Automated Input Generator for Android Applications

Shobhit N. Dutia, Youngho Kim, and Tae Oh
Rochester Institute of Technology
Rochester, NY, USA
{snd7555, yhkist, tom.oh}@rit.edu

Abstract—Android applications have been tested without any knowledge about them using a variety of tools such as Monkey [2], Monkey Runner [3] etc. In this paper, we evaluate existing Android testing techniques by comparing each of these tools and evaluate their efficiency based on a number of factors. Next, we propose requirements of an ideal input generator and present an automated input generator using the Robotium [1] test automation framework to test Android applications while generating valid inputs. The proposed system is targeted to automate the use of Robotium and create a generic test script that tests Android applications with valid inputs without the requirement of their source code. Further, we discuss issues encountered during the development phase of the system and by increasing the complexity of Android applications. We evaluated the system on Android applications with a simple user interface and conclude that the proposed framework using UI testing more efficient as compared to a randomized testing approach.

Keywords—Android; Applications; UI Testing

I. INTRODUCTION

According to International Data Corporation (IDC), Android accounted for 81% of all smartphone shipments in the third quarter of 2013 [4]. With the increase in smartphones, mobile application installs from Google Play have reached more than 48 billion app installs as revealed by Sundar Pichai, senior vice president, Chrome and Apps at Google I/O 2013 [5]. Thus, Android application development has become increasingly popular with the need of an effective software solution for application development and testing. While most Android applications are developed in Java, the Android framework is quite different from typical Java frameworks. Thus, while development or testing of Android applications, the user needs to be aware of all the terms and concepts specific to the Android framework such as activity [9], views [10] and the Android SDK [7] in general.

A key to successfully develop Android applications is not just understanding the core Android concepts but also testing the functionality of the application before it is published in the Android market. The most important part of testing is simulating inputs to the application that closely resemble the inputs that users are actually going to provide once the application is downloaded. Coupled with inputs, another important facet of Android testing is to ensure maximum code coverage while the application is tested. This is largely dependent on the inputs supplied while testing the application.

To achieve automated testing with these criteria in mind, it is challenging to ensure ideal testing of Android applications. Our focus of Android testing was therefore to devise a solution which matches or at least closely resembles the ideal testing of Android applications.

In this paper, we present an automated input generator which systematically scans the user interface of an application and simulates clicking of each object one-by-one till there are no objects left or till the system has covered the maximum amount of possible views in the user interface of the application. The simulation of inputs is not just random inputs but systematic, valid inputs. The underlying principle behind this approach is the depth first search algorithm which traverses throughout the application to cover as many layers of the application’s user interface as possible.

The proposed input generator is fully automated and it is therefore difficult to devise a generalized automated solution for all Android applications. Moreover, the system takes the user interface of the application into account and it is therefore challenging to include applications with complex user interfaces such as large application or games. Therefore, the proposed idea is currently tested only on simple applications. Future functionality takes into account complex applications while the basic idea of traversing the application’s user interface will remain the same.

We evaluated the technique by comparing it on a high level detail with the existing input generating tools such as Monkey and Monkey Runner. The proposed input generating technique is not only more efficient while generating valid inputs as compared to randomized inputs but also more easier to use than tools such as Monkey Runner. In other words, the idea of the proposed input generator couples the simplicity of the random approach in the Monkey tool along with the benefit of ensuring valid inputs in the Monkey Runner tool while ensuring maximum code coverage.

The rest of the paper discusses the background, architecture and the implementation of the proposed system and finally evaluates the difficulties while implementing the system and compares the system with other input generators such as the Monkey and Monkey Runner tools.

