Static Analysis of Android Programs

Étienne Payet
LIM-IREMIA, Université de la Réunion, France

Fausto Spoto
Dipartimento di Informatica, Università di Verona, Italy

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 cast, 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 minutes and on standard hardware. Hence it is ready for a first industrial use.

Keywords: Program verification, static analysis, abstract interpretation, Android

Email addresses: etienne.payet@univ-reunion.fr (Étienne Payet), fausto.spoto@univr.it (Fausto Spoto)
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 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 \cite{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, 6 and 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 \cite{22}. The full experimental evaluation of Sect. 8 does not appear in \cite{22}. Moreover, Sect. 4, Sect. 5 and Sect. 6 provide a deeper presentation of the way we extended Julia, compared to the corresponding sections of \cite{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 \cite{24} and dex2jar \cite{8}). But Android applications developed inside the Eclipse IDE \cite{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 deadcode 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 points. It must be stated that the Android event handlers are executed by a single thread, so we had not to consider the difficult problem of the analysis of multithreaded applications. Entry points are analyzed from a worst-case assumption, stating for instance that their parameters and receiver can belong to any class compatible with their static type, or can always be null or hold overlapping data structures.

A much more complex problem is the declarative specification of user interfaces through XML files, used by Android. 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. This problem must not be underestimated by thinking that layout code only contains graphical aspects, irrelevant to static analysis. Instead, in Android programs, XML-inflatable classes, such as views, menus and preferences, contain most or even all the code of an application, including its business logic. Moreover, the link between XML inflated code and the explicit application code introduces casts and potential null pointer exceptions. Hence, the analyzer must consider XML inflation in detail if we want it to be correct.

Finally, a real challenge is the size of the libraries: in general, Android programs use both a restricted java.* and the new android.* hierarchies. Their classes must be analyzed along with the programs, which easily leads to the analysis of 10,000 methods or more.

### 3. Android Basics

We describe here only the concepts of Android that are useful in this paper. For more information, see [1].

Android applications are written in Java and run in their own process within their own virtual machine. They do not have a single entry point but can rather use parts of other Android applications on-demand and can require their services by calling their event handlers, directly or through the operating system. In particular, Android applications contain activities (code interacting with the user through a visual interface), services (background operations with no interaction with the user), content providers (data containers such as databases) and broadcast receivers (objects reacting to broadcast messages). Event handlers are scheduled in no particular ordering, with some notable exceptions such as the lifecycle of activities.

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
public class LunarLander extends Activity {
private LunarView mLunarView;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // tell system to use the layout defined in our XML file
    setContentView(R.layout.lunar_layout);
    // get handles to the LunarView from XML
    mLunarView = (LunarView) findViewById(R.id.lunar);
    // give the LunarView a handle to a TextView
    mLunarView.setTextView((TextView) findViewById(R.id.text));
}
}

Figure 1: A portion of the source code Android file LunarLander.java.

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <com.example.android.lunarlander.LunarView android:id="@+id/lunar"
        android:layout_width="match_parent" android:layout_height="match_parent"/>
    <RelativeLayout
        android:layout_width="match_parent" android:layout_height="match_parent">
        <TextView android:id="@+id/text"
            android:text="@string/lunar_layout_text_text" android:visibility="true"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:layout_centerInParent="true" android:gravity="center_horizontal"
            android:textColor="#88ffffff" android:textSize="24sp"/>
    </RelativeLayout>
</FrameLayout>

Figure 2: The XML layout file lunar_layout.xml.

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 2.2. 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 the file in Fig. 2, it is clear that the view identified as lunar at line 9 of Fig. 1 belongs to the user-defined view class com.example.android.lunarlander.LunarView. The cast at line 9 in Fig. 1 is hence correct. Constants R.layout.lunar_layout and R.id.lunar are automatically generated at compile-time from the XML layout file names and from the view identifiers that they contain, respectively. The user can call setContentView many times and everywhere in the code; he can pass the value of any integer expression to it and to findViewById, although the usual approach is to pass the compiler generated constants. This declarative construction of objects also applies to preferences (graphical application options) and menus.
4. The Set of Static Analyses that we Apply

We describe here the main analyses that we let Julia apply to Android programs. The first six are relatively simple, compared to the last two. Except for the method redefinition checks, that are purely syntactical, they all exploit the class analysis, already performed during the extraction of the application, which computes an over-approximation of the control-flow graph of the program. Hence they exploit semantical, whole-program and inter-procedural information about the program.

These analyses are not new. They are very well-known static analyses of object-oriented code, already performed by tools such as Klocwork, Coverity and FindBugs, see their respective homepages. The novelty is their implementation inside a sound tool for Android code.

