

# Testing Android Apps Through Symbolic Execution

Nariman Mirzaei<sup>\*</sup>, Sam Malek<sup>\*</sup>, Corina S. Păsăreanu<sup>†</sup>, Naeem Esfahani<sup>\*</sup>, Riyadh Mahmood<sup>\*</sup>

<sup>\*</sup> Department of Computer Science  
George Mason University

{nmirzaei, smalek, nesfaha2, rmahmoo2}@gmu.edu

<sup>†</sup> NASA Ames Research Center  
corina.s.pasareanu@nasa.gov

## ABSTRACT

There is a growing need for automated testing techniques aimed at Android apps. A critical challenge is the systematic generation of test cases. One method of systematically generating test cases for Java programs is symbolic execution. But applying symbolic execution tools, such as Symbolic Pathfinder (SPF), to generate test cases for Android apps is challenged by the fact that Android apps run on Dalvik Virtual Machine (DVM) instead of JVM. In addition, Android apps are event driven and susceptible to path-divergence due to their reliance on an application development framework. This paper provides an overview of a two-pronged approach to alleviate these issues. First, we have developed a model of Android libraries in Java Pathfinder (JPF) to enable execution of Android apps in a way that addresses the issues of incompatibility with JVM and path-divergence. Second, we have leveraged program analysis techniques to correlate events with their handlers for automatically generating Android-specific drivers that simulate all valid events.

## Keywords

Java Pathfinder, Symbolic Pathfinder, Android, Testing

## 1. INTRODUCTION

In 2008, Google and Open Handset Alliance launched Android Platform for mobile devices. Android is a comprehensive software framework for mobile communication devices including smartphones and PDAs. Android has had a meteoric rise since its inception partly due its vibrant app market that currently provisions over half a million apps, with thousands added and updated on a daily basis. Not surprisingly there is an increasing demand by developers, consumers, and market operators for automated testing techniques applicable to Android apps.

A promising automated testing technique is symbolic execution [12], a program-analysis technique that uses symbolic values, rather than actual values, as program inputs. It gathers the constraints on those values along each path of the program and with the help of a solver generates inputs for all reachable paths. Although Android apps are developed in Java, they introduce three challenges for symbolic execution tools targeted at Java, namely Symbolic PathFinder (SPF) [22].

First challenge is that Android apps depend on a proprietary set of libraries that are not available outside the device or emulator. Android code runs on *Dalvik Virtual Machine (DVM)* [6] instead of the traditional *Java Virtual Machine (JVM)*. Thus, Android apps are compiled into Dalvik byte-code rather than Java byte-code. For using SPF to symbolically executed Android apps, they need to be transformed into the corresponding Java byte code representation.

The second challenge is that Android programs' dependence on framework libraries makes them prone to path-divergence problem, and more so than traditional Java programs. In general, path-divergence problem may occur when a symbolic value flows

outside the context of the program that is being symbolically executed and to the context of the bounding framework or any external library. In Android, however, path-divergence is the norm, rather than the exception. A typical Android app is composed of multiple *Activities* and *Services*, playing the role of software components, which communicate extensively with one another using *Intents*, Android's messaging system. An *Intent* is used to carry over a value to another *Activity/Service* and as a result that value leaves the boundaries of the app and is passed through Android libraries before it is retrieved in the new *Activity/Service*.

Finally, Android is an event driven system and extracting program input values is highly dependent on user action most of the times, meaning that the symbolic execution engine has to wait for the user to interact with the system and tap on a button or initiate some other type of event for the program to continue the execution of a certain path. Furthermore, the system itself or a third application can initiate an event and cause the app to behave in a certain way. Such events are far more frequent in smartphone apps than traditional software systems, due to the context sensitive nature of smartphones.

Current techniques for dealing with traditional event based systems either use Capture-replay or model driven approaches. In Capture-replay approaches [1, 16, 17], user records her interaction sequences with the GUI, which are replayed in time of testing. Model driven techniques [15, 24] require user to provide a model of the software system's usages. Both Capture-replay and model driven approaches depend on manual effort, thus not very convenient, and prone to missing ways in which an app could be engaged that are not readily known. In fact, in the context of security testing, an app may intentionally incorporate hidden ways in which it can be engaged. There have also been efforts to extract directed graph models automatically by crawling the GUI [1, 16, 17], and use those graphs to generate test sequences, but again may fail to identify other ways in which a system can be engaged.

