Droidel: A General Approach to Android Framework Modeling

Sam Blackshear
University of Colorado Boulder, USA
samuel.blackshear@colorado.edu

Alexandra Gendreau
University of Colorado Boulder, USA
alexandra.gendreau@colorado.edu

Bor-Yuh Evan Chang
University of Colorado Boulder, USA
evan.chang@colorado.edu

Abstract

We present an approach and tool for general-purpose modeling of Android for static analysis. Our approach is to explicate the reflective bridge between the Android framework and an application to make the framework source amenable to static analysis. Our DROIDEL tool does this by automatically generating application-specific stubs that summarize the reflective behavior for a particular app. The result is a program with a single entry-point that can be processed by any existing Java analysis platform (e.g., Soot, WALA, Chord). We compared call graphs constructed using DROIDEL to call graphs constructed using a state-of-the-art Android model and found that DROIDEL captures more concrete behaviors.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords static analysis, framework modeling, soundness, reflection, Android

1. Introduction

Reflection is a notoriously thorny issue that most static analyses do not handle soundly [11]. Thus frameworks like Android that make heavy use of reflection pose problems for static analysis. Because the Android framework is complex and full of reflection, static analyses for Android typically choose to create models of the Android framework rather than analyzing the framework code itself. Creating these models is both tedious and error-prone, as it requires careful study of the framework's source code, documentation, and dynamic behavior. However, carefully crafted models are extremely important because an incomplete or incorrect model can compromise both the soundness and the precision of an analysis.

Since carefully crafted framework models are so important, we would hope that once a well-tested, authoritative framework model for Android has been created, all static analyses for Android would be able to re-use it. Unfortunately, to our knowledge, no such general framework model exists. The primary reason for this current state of affairs is that framework models tend to be *client-specific*—they summarize only the semantics of the framework with respect to a particular analysis client. This enables the framework model designer to abstract away the complex behavior of the framework code that is not relevant to the client of interest. For example, the FLOWDROID taint analysis tool [2] models calls to framework methods from application code via handwritten *taint wrappers* that summarize the framework's behavior for the taint analysis client. Although the creators of FLOWDROID have spent an immense amount of effort understanding and modeling the Android framework, their models cannot be readily reused by other analyses for Android. To see the problem concretely, consider this response on the Soot mailing list[1] from a FLOWDROID developer to a frustrated analysis designer who wishes to build a new analysis client on top of FLOWDROID:

> Question: "The call graph is missing edges. . . ."
>
> Response: "Another idea would be to just live with the incomplete call graph. . . . We know that we don't have call edges for some call sites. . . . You write that you do not want to perform taint tracking. In that case, the taint wrappers provided by FlowDroid will not be of much help."

Clearly, the Android static analysis community would benefit from a general model of the framework that is independent of any particular client and can be used with any program analysis platform. In this paper, we present an approach to fill this void and an implementation of this approach in the DROIDEL tool.

Android applications (apps) hook into the framework by extending special framework classes such as Activity or Service and overriding known callback methods such as `onCreate` or `onDestroy`. The framework executes an app by using reflection to look up the application classes that extend these special types and to invoke the appropriate callback methods in response to user interaction. In brief, DROIDEL works by explicating this reflection. That is, our approach to "the modeling problem" is to analyze the Android framework code itself, but to de-obfuscate the library's usage of reflection. DROIDEL does this de-obfuscation automatically by replacing reflective method calls with automatically generated app-specific stubs that invoke the appropriate app code.

The key observation underlying our approach is that most uses of reflection are simply to make the Android framework generic for all apps. DROIDEL takes advantage of this observation to create a non-reflective, app-specific version of the Android framework for each application it analyzes. The replacement of reflective calls and the generation of stubs is performed entirely at the Java source code level; the output of DROIDEL is a Java program with a single entry-point that can be processed by any existing Java analysis platform (e.g., Soot, WALA, Chord).

One of our contributions is the open-source DROIDEL implementation[2] following the approach advocated in this paper. DROIDEL is already being used by researchers from IBM Research, the University of Texas, the University of Maryland, and the University of Colorado for a wide variety of analyses including taint analysis, malware detection, permission analysis, and null dereference checking.

SOAP'15 June 14, 2015, Portland, OR, USA

Reprinted from SOAP'15, Proceedings of the 4th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis, June 14, 2015, Portland, OR, USA, pp. 19–25.

[1] `https://mailman.cs.mcgill.ca/pipermail/soot-list/2015-February/007745.html` and `007747.html`.