Equality Checks. Java programmers can compare objects with a pointer identity check $==$ and with a programmatic check $\text{equals}$. In most cases, the latter is preferred. But $==$ is used for efficient comparisons of interned objects, when the programmer knows that identity equality corresponds to programmatic equality. The use of both kinds of checks on the same class type is hence a symptom of a potential bug. Moreover, the use of $\text{equals}$ or $==$ on arrays is very likely a bug. In order to determine the classes $==$ and $\text{equals}$ are applied to, Julia uses an improvement of the 0-CFA class analysis defined in [20].

Classcast Checks. Incorrect classcasts are typical programming bugs. The introduction of generic types into Java has reduced the use of casts, but programmers still need them sometimes. Unfortunately, Android has actually introduced new situations where casts are needed, such as at lines 9 and 11 in Fig. 1. Julia applies its class analysis to prove casts correct. We had to consider the idiosyncracies of the Android library to keep this class analysis so precise to prove those casts correct.

Static Update Checks. The modification of a static field from inside a constructor or an instance method is legal but a symptom of a possible bug or, at least, of bad programming style. For this reason, we check when that situation occurs. We only do that inside the reachable code, by exploiting the class analysis computed by Julia.

Dead Code Checks. By dead code we mean here a method or constructor never invoked in the program and hence useless. This is often consequence of a partial use of a library but also the symptom of an actual bug. Spotting dead code is hence important for debugging. The identification of dead code is quite complex in object-oriented programs, since method calls have no explicit target but are resolved at run-time. Here, the class analysis of Julia comes to help again, by providing a precise static over-approximation of the set of run-time resolved targets. Android complicates this problem, since event handlers are called by the system,
implicitly, and since some constructors are invoked, implicitly, during the XML layout inflation.

**Inconsistent Definition of `hashCode` and `equals`**. Redefining one of those methods but not the other is very often a programming bug since it leads to unpredictable behaviors when the objects are stored into a collection. That situation is easy to check. Julia adds a more clever test that uses aliasing: it checks that a redefinition of `equals` definitely calls `super.equals` with the same arguments; in that case, missing to redefine `hashCode` is still safe.

**Method Redefinition Checks**. In Java, method redefinition may be a source of bugs when the programmer does not use the same name and argument types for both the redefining and redefined methods. This may happen as a consequence of incomplete renaming or incorrect refactoring. Similarly, the programmer might use an inconsistent policy while calling `super`, forgetting some of those calls. This check controls such situations.

**Nullness Checks**. In Java, dereferences occur when an instance field or an array is accessed, when an instance method is called and when threads synchronize. They must not occur on the special value null, or a run-time exception is raised. This is, however, a typical and frequent programming bug. Its automatic identification has been the subject of many articles, starting from the original work in [5]. Julia performs a very precise nullness analysis for Java, described in [25, 26]. Android complicates the problem, because of the XML layout inflation and of the use of the `onCreate` event handler to perform tasks, such as state initialization, that in Java are normally done in constructors. Hence the precision of the nullness analysis of Julia, applied to Android code, is not so high as for Java. For instance, it cannot determine that field `mLunarView` in Fig. 1 is non-null when it is dereferenced. Thus we had to improve its precision by considering some specific features of Android, as we describe in Sect. 6.

**Termination Checks**. A non-terminating program is often considered incorrect. Hence, termination analysis can be used during debugging to spot those methods or constructors that might not terminate. Many termination analyses have been defined for logic, functional and imperative programs, starting from [13]. Julia already performs termination analysis of Java code [28], has won the international competition of termination analysis for Java bytecode on July 2010 and was second in the same competition in 2011. The application of its termination analysis to Android code is challenging because of the size of the Java and Android libraries together.

5. Class Analysis for Android

Before a static analysis tool can analyze a program, the latter must be available and its boundaries clear. This might seem obvious, but it is not the case
for object-oriented languages. They allow dynamic lookup of method implementations in method calls, on the basis of the run-time class of their receiver. Hence, the exact control-flow graph of a program is not even computable, in general. An over-approximation can be computed instead, where each method call is decorated with a superset of its actual run-time targets. This is obtained through a class analysis that computes, for each program variable, field or method return value, a superset of the class types of the objects that it might contain at run-time. Some traditional class analyses are compared, formalized and proved correct in [10, 27]. In particular, Julia uses an improvement of the 0-CFA class analysis defined in [20]. The latter builds a constraint graph whose nodes are the variables, fields and method return values in the program. Arcs link these nodes and mimic the propagation of data in the program. The new statements inject class types in the graph, that propagate along the arcs. The receiver of a virtual method call uses a node that, once reached by a class tag, resolves the method from that class tag and starts the analysis for the resolved method. After propagation, each node over-approximates the set of classes for the variable, field or method return value that it stands for. Julia improves that analysis by deploying it in a flow-sensitive way. This means that the same local variable in a method body can have distinct approximations at distinct program points. This is important since Java compilers often recycle the same bytecode local variable for distinct source code local variables, for optimization purposes. But then, a flow-sensitive analysis is needed to distinguish the approximation of two conceptually distinct uses of the same bytecode local variable. In order to keep the cost of the analysis tractable, the same node of the graph is used when it is clear that the approximation of a local variable does not change from a program point to a subsequent one, as it is almost always the case.

