Verification of Android Applications

Heila van der Merwe
Computer Science Department
Univeristy of Stellenbosch, South Africa
hvdmerwe@cs.sun.ac.za

***Abstract*—This study investigates an alternative approach to analyze Android applications using model checking. We develop an extension to Java PathFinder (JPF) called JPF-Android to verify Android applications outside of the Android platform. JPF is a powerful Java model checker and analysis engine that is very effective at detecting corner-case and hard-to-find errors using its fine-grained analysis capabilities. JPF-Android provides a simplified model of the Android application framework on which an Android application can run and it can generate input events or parse an input script containing sequences of input events to drive the execution of the application. JPF-Android traverses all execution paths of the application by simulating these input events and can detect common property violations such as deadlocks and runtime exceptions in Android applications. It also introduces user defined execution specifications called *Checklists* to verify the flow of application execution.**

I. Introduction

Software applications with many external references and dependencies are notoriously difficult to analyze and verify [1]. These external dependencies form part of the context of the application's execution, also called the environment of the application. For the application under analysis to execute soundly and correctly, an adaptable, controlled environment is required that reacts correctly and consistently [2].

Android applications form part of this group of *open* applications since they have many external references and dependencies. The environment of an Android application consists of the Android software stack, which internally has a client-server design (Figure 1.) The applications are built on top of the Android application framework. The framework provides the core implementation of an application including event handling and processing, management of application components and resources as well as facilitation of Inter Process Communication (IPC) with other applications and system services. The state of all applications on the device is controlled by the system server. The system server contains a set of shared services such as a window manager, package manager as well as an activity manager controlling the state of each application component running on the device. Android applications are driven by input events from system services and other applications. These events include User Interface (UI) and system events sent to the appropriate application via the system server.

The Android application environment presents three challenges to analysis tools: (1) Android applications are heavily dependent on the Android application framework and system services not runnable outside of the software stack, (2) since

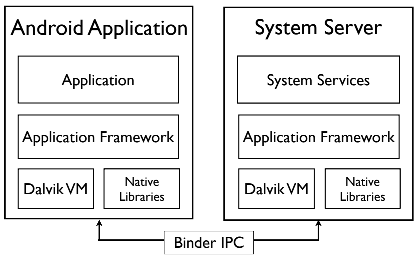

Fig. 1. The Android environment

Android applications have an asynchronous design with many entry points, analysis requires a driver to execute the application and (3) it is difficult to reliably monitor the flow or execution of applications for errors due to the asynchronous, multi-threaded design of the applications as well as the usage of reflection and native method calls. If the application is executed on the software stack, validating the execution is even more challenging since the software stack needs to be edited and recompiled or instrumented for each version of Android.

The commonly used approaches to analyze Android applications are static and dynamic analyses. Static analysis usually requires the system under test to be compilable (not runnable) and as such does not require generation of test drivers. However, static analysis still requires modeling native code and implicit dependencies, as well as specifying properties to check (e.g., memory leaks, null pointer exceptions, specific usage patterns). The results are usually an over-approximation, but for Android applications errors can be missed due to the models required to run the tool on the applications. Static analysis can also not provide users with a sequence of events leading to a possible error for validation purposes.

In contrast, dynamic analysis requires the system under test to be runnable and requires test driver generation. In addition, unit testing requires generation of mock objects. This type of testing uses test oracles and assertions to ensure the component executes correctly. Dynamic analysis provides an under-approximated analysis of the application and its main challenge is generating input events that obtain satisfactory coverage of the application code [3].

Model checking is a mature verification technique that could be used to prove the absence of property violations in an application if the state space is finite (and small enough). However, most often it is used simply for bug finding, even if the state space is infinite. The search space is reduced using state matching and backtracking while dynamically executing the application in a closed environment. Even with state

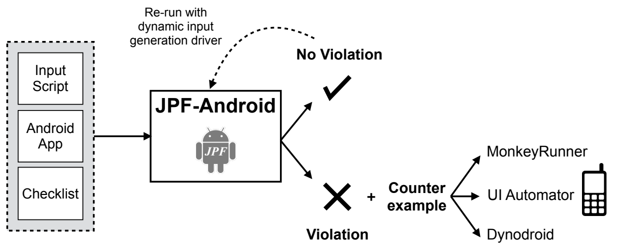

Fig. 2. Our approach

matching, model checking suffers from the state space explosion problem and the actual environment is often abstracted to reduce the state space. Since model checking requires fine-grained control over the application's execution, an adaptable and customizable model of the environment is required.

