SIG-Droid: Automated System Input Generation for Android Applications

Nariman Mirzaei*, Hamid Bagheri†, Riyadh Mahmood* and Sam Malek†
*Department of Computer Science, George Mason University, {nmirzaei, rmahmoo2}@gmu.edu
†Department of Informatics, University of California, Irvine, {hamidb, malek}@uci.edu

***Abstract*—Pervasiveness of smartphones and the vast number of corresponding apps have underlined the need for applicable automated software testing techniques. A wealth of research has been focused on either unit or GUI testing of smartphone apps, but little on automated support for end-to-end system testing. This paper presents SIG-Droid, a framework for system testing of Android apps, backed with automated program analysis to extract app models and symbolic execution of source code guided by such models for obtaining test inputs that ensure covering each reachable branch in the program. SIG-Droid leverages two automatically extracted models: *Interface Model* and *Behavior Model*. The Interface Model is used to find values that an app can receive through its interfaces. Those values are then exchanged with symbolic values to deal with constraints with the help of a symbolic execution engine. The Behavior Model is used to drive the apps for symbolic execution and generate sequences of events. We provide an efficient implementation of SIG-Droid based in part on Symbolic PathFinder, extended in this work to support automatic testing of Android apps. Our experiments show SIG-Droid is able to achieve significantly higher code coverage than existing automated testing tools targeted for Android.**

***Index Terms*—Android, Automated Testing, Input Generation**

I. Introduction

Android is a platform for mobile communication devices, including smartphones and PDAs. It has had a meteoric rise since its inception partly due to its vibrant app market that currently provisions nearly a million apps, with thousands added and updated on a daily basis. Not surprisingly there is an increasing demand by developers, consumers, and app market operators for automated testing techniques applicable to Android apps. One of the key obstacles is the lack of practical techniques for generating test inputs.

Android apps are built using a common application development framework (ADF) that ensures apps developed by a wide variety of suppliers can interoperate and coexist together in a single system (e.g., a phone) as long as they conform to the rules and constraints imposed by the framework. An ADF exposes well-defined extension points for building the application-specific logic, setting it apart from traditional desktop software that is often implemented as a monolithic independent piece of code. Android also provides a container to manage the lifecycle of components comprising an app and facilitates the communication among them. As a result, unlike a traditional monolithic software system, an Android app consists of code snippets that engage one another using the ADFs sophisticated event delivery facilities. This poses a challenge to test automation, as the app's control flow frequently interleaves with the ADF. On the other hand, the knowledge of ADF along with the metadata associated with each app can be used to automate many of the software testing activities, in particular test input generation, as illustrated in this paper.

The main focus of this paper is on the problem of system-level input generation for Android applications. While there has been tremendous progress in Android testing at both unit-level and GUI testing, random testing still remains the major player in automated system testing of Android apps [24]. Monkey [2], a popular fuzzing tool provided by Google generates random touchscreen presses, gestures, and other system-level inputs. Dynodroid [23] performs more effective event-aware random testing, through inferring representative set of events and employing certain heuristics. Other techniques [12], [14], [17], [20], [33] mainly focus on testing the program through GUI elements. System behaviors dependent on data values, though, have not been adequately considered to a large extent, as data widgets are abstracted away, which may cause shallow code coverage.

Hence, system testing of interactive applications, such as Android apps, can be broken down into two distinct, yet interwoven problems. The first is to generate sequences of unique events, where each sequence represents a particular app use and causes a change in the state of the app. Here, the whole set of sequences exhaustively cover all possible use scenarios. The second is to generate proper values for GUI data widgets that take user inputs, such as textboxes. Here, the input domain can be quite large. For example, in a numeric textbox that accepts 5 unsigned digits and involves a conditional statement satisfied when the input value equals a certain integer, random input generation has only $\frac{1}{10^5}$ chance to reach the state satisfying that condition.

In this paper, we present SIG-Droid, an automated **S**ystem **I**nput **G**eneration framework for Android apps that tackles these challenges. SIG-Droid combines program analysis techniques with symbolic execution [22] to systematically generate inputs for Android apps that achieve high code coverage. SIG-Droid leverages the knowledge of Android's ADF specification to automatically extract two models from an app's source code. These models are used to guide the generation of event sequences aimed at simulating actual user behaviors. The *Behavior Model (BM)* captures the event-driven behavior of the app, including the relationships among the event generators and handlers, and the *Interface Model (IM)*, represents all of the input interfaces in the app and the widgets they contain, including buttons, input boxes, etc.

To prune the domain of data inputs, SIG-Droid employs symbolic execution, a promising automated testing technique that can effectively deal with constraints. Symbolic execution

978-1-5090-0406-5/15/$31.00 ©2015 IEEE

uses symbolic values, rather than actual values, as program inputs. It gathers the constraints on input values along each path of a program execution, and with the help of a constraint solver generates actual inputs for all reachable paths. While symbolic execution has proven to be effective for unit level testing, our goal is to utilize symbolic execution for end-to-end system testing of Android apps.

Symbolic execution of programs that are developed on the top of an ADF, however, has always been challenging due to problems such as *path-divergence* that occurs when a symbolic value flows outside the context of the program to the context of the underlying ADF [13]. In addition, Android is an event-driven system, which makes symbolic execution highly dependent on sequence of events; 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, although Android apps are developed in Java, they run on Dalvik Virtual Machine (DVM) [4], instead of the traditional Java Virtual Machine (JVM). This is problematic, as current symbolic execution engines that are targeted at Java cannot be used for Android apps.

SIG-Droid uses the extracted models to exhaustively pinpoint possible ways an app can receive inputs. It then exchanges all concrete inputs with symbolic values, and gathers the constraints around those inputs. To determine the execution paths that should be symbolically analyzed, it automatically generates sequences of event handler methods from the inferred *BM*. We call these sequences *Drivers*. Furthermore, to enable our symbolic execution engine to run the apps on JVM, and to resolve any possible external method calls resulting in path-divergence, models of Android library classes are created. After symbolically executing the app using the drivers, the solved values are used along with the corresponding events to create test inputs. Our experiments corroborate SIG-Droid's ability to systematically generate test cases for end-to-end testing of Android apps that achieve high code coverage.