Since the control-flow graph of the program is not yet available when the class analysis starts, the latter extracts the control-flow graph on-demand, starting from the main method of the program, during the same propagation of the class types. This is problematic for Android programs, that do not have a single main entry point, but many event handlers, that the system calls when a specific event occurs. They are syntactically identified as implementations overriding some method in the android.* hierarchy. Class analysis must hence start from all event handlers and use, at their beginning, a worst-case assumption about the state of the system: any class type may be bound to the receiver or parameters of the handler, as long as it is compatible with their static type. Moreover, the constructors of those classes (if these are not abstract) must be considered as entry points as well, recursively, since their instances might have been constructed by the operating system, by reflection.

This does not solve the problem of class analysis for Android programs yet. As we said above, new statements inject class types in the constraint graph. But, for instance, there is no new LunarView statement in the program in Fig. 1. So consider the LunarView object held in field mLunarView. How and where is it created? It turns out that Android does heavy use of reflection inside setContentView, to inflate graphical views from XML layout files. It instantiates the views from the strings found in the XML file, such as
com.example.android.lunarlander.LunarView at line 3 in Fig. 2. This can only happen through reflection, since the user can define new view names, as in this example. It is well known that class analyses are in general incorrect in the presence of reflection, but for the simplest ones. Here, we want to stick with Julia’s class analysis and we want it to work on Android code.

The first step in that direction has been to instrument the code of the library class android.view.LayoutInflater, that performs the inflation. Namely, Julia replaces reflection, there, with a non-deterministic execution of new statements, for all view classes reported in the layout files of the application. This makes the class analysis of Julia correct w.r.t. layout inflation. But we have a problem of precision here: both class types LunarView and TextView are computed for the return value of the findViewById calls in Fig. 1, since both class names occur in the layout file in Fig. 2 and hence two new statements are instrumented in the code of the inflator. This is correct but imprecise: we know that the first call yields a LunarView, while the second call yields a TextView, consistently with the constants passed to findViewById and with the identifiers declared in Fig. 2, at lines 3 and 7. Without such knowledge, specific to the way the Android library behaves, Julia will not be able to prove correct the two casts on the return value of findViewById in Fig. 1 and it will issue annoying, spurious warnings about apparently incorrect classcasts.

Thus, the second step has been to improve the precision of the class analysis of Julia, with explicit knowledge on the view identifiers. We introduced new nodes views(x) in the constraint graph, one for each view identifier x occurring in the XML layout files. Node views(x) contains a superset of the class types of the views labelled as x. Note that the same identifier x can be used for many views in the same or different layout files and this is why, in general, we need a set. Node views(x) is used for the return value of the findViewById(R.id.x) calls. Moreover, we build the arc

\{ name \mid x \text{ identifies a view of class } name \text{ in some layout file} \} \rightarrow \text{views}(x)

to inject into views(x) all class types explicitly bound to the identifier x. Since it is possible, although unusual, to set the identifier of a view explicitly, through its setId method, in that case we build an arc from the receiver of setId to all views(y) nodes, for every y that identifies a view whose class is consistent with the static type of the receiver of setId() (i.e., of the view whose identifier is modified). This is imprecise but correct. Moreover, we let (very unusual) calls findViewById(exp), for an expression exp that is not, syntactically, a constant view identifier, keep their normal approximation for the return value, containing all views referenced in the XML layout files. Again, this is imprecise but correct and only applies in very unusual situations. This same technique is used also for menus and preferences, that work similarly in Android.

We have performed other improvements to the class analysis of Julia, for better precision, although they are less important than those described above. For instance, we determine, precisely, the class type of the return value of method android.content.Context.getSystemService. The latter receives a
string $s$ as parameter and yields a service, such as a layout manager, a location service, a vibrator service etc. But that method is defined as returning a java.lang.Object, which requires a cast of its return value to the required service class. The correctness of these casts depends on $s$. The standard class analysis of Julia infers that the return value of getSystemService belongs to any service class, which is too imprecise to prove the correctness of such casts. Since this method is used often, this would induce Julia to issue many spurious classcast warnings. Hence, we have instructed Julia to check if $s$ is equal to one of the constant service strings defined in the Android library. In that case, the class of the return value of getSystemService is uniquely determined and the casts can be proved correct. We do this only if the user does not redefine getSystemService, since otherwise he might violate the contract on the class type of the returned service.

