Static Analysis of Android Programs

Étienne Payet[1] and Fausto Spoto[2]

[1] IREMIA, Université de la Réunion, France
[2] Dipartimento di Informatica, Università di Verona, Italy

Abstract. Android is a programming language based on Java and an operating system for embedded or mobile devices whose upper layers are written in that 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. This article is an in-depth description of how we extended the Julia system, based on abstract interpretation, to run formally correct analyses of Android programs, of the difficulties that we faced and of the results that we obtained. Namely, we have analyzed with Julia the whole set of Android sample applications by Google and a few larger open-source programs. We have applied seven static analyses, including classcast, dead code, nullness and termination analysis. Julia has found, automatically, bugs and flaws both in the Google samples and in the open-source applications.

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.

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. For such reasons, an industrial actor such as Klocwork [7] 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. In any case, it must be stated that their tools are incorrect: if the analyzed program contains a bug, they 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, based on abstract interpretation [3], 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. However, 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. In this article, we present those problems and our solutions 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 their adaptation to Android. In particular, our class analysis, at the heart of *simple* checks such as classcast and dead code analysis, is described in [15]; our nullness analysis is described in [13, 14]; our termination analysis is described in [16].

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 seven static analyses that we performed on Android code. Sections 5, 6 and 7 describe how we improved Julia to work on Android. Section 8 presents experimental results over 25 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. Our analyzer is a commercial product (`http://www.juliasoft.com`). It can be freely used through the web interface at `http://julia.scienze.univr.it`, whose power is limited by a time-out and a maximal size of analysis.

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` [12]). But Android applications developed inside the Eclipse IDE [4] 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. 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.

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, layout 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 and precise.

Finally, a real challenge is the size of the libraries: in general, 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. Keeping the cost of a global, correct, precise and interprocedural static analysis under an acceptable threshold has been an actual feat, obtained after two months of software profiling.

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, 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 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 this file, 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

```
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));
  }
}
```

Fig. 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="visible"
    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>
```

Fig. 2. The XML layout file `lunar_layout.xml`.

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 Our Set of Static Analyses

We describe here the analyses that we let Julia apply to Android programs. The first five are relatively simple, compared to the last two. They all exploit the class analysis already performed during the extraction of the application (Sect. 5). Hence none of them is a simple syntactical check, but they all exploit semantical, whole-program and inter-procedural information about the program.

Equality Checks Java programmers can compare objects with a pointer identity check `==` and with a programmatic check `.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. In order to determine the classes `==` and `.equals` are applied to, Julia uses its very precise class analysis.

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 introduced new situations where casts are unavoidable, such as at lines 9 and 11 in Fig. 1. Julia applies its very precise class analysis to prove casts correct. We had to consider the idiosyncracies of the Android library to keep this class analysis precise for those casts (see Sect. 5).

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 precise 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 very precise 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.

Method Redefinition Checks Method redefinition in Java might 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. Julia performs a very precise nullness analysis for Java, described in [13, 14]. 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. Julia already performs termination analysis of Java code [16], and has won the latest international competition of termination analysis for Java bytecode on July 2010. 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, 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 formalized and proved correct in [15]. In particular, Julia uses an improvement of the very precise class analysis defined in [10]. The latter builds a constraint graph whose nodes are the variables, fields and method return values in the program. Arcs link these nodes and mimick the propagation of data in the program. The `new` statements inject class types in the graph, that propagate along the arcs. After propagation, each node over-approximates the set of classes for the variable, field or method return value that it stands for.

Since the control-flow graph of the program is not yet available when the class analysis starts, the latter *extracts* the program on-demand, starting from its `main` method, 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.

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. How is ever created the `LunarView` object stored into field `mLunarView`? It turns out that Android does heavy use of reflection inside `setContentView`, to *inflate* graphical views from layout XML 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 for reflection, but for the simplest ones. Here, we want to stick on Julia's precise 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, 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

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

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. 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 the one 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. 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 intepretation. A basic analysis is strengthened with others, to get a high degree of precision [13, 14]. 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 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 `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.

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. 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 [6] and is used by Julia as well [14]. It is important since it is less *fragile* than the local non-nullness of a field at a given program point, that can easily be broken by imprecise side-effect information. Moreover, it is important since it can be used by automatic type-checkers for nullness, such as [11]. 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 derefenced, 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 for 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 activities and give them an extra constructor whose code is

```
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 [14].) Thus, `mLunarView` is correctly marked as globally non-`null`.

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.