II. BACKGROUND

In general, Android applications are distributed in the form of a package which consists of a manifest file (.xml file), resource files, and application code (.dex files). The .dex (Dalvik Executable) files are generated from the application’s source code written in the Java programming language. As
with class files in Java VM, the Android application code will be processed by Dalvik VM at runtime. Android also provides a sandbox execution model, isolating each application from the others signed by a different private key. In addition, its permission-based access control mechanism limits the application’s capability to only the permissions granted explicitly from the user. Hence, this security-enhanced mobile platform does not allow an external examiner to look into the layout of the application, let alone press a button. Most of the time, Android applications are triggered by user inputs or events from services. For this reason, one of the challenges in automating test procedures for an application is the user interaction without using real user inputs. Therefore, the input generator should control the application under test by generating possible events to trigger all activities inside the application. Without the source code of the application, it is difficult to generate enough events to traverse all possible paths of execution. To make things worse, some applications require a valid form of input to advance to the next activity. For example, a text input area requesting a phone number only understands numerical types where alphabetical characters could yield an incorrect input error and may force the application to end. Accordingly, the accuracy of an automatic test system on an application would be determined by the performance of the input generator. Therefore, we need to design a more application-aware input generator which will create valid inputs in order to traverse all possible execution paths automatically. In addition, the new input generator should address the critical issues introduced in the previous input generators.

III. ARCHITECTURE

This section first presents the requirements of an ideal input generator. Next, it presents the overall architecture of the automated input generator to meet the ideal requirements.

1) Requirements of an ideal input generator:

a) Travel through all possible activities or views of an application: Every Android application consists of a set of activities and views. An activity is a component that provides a screen with which users can interact. A view is used to build the user interface elements of an Android application. To ensure that the application is completely tested, the input generator should test all the activities or views of an application. In other words, the input generator should cover as many lines of code as possible so that all possible execution paths of an application are covered.

b) Generate valid inputs as it traverses through activities: One of the most important aspects of an input generator is generating valid inputs as opposed to random inputs to test an application. For example, some applications might require an email address to be entered in a text box. If a random input generator does not enter a valid email address, the application may not proceed further. Input generators such as 'UI/Application Exerciser Monkey' often have this issue. On the other hand, testing illegal inputs are also as important as valid inputs as they might be a source of other, possibly unexplored, execution paths. Thus, an ideal input generator should consist of a set of valid and invalid inputs to test an application.

c) Never crash: This issue is similar to the above issue where the application may crash if the inputs entered are not valid. To ensure that the application is tested completely, the inputs entered should be valid and must not result in a crash.

d) Easy to use: Tools such as Monkey Runner require the knowledge of Python to test an application. This makes testing difficult and requires an overhead of learning Python. On the other hand, automation tools such as Robotium only require the knowledge of Java to test an application. This makes testing easy to implement as well as requires less time to write robust test cases. Thus, writing test cases in tools such as Robotium are easier as compared to monkeyrunner.

e) Requirement of the application source code: An input generator should be able to test the application without requiring the application source code. This is often an overhead when writing robust, automatic black box test cases. Tools such as monkeyrunner work without the application source code; however, without the code, the testing may need the object’s coordinates, to say, click on a button. Hence a slight change in an object’s UI will require the test cases to be re-written.

f) Work with multiple applications: Often, an input to an application opens another application. For example, clicking the address field in the contacts page opens the google maps application. An input generator should successfully work across such applications. The tool Robotium is not yet supported to develop test cases across such applications.

2) Proposed input generator: The above requirements generally apply for UI testing of Android applications. The algorithm given below is a high level overview of the Robotium script which generates all possible inputs in an application. It assumes that the application is installed on the emulator and is ready to be tested. Further details about how an application is made ready for testing using Robotium is explained in the implementation section later. The proposed technique seeks to resolve two major hurdles of an ideal input generator.

a) Code coverage of an application: i.e. travel through all possible activities or views of an application

b) Generate valid inputs as it traverses through the application.

The algorithm takes as an input a running Android application and outputs all possible execution paths traversed while entering valid inputs. The basic idea of the algorithm is a depth-first search approach which keeps track of the activities in a stack and visits them one by one thereby traversing through all possible activities in an application. This is illustrated by the following example.

Consider a simple application with three activities A, B and C.

- Activity A represents a view containing a text box and two buttons which open activity B and C respectively.
• Activity B represents a view containing a text box with expected input as an email address and a back button which redirects the user back to activity A.

• Activity C represents a view containing a text box with expected input as a number and a back button which redirects the user back to activity A.