6. Nullness Analysis for Android

Julia includes one of the most precise correct nullness analyses for Java. It combines many distinct static analyses, all based on abstract interpretation. A basic analysis is strengthened with others, to get a high degree of precision [25, 26]. We can apply it to Android, without any modification. The results are precise, with some exceptions that we describe below, together with our solutions.

Consider Fig. 1. The nullness analysis of Julia, without any improvement, issues a spurious warning at line 11, complaining about the possible nullness of field mLunarView there. This is because Julia is not so clever to understand that the setContentView at line 7 inflates a layout XML file where a view identified as lunar exists, so that the subsequent findViewById call at line 9 does not yield null. Since this programming pattern is extensively used in Android, failing to cope with this problem would generate many spurious nullness warnings.

The nullness analysis of Julia includes, already, an expression non-nullness analysis that computes, at each program point, sets of expressions that are locally non-null. For instance, this analysis knows that if a check this.foo(x) != null succeeds, then the expression this.foo(x) is subsequently non-null, if foo does not modify any field or array read by foo itself. This local non-nullness is lost as soon as the subsequent code modifies a field or array read by foo. To check these conditions, Julia embeds a side-effect analysis of method calls. We exploited this analysis to embed specific knowledge on the setContentView method. Namely, after a call to setContentView(R.layout.file), we let the expression non-nullness analysis add non-null expressions that have the form findViewById(R.id.z), for every identifier z of a view reported in file.xml. These expressions remain non-null as long as no field or array is modified, that is read by findViewById (for instance, by a setId or another setContentView), but this is the standard way of working of our expression non-nullness analysis, so we had to change nothing for that and the correctness of this approach follows from the correctness of the expression non-nullness analysis that we have already in Julia.
This work removes the spurious warning at line 11 in Fig. 1, but it is not completely satisfactory. Namely, Julia has just proved field mLunarView non-null at line 11 in Fig. 1, but it is not necessarily able to prove the same at other program points, in other methods of LunarLander.java not shown in Fig. 1, since it does not know that the event handler onCreate is always called before any other event handler of the activity. This is a consequence of the activity lifecycle of Android. It specifies that activities are controlled by the operating system that calls their event handlers in a specified order. In particular, onCreate is called, first and always, just after the instantiation of the activity object by the system. The aim of the programmer here was to make mLunarLander always non-null, in the sense that it is never accessed before being initialized to a non-null value at line 9 and is never reassigned null later, during the life-cycle of the activity. (Fig. 1 does not show the whole activity code, but Julia checks it all.) This notion of globally non-null fields comes from [15] and is used by Julia as well [26]. It is important since it is less fragile than the local non-nullness of a field at a given program point, that can be easily broken by imprecise side-effect information. Moreover, it is important since it can be used by automatic type-checkers for nullness, such as [21]. But global non-nullness works for fields that are definitely initialized to a non-null value in all constructors of their defining class directly called in the program (hence not only through the constructor chaining mechanism of Java), and are never read before that moment. Method onCreate in Fig. 1 is not a constructor. Hence, Julia does not prove mLunarLander globally non-null, which typically leads to spurious warnings wherever it is dereferenced, outside onCreate.

The problem, here, is that Android engineers have introduced the onCreate event handler to put code that, in Java, would normally go into constructors. This comes with some drawback: mLunarView cannot be declared final, although, conceptually, it behaves so. (final fields can only be assigned in constructors.) More interestingly to us, Julia does not spot mLunarView as globally non-null, although it does behave as such. Our solution has been to instrument the code of the activities and give them an extra constructor whose code is

```java
public LunarLander(...) { this(); onCreate(null); }
```

That is, it calls the standard constructor of the activity, normally empty, and then the onCreate event handler, passing a null Bundle, exactly as it happens at activity start-up. Class LunarLander.java has two constructors now: the standard one, typically never used directly, and this extra one, that Julia uses to simulate the creation of the activity. They are syntactically distinguished by adding extra, dummy parameters to the instrumented constructor. This solves our problem: the instrumented constructor is now the only constructor called, directly, in the program, to create the activity. It makes mLunarView non-null. (onCreate becomes a helper function of the instrumented constructor, see [26].) Thus, mLunarView is correctly marked as globally non-null. The correctness of this technique follows from that of the general technique for proving fields globally non-null in Java, that we use here [26], since onCreate is always called after the construction of a view during XML inflation.
Note that this second technique does not replace the previous one on the local non-nullness of mLunarView at line 11. Instead, the two techniques are complementary: the first proves that a non-null value is written at line 9 of Fig. 1, the second proves that mLunarView keeps being non-null during the subsequent execution of the activity.