The use of `findViewById` far away from `setContentView` (possibly in other methods than `onCreate`) remains problematic despite what done above. 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.`z`)` 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` 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.

7 Termination Analysis for Android

Our termination analysis for Android is basically the same that we apply to Java [16]. 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. For efficiency, Julia uses zones [9] for the linear approximation. It can also use polyhedra, but their cost is much higher and we have not experienced significant improvements in precision. 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

Fig. 3 presents the result of our analyses of the sample programs in the Google distribution of Android 2.2 (ApiDemos–Wiktionary) and of some larger open-source programs (Mileage–TippyTipper [8]). We used a Linux quad-core Intel Xeon machine running at 2.66GHz, with 8 gigabytes of RAM. We have manually checked all the warnings in Fig. 3. 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.

8.1 Simple Checks

OpenSudoku defines the `SudokuListActivity` with an inner class implementing a `setViewValue` method and containing the snippet of code in Fig. 4.

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.

program	source lines	analyz. lines	acti-vities	ser-vices	prov-iders	rece-ivers	simple checks					nullness			termination		
							time	eq	cast	static	uncalled	time	ws	prec	time	ws	prec
ApiDemos	23134	163178	228	7	1	6	113.37	0	42/638	0	218	-	-	-	-	-	-
BackupRestore	393	60831	1	0	0	0	15.94	0	0/3	0	2	147.97	8	98.81%	62.58	2	0.00%
BluetoothChat	703	90307	2	0	0	0	21.84	0	3/14	0	0	300.01	34***	94.89%	129.34	2	33.33%
ContactManager	466	93015	2	0	0	0	25.94	0	1/20	0	0	331.67	8	97.62%	153.55	0	100.00%
CubeLiveWallpaper	414	34514	1	2	0	0	2.76	0	0/66	0	0	44.84	5	98.48%	21.95	0	100.00%
GestureBuilder	563	89972	2	0	0	0	22.38	0	3/23	1	1	279.49	20	94.74%	134.92	0	100.00%
Home	947	93213	2	0	0	0	24.83	0	2/23	3	3	412.24	45*	94.51%	157.26	3	62.50%
JetBoy	820	73997	1	0	0	0	17.78	0	0/31	0	0	181.58	27	98.54%	85.86	3	57.14%
LunarLander	613	61931	1	0	0	0	12.70	0	0/44	0	0	131.46	6	99.29%	65.40	3*	0.00%
MultiResolution	95	62437	1	0	0	0	13.72	0	0/3	0	0	134.00	0	100.00%	62.67	0	100.00%
NotePad	676	78275	4	0	1	0	18.18	0	0/13	0	1	208.95	4	99.60%	102.20	0	100.00%
SampleSyncAdapter	1266	67790	1	2	0	0	14.06	0	0/9	1	14	152.15	23	97.00%	79.39	2	60.00%
SearchableDictionary	429	93136	2	0	1	0	23.44	0	0/4	0	0	281.79	3	99.33%	138.20	1	0.00%
SkeletonApp	93	60045	1	0	0	0	13.10	0	0/3	0	0	143.06	1	98.11%	60.10	0	100.00%
Snake	445	61332	1	0	0	0	12.02	0	0/17	5	3	127.72	4	99.18%	65.53	1	90.00%
SoftKeyboard	779	58263	0	1	0	0	10.49	0	0/25	0	4	86.91	24	96.61%	52.83	0	100.00%
Spinner	118	64718	1	0	0	0	12.67	0	0/3	0	3	156.25	1	98.44%	71.48	0	100.00%
TicTacToe	624	63434	2	0	0	0	14.28	0	0/31	0	0	134.36	2	99.61%	68.98	1	85.71%
VoiceRecognition	71	33393	1	1	0	0	2.51	0	0/0	0	0	42.94	0	100.00%	21.93	0	100.00%
Wiktionary	600	116457	1	1	0	1	35.66	0	0/8	0	2	745.36	22*	95.10%	367.30	2	33.33%
Mileage	7253	111188	21	0	1	1	41.32	1	18/175	6	50	470.67	113*	98.50%	302.44	13	65.79%
OpenSudoku	6968	128216	10	0	0	0	56.50	2	27/276	0	58	410.19	240*	96.06%	573.37	7	88.52%
Solitaire	4440	66637	1	0	0	0	14.93	0	10/262	0	12	185.10	374	92.42%	160.38	10	86.49%
TiltMazes	2040	95591	2	0	0	0	26.84	0	0/64	0	6	285.65	28	99.06%	152.76	1	88.89%
TippyTipper	2437	68971	5	0	0	0	15.70	0	4/75	0	14	174.06	26	98.34%	83.10	0	100.00%