For this research we focus on verifying Android applications using verification tools developed for general Java applications. More specifically we extend JPF, a powerful model checker and analysis engine to verify and analyze Android applications. We use JPF since it is developed for Java applications and it has many extensions that can be applied to Android applications (such as Symbolic PathFinder). Since Android applications are executed dynamically we face the same challenges as some of the dynamic analysis tools. We need to generate input events that obtain good coverage of the application code. Additionally we require an extensive environment model on which to run the application. JPF allows us to provide a more controllable and adaptable environment that can simplify the detection of hard-to-find and corner-case errors that the currently available Android analysis tools struggle detecting due to the application's reliance on the external environment of the application. JPF also provides us with a property listener framework to monitor the execution of an application at byte-code level. Furthermore, if a property violation is reached, it can provide the user with a sequence of input events to reach the violation.

The expected contribution of this research is to show how model-checking Android applications in a controlled and adaptable environment, using the fine-grained analysis capabilities of JPF, can simplify the detection of errors that are hard-to-find due to the complex design of Android applications.

II. Overview of our approach

As part of this research we develop a tool called JPF-Android built on JPF. Our approach is shown in Figure 2. The tool takes as input a script containing non-deterministic sequences of input events to drive the application's execution, the Android application code compiled to Java byte-code and *Checklists* used as property specifications to verify the application's execution. JPF-Android provides a model environment on which to execute the application using the events from the input script. It then tracks the execution of the application and verifies it against property listeners including the Checklists listener.

There are three possible outcomes of running the tool on an application. The first outcome is that the tool finds a property violation or error in the application. It immediately stops execution and reports the error and the sequence of events that leads to the violation/error to the user. Since the correctness of the tool is based on the model environment, we need to verify that the error exists in the actual environment. For this reason we also provide a translator that can run the erroneous event sequence on a device using dynamic testing tools such as MonkeyRunner[1] or Dynodroid [3]. The second outcome is that the tool completes the search of the state space and finds no property violations. In this case the user can choose to perform a more exhaustive search of the application using the dynamic input generation driver of the tool. We expect to find that model-checking can achieve a high coverage of the application code due to its exhaustive search driver which executes all possible event sequences up to a certain length combined with the state matching to reduce the search space. In the third case, the state space is too large and JPF-Android never stop running or runs out of memory. In this case we can try to limit the state of the application by removing fields from it that change so often that no state matching can take place.

In the following sections we discuss how JPF-Android solves the three main challenges to analyzing Android applications.

A. Environment modeling

The are two main reasons to model components in the Android environment. The first reason is to model behavior of unavailable or native code that can not be executed or analyzed. Android applications are dependent on the Android application framework and system server to run. Unfortunately these libraries cannot execute directly on the Java virtual machine as they require native libraries and drivers that are not available. For this reason we need to model their native behavior to allow Android applications to execute outside of the software stack. The second reason for modeling the behavior of components in the environment is to abstract or simplify their implementation. Abstraction is very important for model checking since it reduces the size of the exploding state space.

In our previous work [4], [5] we created an Android environment model manually to verify Android applications using JPF. We found that the Android environment is very complex and that components often have references and dependencies on other parts of the system or native libraries without which they would not be setup correctly or they might not run at all. Creating models manually is a time-consuming task that requires expert domain knowledge. For our current research we improve our manual approach by identifying where we can apply tools to replace or assist our manual effort.

We use different approaches to create models depending on the functionality required from them. The most basic way to model a class or component is to create empty stubs. These stubs keep public methods and fields and return default

[1]developer.android.com/tools/help/monkeyrunner_concepts.html

values if accessed or called, but they have no side-effects on the application or environment. They are useful for modeling classes that do not influence the verification of the application for example the Logging class. This type of modeling works well for abstraction and the models can easily be generated automatically using static analysis tools such as OCSEGen [1] and Modgen [6] geared towards environment generation.