Even by using the technique described above, proving that a findViewById yields a non-null value is difficult when it occurs far away from setContentView (for instance, when it occurs in other methods than onCreate). This is because the imprecisions in the side-effects analysis and the worst-case assumption at the beginning of the event handlers typically erase any information on the non-nullness of distant findViewById(R.id.) calls. Hence, we have further improved our nullness analysis by exploiting information on the creation points of the receiver of each findViewById in the program. Those creation points are then compared with those of the receivers of the setContentView calls in the program, to determine an over-approximation $S$ of the setContentView's that might affect any given findViewById. If a view is defined in all layout files inflated in $S$, then the return value of findViewById is assumed as non-null. A creation points analysis was already performed by the nullness analyzer of Julia, hence we are not increasing the cost of the analysis here. We observe that this procedure is correct only if we are sure that at least a setContentView has been performed before the given findViewById is executed. To check this condition, we have used the same technique that we use to identify globally non-null fields, that must not be read before being assigned at least once. The correctness of this last technique follows from the correctness of our creation points analysis. It is a concretisation of the class analysis described and proved correct in [27], where instead of the simple class tag, the exact creation point of an object is tracked.

The use of a worst-case assumption at the beginning of the event handlers is sometimes a too pessimistic hypothesis, that leads to a large number of spurious warnings. Hence, for a few hundreds, frequently used event handlers, we have manually provided annotations that specify how they are called by the operating system. For instance, we provide the following annotations for two methods of the library class android.app.Activity:

```java
void onPrepareDialog(int id, @NonNullFromSystem Dialog dialog);  
void onPrepareDialog(int id, @NonNullFromSystem Dialog dialog, Bundle args);
```

stating that a non-null value is always passed by the operating system for the dialog parameter, but not for args. Without that information, the analyzer would signal a spurious null-pointer warning at the first dereference of dialog inside any implementation of those event handlers.

7. Termination Analysis for Android

Our termination analysis for Android is basically the same that we apply to Java [28]. It builds linear constraints on the size of the program variables.
This results in a constraint logic program whose termination is proved by the BinTerm tool [28]. For efficiency, Julia uses zones [18] for the linear approximation. It can also use polyhedra, but their cost is much higher and we have not experienced significant improvements in precision. This is probably due to the relative simplicity of the algorithms used in Android applications, at least in those that we have analyzed. Usually, loops over integer variables and unitary increments are used; loop limits are constraints between a single variable and a constant. More complex loops are typical of mathematical software, where polyhedra might show their power. BinTerm uses polyhedra anyway. For extra precision, we have defined the size of Android cursors as the number of elements that must yet be scanned before reaching their end. This lets Julia prove termination of the typical loops of Android code, where a cursor over a database is used to scan its elements.

We observe that our tool proves termination of loops and recursive methods. Most Android programs might diverge if the user or the system keep interacting with their event handlers. Our work does not consider this case of non-termination, which is typically always possible.

8. Experiments, False Alarms and Actual Bugs

Table 1 presents the result of our analyses of most sample programs in the Google distribution of Android 3.1 and of some larger open-source programs (Mileage 2.2.5, OpenSudoku 1.1.1, Solitaire 1.12.1 and TiltMazes 1.2 [17]; ChimeTimer, Dazzle, OnWatch and Tricorder [19]; TestAppv2 and TxWthr [29]).

We have used a Linux quad-core Intel Xeon machine running at 2.66GHz, with 8 gigabytes of RAM. The latest version of these experiments is available at http://julia.scienze.univr.it/runs/android/results.html, where it is possible to download the exact bytecode that we have analyzed, for better repeatability of the experiments.

The choice of those tests is a consequence of the fact that we wanted Android applications whose source code is available, since we must be able to verify which warnings are actual bugs and doing that on bytecode would require too much effort. Moreover, there is no standard set of benchmarks for the static analysis of Android. In particular, the work on SCanDroid does not report any experiment [14] and applies analyses different from ours. Hence, a threat to the validity of our experiments comes from the same selection of our tests, but we cannot find a representative standard alternative set of tests.

The notion used for precision in Table 1 is the most conservative we could think about: for instance, in the case of nullness analysis, the most precise analysis is an analysis that reports only the actual nullness bugs and no false alarm. This means that its precision (according to our metrics) is 100% if there is no actual nullness bugs and slightly below 100% otherwise. The same for class casts or non-terminating methods. This notion of precision is strictly related to the time spent by a programmer for checking the warnings. As we said in the introduction, a reduced number of false positives is acknowledged as a key aspect for a static analyzer.
We have manually checked all the warnings in Table 1. Most of them look to us as false alarms, but a definite answer is difficult, since we are not the authors of those programs. However, we recognized a few of them as actual bugs and we discuss them below. Note that we might not have spotted all warnings that are actual bugs, since there is no way to decide if a warning is definitely a bug or not.