In this paper, we provide an overview of a multi-faceted approach to tackle these challenges. We describe an extension to SPF that provides stubs modeling Android libraries. These models enable us to compile Android apps on JVM, and run them on Java PathFinder (JPF) to address the first challenge, i.e., incompatibility of DVM and JVM. In addition, we provide the logic in certain stubs to simulate the behavior of Android library classes to address the problem of path-divergence. Finally, we leverage both our knowledge of Android specification and a specialized call-graph model of the app to correlate the events with their handlers [13]. Using this model we can automatically generate drivers for extracting the user input values and generating the sequence of valid events for exercising an app. Drivers address the last challenge by guiding the generation of event sequences aimed at simulating an actual user's behavior. The resulting extended version of SPF enables us to symbolically execute Android apps to generate test cases that achieve high code

coverage.

This paper is organized as follows. Section 2 provides the background on Symbolic PathFinder and Android. Section 3 presents an Android app that is used for illustrating the research. Section 4 outlines an overview of our approach. Section 5 provides a detailed view of how we model Android libraries, while Sections 6 provide the details of our approach in generating the drivers for Android apps. The paper concludes with an overview of the related research in Section 7, and a discussion of our future work in Section 8.

## 2. BACKGROUND

In this section we first provide a brief background on SPF, followed by a more detailed overview of Android framework.

### 2.1 Symbolic PathFinder

JPF [11] is a general purpose model checker for Java programs that uses its own Virtual Machine rather than traditional JVM. Symbolic Pathfinder (JPF-Symbc) is built on top of JPF as an extension implementing a non-standard interpretation of Java bytecode using a modified JPF JVM [5]. SPF analyzes Java bytecode and can handle mixed integer and real constraints, as well as complex mathematical constraints via heuristic solving. SPF can be used for test input generation and for finding counterexamples to safety properties [23]. We are extending SPF to model Android apps and leverage it to generate inputs and test cases for them.

### 2.2 Android

Android is a comprehensive software framework for mobile communication devices including smartphones and PDAs. 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 DVM for executing and containing programs written in Java. Android also comes with an *application development framework*, which provides an environment 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.

Each Android app has a mandatory *manifest* file. This is a required XML file for every app and provides essential information for managing the life cycle of an app in 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

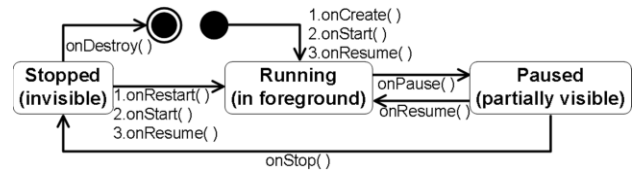


Figure 1. Lifecycle of Activity in Android.

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 pre-specified lifecycles [3]. 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 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.

## 3. ILLUSTRATIVE EXAMPLE

For illustrating the approach, we will use a *Driving Directions* app, a subset of a software system, called Emergency Deployment System (EDS) [14]. 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).

*Driving Directions* app can be used to calculate off-road driving directions between two geographic points, while considering cost objectives such as distance, time, and safety. It also provides the user with the closest dispatches to source, destination or on the route. Figure 2 depicts the GUI of this Android app. 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.

## 4. APPROACH OVERVIEW

The main objective of our tool is to automatically generate test inputs for Android apps using SPF. In order to do so, we first need to generate the Java byte-code of the app that can be executed on JVM. Hence, we have to compile the app's source code with Java compiler, instead of Android's Software Development Kit (SDK). This is achieved by first replacing platform-specific parts of the Android libraries that are needed for

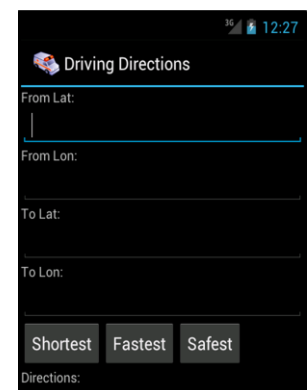


Figure 2. Screen shot of the *Driving Directions* app.

each app with our models. These models are essentially stubs that are created in a way that each component’s composition and callback behavior is preserved. This allows us to execute an Android app on JPF without modifying the app’s implementation.