To retain certain behavior of the component/class but to reduce its complexity, we can also generate smarter stubs using OCSEGen and Modgen. OCSEGen can generate stubs retaining side effects (changes) to certain fields by making use of side-effect analysis. Modgen is a code slicing tool focused on optimization of library classes by stubbing out functionality to reduce complexity. This type of modeling is useful for GUI components where functionality related to the visual aspects of the components is not important for verification purposes and can be removed but functionality such as containment, visibility and listener registration must be retained.

Sometimes we require more complex models of components to verify the usage of the component or when the component is crucial to the execution of the application. In this case we need to study the specifications of the component or use runtime monitoring to extract a model of the component's behavior. We can then write models manually to simulate the behavior or use tools such as OCSEGen to generate models given a set of specifications. This type of modeling is useful for resources such as the MediaPlayer or Camera where we need more extensive models that follow the specifications of the component to ensure correct usage of the resource by the application.

B. Driving the application execution

Android applications have an event driven design where their execution is triggered by input events. Android applications respond to many different types of input events including physical button presses, the rotation of the device, the application being started from the launcher and the WiFi connection dropping. These input events can be combined into millions of unique event sequences of varying lengths. Each of these sequences can potentially trigger erroneous application behavior. Since it is impractical to execute all event sequences, we need to generate input events that obtain maximum code coverage given the minimum amount of input events.

JPF-Android supports driving the application in two ways. Users can either script input sequences or they can perform a more exhaustive search using JPF dynamic input generation driver. The scripting environment allows developers to script compact, non-deterministic sequences of input events. The events are then fired one by one each time the application is idle and waiting for input events. Although this approach provides the tool with optimized input sequences, writing the input scripts becomes a tedious process and scripts might miss uncommon sequences leading to errors. For this reason we are developing an input generation driver that can generate input sequences automatically at runtime bound to a certain number of events. We expect that JPF-Android will reduce the number and length of the input sequences using state matching to stop execution if a previous state that has already been explored is reached again. For the data parameters of generated events we plan on using symbolic execution on bounded and small components of the application to identify interesting values before model checking the application.

We have also developed a translator that can execute event sequences returned by JPF-Android when a property violation is reached on the actual device or emulator. This allows the user to validate errors detected by JPF-Android.

C. Property specifications

In model checking, logical formula specifications are used to verify the application's execution. These specifications can describe safety properties such as unchecked exceptions and liveness properties including deadlock and race-conditions. Property specifications can also be used to verify that the implementation conforms to its design requirements.

JPF-Android simplifies specifying properties by providing a Checklists property listener to track the execution of the application at method level. Checklists can be used to verify that certain methods are executed in a specific order. This is useful as it provides an automatic way to verify that an application executes as expected during each test run. Checklists can be written to verify a variety of properties including liveness and safety properties. Checklists is just one type of property specification that can be implemented using JPF's property listener's functionality. Once Android applications can execute on JPF, many other property listeners can be created for example resource and memory leak detection as well as incorrect usage of APIs. JPF-Android also supports code coverage calculation using Java PathFinder's built-in coverage calculation listener. Code coverage will indicate how thoroughly the code was checked for Checklists and other property violations. For our future work we plan to create Android specific listeners to identify common errors such as these in Android applications to show the powerful analysis that can be performed using JPF-Android.

D. Evaluation

To evaluate our approach we will run JPF-Android on a set of open source Android applications with previously identified errors. We can validate the existence of the errors found during analysis using the translator to execute the sequences leading to the error on the device. We will also compare results and coverage statistics with other dynamic analysis tools such as Dynodroid and Monkey.

III. Progress

This research has lead to three papers published in the ACM SIGSOFT Software Engineering Notes. The first paper entitled "Verifying Android Applications on Java PathFinder" [4], describes the design and implementation of JPF-Android. The second paper, "Execution and Property Specifications for JPF-Android" [5], discusses the syntax of JPF-Android's input script and how events are simulated on the application under test and the addition of Checklists to monitor and verify the execution of Android applications during runtime.

The main challenge to evaluating our approach it that it requires modeling of the environment for the set of applications and the writing of test scripts to drive the application's execution. To allow the tool to be run on a larger set of applications, we are automating some of this effort. To facilitate the creation of the environment model, we investigated currently available modeling techniques and how they can assist the user in our paper entitled "Generation of library models for verification of Android Applications" [2]. We are also currently extending JPF-Android to generate input sequences automatically.