[2] `https://github.com/cuplv/droidel`

In the rest of this paper, we present an example showing how surprisingly subtle soundness issues can arise in framework modeling (Section 2), explain in detail how DROIDEL explicates reflection in the Android framework by generating stubs summarizing the behavior of key reflective method calls (Section 3), and demonstrate that call graphs constructed using DROIDEL-processed apps capture more concrete behaviors than a state-of-the-art model (Section 4).

2. Overview: Modeling Versus Explicating

In this section, we show how framework models can have subtle soundness problems. Even when models seem over-approximate for the analysis of interest, failure to fully model the execution context of application code can lead to surprising unsoundnesses.

Our approach in DROIDEL is to prevent soundness concerns by avoiding modeling whenever possible. Instead, we focus on explicating the parts of the Android framework that are polymorphic with respect to apps. By instantiating this polymorphism for a given app, we can eliminate difficult-to-analyze code and sources of unsoundness, such as uses of reflection.

Background: Modeling the Android Framework Since Android applications are event-driven, apps do not have a single `main` method for a static analysis to use as an entry point. Instead, control flow occurs primarily through callbacks. Applications create callbacks by extending special framework classes (e.g., Activity and Service) that expose known callback methods (e.g., `onCreate` and `onDestroy`). Apps override these callback methods and then register the extended class with the framework. Developers can register callback classes either at run time via a programmatic API or in the application manifest and application resources via special XML files that are packaged with the application. For example, the LoginActivity class in Figure 1a is application code that defines two callback methods `onCreate` and `onCancel` (highlighted in purple).

At run time, callbacks registered by the application are invoked by the framework both in response to user interaction and to signify changes in the *lifecycle* of core components. As an example of the first kind of callback invocation, when the user cancels the dialog `d` from Figure 1a (e.g., if the user presses the hardware "Back" button), the `onCancel` callback will be invoked. As an example of the second kind of callback invocation, when an instance of the LoginActivity class is launched by the Android operating system, its `onCreate` method will be invoked.

The Android framework code that allocates application objects and invokes callbacks is complex. It uses dynamic language features such as reflection to support configuration via XML files. To achieve reasonable precision and scalability in practice, nearly all Java static analyses choose to handle reflection in an unsound way [11]. Though compromising on reflection soundness often produces acceptable results for ordinary Java programs, adopting this strategy to analyze an Android app using framework code as entry points yields a useless call graph in which no application methods are reachable. This situation puts analysis developers in a bind: unsound reflection handling is necessary to achieve precision and scalability, but it is too unsound to understand the reflective bridge that connects the Android framework with apps.

Most static analysis tools for Android address this issue by abandoning analysis of the framework code altogether. Instead, tools typically generate an application-specialized *harness*, or a single "main" method that models the callback invocations performed by the framework. Figure 1b is an example harness for exercising the code in Figure 1a. The `loginActivityHarness` method models the lifecycle of a LoginActivity instance, and the `androidMain` method models the top level of the Android framework that manages the lifecycle of all components. Most models described in the literature (e.g., [2, 8, 12]) generate a harness similar to Figure 1b

```
   class LoginActivity extends Activity {
     AsyncTask mAuthTask = null;

     @Override void onCreate() {
1      mAuthTask = new AsyncTask(...);
2      AlertDialog d = ProgressDialog.create(...);
3      OnCancelListener l =
4        new OnCancelListener() {
5          @Override void onCancel() {
6            mAuthTask.cancel();
7          }
8        };
9      d.setOnCancelListener(l);
     }
   }
```

(a) A code snippet with two callback methods `onCreate` and `onCancel`.

```
   void loginActivityHarness() {
10   Activity a = new LoginActivity();
11   OnCancelListener l =
       new LoginActivity().new OnCancel();
12   a.onCreate();
13   while (*) {
14     if (*) { l.onCancel(); }
15     ... // other callbacks
16   }
17   a.onDestroy();
   }

   void androidMain() {
     while (*) {
       if (*) { loginActivityHarness(); }
       ... // other components
     }
   }
```

(b) A harness model for the app snippet in (a) with a subtle soundness bug.

Figure 1. Modeling: an Android app with a harness model.

using a procedure like the following: (1) parse the application manifest and resources to identify callback classes that will be instantiated by the framework (e.g., LoginActivity), (2) scan the application code to identify allocations of special callback classes (e.g., OnCancelListener), and (3) for each callback class (both framework-allocated and application-allocated), create a harness that allocates an instance of the callback class and invokes each of its known callback methods (e.g., `loginActivityHarness`).