This paper is organized as follows. Section II provides the background on Android and Symbolic Execution. Section III presents an Android app that is used for illustrating the approach. Section IV outlines an overview of our approach. Section V provides the details of our approach in extracting the models used in SIG-Droid. Sections VI describes SIG-Droid's symbolic execution engine. Section VII goes through the details of test case generation. The paper concludes with the presentation of our experimental results in Section VIII, an overview of the related research in Section IX, and a discussion of the limitations and our future work in Section X.

II. Background

In this section, we first provide an overview of Android, followed by a brief background on Symbolic Execution.

A. Android

The Google Android framework includes a full Linux operating system based on the ARM processor, system libraries, middleware, and a suite of pre-installed applications. 88% of the Android users still use older versions of Android [1] which rely on Dalvik Virtual Machine (DVM) [4] for executing programs written in Java. Android also comes with an application development framework (ADF), which provides an API 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 certain constraints.

Android apps are built using a mandatory XML manifest file. The manifest file values are bound to the application at compile time and cannot be changed afterwards unless the application is recompiled. This file provides essential information for managing the life cycle of an application to the Android ADF. Examples of the kinds of information included in a manifest file are descriptions of the application's components among other architectural and configuration properties.

Components can be one of the following types: Activities, Services, Broadcast Receivers, or Content Providers. 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 their controls are typically 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.

All Android components 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 handled through event messaging, a key property of event driven systems.

All major components, including Activity and Service, follow pre-specified lifecycles [1] managed by Android. The lifecycle event handlers (e.g., `onCreate()` and `onStart()`) are called by Android and play an important role in our research as explained later.

B. Symbolic Execution

Symbolic Execution [22] is a program analysis technique that uses symbolic values, rather than actual values as program inputs. Consequently, the outputs of the program are transformed to a function of the symbolic inputs. The path condition is a Boolean formula over the symbolic values representing the constraints which must be satisfied in order for an execution to follow a specific path. Using the path conditions around symbolic values, a decision tree, called symbolic execution tree, is created.

For illustration of this technique, we use a simple Java program depicted in Figure 1b, where S0, S1, S2, and S3 denote statements that can be invoked in different paths of the program. Clearly, random testing is not likely to result in good coverage for this program. Consider that the input value for y has to be exactly three times the value of variable x to cover statement S0. This is precisely where the symbolic execution is shown to be fruitful. Figure 2b shows the symbolic execution

[1] http://developer.android.com/about/dashboards/index.html

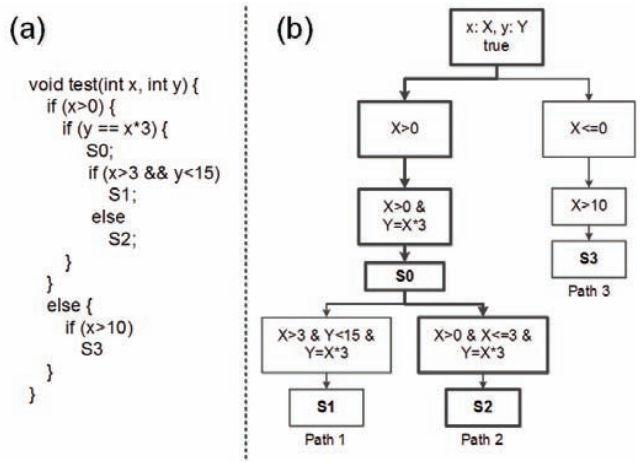

Fig. 1: Symbolic execution: (a) sample code, and (b) the corresponding symbolic execution tree, where X and Y are the symbolic representations of variables x and y

tree for this program. With the help of an off-the-shelf SAT solver, actual input values that result in paths shown in Figure 2b can be generated. These inputs can be used to generate test cases that cover different paths.

As an example, let X and Y be the symbolic representation of variables x and y, respectively. By solving the following constraint "$X > 0 \ \& \ X \leq 3 \ \& \ Y = X \times 3$", we obtain two values "$X = 3$" and "$Y = 9$", which result in taking the bold path in Figure 1b and executing S0 and S2. Similarly, using symbolic execution, we can generate all possible inputs for test method in such a way that all feasible paths in the program are explored. Moreover, symbolic execution can determine infeasible or unreachable paths and report an assertion violation (path 3).

Symbolic Pathfinder (SPF) [11] is a symbolic execution engine for Java programs. It is built on top of Java Pathfinder (JPF) [7], an open source general-purpose model checker for Java programs. Unlike other symbolic execution engines, SPF does not work with code instrumentation. It works with a non-standard interpretation of Java byte-code using a modied JVM [7]. SPF analyzes Java byte-code and handles mixed integer and real constraints, as well as complex mathematical constraints through heuristic solving. SPF can be used for test input generation and finding counterexamples to safety properties [11]. We have extended SPF to support Android apps. By addressing SPF limitations in dealing with event driven nature of Android, we are able to generate inputs and test cases for Android programs. Furthermore, symbolic execution is conventionally used for unit level testing, while our approach presents a novel approach for leveraging symbolic reasoning in generating system level test cases for Android apps.

III. Illustrative Example

For illustrating the approach, we will use a mobile banking app as a running example. Figure 2a shows two of the screens comprising this app: `MainActivity` and `TransferActivity`. The `MainActivity` is the first screen that the user sees when the app is launched. It allows the user to work with her checking or savings account (e.g., see the details of transactions occurring in each account). The `TransferActivity` screen allows the user to transfer money between the checking and savings accounts.

Figure 2b shows code snippets realizing one of the functionalities provided by this app. When the Transfer button (see Figure 2a) is clicked, the `onClick` method is called by the ADF. Subsequently, if the transfer amount is less than $5,000, an Intent is sent to the `MainActivity` including as payload the transfer amount, source and destination accounts. Finally, `MainActivity` updates the balance in each account to reflect the transfer amount, and displays the updated result to the user.

As mentioned in the previous section, the widgets on each activity are defined in an XML layout file. Figure 2c presents a snippet of the layout file for `TransferActivity`.

One conceivable test case generated using one of the existing techniques is clicking on the Transfers tab (thus bringing up the screen corresponding to `TransferActivity`), entering a random value as the amount to be transferred, and clicking on Transfer button. But considering the constraint in the code that the transfer amount cannot be more than $5,000, there is no systematic way of generating tests that cover both possible paths following the constraint. Our approach symbolically executes the parts of the code corresponding to the sequences of events and generates test cases that cover both paths.

