Static Analysis of Android Programs

Étienne Payet\textsuperscript{1} and Fausto Spoto\textsuperscript{2}

\textsuperscript{1} LIM-IREMIA, Université de la Réunion, France
\textsuperscript{2} Dipartimento di Informatica, Università di Verona, Italy

Abstract. Android is a programming language based on Java and an operating system for mobile or embedded devices. It features an extended event-based library and dynamic inflation of graphical views from declarative XML layout files. A static analyzer for Android programs must consider such features, for correctness and precision. This article is a description of how we extended the Julia system, based on abstract interpretation, to run formally correct analyses of Android programs. We have analyzed with Julia the Android sample applications by Google and a few larger open-source programs. Julia has found, automatically, bugs and flaws both in the Google samples and in the open-source programs.

1 Introduction

Android is a main actor in the operating system market for mobile or embedded devices. It is an operating system for such devices and a programming language, based on Java, with an extended event-based library for mobile applications. Any Java compiler can compile Android applications, but the resulting Java bytecode must be translated into a final, very optimized, Dalvik bytecode.

Static analysis of Android applications is important as quality and reliability are keys to success on the Android market (http://www.android.com/market). The company Klocwork (http://www.klocwork.com) has already extended its analysis tools from Java to Android, obtaining the only static analysis for Android that we are aware of, currently limited in power and incorrect: if the analyzed program contains a bug, it will often miss it. Nevertheless, this shows that industry recognizes the importance of the static analysis of Android code.

Julia is a static analyzer for Java bytecode programs that do not use reflection nor multithreading, based on abstract interpretation [1]. It ensures, automatically, that the analyzed applications do not contain a large set of programming bugs. It applies non-trivial whole-program, interprocedural and semantical static analyses, including classcast, dead code, nullness and termination analysis. It comes with a correctness guarantee, as typical in the abstract interpretation community: if the application contains a bug, of a kind considered by the analyzer, then Julia reports it. This makes the result of the analyses more significant. However, its application to Android is not immediate and we had to solve many problems before Julia could analyze Android programs in a correct and precise way. This article presents those problems and our solutions. The resulting system analyzes non-trivial Android programs with high degree...
of precision and finds bugs in third-party code. Our experimental results are available at [http://julia.scienze.univr.it/runs/android/results.html](http://julia.scienze.univr.it/runs/android/results.html). This paper does not describe in detail the static analyses provided by Julia, already published elsewhere, but only their adaptation to Android. Our analyzer is a commercial product ([http://www.juliasoft.com](http://www.juliasoft.com)). It can be freely used through the web interface at [http://julia.scienze.univr.it](http://julia.scienze.univr.it) whose maximal analysis size has been limited.

The analysis of Android programs is non-trivial since we must consider some specific features of Android, both for correctness and precision of analysis. First of all, Julia analyzes Java bytecode while Android applications are shipped in Dalvik bytecode. Eclipse ([http://www.eclipse.org](http://www.eclipse.org)) is the standard development environment for Android at the moment. It can export Android applications in jar format i.e., in Java bytecode. Hence we have generated the jar files of our experiments from Eclipse. Another problem is that Julia starts the analysis of a program from its main method while Android programs start from many event handlers, executed by a single thread. Hence, we had to modify Julia so that it starts the analysis from all such handlers, considering them as potentially concurrent entry points. Layout classes, such as views, menus and preferences, contain most or even all the code of an Android application, including its business logic. A complex problem is that, in Android, user interfaces are specified declaratively by XML files. This means that the code is not completely available in bytecode format, but is rather inflated, at runtime, from XML layout files into actual bytecode, by using Java reflection. Moreover, the link between XML inflated code and the explicit application code introduces casts and potential null pointer exceptions. Finally, a real challenge is the size of the libraries: Android programs use both the java.* and the new android.* hierarchies. Their classes must be analyzed along with the programs, which easily leads to analyze 10,000 methods and more.

2 Android Basics

Android applications are written in Java and made out of activities, or event-driven user interfaces. Programs do not have a single entry point but can rather use parts of other Android applications on-demand and ask services by calling their event handlers, directly or through the operating system.

An XML manifest file registers the components of an application. Other XML files describe the visual layout of the activities. Activities inflate layout files into visual objects (a hierarchy of views), through an inflater provided by the Android library. This means that library or user-defined views are not explicitly created by new statements but rather inflated through reflection. Library methods such as findViewById access the inflated views. As an example, consider the activity in Fig. 1 from the Google distribution of Android 2.1. The onCreate event handler gets called when the activity is first created, after its constructor has been implicitly invoked by the Android system. The setContentView library method calls the layout inflator. Its integer parameter uniquely identifies the XML layout file shown in Fig. 2. From line 3 of this file, it is clear