Static Analysis of App Dependencies in Android Bytecode

-Technical Note-*
updated version 2.1 (12/2014)
updated version 2.0 (04/2014)
(version 1.0: 03/2012)

Steffen Dienst[1] and Thorsten Berger[2,1]

University of Leipzig, Germany
sdienst@informatik.uni-leipzig.de
[2]University of Waterloo, Canada
tberger@gsd.uwaterloo.ca

Abstract. Android applications (apps) are highly interactive, but have—by design—no facilities to declare dependencies reflecting such interactions. Dependencies are hidden in code and uncovering them requires static analysis techniques. This technical note presents our static analysis infrastructure to extract dependency information from Android (Dalvik) bytecode. This infrastructure is used in a study of variability mechanisms in software ecosystems [5]. In the present paper, we provide details on the implementation, our dataset, and the statistics we calculated.

1 Introduction

In this section, we provide details on our static analysis infrastructure that extracts dependencies from Android (Dalvik) bytecode. We describe our implementation, limitations, and how the resulting dataset should be interpreted.

1.1 Intent Mechanism

Android apps interact by instantiating data structures called *intents* and throwing them at runtime using certain API methods (see Table 2). This highly-dynamic facility for interaction gives rise to dependencies that are either soft (if the app handles missing targets dynamically) or hard (otherwise). However,

Android apps cannot declare such dependencies statically in their manifest; thus, detecting dependencies requires static analysis techniques.

Targets of interactions are individual components of apps (*Activities*, *Services*, *Broadcast Receivers*, or *Content Providers* [1]), which are described in the manifest. To be open for interaction, components are declared public by either setting an *export* flag or specifying an *intent filter*—*capabilities* (cf. main paper) advertised to the runtime. Intents classify into *explicit* (direct dependency) and *implicit* (capability-based dependency): explicit intents directly target components that have an export flag; implicit intents target capabilities, that is, components that have an intent filter. The runtime resolves implicit intents by matching them against all registered intent filter—requiring user interaction if several match, for example.

Intent Resolution Intent filters declare several *action* and *category* keys, together with a *data* specification (a complete or partial URI) that the corresponding component can handle. Intents contain *action* and *category* keys, and a *data* field (URI). They may also carry extra (but irrelevant for matching) information, such as key-value pairs (*Bundles*) and *flags*. An implicit intent matches an intent filter if its information is a subset of the intent filter's information. Thus, implicit intents can be seen as a minimal, and intent filters as a maximal specification of app capabilities. If an intent's `component` field is set using a fully-qualified class name, the intent is explicit and directly targets a concrete component.

Many *action* and *category keys* are predefined by the Android API, but in principal, arbitrary values can be used. Such third-party keys have to be documented and published, together with a specification (URI format) of expected data. Community efforts trying to establish intent registries emerged, such as OpenIntents [2].

Example Listing 1.1 shows an example of an intent filter adapted from Android's reference documentation [1]. It matches all intents instantiated in Listing 1.2.

Listing 1.1: Intent filter example

```
<activity android:name="DemoItemsList"
  <intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <action android:name="android.intent.action.EDIT"/>
    <category android:name="android.intent.category.
        DEFAULT"/>
    <data android:mimeType="vnd.android.cursor.dir/vnd.
        demo.item"/>
  </intent-filter>
</activity>
```

Listing 1.2: Intent instantiations

```
1 ...
2 Intent imp1 = new Intent("android.intent.action.VIEW");
3 imp1.addCategory("android.intent.category.DEFAULT");
4 imp1.setType("vnd.android.cursor.dir/vnd.demo.item");
5 ...
6 // reference a DataStore that provides data with mimetype
      vnd.android.cursor.dir/vnd.demo.item
7 Intent imp2 = new Intent("android.intent.action.EDIT");
8 imp2.addCategory("android.intent.category.DEFAULT");
9 imp2.setData(URI.parseUri("content://com.demo.android.
      provider.DemoItems/items");
10 ...
11 // explicitely reference the component
12 Intent exp = new Intent(this, DemoItemsList.class);
13 ...
```

2 Dataset

Android's main distribution channels are repositories with commercial and free applications (*app stores*). Google's official Android Market is by far the largest, but there are also many smaller app stores, such as Amazon Market[1], Opera Mobile Store[2], AndAppstore[3], Freewarelovers[4], Slide.me[5], or AndroidPit[6]. Online catalogs[7] list many more.