Fig. 3. 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. *activities*, *services*, *providers* and *receivers* count the Android components declared in the application's manifest file. Times are in seconds. Those for simple checks include all simple checks. Columns *eq*, *cast*, *static* and *uncalled* refer to warnings issued by the first four 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). Column *uncalled* counts the constructors or methods 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). 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). Asterisks stand for actual bugs in the programs.

```
if (note == null || note.trim() == "")
   ((TextView) view).setVisibility(View.GONE);
else
   ((TextView) view).setText(note);
```

Fig. 4. A portion of method `setViewValue` defined in OpenSudoku.

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. 5. `openDatabase` creates

```
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;
   }
   ...
}
```

Fig. 5. Field `m_db` and methods `openDatabase` and `closeDatabase` defined in Mileage.

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 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. 6) calls `calcEconomy`. The latter, in turn, executes a database operation bracketed 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
```

```
439   public long save() {
440      openDatabase();
441      calcEconomy(); ...
442      if (...)
443         m_id = m_db.insert(...);      // save a new record
444      else
445         m_db.update(...);             // update an existing record
446      closeDatabase(null); ...
447   }
```

Fig. 6. Method `save` defined in Mileage.

```
   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;
   }
```

Fig. 7. The constructor of class `ConnectedThread` in BluetoothChat.

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. 7. 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 and in OpenSudoku.

Julia also reports the warning

```
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.

Finally, for Home, Julia issues the warning

```
Home.java:383: call with possibly-null receiver to resolveActivity
```

and at that line we actually find `manager.resolveActivity(intent, 0)`. Variable `manager` is initialized at line 282 as

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

Fig. 8. A portion of method `doStart` defined in LunarLander.

```
PackageManager manager = getPackageManager();
```

The Android method `getPackageManager` can yield `null` sometimes, as we have verified by looking at its source code, but we could not find any documentation about that behavior. Some user actually got `null` as return value of `getPackageManager`. (See the discussion thread at [5], apparently never answered.)

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. 8. 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 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 Fig. 3 are proved to be bug-free, *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, Fig. 3 shows far from optimal precision for the nullness analysis of OpenSudoku and Solitaire, with an unacceptable 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 (Fig. 3) because they ran into out of memory. Hence, our tool still requires improvements *w.r.t.* precision and efficiency and we are actively working at them.

References

1. The Android Developers Website. `http://developer.android.com`.
2. The Android Market. `http://www.android.com/market`.
3. P. Cousot and R. Cousot. Abstract Interpretation: A Unifed Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.
4. `http://www.eclipse.org`.
5. `http://stackoverflow.com/questions/3675375/java-lang-nullpointerexception-at-android-content-contextwrapper-getpackagemanager`.
6. L. Hubert, T. Jensen, and D. Pichardie. Semantic Foundations and Inference of non-null Annotations. In G. Barthe and F. S. de Boer, editors, *Proc. of the 10th Int. Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 132–149. Springer, 2008.
7. `http://www.klocwork.com`.
8. `http://en.wikipedia.org/wiki/List_of_Open_Source_Android_Applications`.
9. A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Proc. of the 2nd Symposium on Programs as Data Objects (PADO II)*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172, Aarhus, Danemark, May 2001. Springer.
10. J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proc. of OOPSLA'91*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161. ACM Press, November 1991.
11. M. M. Papi, M. Ali, T. L. Correa, J. H. Perkins, and M. D. Ernst. Practical Pluggable Types for Java. In B. G. Ryder and A. Zeller, editors, *Proc. of the ACM/SIGSOFT 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*, pages 201–212, Seattle, WA, USA, July 2008. ACM.
12. M. Schönefeld. Reconstructing Dalvik Applications. Presented at the 10th annual CanSecWest conference, March 2009.
13. F. Spoto. The Nullness Analyser of Julia. In E. M. Clarke and A. Voronkov, editors, *Proc. of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, volume 6355 of *Lecture Notes in Artificial Intelligence*, pages 405–424, Berlin Heidelberg, 2010. Springer-Verlag.
14. F. Spoto. Precise null-Pointer Analysis. *Software and Systems Modeling*, 2011. To appear.
15. F. Spoto and T. Jensen. Class Analyses as Abstract Interpretations of Trace Semantics. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5):578–630, September 2003.
16. F. Spoto, F. Mesnard, and É. Payet. A Termination Analyzer for Java Bytecode Based on Path-Length. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(3):70 pages, March 2010.