A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud

Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, Angelos Stavrou
Computer Science Department
George Mason University
{rmahmoo2, nesfaha2, tkacem, nmirzaei, smalek, astavrou}@gmu.edu

Abstract— **By changing the way software is delivered to end-users, markets for mobile apps create a false sense of security: apps are downloaded from a market that can potentially be regulated. In practice, this is far from truth and instead, there has been evidence that security is not one of the primary design tenets for the mobile app stores. Recent studies have indicated mobile markets are harboring apps that are either malicious or vulnerable leading to compromises of millions of devices. The key technical obstacle for the organizations overseeing these markets is the lack of practical and automated mechanisms to assess the security of mobile apps, given that thousands of apps are added and updated on a daily basis. In this paper, we provide an overview of a multi-faceted project targeted at automatically testing the security and robustness of Android apps in a scalable manner. We describe an Android-specific program analysis technique capable of generating a large number of test cases for fuzzing an app, as well as a test bed that given the generated test cases, executes them in parallel on numerous emulated Androids running on the cloud.**

Keywords-Android; Security Testing; Program Analysis

I. Introduction

Mobile App markets are creating a fundamental paradigm shift in the way software is delivered to the end users. The benefits of this software supply model are plenty, including the ability to rapidly and effectively acquire, introduce, maintain, and enhance software used by the consumers. By providing a medium for reaching a large consumer market at a nominal cost, app markets have leveled the software development field, allowing small entrepreneurs to compete head-to-head against prominent software development companies. The result of this has been an explosive growth in the number of new apps for platforms, such as Mac, Android, and iPhone, that have embraced this model.

This paradigm shift, however, has given rise to a new set of security challenges. In parallel with the emergence of app markets, we are witnessing an increase in the security threats targeted at platforms that have embraced this paradigm. This is nowhere more evident than in the *Android market*, where many cases of apps infected with malwares and spywares have been reported [1]. Numerous culprits are in play here, and some are not even technical, such as the general lack of an overseeing authority in the case of open markets and inconsequential implication to those caught provisioning applications with vulnerabilities or malicious capabilities.

From a technical standpoint, however, the key obstacle is the lack of practical techniques to rapidly assess and test the security of applications submitted to the market. Security testing is generally a manual, expensive, and cumbersome process. This is precisely the challenge that we have begun to address through the development of a framework that aids the analysts in testing the security of Android apps. The framework is comprised of a tool-suite that given an app automatically generates and executes numerous test cases, and provides a report of uncovered security vulnerabilities to the human analyst. We have focused our research on Android as (1) it provides one of the most widely-used and at the same time vulnerable app markets, (2) it dominates the smartphone consumer market, and (3) it is open-source, lending itself naturally for experimentation in the laboratory.

Security testing is a notoriously difficult task. This is partly because unlike functional testing that aims to show a software system complies with its specification, security testing is a form of *negative testing*, i.e., showing that a certain (often apriori unknown) behavior does not exist.

A form of security testing that does not require test case specification or significant upfront effort is *fuzz testing,* or simply *fuzzing* [2]. In short, fuzzing is a form of negative testing that feeds malformed and unexpected input data to a program with the objective of revealing security vulnerabilities. Programs that are used to create and examine fuzz tests are called *fuzzers*. Fuzzers have been employed in the past by the hacking community as one of the predominant ways of breaking into a system [2]. For instance, an SMS protocol fuzzer [3] was recently shown to be highly effective in finding severe security vulnerabilities in all three major smartphone platforms. In the case of Android, fuzzing found a security vulnerability triggered by simply receiving a particular type of SMS message, which not only kills the phone's telephony process, but also kicks the target device off the network [3].

Despite the individual success of fuzzing as a general method of identifying vulnerabilities, fuzzing has traditionally been used as a brute-force mechanism. There has been a lack of sophisticated or guided techniques for fuzz testing apps, in particular those targeted at smartphone platforms. Using fuzzing for testing is generally a time consuming and computationally expensive process, as the space of possible inputs to any real-world program is often unbounded. Existing fuzzing tools, such as Android's Monkey [4], generate purely random test case inputs, and thus are often ineffective in practice.

In this paper, we are addressing these limitations by developing a scalable approach for intelligent fuzz testing of Android applications. Our approach *scales* in terms of code size and number of test cases. We achieve that by leveraging the unprecedented computational power of cloud computing. The framework employes numerous heuristics and software analysis techniques to *intelligently* guide the generation of test cases aiming to boost the likelihood of discovering vulnerabilities. The proposed testing mechanisms empower the broader app market community to harness the immense computational power of cloud together with novel automated testing techniques to quickly, accurately, and cheaply find security vulnerabilities.

This paper is organized as follows. Section II provides a background on Android and its security model. Section III outlines an Android app that is used for illustrating the research. Section IV provides an overview of our approach, while Sections V to IX provide the details. The paper concludes with an overview of the related research in Section X and a discussion of our future work in Section XI.