In addition, we have to take into account the fact that a typical app is composed of multiple components (i.e., *Activities* and *Services*) that communicate using Android’s *Intent* messages. Path-divergence issue arises when an *Intent* is used to pass a symbolic value to another *Activity* or *Service*. To solve this problem, we have implemented the appropriate logic for simulating how Android platform mediates the event-based interaction of components, and incorporated that logic in the *Activity*, *Service*, and *Intent* stubs. We refer to these enhanced stubs as *mock* classes, as they emulate how the corresponding Android constructs behave. This way we are able to address the path-divergence problem caused by the basic mechanism in which Android components communicate with one another.

The second step in our approach is to generate drivers for an app using its call graph model. Unlike traditional Java programs, Android apps do not contain a *main* class that becomes the root node of the call graph, where the program is always initiated. Android apps are event driven, meaning that the thread of execution constantly changes context between the application logic, system, and user. Therefore, instead of a connected call graph that represents the complete control flow of the application, an Android app is composed of a set of disconnected sub-call graphs that collectively represent the app’s logic. These sub-call graphs correspond to all the ways in which an app can be initiated, accessed by the user or Android platform. Figure 3 illustrates a hypothetical call graph model for an Android app, where *A*, *B* and *C* are each a sub-call graph with root nodes *a*, *b*, and *c*, while the black circle represents the start of the app.

Following the generation of the sub-call graphs, we parse the source code, the resources and configuration information, including the *manifest* file, to find the event handlers. This allows us to automatically connect sub-call graphs in order to construct the call graph model of the system. Figure 3 illustrates a subset of the *Driving Directions* app’s call graph model. The dotted arrows that connect two sub-graphs illustrate the initiation of an event either by the user or the system. The call graph model represents all possible method invocation sequences (execution traces) within an app, which is depicted by black arrows in the figure.

Finally, we use the call graph model to derive a *Context Free Grammar (CFG)* that is used to generate drivers. A driver is a sequence of events that simulate user’s interaction with the app.

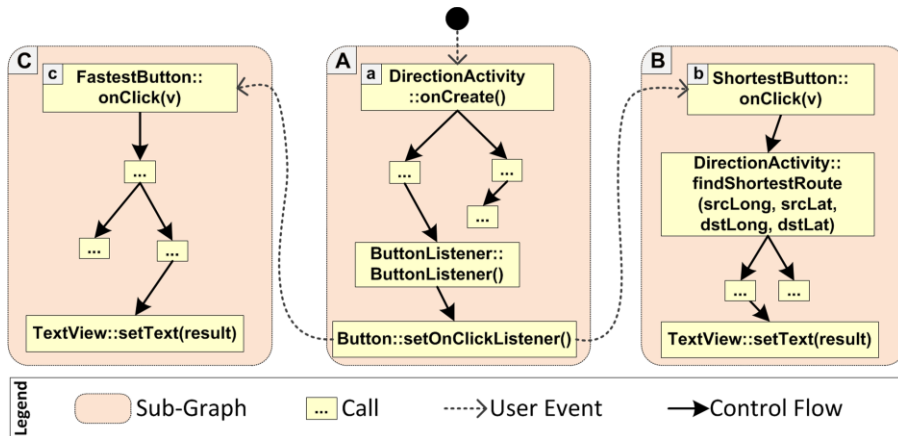


Figure 3. a subset of the call graph model for *Driving Directions* app.

Since the root node of each sub-graph represents an event being handled, every valid combination of the root nodes represents a scenario of user interaction with the app. For instance, based on the call graph model in Figure 3, {a,c} and {a,c,b} are two possible event sequences.

We monitor the code coverage from the execution of tests using *EMMA* [7], an open source toolkit that monitors and reports Java code coverage. We start with a single event driver and continue generating drivers with more event sequences iteratively. The testing stops when we hit a pre-specified code coverage threshold. We describe the approach in more detail in the next two sections.