IV. Overview of SIG-Droid

Figure 3 depicts a high level overview of SIG-Droid, which is comprised of three major components. The first component is the Model Generator that takes an app's source code and outputs two models:

- The Behavior Model (BM) represents the event-driven behavior of the app, including the relationships among the event generators and handlers. SIG-Droid uses the *BM* to generate possible use cases of the system (sequence of events), known as *Drivers* in the symbolic execution literature [11].
- The Interface Model (IM) provides a representation of an app's external interfaces and in particular ways in which it can be exercised, e.g., the inputs and events that are available on various screens to generate test cases that are valid for those screens. SIG-Droid uses the *IM* to determine the candidate input values that should be exchanged with symbolic values.

The second component of SIG-Droid is the Symbolic Execution Engine. As mentioned in Section II-B, SIG-Droid is built on top of JPF, which uses the byte-code interpretation of the program under test. Hence, the app's source code has to be compiled with Java compiler, instead of Android's Software Development Kit. This task is achieved by replacing platform-specific parts of the Android libraries that are needed for each app with stubs. These stubs are created in a way that each component's composition and callback behavior is preserved. This allows SIG-Droid to execute an Android app on JPF virtual machine without modifying the app's implementation.

The symbolic execution engine heavily utilizes the two generated models. The *BM* is used to generate the app Drivers (i.e., use cases), while the *IM* is used to mark the

Fig. 2: Banking App: (a) Screenshots, (b) code snippet from TransferActivity, and (c) snippet from Transfer.xml layout.

input values that have to be exchanged with symbolic values. Furthermore, prior to running the symbolic analysis, the code is instrumented in order to track the sequence of events that occur in each path. The results are stored in the symbolic execution report that is used later in generating test cases.

Finally, the third component of SIG-Droid is the Test Case Generator. It takes the *IM* along with the symbolic execution report as inputs and generates test cases that can be executed on top of Robotium [10], which is an Android test bed. The focus of this paper is on generating test cases that achieve high code coverage, not on whether the test cases have passed/failed. We acknowledge that automatically generating test oracles is a significant challenge, if not infeasible in many cases. This has been and continues to be the focus of many research efforts. Currently, we collect two types of results from the execution of tests: any exceptions that may indicate certain software faults as well as code coverage information. We use EMMA [5], an open source toolkit, for obtaining code coverage information. The next three sections describe the three components of SIG-Droid in more detail.

V. Model Generation

Model Generator extracts two models for each app: *Behavior Model (BM)* and *Interface Model (IM)*. In this section, we describe the details of each.

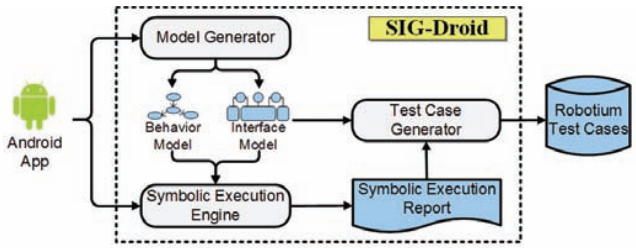

Fig. 3: High level overview of SIG-Droid.

A. Behavior Model

The *BM* represents a high-level behavior of the app in terms of the interactions among the event generators and handlers. This model is obtained in three steps: (1) reverse engineering of the app's call graph, which only contains the explicit method calls, (2) using knowledge of the ADF specification to augment the call graph with implicit calls (i.e., event exchange), and (3) pruning the call graph from nodes irrelevant to understanding the event-driven behavior of the app.

The first step of our approach entails using MoDisco [8], an open source program analysis tool, to extract the app's call graph. 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, ADF, and user interface. Therefore, instead of a connected call graph that represents the connected set of possible method calls, 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 and accessed by the user or the Android platform.

Figure 4a shows a subset of the Banking app's call graph obtained from its source code (as described later in this section, the red dashed lines are inferred to create a fully connected graph by extending MoDisco). Boxes in Figure 4a represent the methods, and the lines represent the sequence of invocations.

The second step of our approach is to relate the reverse engineered sub-call graphs to one another, and thus discover the dotted red arrows shown Figure 4a. Note that the links between the subcall graphs are implicit (i.e., these calls are initiated by the Android ADF itself) and not recoverable through simple source code analysis tools, such as MoDisco.

We observe that the root node of each sub-call graph is a method call never explicitly invoked from other parts of the

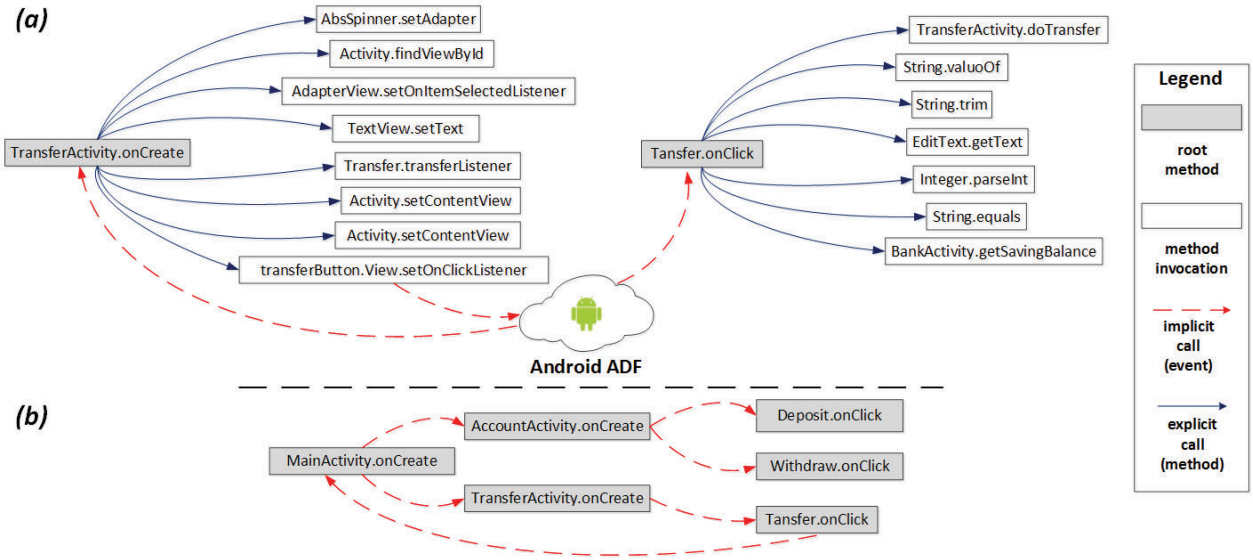

Fig. 4: (a) Examples of two sub-call graphs automatically inferred for the banking app; augmented red lines represent implicit calls used to connect sub-graphs; (b) the *BM* for the banking app automatically generated after pruning sub-graphs.