II. Background

In 2008, Google and Open Handset Alliance launched Android Platform for mobile devices. Android is a comprehensive software framework for mobile communication devices including smart-phones and PDAs.

A. *Android Architecture and its Security Model*

The Android framework includes a full Linux operating system based on the ARM processor, system libraries, middleware, and a suite of pre-installed applications. Google Android platform is based on *Dalvik Virtual Machine (DVM)* [5] for executing and containing programs written in Java. Android also comes with an *application framework*, which provides a platform for application development and includes services for building GUI applications, data access, and other component types. The framework is designed to simplify the reuse and integration of components. Applications publish their capabilities and others can use them, subject to security constraints described further below.

Android enforces its application security mechanisms at two levels. The first level of security is achieved by forcing each application to execute within its own *secure sandbox,* which sets Android apart from other operating systems present in the market. Thus, an instance of an application is isolated from other applications in the memory.

A second level of security enforcement is achieved through Android's permission based security model. Android uses Mandatory Access Control to regulate access to applications and phone resources based on access permission policies. These policies are implemented in the form of permission labels that are assigned to components and applications. The permissions granted to each application are defined in its mandatory *manifest* file. The *manifest* file values are bound to the application at compile time and cannot be changed afterwards unless the application is recompiled.

Android's security mechanisms have been breached numerous times in the past [1]. In general, once the user installs an application infected with a malware, not much protection can be achieved through Android's standard security mechanisms. Since the access permissions in Android are coarse-grained (i.e., high-level all or nothing permissions), a malware embedded in an application can use all of the access permissions granted to the host application. A malicious program can also quickly exhaust, or expose to remote attacks, important system resources. At the same time, unlike desktop computing, it is hard to employ a large, resource intensive program (e.g., Antivirus) to detect, monitor, and control malicious software. Finally, malwares and attackers often leverage bad implementation choices and unintentional bugs to realize their objectives [1].

B. *Android Application Building Blocks*

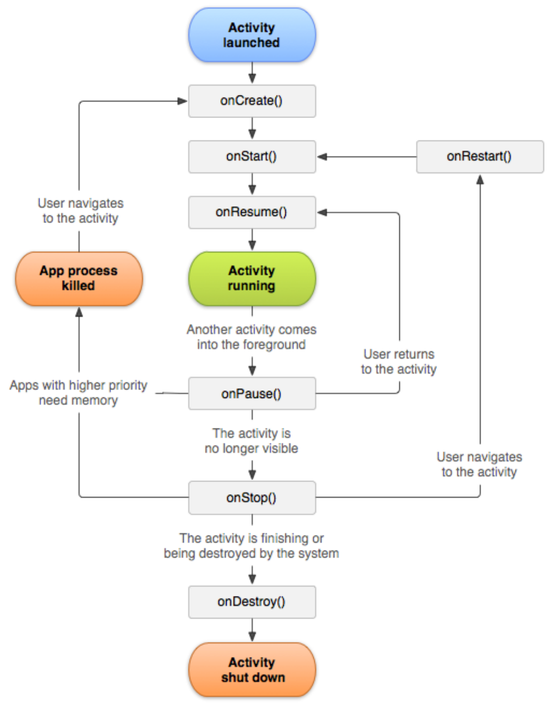
As mentioned earlier, each Android application has a mandatory *manifest* file. This is a required XML file for every application and provides essential information for managing the life cycle of an application to the Android platform. Examples of the kinds of information included in a manifest file are descriptions of the application's *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers* among other architectural and configuration properties.

An *Activity* is a screen that is presented to the user and contains a set of layouts (e.g., *LinearLayout* that organizes items within the screen horizontally or vertically). The layouts contain GUI controls, known as *view widgets* (e.g., *TextView* for viewing text and *EditText* for text inputs). The layouts and its controls are usually described in a configuration XML file with each layout and control having a unique identifier. A *Service* is a component that runs in the background and performs long running tasks, such as playing music. Unlike an *Activity*, a *Service* does not present the user with a screen for interaction. A *Content Provider* manages structured data stored on the file system or database, such as contact information. A *Broadcast Receiver* responds to system wide announcement messages, such as the screen has turned off or the battery is low. *Activities*, *Services*, and *Broadcast Receivers* are activated via *Intent* messages. An *Intent* message is an event for an action to be performed along with the data that supports that action. *Intent* messaging allows for late run-time binding between components, where the calls are *not explicit* in the code, rather connected through event messaging.

Activity and *Service* are required to follow prespecified lifecycles [6]. For instance, Figure 1 shows the events in the lifecycle of an *Activity*: *onCreate()*, *onStart()*, *onResume()*, *onPause()*, *onStop()*, *onRestart()*, and *onDestroy().* These lifecycle events play an important role in our research as explained later.