To analyze a large and sufficient selection of apps, we turned to the official Google Play market, which is usually accessed via the *Google Play* app on an Android device. We used an open source library[8] to download free and not *forward-locked* (a DRM mechanism) apps iteratively over a period of eleven months. Our current dataset contains 625,067 versions of 281,079 unique[9] unique apps. Due to license issues, we are not allowed to publish the whole Android dataset.

[1] http://www.amazon.com/apps
[2] http://mobilestore.opera.com
[3] http://andappstore.com
[4] http://www.freewarelovers.com/android
[5] http://slideme.org
[6] http://www.androidpit.de
[7] http://www.wipconnector.com/appstores
[8] http://code.google.com/p/android-market-api
[9] Due to daily downloads, we also gathered history for 39% of our apps; average number of versions: 3.9, maximum: 53.

3 Static Analysis of Android Bytecode

In this section, we provide details on our static analysis infrastructure that extracts dependencies from Android (Dalvik) bytecode. We describe our implementation, limitations, and how the resulting dataset should be interpreted.

3.1 Dataflow Analysis Implementation

We implemented an intra-procedural dataflow analysis ([8, 7, 6]) on top of the FindBugs[10] [3] framework. Findbugs is primarily used to find bug patterns in Java code, but provides useful facilities for any kind of static analysis, such as building and traversing a control flow graph (CFG), frame data analysis, or efficient fixed-point calculations. Since FindBugs expects Java bytecode, we converted the Android (Dalvik) bytecode to Java bytecode first, using dex2jar[11].

Java Bytecode Instructions Our goal is to identify instantiations of the `Intent` class and constructor parameters by tracing them back to constants using constant propagation techniques. The Java Virtual Machine (JVM) works stack-based: each method owns a stack frame, which consists of a (compiler-determined) number of slots. The first (bottom) slot holds the `this` (current object) reference. On top, the method parameters are stored. All other slots are used for local values. Each bytecode instruction may push or pop values on the stack. Thus, to find constant values being used in Intent constructor calls, we need to observe the effects of bytecode instructions on the stack slot values.

To illustrate relevant bytecode instructions, Listing 1.3 shows a sequence of instractions that originates from the intent call example in Listing 1.4, where the Google Play store is queried.

Listing 1.3: Bytecode of intent creation

```
1  ldc_w #372
2  invokestatic #255
3  astore_2
4  new #171
5  dup
6  ldc_w #381
7  aload_2
8  invokespecial #375
9  astore_3
10 aload_0
11 aload_3
12 invokevirtual #280
```

[10] http://findbugs.sourceforge.net/
[11] http://code.google.com/p/dex2jar

Listing 1.4: Java-Sourcecode of a Google Play Store Query

```
1 Uri marketuri = Uri.parse("market://search?q=pname:com.
      demo.android");
2 Intent intent = new Intent("android.intent.action.VIEW",
      marketuri);
3 this.startActivity(intent);
```

The bytecode instructions in Listing 1.3 provoke the following behaviour:

1. `LDC_W`: Load the string constant number `#372` from this class' constant pool and put it on the stack (e.g. `market://search?q=pname:com.demo.android`).
2. `INVOKESTATIC`: Invoke the static (class-scoped) function `android.net.Uri.parse()` with the string on top of the stack as parameter. Put the resulting `android.net.URI` object on the stack. `#255` is a reference to the method signature `<android/net/Uri/parse(Ljava/lang/String;)Landroid/net/Uri;>` in theconstant pool of this class.
3. `ASTORE_2`: Pop the current stack value (the `android.net.URI` object) and store in stack slot 2.
4. `NEW`: Create a new instance of the class `android.content.Intent` and push on stack. `#177` references the class `<android/content/Intent>`.
5. `DUP`: Duplicate the topmost stack value.
6. `LDC_W`: Load the string constant number `#381` and push on stack (the value in this example is `android.intent.action.VIEW`).
7. `ALOAD_2`: Load the value from stack slot 2 (the `android.net.URI` object) and push on stack.
8. `INVOKESPECIAL`: Call the constructor of the intent class with two parameters (one string and one URI). Consume those objects as well as the topmost intent reference from the stack. Since the intent reference was duplicated in line 5, we still have a reference on the stack. `#375` is again a constant referencing the method `<android/content/Intent/<init>(Ljava/lang/String;Landroid/net/Uri;)V>`
9. `ASTORE_3`: Store the reference to the intent object into slot 3.
10. `ALOAD_0`: Push the `this` reference on stack.
11. `ALOAD_3`: Load stack slot 3 and push on stack.
12. `INVOKE_VIRTUAL`: Invoke the (possibly overloaded) instance method `startActivity()` on the `this` instance with the intent as parameter value. `#280` references the method `<android/content/Context/startActivity(Landroid/content/Intent;)V>`

Dataflow Analysis Classic dataflow analysis traces the effects of code instructions according to a specific aspect, for example the impact on a variable. The first step is the construction of a CFG, whose nodes are sequences of uninterruptible code instructions (basic blocks), and whose edges represent any possible

control flow between basic blocks (normal transfer of control, but also stack unravelling exceptions).

To analyze the value of a variable based on constant propagation, the following functions have to be implemented to emulate the effects of basic blocks and of incoming edges in the CFG:

1. Getting the initial value of the variable
2. Simulating the effect of the basic block on the variable
3. Merging multiple variables, e.g. for loops

To obtain the variable values, a fixed-point algorithm traverses the CFG and applies the functions to each basic block until the variables (stack frame slots) do not change anymore. In this case, the stack frame slots contain the final variable values—given that they could be propagated from a constant.

3.2 Implementation

The variables we are interested in are the stack frame slots that are used as arguments for the constructor and various setter methods of an Intent instance. These arguments originate from the class' constant pool through propagation. We implemented the CFG functions as follows:

1. The stackframe on the entry block is empty.
2. We implemented effects of `LDC` and `LDC_W`, which load values from the class' constant pool; `NEW`, which instantiates (Intent) classes; `INVOKE_SPECIAL`, which calls the constructor on instantiation; and `INVOKE_VIRTUAL`, which invokes setter methods. Constant propagation is based on interpreting the `ASTORE_` and `ALOAD_` instructions. We also implemented heuristics for frequent patterns: Target components of explicit intents are often set by calling a class' `getName()` method; and `android.net.URI.parseUri()` is often used to construct URI instances. Finally, we implemented the effects of the various constructors (varying in terms of arguments) and all setter methods of the `Intent` class.
3. Merging two different stack frames is not implemented, since the effects can be too complex to emulate. Instead, if the two stack frames are different, which means that the loop modified the intent in arbitrary ways, we discard them—a conservative approach that avoids extracting incorrect values.

We run the fixed-point algorithm on the CFG that repeatedly executes all the functions on basic blocks. When a fixed-point is reached, we search for the API methods given in Table 2. We identify the intent instance among the method arguments and log all derived field values of this instance.

3.3 Discovering Intents

We implemented the CFG functions that emulate effects of bytecode instructions (see Sec. 3.1) as follows.