application. There are two types of root nodes:

1. *Inter-component root nodes* represent methods in a component that handle events generated by other components or Android framework itself. For instance, in the example of Figure 4b, `TransferActivity` component sends a `startActivity` event that results in `MainActivity`'s `onCreate()` method to be called.

2. *Intra-component root nodes* correspond to events that are internal to a component. For instance, in the example of Figure 4b, when a Button belonging to `TransferActivity` is clicked, the event is handled by the Transfer class within the same activity that implements the `OnClickListener` interface and overrides the `onClick()` method.

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). To link the different sub-call graphs, we continue down the graph and identify the leaf nodes where implicit method calls are initiated. These nodes would have to be method calls that either set an event handler, start other activities, send Intent messages, or handle 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. Based on Android's specification, we know that the links would have to be from leaf nodes to other root nodes. For example, in Figure 2b there is an implicit call from `startActivity` in `TransferActivity` to `MainActivity`'s `onCreate()`. Algorithm 1 shows how implicit calls are extracted, given a set of disconnected call-graphs and the set of caller methods that initiate implicit calls. These methods are defined by Android's specification. As new sub-call graphs are linked and connected, they are traversed 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.

Finally, the third step of our approach in deriving the *BM* is to remove all of the non-root nodes from the call-graph of an app and connecting the remaining nodes. The *BM* for the Banking app is depicted in Figure 4b. The *BM* only captures a high level behavior of the app in terms of the event interactions and does not include unnecessary details about the sequences of explicit method calls within the Android components.

It must be noted that the obtained call graph and hence the *BM* only represent the chain of possible method calls regardless of constraints, i.e., the call graph of an app does not include any information about conditions and the control flow of an app. As will be explained in detail later, the *BM* is only used to generate the Drivers for symbolic execution and not directly to generate the sequences in test cases. In other words, it is used 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.

B. Interface Model

The *IM* provides information about all of the input interfaces of an app, such as the widgets and input fields belonging to an Activity. It also includes information about the application-level and system-level Intent events handled by each Activity.

Algorithm 1: Implicit Call Extraction

Input: CG : set of sub-call graphs, ψ : set of implicit callers
Output: Υ : implicit calls

```
foreach c ∈ CG do
    rootNodes ← c.getRoot()
foreach c ∈ CG do
    lNodes ← c.getLeafNodes()
    foreach l ∈ lNodes do
        if l ∈ ψ then
            d ← l.getDestinationNode()
            if d ∈ rootNodes then
                Υ.Add(l, d)
```

The *IM* is obtained by combining and correlating the information contained in the configuration files and meta-data included in Android APK (such as Android Manifest and layout XML files). In order to do so, first we list all the activities of the app with the help of information that can be found in the Android Manifest file. Then for each activity we parse the corresponding layout file (recall Figure 2c) and obtain all information on each widget such as name, id, input type and so on. As described in the next section, we use the information provided by *IM* to identify possible symbolic values in the program. Subsequently, the information is used by the test case generator to construct the final test cases.

VI. Symbolic Execution for Android

To build a symbolic execution engine for Android we need to address three major challenges, since Android apps are (1) event-driven, (2) prone to path-divergence, and (3) compiled into Dalvik byte-code. In this section, we explain how SIG-Droid's symbolic execution engine addresses these challenges.

A. Handling Event-Driven Challenge

As Android is an event driven system, symbolic execution is highly dependent on events and their sequencing; 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 another application can initiate an event and cause the app to behave in a certain way.

To address this issue, symbolic execution engines, such as SPF [11], provide a mechanism to specify a Driver, which in the case of SPF is a Java program with a main method that contains the sequence of methods (event handlers) that should be used in a single run of the engine for determining the parts of the code that should be analyzed for gathering constraints. To generate the Drivers for Android apps, we use the *BM* as a finite state machine and traverse all the unique paths that do not contain a loop using a depth first search algorithm. This results in generating many possible sequences of events that represent possible use cases for the app.

As an example, using the *BM* in Figure 4b, if we start at `MainActivity.onCreate()` and follow through with `TransferAtivity.onCreate()` and `Transfer.onClick()`, we arrive at a plausible sequence. Clearly, if the app is comprised of more than one Activity and many events, the generated Driver would be more complex. Listing 1 illustrates a sample Driver for banking app generated in this way using the sample *BM* of Figure 4b. It contains two sequence of events, i.e., creates a `TransferActivity` object by calling its constructor following by calling the `onCreate` method that triggers the start of the activity. Consequently, it simulates the action of user tapping on the Transfer button by calling `onClick` method.

Note that since the *BM* does not model the program's constraints, not all generated Drivers are necessarily valid sequences of events (i.e., can actually occur when the program executes). As will be detailed in Section VII, we do not use the Drivers for the purpose of generating the test cases, but only for the purpose of guiding the symbolic execution and solving the constraints on input values.

B. Handling Path-Divergence and Davlik Byte-Code Challenges

The second challenge is an Android program's dependence on framework libraries that make symbolic execution prone to path-divergence, and more so than traditional Java programs. In general, path-divergence occurs when a symbolic value flows outside the context of the program that is being symbolically executed and into the bounding framework or any external library [13]. Path-divergence leads to two major problems. First, the symbolic execution engine may not be able to execute the external library, as a result extra effort may be needed to support those libraries. Second, the external path may contain its own constraints that result in generating extra test inputs attempting to execute the diverged path rather than the program itself. This creates a scalability problem, as it entails symbolically executing parts of the Android operating system every time there is a path-divergence.