Unsound Modeling Unfortunately, the harness model shown in Figure 1b is unsound for surprisingly subtle reasons.

When a LoginActivity is created, the application code from Figure 1a starts an asynchronous task to authenticate the user (line 1). It then sets up a progress dialog `d` to keep the user informed about the login process and a listener `l` to handle the case where the user wishes to cancel the login process (lines 2–9). If the user cancels the dialog, the `l.onCancel()` callback is invoked by the framework, which in turn cancels the running authentication task (line 7). Concretely, all of the code in this snippet is reachable and each statement can execute without throwing an exception.

The harness shown in Figure 1b models the Android framework by allocating instances of the callback-enabled types Activity and OnCancelListener (lines 10–11), which correspond to LoginActivity

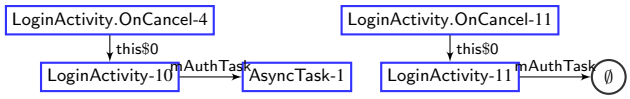

Figure 2. Points-to graph for the code snippet in Figure 1a using the harness in Figure 1b as an entry point. Unsoundness in the harness causes the points-to analysis to conclude that the mAuthTask field of LoginActivity-11 always points to null, which suggests the dispatch at line 7 of Figure 1a always raises a null pointer exception.

and LoginActivity.OnCancel (the anonymous inner class[3]). While the allocation of the LoginActivity.OnCancel class looks a bit odd, the harness needs an instance of the class in order to call its `onCancel` callback. Many tools model anonymous callback classes this way.

Because the anonymous inner class LoginActivity.OnCancel is a non-static inner class, it can (according to Java semantics) only be allocated alongside an instance of its parent class LoginActivity. Notice that the statement at line 11 of the harness actually allocates two objects—a parent LoginActivity object and its inner LoginActivity.OnCancel object. The remainder of the harness (lines 12–17) models the Android lifecycle for Activity objects, or specifically for LoginActivity objects (beginning with `onCreate` and ending with `onDestroy`). The uses of non-deterministic choice ($*$) over-approximate the event loop of the framework and model the possibility that user interactions (e.g., canceling the dialog) may or may not occur.

One subtle soundness issue in this framework model is that the harness allocates its own instance of the LoginActivity.OnCancel class rather than using the instance allocated in the application code (line 4 in Figure 1a). The harness only calls `onCancel` on its own instance (allocated at line 11 in Figure 1b); the `onCancel` callback is never called on the application instance (allocated at line 10). Correspondingly, the `onCreate` callback of the LoginActivity instance that is the parent of the harness instance of LoginActivity.OnCancel does not get called. Thus, a sound static analysis of this code could unsoundly conclude that `mAuthTask` is *always* null and thus the method call to `mAuthTask.cancel()` at line 7 in Figure 1a never executes.

This subtle soundness mistake in modeling can have a significant effect on analysis results. Consider running a 1-object-sensitive Andersen-style points-to analysis using an allocation-site based heap abstraction on the program in Figure 1a using the harness in Figure 1b as an entry point. We show the result of this analysis in Figure 2. The nodes in the graph are allocation sites numbered with the line number of the allocation. For example, LoginActivity.OnCancel-4 is a summary of the instances of the anonymous class LoginActivity.OnCancel allocated at line 4. A directed, labeled edge between a source node and a sink node means that at run time, an object in the concretization of the source node may point to an object in the concretization of the sink node through the field indicated by the label. Now, we see clearly in the points-to graph that the harness-allocated instance of LoginActivity.OnCancel is summarized by LoginActivity.OnCancel-11, and the mAuthTask of LoginActivity.OnCancel-11 can only ever hold the value null. We emphasize that this occurs even though the points-to analysis itself is sound—the problem is that the harness serving as the framework model for Android is unsound.

[3] The Java compiler typically give anonymous inner classes names like LoginActivity$1, but here we use a different naming convention for clarity.

The consequence of this modeling bug is that the effects of the call to the `cancel` method and its callees are (unsoundly) not taken into account by the analysis. This unsoundness can propagate. For example, a taint analysis tool like FLOWDROID wishes to soundly report all leaks of sensitive information to public sinks (e.g., to report all leaks of the user's contacts to the Internet), but if such a leak occurs in the `cancel` method or its callees, FLOWDROID will unsoundly miss the leak.