## 5. MODELING ANDROID LIBRARIES

The conventional technique to tackle the path-divergence problem, which occurs when a Java program depends on external libraries, is to provide stubs for those libraries. We explored three possible techniques for developing stubs for supporting Android apps as describe below.

The first and probably the most straightforward approach for us would have been to leverage the implementation of Android’s library classes. Android platform provides a set of library classes that apps use to access the resources on the phone. These library classes contain native methods and Java Native Interface (JNI) calls that are platform dependent and cannot be executed outside of an actual phone. To support the development activity, Google provides *android.jar* with Android’s Software Development Kit, which allows developers to resolve the dependencies and compile apps in the development environment. At first blush, it may seem that this jar file could be used to execute apps on top of JPF. However, upon further inspection, we found that Google has stripped the library classes in this jar file, and replaced all the method bodies to throw an exception.

A second approach we explored was similar to that employed in Robolectric [20], a framework for running Android unit tests outside of Android emulator and on top of JVM. In this approach, one would use *shadow* classes, which are mock classes that are accessed through reflection. Shadow classes simulate the behavior of the actual Android library classes, and override the calls to them through reflection. This approach may be useful for running test cases on JVM, but not feasible for use with JPF, because currently JPF is not capable of handling reflection.

Finally, a third approach, and the one that we have adopted in our implementation is to provide our own custom built *stub* and *mock* classes. The stub classes are used to compile Android apps into JVM byte-code, while mock classes are used to deal with the

path-divergence problem. We developed stubs that return random values, when the return type of a method is primitive, and return empty instances of the object, when the return type is a complex data type.

Dealing with Android platform, not only we need to provide stub classes to resolve the byte-code incompatibility with JVM, but we also need to address the lack of Android logic outside the phone environment. Android uses its library classes as bolts and nuts that connect the different pieces of an app together. For example, and as shown in Listing 1, in *Driving Directions* app, *DirectionsActivity* uses the *startActivity*

method of the Android library class *Activity.java* to start the app's *DisplayRouteActivity* that displays the shortest route. It creates an *Intent* in which the source and destination activities along with the values to be carried are specified. In this case we provide the appropriate logic for *startActivity* mock, such that when a new instance of *DisplayRouteActivity* is created the control flow moves to its *onCreate* method.

Moreover, we create a mock for the *Intent.class* to address the path-divergence problem. As shown in Listing 1, an instance of *Intent* is passed to *startActivity*. This *Intent* encapsulates symbolic values of *sourceLong*, *sourceLat*, *destLong* and *destLat*, and causes path-divergence. To deal with this issue, we provided our own implementation of *putExtra* and *getExtra* methods in the mock implementation of *Intent.java*, such that the symbolic value of those variables is preserved. Android uses a *HashMap<String, Object>* to store and retrieve the values stored in an *Intent*, making it difficult to reason about a value stored as *Object* symbolically. To solve this problem, we provide our model of a hash map that holds primitive values. Consequently, in our implementation of the *putExtra* and *getExtra* methods we use our own model of hash map to enable JPF to symbolically reason about values that are exchanged using the *Intent* messages.

## 6. GENERATING DRIVERS

The second part of our approach is a technique for generation of drivers that simulate different ways in which an Android app can be engaged. We do that by generating models of app behavior. We parse the app's source code using MoDisco [19] and extract the app's call graph model as shown in Figure 3. 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 3 represents a method, and the lines represent the sequence of invocations. The link between *sub-graph A* and *sub-graph B* is implicit, and hence, shown as a dotted arrow. Initially, implicit links are not present in the model, as MoDisco cannot retrieve the invocation sequences due to the handling of *Intent* messages. Below we describe how these implicit links are determined in our approach.

To generate the drivers, we need to be able to navigate within the app to determine all the ways in which it receives user inputs, system notifications, starts/stops/resumes activities and services, interacts using *Intents*, etc. Note that unlike *Activity*, which only accepts GUI inputs, *Services* may receive inputs from other sources (e.g., system), which are also part of the input surface. We observe the root node of each tree is a method call that no other part of the application logic *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 also 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. In the case of *Driving Directions*, we see that *sub-graph A*'s root is the *onCreate()*

```
public class DirectionsActivity extends Activity {
    ...
    public String findShortestRoute(double sourceLong,
        double sourceLat, double destLong, double destLat){
        ...
        Intent intent = new Intent(this, DisplayRouteActivity.class);
        intent.putExtra("value", sourceLong);
        ...
        startActivity(intent);
    }
}
```

Listing 1. Code snippet from *DirectionsActivity.java*

event handler, and *sub-graph B*'s root node is the *onClick()* event handler (Figure 3).