In addition to these components, a typical application utilizes many resources. These resources include animation files, graphics files, layout files, menu files, string constants, styles for user interface controls. Most of these are described using XML files. An example, as mentioned before are layouts. The layout XML files define the architecture of user interface controls that are used by *Activities*. The resources each have a unique identifier that is used to distinguish and get a reference to them in the application code.

III. Illustrative Example



Activity launched
onCreate()
onStart()
onRestart()
onResume()
User navigates to the activity
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 1. Lifecycle of *Activity* in Android from [6].

We use a subset of a software system, called Emergency Deployment System (EDS) [7], to illustrate our research. EDS is a system previously developed in collaboration with a government agency for the deployment of personnel in emergency response scenarios. EDS is intended to allow a search and rescue crew to share and obtain an assessment of the situation in real-time (e.g., interactive overlay on maps), coordinate with one another (e.g., send reports, chat, and share video streams), and engage the headquarters (e.g., request resources).

EDS has several interrelated apps. One of them is a *Driving Direction* app that can be used to calculate off-road driving directions between two geographic points, while considering objectives such as distance, time, and safety. Figure 3a depicts the GUI for this app. This screen has four input text boxes and three buttons. The input boxes are for latitude/longitude pair and the buttons for alternative ways of computing the directions. The latitude/longitude coordinates can be typed in or selected from a map. The resulting turn-by-turn directions are shown in a separate text box, and optionally displayed on a map.

IV. Approach Overview

Figure 2 illustrates an overview of our approach. The parts of the approach depicted within a dotted bubble run on a cloud platform to allow for the execution of large number of test cases on many instances of a given application.

The input to our framework is an *Android application package file (APK)*. APKs are Java bytecode packages used to distribute and install Android apps. If the source code is not readily available, we first reverse engineer the APK file using one of the available tools for this purpose (e.g., *apktool* [8], *dex2jar* [9], *smali* [10], and *dedexer* [11]). We then leverage *JD-GUI* [12] for decompiling the Java class files to obtain the source files.

The first step is to discover the app's *Input Surface*, which corresponds to all the ways in which an application can be initiated or accessed. We use MoDisco [13] to parse the source code we obtained directly or via decompilation of the bytecode. In addition, we process the resources and configuration information including the *manifest* file. From the generated data, we automatically construct two models of the app: *Call Graph Model* and *Architectural Model*. We base our analysis on these two models. The *Call Graph Model* represents all possible method invocation sequences (execution traces) within an application. The *Architectural Model* represents the application's architecture and user interface layout constructed from the meta-data associated with Android apps (i.e., recall *manifest* and *layout* XML *files* from Section II).

The *Test Case Generator* uses these models together with the Android specifications (for GUI controls, widgets, APIs, etc.) and template Android test case skeletons to generate test cases. A test case template is a skeleton Java file that contains the common static portions of a test case. For example, the *JUnit* methods such as *setUp()* and *tearDown()* come standard as part of the template.

Following the generation of test cases, the *Test Execution Environment* is activated to simultaneously execute the tests on numerous instances of the same application. For scalability, we harness the parallelism of a cloud-based system to execute the tests on virtual nodes running the *Android Emulator.* In addition to code coverage, several other Android-specific *Monitoring Facilities* such as *Intent Sniffer* [14] are instantiated and deployed to collect runtime data as tests execute. These monitoring facilities record program behavior and errors (e.g., crashes, exceptions, access violations, resource thrashing) that arise during the testing in the *Output Repository*.

The *Exception Analyzer* engine then investigates the

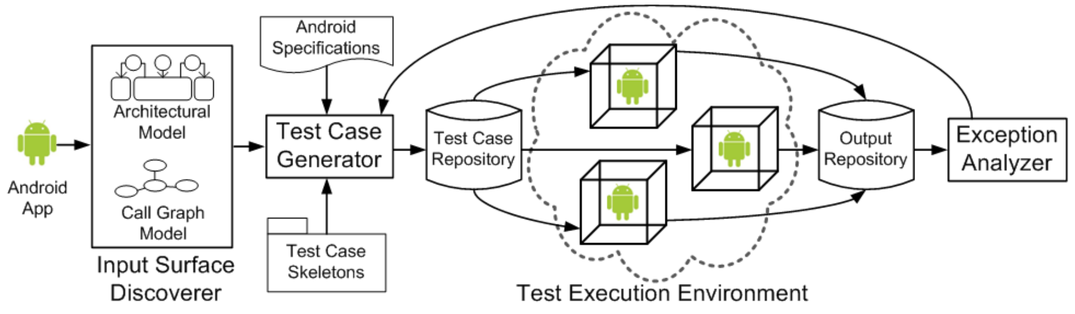


Figure 2. Overview of the approach. Components contained in the dotted bubble execute in parallel on the cloud.