The Problem with Harness Models Our point in presenting this example is not that this specific modeling issue is particularly troublesome; it is a subtle problem, but can easily be fixed. Alternatively, this unsound model is acceptable as-is when using a less precise type-based points-to analysis like RTA [4] or VTA [17] that "hides" the unsoundness of the model—just like the control-flow constructs **while** and **if** in the harness are not needed if the static analysis is flow-insensitive. However, these coarser abstractions may not be precise enough to meet the needs of client analyses, and camouflaging the unsoundness of the model with a coarser abstraction addresses the symptom of the soundness issue rather than the cause.

The true issue is that the harness-based approach to modeling Android is problematic in general. Using a harness abstracts away intricate details of the framework that are difficult to recreate and may (unexpectedly) be important for soundness. We speak from personal experience—an early version of DROIDEL used a harness-based modeling approach that had many problems similar to the one described here. We discovered and fixed this particular unsoundness, only to discover more and more unsoundnesses also caused by abstracting away too many details about the execution context of the Android framework. DROIDEL's current approach of explicating reflection and analyzing the framework code directly (as described in Section 3) is motivated by this bad experience with "unsoundness whack-a-mole."

3. Design and Implementation of DROIDEL

We want DROIDEL to be a tool that can automatically transform an Android application into a form that can be analyzed by any Java program analysis tool. In this section, we explain how we designed and implemented DROIDEL to achieve this goal. Section 3.1 focuses on how an analysis designer can use DROIDEL, while Section 3.2 explains our implementation in detail for the benefit of analysis designers interested in adapting and extending our approach.

3.1 Designing DROIDEL for General Usability

In designing DROIDEL, we focused on the following two principles.

Model the framework as little as possible. Most existing approaches to analyzing Android applications explicitly seek to avoid analyzing the Android framework, but our approach is exactly the opposite. Each bit of framework code that is not analyzed must be carefully modeled to avoid introducing unsoundness (as we argued in Section 2). Instead of replacing the framework code with a large model, we choose to augment it with small models that minimally explicate the reflection and native code that the framework uses to interact with applications.

Be as standalone as possible. Modeling Android is hard work. We want others to benefit from our modeling efforts. In practical terms, this means that our model must be usable by any client analysis or program analysis framework in order to be widely adopted. To avoid being client-specific, we avoid abstracting away any Android framework code so that we do not eliminate any behaviors of potential importance. To avoid being analysis framework-specific, we generate all of our stubs and models at the Java source code level so they can be understood by any Java program analysis tool.

```
public interface DroidelStubs {
  // Reflective allocations of app objects
  Application getApplication(String cls);
  Activity getActivity(String cls);
  Service getService(String cls);
  BroadcastReceiver getBroadcastReceiver(String cls);
  ContentProvider getContentProvider(String cls);
  Fragment getFragment(String cname);
  View inflateViewById(int id, Context ctx);

  // Reflective method invocations
  void callXMLRegisteredCB(Context ctx, View v);
}
```

(a) DROIDEL generates app-specialized stubs that implement this interface.

```
class AppStubs implements DroidelStubs {
  Activity getActivity(String cls) {
    if (cls == "ActivityA") {
      return new ActivityA();
    } else if (cls == "ActivityB") {
      return new ActivityB();
    } else { return new Activity(); }
  }

  View inflateViewById(int id, Context ctx) {
    switch (id) {
      case R.id.passwordView:
        return new TextView(ctx);
      case R.id.tweetView:
        return new TextView(ctx);
      default: return null;
    }
  }

  void callXMLRegisteredCB(Context ctx, View v) {
    if (ctx instanceof ActivityA) {
      ((ActivityA) ctx).myOnClick(v);
    } else if (ctx instanceof ActivityB) {
      ((ActivityB) ctx).myOnClick(v);
    }
  }
}
```

(b) A partial implementation of `DroidelStubs` from (a) for an app with `ActivityA` and `ActivityB`, two TextViews, and an XML configuration-registered callback `myOnClick`.

Figure 3. We manually replace reflective calls in the Android framework with calls to `DroidelStubs` methods.

Note that our explicating approach is entirely compatible with additional modeling such as client-specific modeling (e.g., for increasing precision or improving the scalability of the analysis). We believe that starting from the framework source code and incrementally modeling key portions of the code is bound to lead to more trustworthy analysis results than beginning with a model that has no direct relationship to the framework source.

In building DROIDEL, we began by studying the source code for the Android framework and identifying uses of reflection that may allocate application objects or call methods on application objects. We then manually replaced each such use of reflection in the Android framework with a call to an appropriate method from the `DroidelStubs` interface shown in Figure 3a. This interface acts as a bridge between application and framework code—it allows the framework to obtain pointers to application-space objects. The `DroidelStubs` interface also centralizes and serves to document the instances of framework reflection that it explicates.

