Information and Software Technology 54 (2012) 1192–1201

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Static analysis of Android programs

Étienne Payet [a,*], Fausto Spoto [b]

[a] LIM-IREMIA, Université de la Réunion, France
[b] Dipartimento di Informatica, Università di Verona, Italy

ARTICLE INFO

Article history:
Received 14 December 2011
Received in revised form 24 May 2012
Accepted 25 May 2012
Available online 1 June 2012

Keywords:
Program verification
Static analysis
Abstract interpretation
Android

ABSTRACT

Context: Android is a programming language based on Java and an operating system for embedded and mobile devices, whose upper layers are written in the Android language itself. As a language, 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.
Objective: Our goal is to extend the Julia static analyzer, based on abstract interpretation, to perform formally correct analyses of Android programs. This article is an in-depth description of such an extension, of the difficulties that we faced and of the results that we obtained.
Method: We have extended the class analysis of the Julia analyzer, which lies at the heart of many other analyses, by considering some Android key specific features such as the potential existence of many entry points to a program and the inflation of graphical views from XML through reflection. We also have significantly improved the precision of the nullness analysis on Android programs.
Results: We have analyzed with Julia most of the Android sample applications by Google and a few larger open-source programs. We have applied tens of static analyses, including classcast, dead code, nullness and termination analysis. Julia has found, automatically, bugs, flaws and inefficiencies both in the Google samples and in the open-source applications.
Conclusion: Julia is the first sound static analyzer for Android programs, based on a formal basis such as abstract interpretation. Our results show that it can analyze real third-party Android applications, without any user annotation of the code, yielding formally correct results in at most 7 min and on standard hardware. Hence it is ready for a first industrial use.

1. Introduction

Android is a main actor in the operating system market for mobile and embedded devices such as mobile phones, tablets and televisions. It is an operating system for such devices, whose upper layers are written in a programming language, also called Android. As a language, Android is Java with an extended library for mobile and interactive applications, hence based on an event-driven architecture. Any Java compiler can compile Android applications, but the resulting Java bytecode must be translated into a final, very optimized, *Dalvik* bytecode to be run on the device.

Static analysis of Android applications is important because quality and reliability are keys to success on the Android market [2]. Buggy applications get a negative feedback and are immediately discarded by their potential users. Hence Android programmers want to ensure that their programs are bug-free, for instance that they do not throw any unexpected exception and do not hang the device. But Android applications are also increasingly deployed in critical contexts, even in military scenarios, where security and reliability are of the utmost importance. For such reasons, an industrial actor such as Klocwork [16] has already extended its static analysis tools from Java to Android, obtaining the only static analysis for Android that we are aware of. It is relatively limited in power, as far as we can infer from their web page. We could not get a free evaluation licence.

A tool such as Klocwork is based on *syntactical* checks. This means that *bugs* are identified by looking for typical syntactical patterns of code that often contain a bug. The use of syntactical checks leads to very fast and practical analyses. However, it fails to recognize bugs when the buggy code does not follow the predefined patterns known by the analyzer. The situation is the opposite for *semantical* tools such as Julia, where bugs are found where the artificial intelligence of the tool, based on formal methods, has not been able to *prove* that a program fragment does not contain a bug. This second scenario is much more complex and computationally expensive, but provides a guarantee of soundness for the results: if no potential bug (of some class) is found, then there is no bug of that class in the code. In other terms, *syntactical* tools are fast

* Corresponding author.
E-mail addresses: etienne.payet@univ-reunion.fr (É. Payet), fausto.spoto@univr.it (F. Spoto).

but unsound. Both approaches signal false alarms, that is, potential bugs that are actually not a real bug. Precision (*i.e.*, the amount of real bugs *w.r.t.* the number of warnings) is the key issue here, since the number of false alarms should not overwhelm the user of the tool. This is acknowledged by most developers of static analysis tools. For instance, we can quote the web page of Coverity [7]: "By providing the industry's most accurate analysis solution and the lowest false positive rate, you can focus on the real and relevant defects instead of wasting development cycles". Hence, most of the effort of the developer of a static analyzer is towards the reduction of the number of false positives. This is much more difficult for sound analyzers, since they cannot just throw away warnings and nevertheless stay sound. In any case, the presence of a company such as Klocwork on this market shows that industry recognizes the importance of the static analysis of Android code.

A more scientific approach is underlying the SCanDroid tool [14], currently limited to security verification of Android applications. It performs an information flow analysis of Android applications, tracking inter-component communication through *intents* and the potential illegal acquisition of security privileges through a coalition of applications. Its basis is a constraint-based analysis of the code and there is a soundness guarantee, at least for a restricted kind of bytecodes. Klocwork does not currently perform any information flow analysis of Android applications.