Indeed, in Android, path-divergence is the norm, rather than the exception. A typical Android app is composed of multiple Activities and Services communicating extensively with one another using Intents. An Intent is used to carry 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.

Furthermore, 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) [4] instead of the traditional Java Virtual Machine (JVM). Thus, Android apps are compiled into Dalvik byte-code rather than Java byte-code. To symbolically execute an Android app using SPF , we need to first transform the app into the corresponding Java byte-code representation.

To tackle the path-divergence problem and compile Android apps to Java byte-code, SIG-Droid provides its 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 within a reasonable range, 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 do 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 nuts and bolts that connect the different pieces of an app together.

A common instance of path-divergence in Android occurs when one Activity is initiated from another one and a value is passed from the source to the destination Activity. This process is performed by utilizing an Intent message (recall from Section II-A that in Android inter-component messaging is achieved through Intents). In the case of banking app, as shown in Figure 2b, `TransferActivity` uses the `startActivity` method of the Android library class `Activity.java` to start the app's `AccountActivity` that displays the accounts and their respective balances. 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 Activity.java mock, such

```
public static void main(String[] args) {
   try {
      View v = new View(null);
      TransferActivity ta = new TransferActivity();
      Transfer t = ta.new Transfer();
      ta.onCreate();
      t.onClick(v);
   }
   catch (Exception e) {
      e.printStackTrace();
   }
}
```

Listing 1: Sample Driver for banking app.

that when its `startActivity` method is called, the control flow moves to the `onCreate` method of the recipient activity.

Moreover, we create a mock for the `Intent.java` to address the path-divergence problem in cases where the payload is a symbolic value. As shown in Figure 2b, an instance of Intent is passed to `startActivity`. If this Intent encapsulates a symbolic value for variable `amountValue`, it would result in 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 payload of an Intent, making it difficult to reason about a value stored as Object symbolically. To solve this problem, we provide our own implementation of a hash map that holds primitive values. Consequently, in our implementation of the `putExtra` and `getExtra` methods, we use our hash map implementation to enable the symbolic execution engine to reason about values that are exchanged using the Intent messages.

The last step prior to running symbolic execution of each app is to identify which values need to be executed symbolically. These are the values that the user can input using the GUI, e.g., the transfer amount in the banking app. As an example, for each input box in the *IM*, the source code of the corresponding activity is explored and the value of that input box, retrieved by calling `inputBox.getText()`, is exchanged with a symbolic value. It is important to keep a mapping between each introduced symbolic value and its corresponding widget on the screen. At the same time, the code is instrumented to record the sequence of actions taken. The mapping along with the sequence of actions captured in the *Drivers* are used by the test case generator to reproduce the values and actions in each test case.

VII. Test Case Generation

Following the extraction of models and symbolic execution of an app, SIG-Droid automatically generates test inputs that can be executed on an actual phone or emulator device. Running symbolic execution with each Driver results in a symbolic execution report. Each report specifies the concrete values that are obtained by solving the gathered symbolic conditions.

Each Driver representing a single path in the *BM* may contain several constraints, thus it may result in multiple execution paths. For instance, if `amountValue` is a symbolic value in `TransferActivity` in Figure 3b, the Driver in Listing 1 would result in two different execution paths: One

```
<?xml version="1.0" encoding="utf-8"?>
<Report>
  <Path id="1"
    <Activity name="TransfersActivity">
      <MethodCall name="Transfer.onClick()">
        <SoldvedVariable name="amountValue" value="1" />
      </MethodCall>
    </Activity>
  </Path>
  <Path id="2"
    <Activity name="TransfersActivity"
      <MethodCall name="Transfer.onClick()">
        <SoldvedVariable name="amountValue" value="5001" />
      </MethodCall>
    </Activity>
  </Path>
</Report>
```

Listing 2: Symbolic Execution report for Driver in Listing 1.

where the `amountValue` is less than $5000, and another where it is greater. Hence, the report for each Driver may result in several tests.

Moreover, as mentioned in Section V, the call graph of each app only contains information about the possible chains of method calls regardless of constraints. As a result, the Drivers that are generated using the *BM* may be invalid sequences of events, meaning that the constraints may prevent the execution of certain events. In order to make sure that we only generate valid sequences of events in each test case, the code is instrumented to track the actual method execution sequence during the symbolic execution. Thus, the symbolic execution report contains the sequence of called methods as well. Listing 2 shows the symbolic execution report for the Driver of Listing 1.

Since the report contains only the event handlers and not the actual event generators (e.g., the *ID* of the buttons on a screen), to generate test cases we use the *IM* to determine the event generator corresponding to each event handler in the report. For example, `Transfer.onClick` handler method in Listing 2 is the handler for the `Transfer` button on `TransferActivity` screen of Figure 2.

Listing 3 illustrates one of the Robotium test cases generated by SIG-Droid that corresponds to the report shown in Listing 2. Solo is a Java class provided by Robotium that executes the test (essentially represents the user of the app). This test case inputs `5001` in the `amount` text box, which has the index of zero, meaning it is the first text box on that activity, and then clicks on `Transfer` button.

```
public class TransferActivityTest_1 extends
    ActivityInstrumentationTestCase2<TransferActivity> {

  private Solo solo;
    ...
    @smoke
    public void testMethod() throws Exception {
      solo.enterText(0,5001);
      solo.clickOnButton(Transfer);
  }
  ...
}
```

Listing 3: Code snippet of a Robotium test automatically generated by SIG-Droid for TransferActivity.

VIII. Evaluation

To evaluate SIG-Droid, we formulate three research questions:

- **RQ1:** Is SIG-Droid capable of generating test cases for real-world Android apps?
- **RQ2:** How well does SIG-Droid perform? Can SIG-Droid achieve a better code coverage than state-of-the-art Android system testing frameworks?
- **RQ3:** How scalable is the approach in generating test cases for complex applications, i.e., apps involving highly constrained input values?

For investigating RQ1, we apply SIG-Droid to several real-world apps from an open-source repository, called F-Droid [6]. These apps are picked based on the following criteria: (1) the source code for the applications must be available (2) the app only uses standard GUI widgets that are included in Android API and does not use any third party widgets, and (3) the apps should capture the different application categories, such as productivity, entertainment, and tools. Moreover, neither MoDisco nor SPF handle anonymous classes. As a result, we refactored the source code of the apps to ensure they do not contain any anonymous classes.