After replacing uses of reflection with calls to method stubs from `DroidelStubs`, we made one final change to the Android framework code: we changed `ActivityThread.main`, the "main" method that the Android framework uses to run an application, to take an implementation of the `DroidelStubs` interface as input.

The result is a slightly modified version of the Android framework that calls stubs from `DroidelStubs` rather than using reflection in several key places. This modified framework code can be compiled once and then used to analyze any application. The application-specific part of DROIDEL is generating an implementation of `DroidelStubs`, which we explain further in Section 3.2.

As the Android framework changes, future uses of framework reflection can easily be handled by adding new methods to this interface, updating the framework with calls to the new methods, and updating the application-specific part of DROIDEL to generate implementations of these new methods on a per-app basis.

***Analyzing an App with* DROIDEL** To enable whole-program analysis, DROIDEL creates a special `androidMain` method whose body allocates an instance of the auto-generated `DroidelStubs` implementation and calls the `ActivityThread.main` method with this object as its argument. An Android program analysis that wishes to use a DROIDEL-processed program need only import: (a) the application classes, (b) the DROIDEL-generated stub classes, and (c) the modified Android framework classes. The `androidMain` method can be used as a single entry point for whole-program analysis.

3.2 Implementation

There are two parts to DROIDEL: (1) a one-time manual modification of the Android framework sources to replace uses of reflection with calls to the appropriate methods of the `DroidelStubs` interface and (2) a per-app code generation module to automatically create an application-specific implementation of `DroidelStubs`.

Manually Explicating Reflection in the Android Framework To see a concrete example of replacing uses of reflection with calls to stub methods in `DroidelStubs`, consider the following snippet drawn from the `android.app.Instrumentation` class:

```
// Replace the use of reflection (a call to newInstance) with
// a call to the DROIDEL stub getActivity.
~~Activity a = (Activity) clazz.newInstance();~~
Activity a = droidelStubs.getActivity(clazz.getName());
```

We manually identified that the call to `clazz.newInstance()` might create an Activity object from the application, so we replaced this use of reflection with a call to `droidelStubs.getActivity`, a method of the `DroidelStubs` interface. For a method like `getActivity`, the DROIDEL implementation will generate allocations for each subclass of Activity defined in the application.

Application-Specific Stub Generation When DROIDEL runs on an app, it synthesizes an application-specific implementation of each of the stub methods of the `DroidelStubs` interface. To generate the getter methods for the core Android components Application, Activity, Service, BroadcastReceiver, ContentProvider, and Fragment, DROIDEL parses the application manifest `AndroidManifest.xml` for the app to determine which components have been declared by the developer and then builds the class hierarchy for the app and ensures that it can find each component.

To give an example of what an app-specific implementation of `DroidelStubs` would look like, we continue the discussion of the stub method `getActivity` from above. In Figure 3b, we show an implementation of `DroidelStubs` for an app with two subclasses of Activity (named `ActivityA` and `ActivityB`). The generated

implementation of `getActivity` simply dispatches based on the `cname` parameter. The documentation for `newInstance` states that this reflective call invokes the default (zero-argument) constructor for the given `Class`, so our stub methods allocate each type by invoking its zero-argument constructor. Generating the getter methods for the other core Android components is similar, though there is some special handling of Fragments because their usage has changed slightly as the Android framework has evolved.

Generating the `inflateViewById` stub is slightly different because Views are components of the Android layout instead of core Android components, so they are typically declared in resource files (e.g., `res/layout/filename.xml`) rather than in the application manifest. Additionally, View objects have associated identifiers that the app developer can use to distinguish between different layout components at run time. Understanding the association between View objects and their identifiers is crucial for a static analysis because View objects are frequently retrieved using these identifiers with methods such as `findViewById`. For example, the developer might have one TextView object with identifier `R.id.passwordView` and another TextView object with with identifier `R.id.tweetView`. It is important for the static analysis client to understand that calling `findViewById(R.id.passwordView)` will return a different TextView than calling `findViewById(R.id.tweetView)`, or else the state of the two View objects may be conflated.

Thus, DROIDEL parses all of the layout resource files to identify which View objects the app may use and to associate instances with their identifiers. It then generates stubs that model the reflective instantiation of View objects from the layout XML configuration file (called *layout inflation* in Android). For the simple two-TextView layout objects described above, DROIDEL would generated the stub implementation for `inflateViewById` shown in Figure 3b.