IV. Related work

Static analysis usually performs a shallow analysis of the application and is geared towards detecting very specific properties such as buffer overflow. Since static analysis is very efficient at tracking data flow within an application, most of the static analysis tools for Android applications are focused on performing some kind of data flow analysis for example privacy leaks [7]–[9].

Most dynamic analysis tools execute the application on the actual emulator or device. Although the application can execute soundly in this environment, it is not always possible or simple to control the state of the environment required for controlled testing. The emulator, for example does not have WiFi capabilities and it is not possible to control the state of the battery on the actual device without draining it to the required amount. It provides an under approximated analysis bound by the number of input events.

Android's monkey tool is a dynamic stress testing tool that takes a black-box approach to Android application testing. It automatically generates pseudo-random input events and fires the events while the application is running on the emulator / device. The disadvantage of generating random input events is that it may generate many unimportant event sequences while not triggering important events. Dynodroid [3] makes use of the same approach as Monkey, but instead of generating the events randomly, it intelligently selects the events, careful not to starve certain event types. Another tool shipped with the Standard Development Kit (SDK) is MonkeyRunner. It provides a python API for users to script input event sequences that are simulated by the Android emulator/device.

Android application can also be tested using Android's JUnit testing framework. This framework allows the developer to write unit and integration tests for Android applications. Although we can create mocks and stubs to model external functionality, more complicated application behavior requires the application to execute as a whole and we need to be able to track the execution of the application. Robolectric [10] is also a JUnit testing framework for Android, but runs outside of the emulator in the JVM. As the Android source code can not compile on the JVM, Robolectric makes use of the JavaAssist library to intercept class loading and return a shadow class that model the functionality of the original class.

Other project have also applied verification techniques to Android applications. Anand et al. [11] applied concolic testing to traverse the application on the device and Mirzaei et al. [12] applied symbolic execution to Android applications to generate drivers for JUnit testing application components. For both approaches the state space exploded quite quickly due to the number of possible input events.

V. Conclusion

There are many approaches available to detect errors and bugs in Android applications. Each approach has to deal with the three main challenges of analyzing Android applications to various degrees, e.g., static analysis may not require input generation. The research presents a novel approach for verifying Android applications by developing a model checking tool, called JPF-Android. JPF-Android allows Android applications to be verified outside the Android platform on Java PathFinder (JPF). Android applications are executed on a model of the Android software stack and their execution driven by simulating user and system input events. This approach provides the tool with full control over the Android system and analysis of the execution of the application during runtime. The limitations of using this approach is the upfront work required to create a bounded environment model, whose correctness is imperative to the correctness of the analysis.

Acknowledgment

I would like to thank my two advisors, Willem Visser and Brink van der Merwe, as well as Oksana Tkachuk for their advise and contributions to this research.

References

[1] O. Tkachuk, "OCSEGen: Open components and systems environment generator," in *Proc. 2nd Int. Work. State Art Java Progr. Anal.*, no. 1,

[2] H. van der Merwe, O. Tkachuk, B. van der Merwe, and W. Visser, "Generation of Library Models for Verification of Android Applications," in *ACM SIGSOFT Softw. Eng. Notes*,

[3] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for Android apps," *Proc. 2013 9th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2013*, p. 224, Aug. 2013.

[4] H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying Android applications using Java PathFinder," *ACM SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, p. 1, Nov. 2012.

[5] ——, "Execution and property specifications for JPF-android," in *ACM SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1,

[6] M. Ceccarello and O. Tkachuk, "Automated Generation of Model Classes for Java PathFinder," in *ACM SIGSOFT Softw. Eng. Notes*,

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid," *ACM SIGPLAN Not.*, vol. 49, pp. 259–269, 2014.

[8] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid : Automated Security Certification of Android Applications."

[9] W. Enck, L. P. Cox, P. Gilbert, and P. Mcdaniel, "TaintDroid : An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones."

[10] B. Sadeh and S. Gopalakrishnan, "A Study on the Evaluation of Unit Testing for Android Systems," *Int. J. New Comput. Archit. their Appl.*, vol. 4, no. 1, pp. 926–941, 2011.

[11] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng. - FSE '12*, ser. FSE '12.

[12] N. Mirzaei, S. Malek, C. S. Pasareanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *ACM SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, p. 1, Nov. 2012.