The proposed algorithm works as follows:

• Generate valid inputs to the text field
• Click button 1
• Check whether clicking the button starts a new activity.
• If yes, then recursively call itself to explore that activity and so on.

This depth first search approach is summarized in the figure below:

Algorithm 1 Overall algorithm of input generator

Input: A running Android application

Output: Traversed through all possible execution paths of an application

1. Function:
2. loop through current view
3. if view is TextBox
4. enter valid text inputs
5. loop through the current view
6. if view is CheckBox
7. check it
8. loop through the current view
9. if view is Radio Button
10. click it
11. loop through the current view
12. if view is button
13. click the button
14. if current activity is different from previous activity
15. call Function
16. END

The algorithm starts with the first activity and traverses its view/UI to check for all text boxes. It then checks the type of the text box e.g. if the text box only accepts email addresses, it enters a dummy email address hard coded in the test script. The test script thus, has hard coded values of every possible input type viz. Number, email address, passwords etc. and checks the type of the text box at run time. Next, it traverses the UI again to check for radio buttons and check boxes and clicks them. This click-for-all check boxes and radio buttons behavior is adopted as an application may not proceed further if, say a given check box or radio button is not clicked. Finally, when it encounters a button, it clicks it and checks whether clicking the button resulted in a new activity to be launched and scans the UI of the new activity again otherwise it continues to click other buttons without recursively calling itself. This process is repeated till there are no activities left in the application.

The above approach may not scale well for applications with a complex UI as the script takes into account only text boxes, check boxes, radio buttons, buttons and image buttons. Thus, the current architecture is focused only on simple applications. However, the script can be expanded to take into account additional input methods and the depth first search approach can still be applied as the complexity increases.

IV. IMPLEMENTATION

The automated input generator was implemented using the Robotium framework. Robotium is an easy to use tool which helps to create automatic black-box tests for Android applications in Java. The current use of Robotium requires a number of manual steps. In order to create an automated input generator, this manual process had to be automated right from automatically instantiating an emulator to firing all possible inputs on the application. Moreover, the above algorithm is not specific for a particular application and takes into account a set of simple applications. This includes challenges of scalability while using Robotium. This section first explains the usage of
Robotium for one application and later explains the implementation details of the system to handle multiple applications.

A. Testing an application using Robotium:

To test an application using Robotium requires the following steps:

1) Resigning the application to be tested: An Android application is signed with either a debug key or a release key before it can be installed on an emulator or any device running the Android OS. To test the application using Robotium, it needs to be re-signed with the same debug key as that of the test project. This is the primary requirement to test an application using Robotium. Therefore, the application needs to be unsigned first and then re-signed with the local debug key of the test project.

2) Installing the application on the emulator: After re-signing the application, it needs to be installed on the emulator.

3) Creating a test project in eclipse: Next, a test project needs to be created in eclipse. The major steps to create a test project re:

a) To test a given application: Robotium requires the application’s package name in the manifest file of the test project and the name of the launcher (or the first) activity of the application to be tested in the Robotium script (java) file to identify the application under test.

b) Adding Robotium.jar in the build path of the project: Robotium: Robotium.jar file contains the required files to use the Robotium framework

c) Write a Robotium test script in the test class: The test script contains the generic depth first search algorithm explained in the architecture section.

4) Running the test project: Finally, the test project can be run from within Eclipse. Eclipse will build a test apk file and run the test (This is the apk file which fires inputs on the application based on the Robotium test script). It is important to note that there are two applications into account.

a) The application which is to be tested.

b) The test application built by Eclipse which fires inputs on the application to be tested.