Output Repository to correlate the executed tests cases to the reported issues, and thus potential security vulnerabilities. Moreover, the *Exception Analyzer* engine prunes the collected data to filter any redundancy, since the same vulnerability may be encountered by multiple test cases. It also looks for anomalous behavior, such as performance degradations, which may also indicate vulnerabilities (e.g., an input that could instigate a denial of service attack).

V. MODELS OF APP

As discussed in Section IV, our approach leverages two types of models for generating the test cases. Figure 3 depicts an example of the models that are automatically extracted for the Driving Direction app. The *Architectural Model*, partially depicted in Figure 3b, is generated by combining and correlating information containted in the configuration files and meta-data included in Android APK (i.e., *manifest* and layout XML files). Essentially, this model represents the app's architecture extracted from its configuration and resource files.

In Figure 3b, we use stereotypes to classify the different components types. *Activity* and *Service* are shown in orange rectangles, whereas *Intent* is shown in dark orange circles. Resources such as *Layout*, *Value*, and *Drawable* are shown in light orange rectangles. An example in each is shown using a triple tuple {*Classification*, *Type*, *Instance*}. For example, a *Layout* could be an *EditText* control with identifier *latOneId*. All of the resources used by an *Activity* or *Service* are contained within it. The *Intent* messaging between *Activities* and *Services* are represented by the dotted lines with the arrowhead showing the direction. For example, the Driving Directions Activity sends an *Intent* called *SOLVE* to the Route Solver Service.

Initially, when the *Architectural Model* is built, the associations between *Layouts* and *Activities* are not known since they are set in code and are not present in the configuration files. To extract these associations, we update this model as described in the next section to associate the *Layouts* of an *Activity*. Moreover, we parse the app's source code using MoDisco [13] and extract all of the method invocation sequences. We use this information to derive the app's *Call Graph Model* as shown in Figure 3c. The *Call Graph Model* contains a set of call trees showing the different possible invocation sequences within a given application. Each yellow box in Figure 3c represents a method, and the lines represent the sequence of invocations. In *Tree 1*, the *DirectionsActivity*'s *onCreate()* method is called by the Android system, and it sets the layout of the *Activity* using a unique identifier reference. Then it finds each button using the button's identifiers and attaches a click listener for the button. *Tree 2* begins on the event of a button click. It calculates the appropriate driving directions and sets the output text. The link between *Tree 1* and *Tree 2* is implicit, and hence, shown as a dotted arrow. Initially this link is not present in the model; it is updated as described in the next section.

Notice that the controls within the application are referenced across the models in Figure 3. For example, "*From Lat*" input box in Figure 3a is listed as a layout control with *latOneId* identifier in Figure 3b and accessed using this id in *Tree 2* of Figure 3c.

VI. DISCOVERING THE INPUT SURFACE

In order to automatically perform input fuzzing of an app, we need to discover its input surface. We have developed a technique for automatically identifying the ways in which an Android app can be engaged as described next.

First, we identify the ways in which an app can be started. From the Android specification, we know that the main *Activity* launches when an app starts. We resolve this using our *Architectural Model*, which shows the *Intent* messages that each *Activity* is interested in receiving and responding. As seen in Figure 3b, in the case of Driving Direction app, the main *Activity* is *DirectionsActivity* that handles the *MAIN Intent*. Android specification also tells us that *onCreate()* method of the main *Activity* is the starting point in the *Call Graph Model* (see Tree 1 in Figure 3c).

Second, we need to determine how to navigate within the app to determine all the ways in which it can receive GUI inputs, as well as how it resumes, receives system notifications, and starts a *Service*. Unlike *Activity*, which only accepts GUI inputs, *Services* may receive inputs from other sources, which are also part of the input surface.

To resolve these, we use the *Call Graph Model* described in the previous section. The root node of each tree is a method call that no other part of the application logic