Julia is a static analyzer for Java bytecode, based on abstract interpretation [6], that 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 it is typically the case in the abstract interpretation community: if the application contains a bug, of a kind considered by the analyzer, then Julia will report it. This makes the result of the analyses more significant. Although Java and Android are the same language, with a different library set, the application of Julia to Android is not immediate and we had to solve many problems before Julia could analyze Android programs in a correct and precise way. Many are related to the different library set, others to the use of XML to build part of the application. In this article, we present those problems together with our solutions to them and show that the resulting system analyzes non-trivial Android programs with high degree of precision and finds bugs in third-party code. This paper does not describe in detail the static analyses provided by Julia, already published elsewhere, but only the adaptation to Android of the analyzer and of its analyses. In particular, our class analysis, at the heart of *simple* checks such as classcast and dead code analysis, is described in [27]; our nullness analysis is described in [25,26]; our termination analysis is described in [28].

It must be stated that our Julia analyzer is not sound in the presence of reflection, redefinitions of the class loading mechanism of Java and multithreading. This does not mean that programs using those features cannot be analyzed, but only that the results might be incorrect. Actually, one main achievement of our work has been to teach Julia about the specific use of reflection that is done during the XML layout inflation in Android, so that the results of the analysis remain sound in that case (but not for other uses of reflection).

Our analyzer assumes a closed world assumption, in the sense that, for instance, it assumes that, at the entry points, variables might be bound to every class compatible with their declared type, might share in any possible way or hold null. The same assumption cannot be made for libraries, that can be expanded and whose behavior can be modified by subclassing. Hence ours is not a modular analysis for libraries since we only analyze complete (closed) Android applications.

There are many static analyzers that are able to analyze Java source code and find bugs or inefficiencies. Most of them are based on syntactical analyses (Checkstyle [4], Coverity [7], FindBugs [11,3], PMD [23]) or use theorem proving with some simplifying (and in general unsound) hypotheses (ESC/Java [12]). Since the Android language is Java, only the library changes, it might be in principle possible to apply those analyzers to Android source code as well. However, as we show in the next sections, there are new language features, such as XML inflation, that are not understood by those tools and that affect the same construction of the control flow graph of the program, usually performed through a type inference analysis known as *class analysis* [20]; there are many new kinds of bugs in Android code, because of the way the library is used, that are not typical of Java. Hence, either a static analyzer assumes that those features do not exist and those bugs do not occur (unsoundness) or must deal with them, possibly in a sound way. We think that the solutions that we highlight in this paper can be applied to those static analyzers as well, since they are not limited to our specific static analyzer of choice.

The rest of this paper is organized as follows. Section 2 justifies the difficulties of the static analysis of Android programs. Section 3 introduces the Android concepts relevant to this paper. Section 4 presents the more relevant static analyses that we performed on Android code. Sections 5–7 describe how we improved Julia to work on Android. In particular, Section 5 discusses the construction of a sound control-flow graph through class analysis, in the presence of XML inflation. Section 8 presents experimental results over many non-trivial Android programs from the standard Google distribution and from larger open-source projects; it shows that Julia found some actual bugs in those programs. Section 9 concludes the paper. This article is an extended version of a shorter conference paper presented at CADE in 2011 [22]. The full experimental evaluation of Section 8 does not appear in [22]. Moreover, Sections 4–6 provide a deeper presentation of the way we extended Julia, compared to the corresponding sections of [22].

Julia is a commercial product (http://www.juliasoft.com) that can be freely used through a web interface available from the web site of the company, whose power is limited by a time-out and a maximal size of analysis. Fausto Spoto is the chairman of the company, that he established in November 2010. He is also the main developer of the Julia software. Étienne Payet is currently an associate professor in Reunion (France). He is not a member of Julia Srl but he regularly collaborates with Fausto Spoto on scientific matters.

2. Challenges in the static analysis of Android

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. There are translators from Dalvik to Java bytecode (such as `undx` [24] and `dex2jar` [8]). But Android applications developed inside the Eclipse IDE [9] can always be exported in jar format, that is, in Java bytecode. Eclipse is the standard development environment for Android at the moment, hence we have preferred to generate the jar files 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. This is also a problem for some event-based Java programs, such as Swing programs using the `actionPerformed` event handlers. This is much more problematic for Android code, where the whole program works through event handlers that are often called through reflection, so that they might actually look like dead code to a static analyzer that does not understand reflection. Hence, we had to modify Julia so that it starts the analysis from all such handlers, considering them as potentially concurrent entry