B. Creating an Automated Implementation of Robotium:

As explained in the previous section, Robotium requires a number of manual steps to test an application. The biggest hurdle to this challenge is replacing the use of eclipse to setup the test environment. This can be automated by creating the Android test project and the test environment only once for the first application. Subsequent applications can use the same test project and a shell script can be used on the existing test project to modify the parameters of the files within the project. The shell script is summarized in the figure below:

Fig. 3. Automated implementation of Robotium

a) Extract package name and name of the launcher (first) activity of the application to be tested: The Android SDK contains the Android Asset Packaging Tool (aapt). This tool is used to compile resources to create an application and can also be used to update and view the contents of Android zip-compatible packages (apk, jar, zip). The package name and the name of the launcher (first) activity of the application under test can be extracted using the command: “aapt dump badging fileName.apk”. This returns the required information along with other information of the application such as which permissions does the application use, what is the target sdk version used by the application etc. The output of this command can be piped into the ‘grep’ command utility to extract the package and the activity names. Further text processing can be applied to the output of the grep command using the ‘sed’ command to accurately return the package name and the launcher activity name of the Android application under test.

b) Modify the test project's manifest file with the package name and the test script (java) file with the name of the launcher activity: Once the script extracts the launcher activity and package names, the information can be updated to the required test script file and the manifest file respectively within the already created test project. This can also be done using the ‘sed’ command utility to replace the existing values of package and activity names within the manifest file and the test script files respectively.

c) Build the test apk: After the test project is modified with the change in the manifest file and the script file, a test application is built using the Apache Ant tool. Ant is a tool widely used to build Java applications and can be used in this project to replace the usage of Eclipse to build the test apk file.

d) Resigning the application to be tested: As explained in the previous section, Robotium requires re-signing the application before it can be tested. This can be done by removing the existing signature from the test apk by deleting the ‘META-INF’ folder within the apk file. This folder contains the signature information of an Android application. The folder can be deleted using the ‘zip –d’ command. Next, the application can be re-signed by the local debug key which is the same as the test project using the ‘jarsigner’ command.

e) Install test apk and re-signed apk in the emulator and run the test: The script can then install the re-signed application and the test application in the emulator and run the test using the Android Debug Bridge (adb) tool. For example: adb install ‘Path to apk file’
V. Evaluation

In general, Android applications are implemented as a combination of Activity, Service, Broadcast Receiver, and Content Provider components. In this paper, we explain the idea by taking simple applications into account with mainly view objects which are implemented as Activities. Thus, an increase in the complexity of applications comprises applications with other types of components as well. Moreover, the view objects considered in the current implementation consist of only check boxes, radio buttons, image buttons and buttons. An activity may have other complex view objects such as spinners, pickers etc. Such objects have not yet been taken into account. Thus, the above implementation is suited well for applications consisting of a simple UI. As the complexity and number of applications we tested increased, there were many issues encountered; some which are solved and others which are targeted to be resolved in the future. One of the major drawbacks in this approach is the limited amount of input types taken into consideration. While this issue remains an area of concern, we discovered that applications under test, whether simple or complex in terms of its UI, tend to work well with valid inputs as opposed to random inputs. Randomized approach such as Monkey has been compared with other input generating techniques [6] and results show that GUI-based testing is much more efficient than random testing of Android applications. This section outlines some issues encountered during the implementation of our system and lastly discusses comparison of the proposed input generator with random input generator Monkey and other widely used input generating techniques such as Monkey Runner.

A. Difficulties and Issues while Implementing the System:

a) Splash Screens: In a number of applications, the first activity loaded is a Splash Screen. A Splash Screen is used to show some kind of progress to the user before the application loads completely. This was incorrectly recognized by the algorithm as the first activity and therefore crashes when used with such applications. A workaround for this issue is invoking a delay for a specified period of time (3-5 seconds) for the Splash Screen to load completely before the first activity is loaded. This is based on the assumption that Splash Screens do not last for more than 2 or 3 seconds for most applications. Therefore this delay is sufficient for an activity to load.

b) Incorrect activity recorded: This issue is similar to the Splash Screen issue. If the test script clicks a button and a new activity is launched, there might be a slight lag to load the new activity completely. This might result in an incorrect activity name recorded by the script since the new activity is still yet to be loaded. This issue was resolved in a similar manner as the Splash Screen issue where a delay of 3-5 seconds was introduced from the time when a button is clicked till a new activity is loaded.

c) Resigning application to use Robotium: As explained in the implementation section, an application needs to be resigned in order to test it using Robotium. This was successfully implemented using various commands in our shell script however, a very few number of applications such as the stock ‘Contacts’ application in the Android OS tend to crash if their signature is modified. This is an inherent issue with using Robotium and is no solution for this as yet.