In order to resolve all of the implicit links in the app, we traverse the call graph starting with the *onCreate()* root node of the main *Activity* (the starting point of the app). We continue down the graph and identify implicit method calls in order to link the different sub-graphs. We know that the links would have to be to other root nodes of trees, and achieved through setting event handlers, starting other activities, sending *Intent* messages, and handling system events. System event handlers deal with notification events, such as when a call is received, network is disconnected, or the battery is running low.

As each new sub-graph is 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. The call graph model is updated with the newly found information. Using the call graph model we can determine all the valid sequences in which an app can be engaged. This is critical, because one of the shortcomings of symbolic execution is the fact that it is not possible to reason about sequences of actions/events symbolically [10].

The last step is to use the call graph model and create a *Context Free Grammar (CFG)* that generates all possible sequences of events. We take all the root nodes in our sub-call graphs to be the non-terminals (i.e., alphabets) in the CFG, which are depicted as *a*, *b*, and *c* in the call graph of Figure 3. Each sub-call graph is represented by a variable in the CFG, i.e., *A*, *B* and *C* in Figure 3. We take *S* to be the start variable. Thus, the CFG for the call graph model in Figure 3 is as follows:

- |                                    |                                |
|------------------------------------|--------------------------------|
| (1) $S \rightarrow aA$             | (3) $B \rightarrow A \epsilon$ |
| (2) $A \rightarrow bB cC \epsilon$ | (4) $C \rightarrow A \epsilon$ |