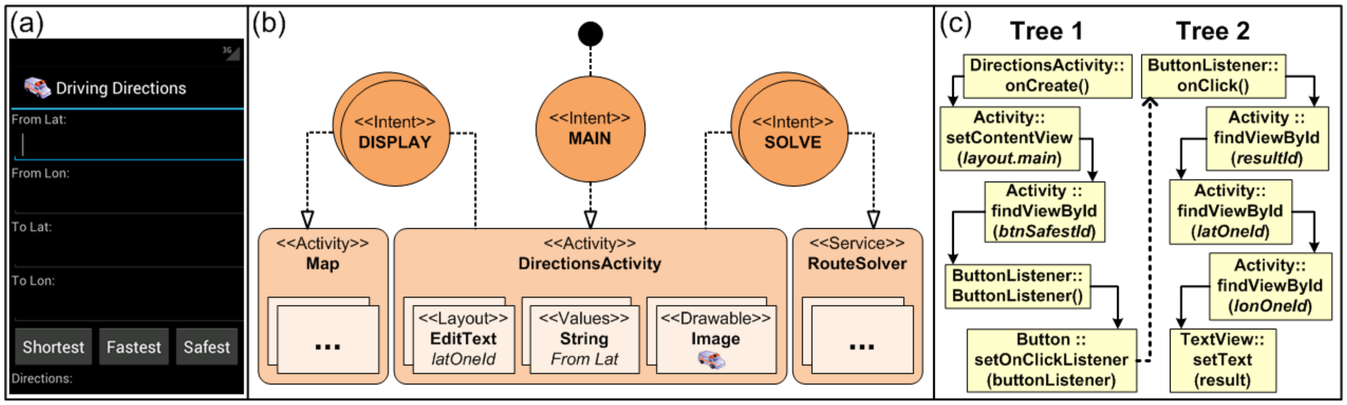

(a)
Driving Directions
From Lat:
From Lon:
To Lat:
To Lon:
Shortest
Fastest
Safest
Directions:
(b)
<<Intent>> DISPLAY
<<Intent>> MAIN
<<Intent>> SOLVE
<<Activity>> Map
...
<<Activity>> DirectionsActivity
<<Layout>> EditText latOneId
<<Values>> String From Lat
<<Drawable>> Image
<<Service>> RouteSolver
...
(c)
Tree 1
Tree 2
DirectionsActivity:: onCreate()
Activity:: setContentView (layout.main)
Activity :: findViewById (btnSafestId)
ButtonListener:: ButtonListener()
Button :: setOnClickListener (buttonListener)
ButtonListener:: onClick()
Activity :: findViewById (resultId)
Activity:: findViewById (latOneId)
Activity:: findViewById (lonOneId)
TextView:: setText (result)

Figure 3. Driving direction app: (a) screen shot of the app, (b) subset of the Architectural Model, and (c) subset of the Call Graph Model.

explicitly calls. Recall from Figure 1 that the lifecycle methods are called by the Android framework only. When these lifecycle methods are overridden in an app's implementation, they form the root nodes of that app's *Call Graph Model*. Similarly, the event methods of a *Service*, *onCreate()* and *onBind()* for example, would be root nodes. Some of these root nodes are the initiating points, where input may be supplied to the app from within or outside.

Additionally, the controls on an *Activity* have handlers for their events. For example, a *Button* often has a click event associated with it. This event is handled by a class that implements the *OnClickListener* interface and overrides the *onClick()* method. We expect these sorts of handlers to be in the root nodes of our call trees as well, since Android is event driven and the event handlers are called by the Android system as opposed to the application logic. For instance, as depicted in Figure 3c, we see that *Tree 1*'s root is the *onCreate()* event handler, and *Tree 2*'s root node is the *onClick()* event handler.

From this point, we need to identify two additional attributes. We need to resolve what layout/view is being used by each *Activity,* since there can be many *Activities* with many layout XML files and their respective controls. Secondly, we also need to be able to link the implicit calls between the trees (recall the dotted arrow in Figure 3c).

In order to resolve this information, we traverse the call graph starting with the *onCreate()* root node of the main *Activity* and look for what layout is used. From the Android specification, we know this is achieved by calling the *setContentView()* method and passing it a reference identifier that describes the layout and controls. For instance, in the Driving Direction app, the view is set to the *Layout* with identifier *layout.main* (Figure 3c). Since this was set in the *DirectionsActivity*, we associate this *Activity* with this *Layout* in the *Architectural Model*. This means that the layout and controls identified in this way are those that get rendered on the screen when the corresponding *Activity* runs.

Finally, we continue down the graph and identify implicit method calls in order to link the different trees. We know that the links would have to be to other root nodes of trees, and achieved through setting event handlers. For example,

```
public class DirectionsTest extends
   ActivityInstrumentationTestCase2<DirectionsActivity>
{
 private Solo solo;

 public DirectionsTest() {
   super("edu.gmu.android", DirectionsActivity.class);}

 public void setUp() throws Exception {
   solo = new Solo(getInstrumentation(), getActivity());}

 public void testDirections() throws Exception {
   solo.enterText(0, "38.95");
   solo.enterText(1, "77.46");
   solo.enterText(2, "38.95");
   solo.enterText(3, "77.46");
   solo.clickOnButton("Safest");   }

 public void tearDown() throws Exception {
   solo.finishOpenedActivities();   }

}
```

Listing 1. Sample auto generated test case.

system event handlers that handle notification events, such as when a call is received, network is disconnected, or the battery is running low. As trees are linked and connected, we traverse them in a similar fashion. By doing so, we are able to connect the entire call graph of the application, from beginning to end. Both the *Architectural Model* and the *Call Graph Model* are updated with the newly found information. Using this algorithm we can discover all the ways an app can be initiated, as well as the inputs it can receive.