Table 1: API methods for throwing intents

Calls	Queries
android.app.Activity	android.content.pm.PackageManager
startIntentSender()	queryBroadcastReceivers()
startActivity()	queryIntentActivities()
startActivityForResult()	queryIntentServices()
startActivityFromChild()	getActivityIcon()
startActivityIfNeeded()	resolveActivity()
startService()	resolveService()
stopService()	
bindService()	
unbindService()	
sendBroadcast()	
sendOrderedBroadcast()	
sendStickyBroadcast()	
sendStickyOrderedBroadcast()	
android.content.Context	
stopService()	
bindService()	
unbindService()	
sendBroadcast()	
sendOrderedBroadcast()	
sendStickyBroadcast()	
sendStickyOrderedBroadcast()	
startService()	
registerReceiver()	

1. The stackframe on the entry block is empty.
2. The variables we are interested in are the slots of the stack frame; more precisely, references to constants from a class' constant pool that are used as constructor and setter arguments of an Intent instance—possibly propagated through many statements. We implemented the effects of the following bytecode instructions: `LDC` and `LDC_W`, which load values from the class' constant pool; `NEW`, which instantiates (Intent) classes; `INVOKE_SPECIAL`, which calls the constructor on instantiation; and `INVOKE_VIRTUAL`, which invokes setter methods. Constant propagation is based on interpreting the `ASTORE_` and `ALOAD_` instructions. Our analysis emulates the effects of these instructions on slot values of the stack frame for every basic block.

   We also implemented heuristics to deal with other frequently occurring patterns. First, we found that the target component of explicit intents is often set by loading a class and calling its `getName()` method. Second, the static method `android.net.URI.parseUri()` is often called to construct a URI instance.

   Finally, we implemented the effects of the various constructors (varying in terms of arguments) and all setter methods of the Intent class.
3. If a basic block is the target of outgoing CFG edges from two other basic blocks, the values of the corresponding stack frames need to be merged. Merging two different stack frames is not implemented, since the effects can become too complex to emulate. Instead, if the two stack frames are different (which means that the loop modified the intent in arbitrary ways), we discard

the stack frames altogether. This conservative approach might loose intents, but avoids extracting incorrect values.

Table 2: API methods for passing intents

Calls	Queries
`android.app.Activity`	`android.content.pm.PackageManager`
`startIntentSender()`	`queryBroadcastReceivers()`
`startActivity()`	`queryIntentActivities()`
`startActivityForResult()`	`queryIntentServices()`
`startActivityFromChild()`	`getActivityIcon()`
`startActivityIfNeeded()`	`resolveActivity()`
`startService()`	`resolveService()`
`stopService()`	
`bindService()`	
`unbindService()`	
`sendBroadcast()`	
`sendOrderedBroadcast()`	
`sendStickyBroadcast()`	
`sendStickyOrderedBroadcast()`	
`android.content.Context`	
`stopService()`	
`bindService()`	
`unbindService()`	
`sendBroadcast()`	
`sendOrderedBroadcast()`	
`sendStickyBroadcast()`	
`sendStickyOrderedBroadcast()`	
`startService()`	
`registerReceiver()`	

We run the fixed-point algorithm on the CFG that repeatedly executes all the functions on basic blocks. When a fixed-point is reached, we search for the API methods given in Table 2. We identify the intent instance among the method arguments and log all derived field values of this instance.

3.4 Counting Dependencies

Since our study [5, 4] analyzed several types of dependencies, we now provide details on how these numbers were acquired. Figure 1 presents the metamodel we introduced.

We calculated numbers for each association end as follows; assuring that our implementation leads to lower bounds of the actual numbers:

① *Direct dependencies* are represented by explicit intents—intent objects whose `component` field is set to a fully-qualified class name. We count all *unique*

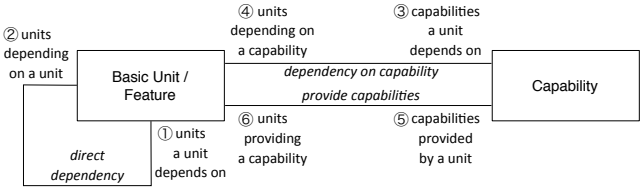

Fig. 1: Dependency metamodel [5]

explicit intents that have such a value and were found by our analysis. We implemented two heuristics to assure that these explicit intents actually target components of another instead of the same app. We omit intents if the target class is declared as a component in the own manifest file, or a physical class with the same name exists in the own app. These heuristics are justified by corner cases we found, for example, intents targeting internal components that are not declared in the manifest file.