Table I lists these apps. LOC, Activities, and Category columns report lines of code, number of activities of each app, and category of each app, respectively. [2]

TABLE I: Open-source apps used in the evaluation.

App	LOC	Activities	Category
CalAdder	276	2	Productivity
Tipster	501	1	Tool
MunchLife	631	2	Entertainment
JustSit	849	4	Productivity
AnyCut	1095	4	Tool
TippyTipper	2953	6	Tool

For addressing RQ2, we compare SIG-Droid with two approaches: Android Monkey [2] and Dynodroid [23]. We also considered other testing tools for the evaluation, but were not able to include them for various reasons. Some focus on other objectives (e.g., A^3E [15] focuses on discovering Activities by covering a model of an app and does not report statement coverage), while there were practical difficulties with others (e.g., SwiftHand [17] exits with an exception when used on our apps).

For answering RQ3, we develop a benchmark-suite that entails a collection of synthetic apps in different levels of complexity. To measure apps' complexity, we use three well-established complexity metrics from literature, namely *Method Call Sequence Depth*, *McCabe Cyclomatic Complexity*, and *Block Depth per Method*. We then measure the impact of input constraint and complexity on scalability of our technique.

All experiments were conducted on an Apple iMac machine with 8GB memory and a dual core 2.4GHz processor. We used Android Virtual Devices (Android emulators) with 1GB RAM and 2GB SD Card. We used Android 4.4 (KitKat), which at the time of writing this paper was the latest version of Android. A fresh emulator was created for each app along with only default system applications. During the experiments, we used EMMA [5] to monitor the statement coverage. The reported line coverage is gathered by running all of the generated test cases on each app.

A. Experiment 1: Open-Source Apps

In our first set of experiments, we measured and compared the source code statement coverage achieved using the test cases generated by SIG-Droid, Monkey, and Dynodroid. Android Monkey, developed by Google, is essentially a fuzzing tool that sends random inputs and events to the app under test. Dynodroid improves on the number of inputs/events Monkey uses, thus achieves a similar coverage with less generated events. Since both Dynodroid and Monkey treat both data widget inputs and events as input events, to achieve a fair comparison among Android Monkey, Dynodroid, and SIG-Droid, we ran each tool with the same number of events. To be more specific, we counted the number of events used in SIG-Droid generated test cases, and used that number as the number of inputs for Monkey and Dynodroid. As both Monkey and Dynodroid are based on random approaches, using the same low number of events that are generated by SIG-Droid may not be fair for a comparison. To address that, we also run both tools with 2,000 input events, which is the maximum number possible for Dynodroid [23].

The line coverage results are summarized in Table II.[3] Column *# of Events* represents the number of input events in Robotium test cases generated by SIG-Droid. An event in a Robotium test case can be either an action, such as a button click, or entering an input value into a widget like a text box. The next six columns then represent the line coverage and the time taken to generate and execute test cases for SIG-Droid, Monkey, and Dynodroid, given the number of input events shown in the first column for each app. Columns *Monkey(2000 Events)* and *Dynodroid(2000 Events)* represent the same information, but when Monkey and Dynodroid are given 2,000 events.

In terms of the total code covered for each app, SIG-Droid easily outperforms both Dynodroid and Monkey, achieving higher coverage for all apps. Given the same number of input events shown in the first column for each app, SIG-Droid's coverage on average outperforms Monkey and Dynodroid by a 57% and a 41% margin, respectively. Even when the other tools are allowed to use more events, the code coverage achieved by them is still clearly outperformed by SIG-Droid's. In addition, SIG-Droid runs 2X faster than Dynodroid, and about 3X slower than Monkey. This is not surprising, given that Monkey is a completely random testing tool. More specifically, when Monkey traverses a path, it does not backtrack or use any other systematic way to test the app. Therefore, Monkey's test coverage does not considerably improve even with significantly higher number of input events. In contrast, SIG-Droid relies on the *BM* to generate input events, thereby leads to unique sequences of events that cover nodes captured in the *BM*.

Although SIG-Droid performs significantly better than the two mentioned methods, it fails to achieve complete code coverage. In some cases this could be due to unreachable

[2]Per our study of 100 F-Droid apps, average number of activities for an app is 4.

[3]We could not run Dynodroid on CalAdder and TippyTipper as Dynodroid runs on Android 2.3 and it does not support apps developed with newer APIs.

TABLE II: Comparison of SIG-Droid with other techniques

App	# of Events	SIG-Droid		Monkey		Dynodroid		Monkey(2000 Events)		Dynodroid(2000 Events)	
		Coverage	Time(sec)	Coverage	Time(sec)	Coverage	Time(sec)	Coverage	Time(sec)	Coverage	Time(sec)
CalAdder	6	82%	122	10%	37	-	-	35%	118	-	-
Tipster	35	83%	159	31%	26	53%	462	67%	104	59%	33825
MunchLife	20	74%	186	36%	44	32%	354	49%	75	54%	31421
JustSit	30	75%	137	20%	46	26%	335	35%	163	53%	41252
AnyCut	18	79%	179	6%	58	38%	282	6%	71	66%	21757
TippyTipper	91	78%	484	26%	45	-	-	42%	106	-	-

code or pieces of code that handle specific events while the app is running, such as phone unlock. It is also partially attributed to the fact that the current implementation does not handle loops in the *BM* as well as well-known symbolic execution shortcomings in dealing with non-primitive data-types. Additionally, our program analysis does not support all possible ways that Android apps could be developed. For instance, there are many ways of handing events in Android, and one of those is through inline class declarations, which we do not support. In principle by extending the program analysis and symbolic execution support, we could increase the code coverage using SIG-Droid. That said, the results show that SIG-Droid is already significantly more effective than existing system testing techniques targeted at Android apps.