VII. Test Case Generation

Now that the controls, their input value domain, the events, and their handlers have been inferred, the next step is to generate test cases for execution. We take a test case template, substitute our inferred information, and output the result as a Java file. Most of the generated text in the file is from the template, while the dynamic parts are replaced with the inferred information. For example, the class name is replaced with activity under test (e.g., *DirectionsActivity*), and the inputs are also generated as follows.

For the test cases, our goal is to obtain sufficient code coverage, while generating inputs with security implications, e.g., an input that makes an application unavailable or violate access permissions. We attempt to generate our inputs so that we start with the first *Activity's* root node and traverse the tree, including any implicit connections, in a recursive manner. We monitor our code coverage by using *EMMA* [15], an open source toolkit that monitors and reports Java code coverage. By comparing the stack-trace reports generated by *EMMA* with the *Call Graph Model*, we can obtain an accurate assessment of code coverage.

We iteratively employ various *fuzzers* in order to generate and improve the inputs. The initial input generation for each *fuzzer* is based on using the Android specifications for each control and includes commonly employed rules, such as boundary values, very small or large values, special characters, empty values, etc. The valid input domain for an interface is derived by checking the specifications of a control. For example, we can tell whether the input domain of a text box is numerical, text, etc by referencing its specification. Based on the input domain, we employ the correct *fuzzer*. For instance, a *fuzzer* for numerical inputs starts off with negative numbers, zero, large values, and so on. In the case of text inputs, the text *fuzzer* generates null values, special characters, UTF characters, etc.

We refine the inputs by using an iterative strategy, where we run test cases in iterations. We assess the depth of the call graph that a test case was able to penetrate. In the next iteration, we revise the inputs in one direction (lower/higher or negative/positive) and repeat the process. This way we are able to observe if we are obtaining coverage in the parts of the code that have not been tested.

As part of our ongoing activity, we are developing more sophisticated input generation algorithms, which are further discussed in Section XI. But even with the current approach we have been very successful at generating a very large number of test cases, achieving substantial code coverage, and detecting real vulnerabilities.

These test cases are stored inside of a test case repository database as shown in Figure 2. Currently, the generated test cases leverage the *Robotium* [16] framework. It is a testing framework built on Android's *JUnit* testing suite that facilitates automated testing.

As an example, Listing 1 shows one of the many *Robotium* test cases automatically generated using our approach for the Driving Direction app. The name of the test case is *DirectionsTest* and it extends *ActivityInstrumentationTestCase2*, an *Android Activity* testing class supplied by the Android testing framework, which in turns extends from JUnit's *TestCase* class. JUnit's *setup()* and *tearDown()* methods are overridden to set up and finalize the *Activities* being tested. The test case has the *Robotium's Solo* object that is used to interact with the application, such as sending inputs. Essentially Solo mimics the human user of the app. Any method that starts with the word "*test*" is run when the test case executes. The generated *testDirections()* method uses the *Solo* instance to enter four numbers (two pairs of lat/long) in the input text boxes and then clicks on the *Safest* button. The input text is entered by using the index of the input field.

In this test, note that both pairs of lat/long are set to the same value. In fact, this particular input along with the *Safest* combination exposed a bug in the routing algorithm: a divide by zero exception. The result of running this test case on the emulator is shown in Figure 4a. The integer divide by zero exception makes the application unavailable, thus making this test case a success for exposing a potential vulnerability.

VIII. Test Execution Environment

Fuzzing usually requires the execution of a large number of tests. This is challenging on both traditional desktops as well smartphones due to the limitation of resources and the length of time it takes to execute a large set of test cases. To mitigate this issue, we have developed a novel technique to execute the tests in *parallel* and on the cloud. This allows us to seamlessly scale up and down as necessary.

We have set up an instance of Amazon EC2 virtual server running Windows Server 2008, and configured it with Java SDK, Android SDK, Android Virtual Device, and a custom test execution manager engine developed by us. The execution manager is responsible for polling the test repository and running the test cases on its host environment. For each test, it launches the emulator, installs the app, and installs and executes the test. It is also responsible for persisting all results, along with log and monitor data to the output repository. We created a virtual machine *image* from our base machine configured in this way to be replicated on demand.

In order to execute the tests cases in parallel, we launch a set number of virtual node instances, built using our image template and using Amazon's EC2 API. The execution manager on each instance polls our test case repository database and executes the tests. As an example, a batch execution output of Listing 1 is shown in Figure 4b.

