentral.com/google-play-store`.

[3] Intent-filter. `http://developer.android.com/tools/docs/Intent.html`.

[4] The monkey ui android testing tool. `http://developer.android.com/tools/help/monkey.html`.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[6] 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*, page 59. ACM, 2012.

[7] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *ACM SIGPLAN Notices*, 48(10):641–660, 2013.

[8] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *ACM SIGPLAN Notices*, volume 48, pages 623–640. ACM, 2013.

[9] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? *arXiv preprint arXiv:1503.07217*, 2015.

[10] 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*, pages 224–234. ACM, 2013.

[11] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[12] 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*, pages 250–265. Springer, 2013.