8.1. Simple Checks

OpenSudoku defines the SudokuListActivity with an inner class implementing a setViewValue method and containing the snippet of code in Fig. 3. There, note is a local variable of type String and trim returns the string obtained by removing white spaces from the beginning and end of note; variable view is a parameter of setViewValue and constant View.GONE is used for setting a view invisible so that it does not use any layout space. Julia spots the test note.trim() == "" as a suspicious use of == instead of .equals. This is an actual bug since, when note.trim() is the empty string, the == check fails, the visibility of view is not set to View.GONE and view still uses some layout space.

All 12 warnings about suspicious == tests, found by Julia in TxWthr, look as actual bugs to us. These are tests such as if (widget.appname == "update"), where the constant string "update" is assigned to widget.appname in a different class and hence a distinct String object is used, according to the semantics of Java.

ApiDemos defines classes GLColor and GLVertex. They override equals but not hashCode. We have manually checked the two warnings issued there by Julia and found that they are actual bugs, that lead to inconsistent behaviors when instances of those two classes are used in collections.

8.2. Nullness Checks

Julia issues two warnings (among others) for the Mileage program:

FillUp.java:443: call with possibly-null receiver to insert
FillUp.java:445: call with possibly-null receiver to update

We have investigated those problems and found that Mileage defines the Model class that declares the field and the methods in Fig. 4. openDatabase creates a database object and stores its reference in m_db, while closeDatabase closes the database and resets m_db to null. The programmer’s intent was to put
<table>
<thead>
<tr>
<th>program</th>
<th>source lines</th>
<th>analyz. lines</th>
<th>simple checks</th>
<th>nullness</th>
<th>termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbdTest</td>
<td>490</td>
<td>56221</td>
<td>22.99</td>
<td>0</td>
<td>119.12</td>
</tr>
<tr>
<td>AccelerometerPlay</td>
<td>306</td>
<td>47128</td>
<td>10.73</td>
<td>0</td>
<td>7.23</td>
</tr>
<tr>
<td>ApiDemos</td>
<td>19110</td>
<td>156105</td>
<td>84.30</td>
<td>0</td>
<td>105.87</td>
</tr>
<tr>
<td>BackupRestore</td>
<td>315</td>
<td>57135</td>
<td>14.60</td>
<td>0</td>
<td>105.87</td>
</tr>
<tr>
<td>BluetoothChat</td>
<td>616</td>
<td>84543</td>
<td>23.07</td>
<td>0</td>
<td>248.52</td>
</tr>
<tr>
<td>ChimeTimer</td>
<td>1990</td>
<td>89700</td>
<td>28.42</td>
<td>0</td>
<td>265.65</td>
</tr>
<tr>
<td>ContactManager</td>
<td>347</td>
<td>87269</td>
<td>21.89</td>
<td>0</td>
<td>260.23</td>
</tr>
<tr>
<td>CubeLiveWallpaper</td>
<td>450</td>
<td>26003</td>
<td>3.65</td>
<td>0</td>
<td>32.61</td>
</tr>
<tr>
<td>Dazzle</td>
<td>1798</td>
<td>72127</td>
<td>27.43</td>
<td>0</td>
<td>149.62</td>
</tr>
<tr>
<td>GestureBuilder</td>
<td>502</td>
<td>84473</td>
<td>22.92</td>
<td>0</td>
<td>225.64</td>
</tr>
<tr>
<td>Home</td>
<td>870</td>
<td>87552</td>
<td>24.35</td>
<td>0</td>
<td>243.13</td>
</tr>
<tr>
<td>HoneycombGallery</td>
<td>948</td>
<td>69423</td>
<td>19.64</td>
<td>0</td>
<td>153.93</td>
</tr>
<tr>
<td>JetBoy</td>
<td>839</td>
<td>64384</td>
<td>15.57</td>
<td>0</td>
<td>129.88</td>
</tr>
<tr>
<td>LunarLander</td>
<td>538</td>
<td>57448</td>
<td>15.54</td>
<td>0</td>
<td>109.38</td>
</tr>
<tr>
<td>Mileage</td>
<td>5879</td>
<td>101412</td>
<td>32.05</td>
<td>1</td>
<td>379.73</td>
</tr>
<tr>
<td>MultiResolution</td>
<td>75</td>
<td>57997</td>
<td>15.57</td>
<td>0</td>
<td>110.42</td>
</tr>
<tr>
<td>NotePad</td>
<td>707</td>
<td>70460</td>
<td>17.39</td>
<td>0</td>
<td>151.75</td>
</tr>
<tr>
<td>OnWatch</td>
<td>6300</td>
<td>112583</td>
<td>40.92</td>
<td>1</td>
<td>419.98</td>
</tr>
<tr>
<td>OpenSudoku</td>
<td>5879</td>
<td>119332</td>
<td>39.07</td>
<td>5</td>
<td>218.09</td>
</tr>
<tr>
<td>Real3D</td>
<td>1230</td>
<td>74574</td>
<td>20.06</td>
<td>0</td>
<td>176.42</td>
</tr>
<tr>
<td>SampleSyncAdapter</td>
<td>978</td>
<td>65673</td>
<td>15.84</td>
<td>0</td>
<td>133.94</td>
</tr>
<tr>
<td>SearchableDictionary</td>
<td>343</td>
<td>87572</td>
<td>20.98</td>
<td>0</td>
<td>222.64</td>
</tr>
<tr>
<td>SkeletonApp</td>
<td>77</td>
<td>55824</td>
<td>12.86</td>
<td>0</td>
<td>101.68</td>
</tr>
<tr>
<td>Snake</td>
<td>420</td>
<td>56918</td>
<td>14.12</td>
<td>0</td>
<td>106.53</td>
</tr>
<tr>
<td>SoftKeyboard</td>
<td>705</td>
<td>56013</td>
<td>11.32</td>
<td>0</td>
<td>84.21</td>
</tr>
<tr>
<td>Solitaire</td>
<td>5916</td>
<td>61805</td>
<td>15.22</td>
<td>5</td>
<td>130.36</td>
</tr>
<tr>
<td>Spinner</td>
<td>85</td>
<td>56707</td>
<td>12.66</td>
<td>0</td>
<td>101.43</td>
</tr>
<tr>
<td>TestAppv2</td>
<td>377</td>
<td>56109</td>
<td>11.35</td>
<td>0</td>
<td>99.57</td>
</tr>
<tr>
<td>TicTacToe</td>
<td>607</td>
<td>58935</td>
<td>15.74</td>
<td>0</td>
<td>107.32</td>
</tr>
<tr>
<td>TiltMazes</td>
<td>1853</td>
<td>89789</td>
<td>23.97</td>
<td>0</td>
<td>240.84</td>
</tr>
<tr>
<td>Tricorder</td>
<td>5321</td>
<td>97755</td>
<td>33.98</td>
<td>0</td>
<td>319.45</td>
</tr>
<tr>
<td>TxWthr</td>
<td>2024</td>
<td>69900</td>
<td>20.27</td>
<td>12**</td>
<td>140.28</td>
</tr>
<tr>
<td>VoiceRecognition</td>
<td>54</td>
<td>22462</td>
<td>3.36</td>
<td>0</td>
<td>23.77</td>
</tr>
<tr>
<td>Wiktionary</td>
<td>420</td>
<td>108158</td>
<td>29.21</td>
<td>0</td>
<td>372.46</td>
</tr>
</tbody>
</table>