The functionality to generate these application-specific stubs is the core of DROIDEL. These stubs explicate Android framework's use of reflection to allocate application objects specified in XML configuration files. However, there is another tricky use of reflection in the Android framework. In Android apps, the developer can register callbacks either in the application code itself or (for certain callbacks) in the layout XML configuration file for the application. The first kind of registration is handled easily because its behavior is apparent in the framework code (i.e., does not use reflection), but the second kind of registration requires special treatment.

To give an example of the XML registration construct, suppose the application developer uses the layout XML configuration file to register a callback on a Button using the following snippet:

```
<Button android:onClick="myOnClick" ... />
```

The semantics of this XML snippet are that when the layout hierarchy containing this Button is attached to an Android Context object (e.g., via `Activity.setContentView`), the `myOnClick` method of that Activity will be registered as a callback to be invoked when the user clicks the Button. The lookup of the `myOnClick` method for a particular Context object is performed reflectively and thus must be explicated to be understood by the static analysis client. DROIDEL deals with this use of reflection by parsing the layout XML configuration file to identify XML-registered callbacks. It then generates an implementation of the `callXMLRegisteredCB` stub method that invokes each method of a Context subclass whose name matches the method name in the layout XML.

Let us assume that the two Activity classes from our running example, `ActivityA` and `ActivityB`, each have an `myOnClick` method with the proper signature for overriding the interface method `OnClickListener.onClick`. The `callXMLRegisteredCB` stub shown in Figure 3b corresponds to the implementation that DROIDEL would generate for this app. Since the layout hierarchy that registers the `myOnClick` method in the layout XML can be used in any `Context` object at run time, this way of generating stubs makes sure that every method matching `myOnClick` gets called.

Limitations There are many uses of reflection in the complex Android framework that DROIDEL does not (yet) explicate (for example, reflective allocation of Preferences objects). In addition, DROIDEL does not generate stubs to summarize the behavior of native methods in Android. Both of these issues are not fundamental problems with our approach, but limitations of the current implementation that we plan to address in the future.

Another issue is that we currently need to perform the manual explication of reflection in the Android framework separately for each version of the Android framework. We believe that this process can be automated in the future, as the explication that needs to be performed is almost identical for each version of framework we have considered. We note that the state of affairs is worse for harness-based approaches since the semantics of each version of the framework must be manually scrutinized in order to ensure that the generated harness over-approximates its behaviors.

4. Empirical Evaluation

We have explained the need for a general and versatile Android framework model and described how we developed DROIDEL to attempt to fill this void. In this section, we present empirical evidence to support the following claims:

1. DROIDEL *represents a viable approach to modeling the Android framework that captures more concrete behavior than current models.*

   We evaluate this claim with respect to sound call graph construction. A sound call graph should (at the minimum) over-approximate the set of concretely-reachable methods. But as discussed in Section 2, framework modeling can easily lead to subtle soundness issues.

2. *Incorporating the Android framework source as advocated by* DROIDEL *is a tractable option for baseline static analyses.*

   We evaluate this claim by considering the feasibility of call graph construction (which includes points-to analysis) using WALA, a commonly-used Java static analysis framework.

Experimental Setup On a set of open source Android applications, we constructed call graphs using FLOWDROID and DROIDEL and compared the missed concretely reachable application methods (Table 1). FLOWDROID is a state-of-the-art tool and Android framework model that was certified by the PLDI 2014 artifact evaluation committee, so it represents a stringent target for comparison. FLOWDROID does not aim to be a general purpose model—its harness generation aims primarily to improve the precision of flow-sensitive taint analysis. However, many developers are building analyses on top of the FLOWDROID model (cf. the Soot mailing list issue in Section 1).

Our experimental process had three steps: (1) first, we instrumented each app and manually exercised it for five minutes; (2) then, we constructed a call graph for each app using both FLOWDROID and DROIDEL; and (3) finally, we compare the number of concretely reachable *application* methods present in each call graph. We compared only the set of application methods reached to avoid unfairly penalizing FLOWDROID for modeling the Android framework rather than analyzing the actual framework code (as DROIDEL does).

To determine a set of concretely reachable application methods, we instrumented each app using ELLA[4] and manually exercised the

[4] `https://github.com/saswatanand/ella`

Table 1. Percent missed methods as a metric for determining how much app behavior is unsoundly missed. For dynamic exploration, we give the number of application methods (Total) and application methods visited (Visited), as well as the percentage of application methods manually executed (% Visited). For FLOWDROID and DROIDEL reachable methods, we show the number of visited methods present in the respective statically-constructed call graphs (Reachable) and the percentage of concrete methods that were missed (% Missed). The bottom row (**Summary**) contains the sum of all methods as well as the geometric means of the percent visited methods and percent missed methods.