② For the *reverse* direction of *direct dependencies*, we used the same procedure as for ①.

③ *Dependencies on capabilities* are represented by implicit intents. We count all *unique* (according to their values) implicit intents for which we could at least identify the action value. We implemented one heuristic to omit internal intents: we exclude intents with action keys that are defined in the own manifest file. However, we found corner cases where intent filters are created at runtime, which is possible via the Android API. Thus, internal intents could have been accidentally counted. We consider this issue a threat to validity, but believe this case being extremely rare.

④ For the *reverse* direction of *dependencies to capabilities*, we cannot give numbers. This calculation requires emulating the complex (and hidden in Android code) matching algorithm for intents. Further, the emulated matching would likely be inaccurate, given that our extractions might be incomplete and represent lower bounds.

⑤ For the number of *capabilities provided* per app, we count their unique intent filters. This measure is an approximation of capabilities (but still a lower bound), since intent filters can subsume other intent filters. Essentially, intent filters represent a partial order with regard to subsumption (intent filters can be specializations of others).

⑥ For the number of apps *providing a certain capability* (i.e. the reverse direction of ⑤), we counted the apps that declare each unique intent filter.

Limitations The intra-procedural character of our dataflow analysis imposes two conceptual limitations. First, we only catch intents that are both constructed and passed to an API call within the same method. Second, we cannot derive values that are the result of method calls. Our analysis only resolves (propagated) string constants, which can lead to missing values, for example, when the *action* key is constructed algorithmically or obtained from a configuration

service. However, we implemented some heuristics to deal with common string construction patterns, see Section 3.3.

Further, we cannot give reliable numbers for reverse dependencies, since we would have to emulate Android's intent matching algorithm (cf. Section 3.4, association end ③).

Given these limitations, we roughly estimated our recall by identifying all intent constructors calls in bytecode (`NEW`) and comparing them with those constructor calls for which our analysis identified at least the *action* key value. This recall is approximately 60%.

4 Other Statistics

App Size. In our study [5, 4], we also estimate the average lines of code (LOC) per app, with the following strategy.

We selected a random sample of 150 apps, which we converted from Dalvik to Java bytecode, in order to reconstruct source code with a Java decompiler[12] and measure using sloccount. As third-party libraries are directly compiled into apps, we also identified and subtracted their sizes by identifying duplicated code over the whole (281k apps) ecosystem subset (based on calculating hash codes for each Java package subfolder in the apps). Finally, using a median bootstrap analysis[13], the average app size excluding libraries amounts to 1,541 LOC (confidence interval of [1164,2239]). Interestingly, around 3/4 of an app's code belongs to libraries. Finally, we multiplied the average app size with a public estimation of currently available apps[14].

References

1. Android Developer's Guide (Sep 2011), http://developer.android.com/guide
2. OpenIntents. http://openintents.org (May 2011)
3. Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W.: Using static analysis to find bugs. IEEE Software 25 (2008)
4. Berger, T.: Variability Modeling in the Real. Ph.D. thesis, University of Leipzig, Faculty of Mathematics and Computer Science (2012)
5. Berger, T., Pfeiffer, R.H., Tartler, R., Dienst, S., Czarnecki, K., Wasowski, A., She, S.: Variability mechanisms in software ecosystems. Information and Software Technology 56(11), 1520–1535 (2014)
6. Cousot, P.: Abstract interpretation: Achievements and perspectives. In: Proceedings of the SSGRR 2000 Computer & eBusiness International Conference (2000)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. POPL '77 (1977)
8. Schwartzbach, M.: Lecture notes on static analysis http://www.brics.dk/~mis

[12] http://java.decompiler.free.fr/?q=jdgui

[13] http://reference.wolfram.com/mathematica/howto/PerformABootstrapAnalysis.html

[14] http://appbrain.com/stats/number-of-android-apps