Table 1: Our experiments of analysis. source lines counts the non-comment non-blank lines of programmatic and XML code. analyzed lines includes the portion of the java.* and android.* libraries analyzed with each program and is a more faithful measure of the analyzed codebase. Times are in seconds. Those for simple checks include all such checks. Columns eq, cast, static, dead and hash refer to warnings issued by the first five analyses in Sect. 4 (method redefinition checks never issued any warning and are not reported). Column cast counts the casts that Julia could not prove safe, over the total number of casts in the program (0/x is the maximal precision, in the absence of cast errors). Column dead counts the constructors or methods, of the analyzed application, found as definite dead code by Julia. For nullness analysis, ws counts the warnings issued by Julia (possible dereference of null, possibly passing null to a library method) and prec reports its precision, as the ratio of the dereferences proved safe over their total number (100% is the maximal precision, if there is no nullness error). For termination analysis, ws counts the warnings issued by Julia (constructors or methods possibly diverging) and its precision, as the ratio of the constructors or methods proved to terminate over the total number of constructors or methods containing loops or recursive (100% is the maximal precision if all methods terminate). Asterisks stand for actual bugs in the programs.
protected SQLiteDatabase m_db = null;

protected void openDatabase () {
    if (m_db == null)
        m_db = SQLiteDatabase.openOrCreateDatabase(...);
}

protected void closeDatabase (Cursor c) {
    ...
    if (m_db != null)
        m_db.close();
    m_db = null;
    ...
}

Figure 4: Field m_db and methods openDatabase and closeDatabase defined in Mileage.

each database access operation inside an openDatabase/closeDatabase pair. Unfortunately, this does not work when database access operations are nested. This happens in the FillUp class, subclass of Model, whose save method (Fig. 5) calls calcEconomy. The latter, in turn, executes a database operation bracketed

public long save () {
    openDatabase();
    calcEconomy(); ...
    if (...)
        m_db.insert(...); // save a new record
    else
        m_db.update(...); // update an existing record
    closeDatabase(null); ...
}

Figure 5: Method save defined in Mileage.

inside the usual openDatabase/closeDatabase calls (not shown in the figure). Hence, after the call to calcEconomy inside save, m_db holds null and the calls m_db.insert and m_db.update inside save raise a NullPointerException.