Benchmark	**Dynamic Exploration of App Methods**			**Reachable methods (FLOWDROID)**		**Reachable methods (DROIDEL)**	
	Total	Visited	% Visited	Reachable	% Missed	Reachable	% Missed
drupaleditor	325	90	28	78	13	88	2
spycamera	254	156	61	40	74	151	3
npr	341	96	28	76	21	90	6
duckduckgo	935	520	56	352	32	449	14
textsecure	4459	1364	31	925	32	1141	16
wordpress	5796	2042	35	1362	33	1961	4
k9	5357	1905	36	1267	33	1773	7
Summary	17467	6173	**38**	4120	**30**	5653	**6**

app. We chose to manually exercise the apps under test since many of them require user logins, API keys, and other features that present problems for automated exploration tools (e.g. DYNODROID [13], A^3E [3], GUITAR [1]).

We exercised each instrumented app on a Nexus 4 phone running Android version 4.4.4 (KitKat) for five minutes, a time we found sufficient to explore the core features of the application and execute many of its methods. On average, we visited 38% of the total app methods instrumented by ELLA (see Table 1). We felt that this was reasonable coverage given that many apps have complex features that are difficult to exercise and may also contain unreachable methods.

Next, we ran FLOWDROID and DROIDEL on each app under test using the same version of Android. We used the final result of FLOWDROID's incremental call graph construction using the default options configured by FLOWDROID, except with the `NoCodeElimination` flag set to true to avoid any pruning of the call graph. We used WALA to construct a call graph with DROIDEL's `androidMain` as an entry point (see Section 3) using the most precise call graph construction algorithm built into WALA (ZeroOneContainer-CFA).

Claim: The* DROIDEL *Approach is Effective for Sound Android Modeling We compared the set of application methods visited during dynamic exploration with the set of reported reachable by the two static analyses (Table 1). On the seven apps we tested, FLOWDROID missed an average of 30% of the dynamically visited methods, whereas DROIDEL missed only 6%. While DROIDEL still missed some dynamically visited app methods, it missed nearly 25% fewer than FLOWDROID and less than 10% overall. This provides supporting evidence for our claim that DROIDEL is a viable approach to sound modeling the Android framework.

Claim: Analyzing the Android Framework is Tractable A frequently cited reason for modeling the Android framework rather than analyzing the source is the size of the framework source code. Since DROIDEL's philosophy is to avoid modeling by analyzing the framework source code directly, one might suspect that constructing a call graph using DROIDEL would be intractable. Table 2 shows that even for fairly large applications (up to 55K LOC), the call graph construction in WALA takes less than five minutes. Though DROIDEL takes longer to construct a call graph than FLOWDROID in almost every case, the reason why is clear: DROIDEL must analyze the 1M+ LOC of the Android framework, whereas FLOWDROID does not. The time taken by DROIDEL is still reasonable and (we feel) a fair price to pay for enhanced soundness.

Table 2. The size of each benchmark (**KLOC**), the time taken to run DROIDEL (**Stub**), to construct the call graph using DROIDEL/WALA (**Graph**), and to construct the call graph using FLOWDROID. All experiments were run on a MacBook Pro with a 2.6 GHz Intel Core i5 processor and 16 GB of memory.

Benchmark	**KLOC**	**DROIDEL**		**FLOWDROID**
		Stub (s)	**Graph** (s)	**Graph** (s)
drupaleditor	3	21	112	43
spycamera	3	19	89	15
npr	5	22	206	19
duckduckgo	10	28	121	48
textsecure	38	48	121	296
wordpress	47	35	220	113
k9	55	35	276	87

Threats to Validity It is well-understood that using a less precise call graph construction algorithm can yield a call graph containing more reachable methods. To try to avoid this effect, we use precise off-the-shelf construction algorithms. For DROIDEL, we use ZeroOneContainer-CFA, which (as stated above) is the most precise algorithm built into WALA. For FLOWDROID, we use the default call graph configuration as configured by the FLOWDROID taint-analysis tool (except for setting `NoCodeElimination` as discussed previously).

We note that both FLOWDROID and DROIDEL unsoundly miss concretely reachable methods in the evaluation. A fundamental problem with unsoundness is that it is difficult to diagnose why a model is unsound—is it because it models some feature incorrectly, or is it because it fails to model some feature altogether? In ongoing work, we are developing general techniques for automatically identifying unsoundnesses in framework models and diagnosing the root cause. In the absence of such techniques, our strategy for ensuring soundness so far has been to use the framework as much as possible and to model as little as possible in order to minimize the surface area for modeling mistakes.