B. Experiment 2: Benchmark Apps

In practice, symbolic execution is predominantly known to be suffering from scalability issues caused by problems such path-explosion [16]. To assess SIG-Droid's performance and scalability, we needed a way of selecting benchmark apps that are nontrivial. Finding real-world apps that fall into a variety of categories defined by a number of complexity metrics is nontrivial. As shown previously [30], [31], an effective way to address this problem is to write benchmark applications that satisfy the requirements. Similarly, we built an Android app generator that produces apps with different levels of complexity for our experiments. These apps provide us with a controlled environment, i.e., these apps do not contain features, such as loops in their control flow, that are not fully supported by symbolic execution tools, including SPF. By using these apps for benchmarking the performance and scalability of SIG-Droid, we can remove the impact of such known limitations and only concentrate on the impact of app complexity. However, we also needed a way of ensuring the synthesized apps were representative of real apps.

To that end, we first conducted an empirical study involving real world apps and analyzed approximately 100 apps chosen randomly from F-Droid repository [6]. The selected apps were in various categories, such as education, communication, gaming, etc. We analyzed these apps according to three major complexity dimensions that could impact SIG-Droid: (1) Method Call Sequence Depth — the longest method call sequence in the app, (2) McCabe Cyclomatic Complexity — the average number of control flow branches per method, and (3) Block Depth per Method — the average number of nested condition statements per method. Figure 5 shows the distribution of these complexity dimensions among the 100 Android apps from F-Droid. Our app generator is able to synthesize apps with varying values in these three dimensions.

We then defined *complexity classes* for generating subject apps in our experiments. For that, we aggregated the data collected through our empirical study, as shown in Figure 5, and produced the overall app complexity classes ranging from 10th to 90th percentile, shown in Table III. For instance, the 10th overall complexity in Table III corresponds to the 10th percentile in all of the three dimensions shown in Figure 5. Essentially this means that an app belonging to a lower class is less complex with respect to all three dimensions compared to an app from a higher class.

We used SIG-Droid to test one generated app from each of the nine complexity classes. We evaluated SIG-Droid by measuring the execution time and the resulting statement coverage. By conducting this experiment, we were able to measure the impact of input constraint and complexity on performance of our technique.

Table III shows the symbolic execution time as well as the overall execution time, which includes the symbolic execution time and the time it took to generate and execute the tests. As one would expect, the increase in the app complexity results in a modest increase in the symbolic execution time, since more constraints need to be solved. We also notice an increase in the overall execution time, due to the higher number of generated test cases and consequently the time needed for their execution. The results demonstrate that SIG-Droid is capable of scaling to even the most complex Android apps. Although not the focus of this experiment, it corroborates our earlier assertion that SIG-Droid 's inability to obtain complete code coverage in the case of real apps is due to the existence of unreachable code as well as incomplete model of app behavior. More advanced program analysis techniques for obtaining complete model of app behavior could reduce the gap between SIG-Droid 's actual code coverage in real apps and its theoretical potential in synthesized apps.

IX. Related Work

The Android development environment ships with a powerful testing framework [3] that is built on top of JUnit. Robolectric [9] is another framework that separates the test cases from the device or emulator and provides the ability to run the tests directly by referencing the Android library files. While these frameworks automate the execution of the tests, the test cases themselves still have to be manually developed.

Prior research [25]–[27], [32], [34] has investigated testing of traditional event-based systems. These approaches rely on techniques, such as ripping and crawling, to automatically

TABLE III: Benchmark apps.

App	Max Method Call Sequence	McCabe Cyclomatic Complexity/Method	Nested Block Depth/Method	Symbolic Exec. Time(sec)	Overall Exec. Time(sec)
1	15.00	1.6590	1.2929	2	40
2	23.60	1.8506	1.4018	2	37
3	27.10	1.9332	1.4742	4	175
4	32.00	2.0416	1.5216	4	135
5	38.00	2.1945	1.5650	6	30
6	41.40	2.3606	1.6860	5	60
7	50.10	2.5575	1.7761	5	299
8	62.20	2.8956	1.8850	6	596
9	90.80	3.2287	1.9867	6	446

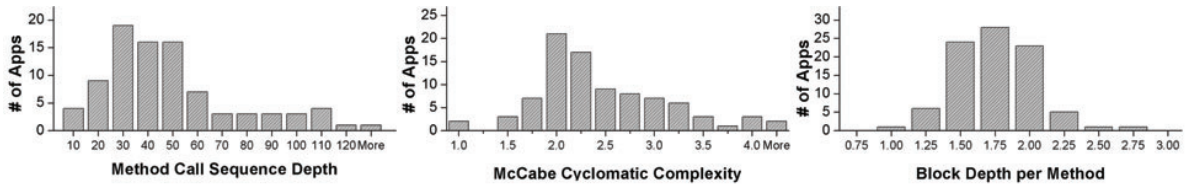

Fig. 5: Android complexity metrics distribution from a random sample of 100 apps.

extract directed graph models. Those models are then used to generate test sequences. However, data widgets are often abstracted away in such models. As such, system behaviors dependent on data values have not been adequately considered.

Barad [19] is perhaps the most closely related work to ours. It symbolicaly executes a sequence of events to infer inputs for data components of the GUI. While Barad relies on a combinatorial technique in generating the sequences of events, SIG-Droid uses a more efficient algorithm applied to inferred app models. In addition, Barad does not target Android framework, whose event-driven nature imposes several research and engineering challenges. More recently, Jensen et al. [21] proposed an approach that combines symbolic execution with sequence generation. Their work is concerned with finding valid sequences and inputs to reach pre-specified target locations, therefore, does not address the problem that we address, i.e., the automation of end-to-end system testing maximizing code coverage, nor is it targeted at Android.

Our research is also related to the approaches described in [14], [15], [23], [29], [33] for testing Android apps. Among others, Anand et al. [14] presented an approach based on concolic testing for generating event sequences for apps. Their approach only works for testing screen tap events and does not address the problem of handling user's input values. A^3E Tool [15] uses static taint analysis capabilities of SCanDroid [18] to build app models used for test generation, whereas SIG-Droid analyzes the source code and the associated metadata (e.g., Android's manifest file) to generate such models. Dynodroid [23], which was used extensively in our evaluation, uses random values and sequences of events. Dynodroid incorporates several heuristics to improve on Android Monkey's performance. Another approach, presented in [29] combines Model-Based testing and Combinatorial Testing. Our work differs from [29] as we use symbolic execution, which is both more effective and efficient than combinatorial testing for input generation, specially in dealing with constraints.