Julia issues the following warning (among others) for BluetoothChat:

BluetoothChatService.java:397: call with possibly-null receiver to read

Line 397 is inside the inner class ConnectedThread of BluetoothChatService and contains bytes = this.mmInStream.read(buffer). Field mmInStream is initialized in the constructor of ConnectedThread, shown in Fig. 6. But, there, mmInStream and mmOutStream are set to null if there is some problem with the bluetooth interface and an IOException is thrown inside the try block. A similar bug has been found at line 253 of BluetoothChatService, in Wiktionary, OpenSudoku, Dazzle and Real3D.

Julia also reports the warning
public ConnectedThread(BluetoothSocket socket) {
    mmSocket = socket;
    InputStream tmpIn = null;
    OutputStream tmpOut = null;
    try { // Get the BluetoothSocket input and output streams
        tmpIn = socket.getInputStream();
        tmpOut = socket.getOutputStream();
    } catch (IOException e) {}    
    mmInStream = tmpIn;
    mmOutStream = tmpOut;
}

Figure 6: The constructor of class ConnectedThread in BluetoothChat.

while (true) {
    mGoalX = (int) (Math.random()*(mCanvasWidth-mGoalWidth)));
    if (Math.abs(mGoalX-(mX-mLanderWidth/2)) > mCanvasHeight/6)
        break;
}

Figure 7: A portion of method doStart defined in LunarLander.

BluetoothChatService.java:236: call with possibly-null receiver
to listenUsingRfcommWithServiceRecord

That line contains tmp = this.mAdapter.listenUsingRfcommWithServiceRecord(NAME, MY_UUID) and it turns out that field mAdapter is initialized at line 71, inside the constructor of BluetoothChatService, as

    mAdapter = BluetoothAdapter.getDefaultAdapter();

However, method getDefaultAdapter yields null on devices that do not feature a bluetooth adapter and this condition is not checked in the program.

8.3. Termination Checks

Most warnings issued by Julia about possibly diverging methods are false alarms. A few are actually diverging methods, that can in principle run for an indefinite time, as long as the user does not decide to stop a game or network connection. An interesting diverging method, although not actually a bug, is found in program LunarLander, whose inner class LunarView.LunarThread has a doStart method containing the snippet of code in Fig. 7. Julia spots this loop as possibly non-terminating: variable mGoalX is assigned a random value at each iteration of the while loop. In principle, that value might keep falsifying the condition of the if and the break statement might never be executed. Although this is statistically improbable, it is, at least, a case of inefficient use of computing resources.
9. Conclusion

This is the first sound static analysis framework for Android programs, based on a formal basis such as abstract interpretation. We have shown that it can analyze real third-party Android applications, without any user annotation of the code, yielding formally correct results in a few minutes and on standard hardware. Hence it is ready for a first industrial use. Formal correctness means for instance that programs such as VoiceRecognition in Table 1 are proved to be bug-free, \textit{w.r.t.} the classes of bugs considered by Julia.

The problems of the analysis of real Android software are far from trivial and we do not claim to have solved them all. For instance, Table 1 shows far from optimal precision for the nullness analysis of OpenSudoku and Solitaire, with a relatively high number of warnings. It turns out that those programs use arrays of references and Julia could not prove their elements to be non-null when they are dereferenced. The size of the analyzed code is also problematic. For instance, we could not perform the nullness and termination analyses of ApiDemos (Table 1) because they ran into out of memory. Hence, our tool still requires improvements \textit{w.r.t.} precision and efficiency and we are actively working at them.

We have recently extended our Julia static analyzer in order to analyze GWT and Play applications. The sad point about those extensions is that there is little in common between them, since most of the problems faced when analyzing a Java framework are different for each framework. Hence, we have actually provided a general plug-in mechanism for Julia, so that the developer of Julia can specify which are the entry points of a program in a given Java framework, how the code must be instrumented and how the XML files must be processed. This is not a definite solution to the problem of analyzing Java frameworks. In particular, the emergence of frameworks that do heavy use of annotations (such as Spring or Hibernate) introduces new challenges, since the semantics of the annotations is unknown to the analyzer.

References