5. Related Work

Modeling the Android Framework for Static Analysis Previous work has developed numerous techniques for modeling various features of Android. SCANDROID [8] and FLOWDROID [2] were the first static analysis tools to consider modeling the event-driven

lifecycle of the core components in Android. DROIDSAFE [9] attempts to analyze some of the framework code while replacing other parts with *accurate analysis stubs* that summarize framework behavior with respect to tainting and points-to analysis. These stubs are correct for points-to and taint analysis, but abstract away other behaviors of the framework. The GATOR tool [15, 18] of Yang et al. and the SMARTDROID [19] tool focus on precisely modeling the control-flow not only between lifecycle callbacks, but also between callbacks registered to GUI components. COMDROID [6], EPICC [14], and APPOSCOPY [7] specialize in modeling the Intent mechanism that Android uses to implement *inter-component-communication* between core components of a single app and (in some cases) between core components of different apps on the same device.

Though not all of these approaches explicitly reify a harness modeling the application callbacks invoked by the Android framework, they are all (to the best of our understanding) *harness-based* in the sense that they model the invocation behavior of the framework by considering a hard-coded set of callback methods to be entry points for analysis. By contrast, DROIDEL works by explicating reflection in the framework and then analyzing the framework code to allow the analysis itself to determine what callbacks may be invoked.

Handling Java Reflection As mentioned previously, reflection is a challenging feature for static analyses to handle soundly both in Java and in other languages [11]. Several approaches to handling Java reflection more soundly have been proposed. Tamiflex [5] uses dynamic analysis to observe the targets of reflective method calls at run time, then uses this information to generate reflective summaries that are sound with respect to the observed concrete behavior. The solution offered by Tamiflex is much more general than our Android-specific reflection handling, but the instrumentation Tamiflex performs does not work with Android applications.

Recent work by Li et al. [10] and Smaragdakis et al. [16] present promising new approaches to fully-static resolution of reflective calls in Java. Both techniques leverage meaningful operations performed on the return value of reflective calls (such as downcasts) to provide a more sound handling of reflection without compromising scalability.

6. Conclusion

We have presented an approach for modeling the Android framework that explicates the reflective bridge between applications and the framework. Built using this philosophy, DROIDEL attempts to avoid unsoundnesses and client-specific design choices by avoiding direct modeling in favor of analyzing the framework source code. We have found DROIDEL to be a useful tool in our own research, and we hope that DROIDEL can evolve as a community-driven building block for many other Android static analyses going forward.

Acknowledgments

We thank Manu Sridharan, the anonymous reviewers, and DROIDEL's early adopters for their helpful comments. We also thank Emily Kowalczyk and the Maryland-Colorado APAC team for useful discussions. This material is based on research sponsored in part by DARPA under agreement number FA8750-14-2-0039, the National Science Foundation under CAREER grant number CCF-1055066, and a Facebook Graduate Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of Android applications. In *Automated Software Engineering (ASE)*, 2012.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Programming Language Design and Implementation (PLDI)*, 2014.

[3] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2013.

[4] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1996.

[5] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *International Conference on Software Engineering (ICSE)*, 2011.

[6] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[7] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *Foundations of Software Engineering (FSE)*, 2014.

[8] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated security certification of Android applications. Technical Report CS-TR-4991, University of Maryland, College Park, 2009.

[9] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of Android applications in DroidSafe. In *Network and Distributed System Security (NDSS)*, 2015.

[10] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for Java. In *Object-Oriented Programming (ECOOP)*, 2014.

[11] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2), 2015.

[12] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Computer and Communications Security (CCS)*, 2012.

[13] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for Android apps. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 2013.

[14] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security*, 2013.

[15] Atanas Rountev and Dacong Yan. Static reference analysis for GUI objects in Android software. In *Code Generation and Optimization (CGO)*, 2014.

[16] Yannis Smaragdakis, George Kastrinis, George Balatsouras, and Martin Bravenboer. More sound static handling of Java reflection. Technical report, 2014. doi: 10.5281/zenodo.12729.

[17] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2000.

[18] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *International Conference on Software Engineering (ICSE)*, 2015.

[19] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. SmartDroid: an automatic system for revealing UI-based trigger conditions in Android applications. In *Security and Privacy in Smartphones and Mobile Devices (SPSM@CCS)*, 2012.