Finally, in our own prior research, we have developed an evolutionary testing approach, called EvoDroid [24]. EvoDroid's objective is to generate sequences of events using a genetic algorithm, while SIG-Droid's objective is to generate suitable input values for data widgets. Our prior publication [28] describes the major challenges of symbolically executing Android apps and provides a high-level sketch of a solution for solving those challenges. In addition, the solution described in our prior work was limited to the scope of a particular Activity, rather than generating system level test inputs. SIG-Droid extends our prior research by generating system level inputs using an improved model extraction and subsequently event sequence generation.

X. Conclusion

We have presented SIG-Droid, a novel framework for automated testing of Android apps. The key contributions of our work are (1) a fully automated technique for extracting models of an app's behavior and interface to support testing, (2) a symbolic execution engine that supports Android apps, (3) combining model-based testing with symbolic execution to systematically generate test inputs for Android apps, and (4) a supporting framework that generates effective system level test inputs for Android apps.

Although SIG-Droid has shown to be significantly better than existing tools for automated testing of Android apps, there are several avenues of future research and improvement. Currently we generate the sequence of events through a depth first search on the *BM*, which does not guarantee to generate all possible sequence of events. For instance, consider a situation in which a particular execution path is taken only when the same button is clicked several times. Covering such sequences requires our depth-first search algorithm to include loops in its search for all unique sequences of events, the space for which is infinite. In our previous [24], we have developed an evolutionary testing technique for Android apps to support generation of more complex sequences. We plan to use both techniques in tandem to complement the shortcomings of each. The symbolic execution engine presented in this paper will be used to solve the constraints in the code, while the proposed evolutionary algorithm will be used to find sequences that reach a particular location in code.

SIG-Droid currently only focuses on generating values for GUI data-input widgets. In our future work, we plan to extend SIG-Droid for symbolically reasoning about other types of input, such as system inputs. In addition, expanding support for Android libraries through the development of additional stubs and mock classes requires significant manual engineering effort; hence, we plan to investigate possible techniques to automate this process. Finally, we are enhancing SIG-Droid's program analysis to support inline declaration of event handlers to generate more accurate model of app behavior.

XI. Acknowledgments

We would like to thank Corina Pasareanu from NASA Ames Research Center for assisting us with making changes to the Symbolic PathFinder, as well as Aravind Machiry and Mayur Naik from Georgia Tech for assisting us with the setup of Dynodroid. This work was supported in part by awards D11AP00282 from the US Defense Advanced Research Projects Agency, H98230-14-C-0140 from the US National Security Agency, HSHQDC-14-C-B0040 from the US Department of Homeland Security, and CCF-1550206 from the US National Science Foundation.

REFERENCES

[1] Android developers guide. http://developer.android.com/guide/topics/fundamentals.html

[2] Android monkey. http://developer.android.com/guide/developing/tools/monkey.html

[3] Android testing framework. http://developer.android.com/guide/topics/testing/index.html

[4] Dalvik - code and documentation from android's VM team. http://code.google.com/p/dalvik/

[5] EMMA. http://emma.sourceforge.net/

[6] F-droid. https://f-droid.org/

[7] Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/

[8] MoDisco. http://www.eclipse.org/MoDisco/

[9] Robolectric. http://pivotal.github.com/robolectric/

[10] Robotium. http://code.google.com/p/robotium/

[11] Symbolic PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc

[12] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012, 2012.

[13] S. Anand, "Techniques to facilitate symbolic execution of real-world programs," Ph.D., Georgia Institute of Technology, 2012.

[14] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th Int. Symp. on the Foundations of Software Eng.*, ser. FSE '12, 2012.

[15] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN Int. Conf. on Object Oriented Programming Systems Languages & Apps.*, ser. OOPSLA '13, 2013.

[16] C. Cadar *et al.*, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011.

[17] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN Int. Conf. on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13, 2013.

[18] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications," Dept. of Comp. Science, University of Maryland, College Park, Tech. Rep. CS-TR-4991, Nov 2009.

[19] S. Ganov, C. Killmar, S. Khurshid, and D. E. Perry, "Event listener analysis and symbolic execution for testing gui applications," in *Proceedings of the 11th Int. Conf. on Formal Engineering Methods: Formal Methods and Software Engineering*, ser. ICFEM '09, 2009.

[20] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th Int. Workshop on Automation of Software Test*, ser. AST '11, 2011.

[21] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 2013 Int. Symp. on Software Testing and Analysis*, ser. ISSTA 2013, 2013.

[22] J. C. King, "A new approach to program testing," in *Proceedings of the International Conference on Reliable Software*, 1975.

[23] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Softw. Eng.*, ser. ESEC/FSE 2013, 2013.

[24] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 2014 ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14, November 2014.

[25] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "Autoblacktest: Automatic black-box testing of interactive applications," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, April 2012.

[26] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th Working Conf. on Reverse Eng.*, ser. WCRE '03, 2003.

[27] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Automated test oracles for guis," in *Proceedings of the 8th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering: Twenty-first Century Applications*, ser. SIGSOFT '00/FSE-8, 2000.

[28] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, Nov. 2012.

[29] C. D. Nguyen, A. Marchetto, and P. Tonella, "Combining model-based and combinatorial testing for effective test case generation," in *Proceedings of the 2012 Int. Symp. on Software Testing and Analysis*, ser. ISSTA 2012, 2012.

[30] S. Park *et al.*, "Carfast: achieving higher statement coverage faster," in *Proceedings of the ACM SIGSOFT 20th Int. Symp. on the Foundations of Software Engineering*. ACM, 2012.

[31] D. R. Slutz, "Massive stochastic testing of sql," in *VLDB*. Citeseer, 1998.

[32] L. White and H. Almezen, "Generating test cases for gui responsibilities using complete interaction sequences," in *Proceedings of 11th Int. Symp. on Software Reliability Engineering, (ISSRE) 2000.*, 2000.

[33] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Proceedings of the 16th Int. Conf. on Fundamental Approaches to Software Engineering*, ser. FASE'13, 2013.

[34] X. Yuan and A. M. Memon, "Generating event sequence-based test cases using gui runtime state feedback," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, Jan. 2010.