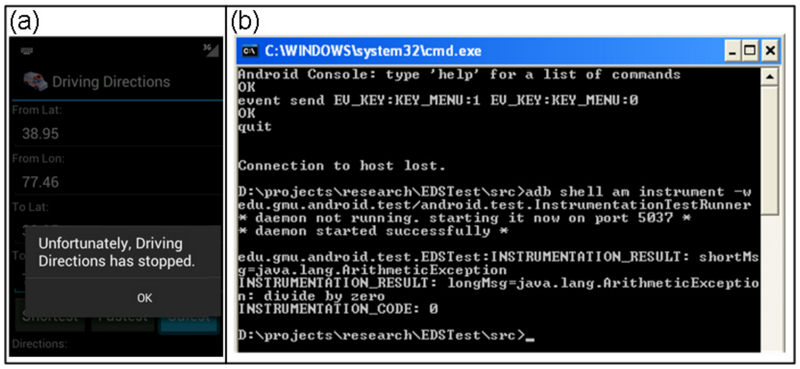
Using the above setup, we were able to run 1,000 test cases for the suite of apps shipped with EDS in less than *25 minutes* by using 100 parallel instances (already running) each processing 10 test cases. The same test cases took over *77 hours* to complete on a single workstation executing the tests sequentially. Note that the reported times are not just the execution time of tests, but also the time associated with loading the Android emulator, test setup, and clean up.

IX. Test Case Results Analysis

Currently, we categorize the observed/logged output information into *Interface*, *Interaction*, *Permissions*, and *Resources* types. *Interface* exceptions are caused by direct inputs at the surface, usually to GUI controls. *Interactions* exceptions are a result of communication failures with other components within the application or with external applications. *Permissions* exceptions are a result of access violations, such as the application attempting to access components or system APIs that it explicitly has not requested. *Resources* exceptions are caused by abnormal system usage causing resource depletion, unavailability, or certain operation to time out.

Furthermore, we collect various attributes for each exception such as the exact Java exception type and the number of such exceptions, whether it was checked or unchecked, and the Android system APIs that threw the exception. Example APIs are *Networking*, *Telephony*, *Internet*, *Media,* etc. We correlate exceptions information with the method, class, and package that contained the exception and the frequency with which the respective code block was exercised during the testing. This helps us identify and cluster the most defective components of the application, which could potentially aid with bug fix prioritization.

We also analyze the code coverage by checking method invocations. A lack of depth in tree traversal or being unable to go beyond surface trees indicate that input is being filtered, application is branching early in different directions, or that only a certain input range is allowed. For example, in the Driving Direction app, we noticed that only a subset of test cases were able to penetrate deep into *Tree 2* in Figure 3c. This is because of input validation at the beginning of *Tree 2* that prevents the execution from going further unless valid latitude and longitude coordinates are entered. Since we knew all the nodes in *Tree 2*, we revised the inputs in the


(a)
Driving Directions
From Lat:
38.95
From Lon:
77.46
To Lat:
Unfortunately, Driving Directions has stopped.
OK
Directions:
(b)
C:\WINDOWS\system32\cmd.exe

```
Android Console: type 'help' for a list of commands
OK
event send EV_KEY:KEY_MENU:1 EV_KEY:KEY_MENU:0
OK
quit


Connection to host lost.

D:\projects\research\EDSTest\src>adb shell am instrument -w
edu.gmu.android.test/android.test.InstrumentationTestRunner
* daemon not running. starting it now on port 5037 *
* daemon started successfully *

edu.gmu.android.test.EDSTest:INSTRUMENTATION_RESULT: shortMs
g=java.lang.ArithmeticException
INSTRUMENTATION_RESULT: longMsg=java.lang.ArithmeticExceptio
n: divide by zero
INSTRUMENTATION_CODE: 0

D:\projects\research\EDSTest\src>_
```

Figure 4. Sample Test Case Result: (a) Manual (b) Batch

subsequent iterations, and honed in on the inputs that could penetrate into Tree 2, and eventually all the way to the end of it. The iterative strategy helps us with obtaining greater code coverage, while at the same time be able to reason about valid input ranges for an app. This knowledge also helps us with finding inputs that are *outside* of the valid range, thus enabling negative test cases as well.

X. Related Work

The Android development environment ships with a powerful testing framework [17] that is built on top of *JUnit*. *Robolectric* [18] is another framework that separates the test cases from the device or emulator and provides the ability to run them directly by referencing their library files. While these frameworks automate the execution of the tests, the test cases themselves still have to be written by the engineers.

Traditionally *fuzz* testing tools use random inputs, but modern approaches utilize grammars for representing mutations of possible inputs [19][20] or achieve white-box *fuzz* testing using symbolic execution and dynamic test generation [21]. *SPIKE*, *Peach*, *File-Fuzz*, *Autodaf´e* are examples of *fuzzers* that support some form of grammar representation. Applying exhaustive approaches are typically not feasible due to the path explosion problem.