d) Support of Robotium across multiple applications: Currently, Robotium is only provided to support multiple Activities of an application. However, some applications direct the user into a different application if the user initiates some kind of input. For example many applications have a ‘contacts’ page where clicking an address redirects the user to the Google Maps application. Robotium does not support testing of such applications. The test may crash abruptly if a new application is launched.

e) Avoiding cycles among the views: It is critical to detect cycles in the view layout and avoid infinite loop while searching. To this end, we store all object references that have been traversed before and compare them with the current view object reference to see if there is a cycle. Upon identifying a cycle, the traversal flow will skip the current view object reference and jump to the next view object.

f) Recovery step in case of crash: It is assumed that the application consists of only simple view objects such as text boxes, radio buttons, check boxes and image buttons/buttons. Complex applications may crash since the current solution is not suited for them however a robust solution is planned by extending the current solution for such applications. This will include the necessary crash detection module. Another scenario in which an application crash may occur is when an application itself has a flaw. After all, the input generator is used to test an application to detect failures. The crash error is reported to the command line (or an IDE such as Eclipse if it is used to test the application) and the developer can make necessary changes to the application to patch the flaw and re-run the test scripts. Other crashes are related to the inherent problem of the Robotium framework where the application cannot be tested using Robotium since it cannot be re-signed.

B. Comparison of Existing Input Generators

Table I below summarizes the high level comparisons between Robotium, Monkey and Monkey Runner tool and encapsulates details of why it was used/not used to implement the automated input generator.

<table>
<thead>
<tr>
<th>Category</th>
<th>Proposed automated input generator</th>
<th>Monkey Runner</th>
<th>UI/Application Exerciser</th>
<th>Monkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input type</td>
<td>User defined inputs</td>
<td>User defined inputs</td>
<td>Random input generator</td>
<td></td>
</tr>
<tr>
<td>Complexity</td>
<td>Easy to use. Only requires knowledge of java to test an application</td>
<td>Relatively complex. Requires knowledge of Python to test an application</td>
<td>Easy to use. Pseudo random input generator started from the command line</td>
<td></td>
</tr>
<tr>
<td>Robustness</td>
<td>Application does not</td>
<td>Application does not tend to crash</td>
<td>Application may crash</td>
<td></td>
</tr>
</tbody>
</table>

TABLE I. EXISTING INPUT GENERATOR COMPARISON


<table>
<thead>
<tr>
<th>Category</th>
<th>Proposed automated input generator</th>
<th>Monkey Runner</th>
<th>UI/Application Exerciser Monkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td>tend to crash due to valid inputs specific to application</td>
<td>due to valid inputs specific to application</td>
<td>because of randomly generated inputs</td>
</tr>
<tr>
<td>Application source code</td>
<td>Works without the application source code</td>
<td>Works without the application source code</td>
<td>Works without the source code</td>
</tr>
<tr>
<td>Multiple applications support with a single script</td>
<td>An attempt to create a common test script across multiple applications is possible</td>
<td>Difficult to use the same test script, e.g., to click a button, the script may need its coordinates which is not the same for all applications</td>
<td>Possible, with the chance of an application crash if inputs are not valid</td>
</tr>
<tr>
<td>Used/not used in the project</td>
<td>Used since it is easy to use, only requires knowledge of java and exercises more control over the application</td>
<td>Not used since it is complex and requires knowledge of python as well as other issues described above</td>
<td>Not used since application may crash if inputs entered are not valid, e.g., an email address entered instead of a number</td>
</tr>
</tbody>
</table>

VI. CONCLUSION

We presented an automated input generator that generates valid inputs to simple mobile applications in the Android platform. Along with generating valid inputs, the system covers as much code as possible by traversing the user interface of the application in a depth first search approach. We also documented the requirements of an ideal input generator and developed the system by keeping these requirements in mind. We applied this approach to simple android applications and compared it on a high level detail with existing input generators such as Money and Monkey Runner tool. Finally, we conclude that generating valid inputs as opposed to random inputs is a much more efficient technique of testing Android applications.

REFERENCES