The production rule (2) captures the implicit calls from the activity to the event handlers, while productions rules (3) and (4) capture the return of the control flow to the activity where the call is initiated after an event is handled. It is clear that if the app is comprised of more than one *Activity* and many events, the production rules would subsequently become more complex. Since infinite number of sequences can be generated from a grammar such as this, we can generate drivers with all kinds of valid combinations of event sequences. We start with drivers that

```
public static void main(String[] args) {
    try {
        View v = new View(null);
        DirectionsActivity da = new DirectionsActivity();
        ShortestRoute shortestButton = da.new ShortestRoute();

        da.onCreate();
        shortestButton.onClick(v);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

Listing 2. Sample Driver for *Driving Directions* App



have a single event and iteratively generate more complex drivers. By symbolically executing the apps using the generated drivers and monitoring the code coverage using *EMMA* [7], we obtain an accurate assessment of code coverage. This iterative process of generating drivers and testing would continue until a pre-specified code coverage threshold has been reached.

Listing 2 illustrates a sample driver for Driving Directions app generated using this method and using the sample call graph model in Figure 3. It contains two sequence of events, i.e., creating a *DirectionsActivity* object by calling its constructor that triggers the start of the app and calling *shortestButton.onClick* that simulates the action of user tapping on the “Shortest” button. That triggers *findShortestRoute(double srcLong, double srcLat, double dstLong, double dstLat)* method of Listing 1, which is specified to be executed with symbolic values.

## 7. RELATED WORK

The Android development environment ships with a powerful testing framework [4] that is built on top of *JUnit*. *Robolectric* [20] 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, testing tools use random inputs, but modern approaches utilize grammars for representing mutations of possible inputs [9, 21] or achieve white-box *fuzz* testing using symbolic execution and dynamic test generation[8]. None of these methods provide a fully automatic while comprehensive technique to generate test inputs for applications developed on an event driven framework, such as Android.

While approaches presented in [16, 18] use GUI crawling techniques to automatically extract directed graph models to model user interaction and generate test sequences, we look in the source code and use program analysis to derive the test generation and by targeting our framework to Android, we are able to achieve significant automation.

Our research is related to the approaches described in [2, 10] for testing Android apps. In [10], a new testing framework that integrates evolutionary testing and symbolic execution is proposed to address the weaknesses of symbolic execution in generating event sequences. [2] presents an approach based on concolic testing for generating event sequences for Android apps. While the first approach does not address the problem of event sequence generation for frameworks, such as Android, the second one only works for testing screen tap events and does not address the problem of handling input values.

## 8. CONCLUSION

We have provided an overview of what we believe to be the first approach for symbolic execution of Android apps. We have extended SPF and used it together with the system’s call graph model to generate test cases for Android apps. The key contributions of our work are (1) creating stubs that let us compile and run Android apps on JPF, (2) creating mock classes for Android library classes to address path-divergence, and (3) automatically generating drivers that simulate the user’s behavior based on the valid use cases obtained from analyzing the system’s source code. In our on going work, we are developing support for a larger subset of Android library classes by implementing the appropriate stubs and mock classes. We are also planning to integrate the extended SPF tool suite with our cloud-based Android testing infrastructure, described in [13], to detect both

functional and security vulnerabilities.

## 9. ACKNOWLEDGMENTS

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

## 10. REFERENCES

- [1] Amalfitano, D. et al. 2011. A GUI Crawling-Based Technique for Android Mobile Application Testing. *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on* (2011).
- [2] Anand, S. et al. 2012. Automated Concolic Testing of Smartphone Apps. *ACM Symposium on Foundations of Software Engineering (FSE’12)*, (2012).
- [3] Android Developers Guide: <http://developer.android.com/guide/topics/fundamentals.html>
- [4] Android Testing Framework: <http://developer.android.com/guide/topics/testing/index.html>.
- [5] Cadar, C. et al. 2011. Symbolic execution for software testing in practice: preliminary assessment. *International Conference on Software Engineering (ICSE)*, (2011).
- [6] Dalvik - Code and documentation from Android’s VM team: <http://code.google.com/p/dalvik/>.
- [7] EMMA: <http://emma.sourceforge.net/>.
- [8] Godefroid, P. et al. 2008. Automated whitebox fuzz testing. *Network and Distributed System Security Sym.* (2008).
- [9] Godefroid, P. et al. 2008. Grammar-based whitebox fuzzing. *ACM SIGPLAN Notices* (2008), 206–215.
- [10] Inkumsah, K. and Xie, T. 2008. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. *23rd IEEE/ACM Int’l Conf. on Automated Software Engineering (ASE ’08)* (2008).
- [11] Java PathFinder: <http://babelfish.arc.nasa.gov/trac/jpf/>.
- [12] King, J. 1975. A new approach to program testing. *Programming Methodology.* (1975), 278–290.
- [13] Mahmood, R. et al. 2012. A whitebox approach for automated security testing of Android applications on the cloud. *Int’l Wrkshp. on Automation of Soft. Test.* (Jun. 2012).
- [14] Malek, S. et al. 2005. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *IEEE Trans. Softw. Eng.* 31, 3 (Mar. 2005), 256–272.
- [15] Mehlitz, P. et al. 2011. Jpf-awt: Model checking gui applications. *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on* (2011).
- [16] Memon, A. et al. 2003. GUI ripping: Reverse engineering of graphical user interfaces for testing. *proceedings of the 10th working conf. on reverse engineering (WCRE ’03)* (2003).
- [17] Memon, A.M. et al. 2000. Automated test oracles for GUIs. *ACM SIGSOFT Software Engineering Notes* (2000), 30–39.
- [18] Memon, A.M. and Xie, Q. 2005. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *Software Engineering, IEEE Transactions on.* 31, 10 (2005).
- [19] MoDisco: <http://www.eclipse.org/MoDisco/>.
- [20] Robolectric: <http://pivotal.github.com/robolectric/>.
- [21] Sen, K. et al. 2005. *CUTE: A concolic unit testing engine for C.* ACM.
- [22] Symbolic PathFinder: <http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc>.
- [23] Visser, W. et al. 2004. Test input generation with Java PathFinder. *ACM SIGSOFT Soft. Eng. Notes.* 29, 4 (2004).
- [24] White, L. and Almezen, H. 2000. Generating test cases for GUI responsibilities using complete interaction sequences. *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on* (2000).