Our research is related to the approaches described in [22][23] for testing Android apps. In [22], a crawling-based approach that leverages completely random inputs is proposed to generate unique test cases. [23] presents a random approach for generating GUI tests and uses the Android *Monkey* platform to execute them. We are leveraging reverse engineering techniques to obtain the app's implementation, and use program analysis to derive the test generation process. This sets us apart from these works that employ black-box testing techniques. Moreover, these approaches have neither targeted security issues, nor have they considered the scalability implications of their solutions.

There has been a recent interest in using cloud to validate and verify software. TaaS is an automated testing framework that automates software testing as a service on the cloud [24]. *Cloud9* provides a cloud-based symbolic execution engine [25]. Similarly, our framework is leveraging the computation power of cloud to scale fuzz testing. Unlike prior research, however, by targetting our framework to Android, we are able to achieve significant automation.

XI. Concluding Remarks

We have presented a novel framework for automated security testing of Android applications on the cloud. The key contributions of our work are (1) a fully automated test case generation, (2) iterative feedback loop to generate and guide our input in an intelligent manner that ensures code coverage and uncovers potential security defects, and (3) highly scalable *fuzzing* by leveraging the cloud.

In our on going work, we are exploring two approaches for improving the test case generation facet of our framework. First, we are developing an evolutionary algorithm for generating tests, as part of which we are modeling the problem of testing an Android app as a genetic problem and developing an appropriate fitness function to evaluate the quality of test cases. Second, we are developing an Android-specific symbolic execution engine for automatically generating test cases. We are extending *Java Pathfinder*, which is capable of symbolically executing pure Java code, to work on Android. In addition, we are creating a graphical reporting environment that would allow the security analyst to visually explore the results of the testing, and in particular obtain metrics (e.g., achieved code coverage, bugs per KSLOC) that could then be used for making decisions as to the overall security and robustness.

Acknowledgments

This research is supported by grant D11AP00282 from Defense Advanced Research Projects Agency.

References

[1] A. Shabtai, et al., "Google Android: A comprehensive security assessment," *Security & Privacy, IEEE*, 8(2), pp. 35–44, 2010.
[2] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for software security testing and quality assurance*. Artech House Publishers, 2008.
[3] C. Miller and C. Mulliner, "Fuzzing the Phone in your Phone," in *Black Hat Technical Security Conference USA*, 2009.
[4] "Android Monkey." [Online]. Available: http://developer.android.com/guide/developing/tools/monkey.html.
[5] "Dalvik - Code and documentation from Android's VM team." [Online]. Available: http://code.google.com/p/dalvik/.
[6] "Android Developers Guide." [Online]. Available: http://developer.android.com/guide/topics/fundamentals.html.
[7] S. Malek, et al., "A style-aware architectural middleware for resource-constrained, distributed systems," *Software Engineering, IEEE Transactions on*, vol. 31, no. 3, pp. 256–272, 2005.
[8] "Apktool." [Online]. Available: http://code.google.com/p/android-apktool/.
[9] "Dex2jar." [Online]. Available: http://code.google.com/p/dex2jar/.
[10] "Smali." [Online]. Available: http://code.google.com/p/smali/.
[11] "Dedexer." [Online]. Available: http://dedexer.sourceforge.net/.
[12] "JD-GUI." [Online]. Available: http://java.decompiler.free.fr/?q=jdgui.
[13] "MoDisco." [Online]. Available: http://www.eclipse.org/MoDisco/.
[14] "Intent Sniffer." [Online]. Available: http://www.isecpartners.com/mobile-security-tools/intent-sniffer.html.
[15] "EMMA." [Online]. Available: http://emma.sourceforge.net/.
[16] "Robotium." [Online]. Available: http://code.google.com/p/robotium/.
[17] "Android Testing Framework." [Online]. Available: http://developer.android.com/guide/topics/testing/index.html.
[18] "Robolectric." [Online]. Available: http://pivotal.github.com/robolectric/.
[19] K. Sen, D. Marinov, and G. Agha, *CUTE: A concolic unit testing engine for C*, vol. 30. ACM, 2005.
[20] P. Godefroid, et al., "Grammar-based whitebox fuzzing," in *ACM SIGPLAN Notices*, 2008, vol. 43, pp. 206–215.
[21] P. Godefroid, et al., "Automated whitebox fuzz testing," *Network and Distributed System Security Symposium*, 2008, vol. 9.
[22] D. Amalfitano, et al., "A GUI Crawling-Based Technique for Android Mobile Application Testing," in *Software Testing, Verification and Validation Workshops (ICSTW),* 2011, pp. 252–261.
[23] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceeding of the 6th international workshop on Automation of software test*, 2011, pp. 77–83.
[24] G. Candea, S. Bucur, and C. Zamfir, "Automated software testing as a service," *ACM symposium on Cloud computing*, 2010, pp. 155–160.
[25] L. Ciortea, et al., "Cloud9: A software testing service," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 5–10, 2010.