Static Taint Analysis Applied to Detecting Bad Programming Practices in Android

Sergio Yovine[1] and Gonzalo Winniczuk[2]

[1] Universidad ORT Uruguay
yovine@ort.edu.uy
[2] Universidad de Buenos Aires

Abstract. Frameworks and Application Programming Interfaces (API) usually come along with a set of guidelines that establish good programming practices in order to avoid pitfalls which could lead, at least, to bad user experiences, but also to program crashes. Most often than not, such guidelines are not at all enforced by IDEs. This work investigates whether static taint analysis could be effectively used for automatically detecting bad programming patterns in Android applications. It presents the implemented tool, called CheckDroid, together with the preliminary experimental evaluation carried out.

1 Introduction

Since its introduction in 2008, Android has been boldly increasing its share of the smartphone market. During the last 4 years, Android has consistently captured between 80% to 90% of the worldwide smartphone volume[3]. Besides smartphones, Android is used as software platform of wearable technology, such as watches, in-car entertainment, and embedded devices in IoT applications in a variety of domains, such as retailing. In other words, Android-based devices are pervasive in our daily life.

Therefore, it is very important to make sure that Android applications are correct with respect to their functional and non-functional requirements. That is, applications must provide the functionality they have been programmed for and they must do it through an efficient use of the available resources, such as memory, energy and processing power.

Carelessly coded applications are likely to be unreliable and inefficient. Thus, the documentation of the Android API exhibits a number of recommended coding practices which should be respected by the software developer. Consistently and carefully applying these rules is key to avoid bad user experiences, such as frozen screens and Application Not Responding (ANR) messages, poor performance caused by memory leaks, unexpected faults causing application crashes and loss of data, etc.. However, despite the existence of such recommendations, current Android IDEs do not provide any means to enforce them. Besides, because of the intrinsic asynchronous behavior of the Android runtime, faults originated in not respecting the programming guidelines are very difficult to capture

[3] https://www.idc.com/promo/smartphone-market-share/os

during testing and debugging. Hence, it makes sense to resort to static analysis to verify whether a program complies with the recommendations.

There is a significant amount of research effort devoted to developing static analysis techniques for Android. A thorough review of the state-of-the-art is done in [9]. A major conclusion of [9] is that Android code inspection through static analysis is mainly addressed to seeking privacy and security vulnerabilities [11]. Regarding performance issues, most effort has been put in developing tools for searching resource (memory, energy, ...) leaks [8], estimating memory usage [2,5], and testing applications for poor responsiveness [10, 17]. An important observation reported in [9] is that very few approaches provide path-sensitive analyses. FlowDroid [1] is one of the exceptions.

In fact, many of the performance-related bugs turn up to be consequences of bad programming practices in the first place. Therefore, it makes sense to automatically scrutinize application code early in the development cycle to clean it up of bad patterns which could be susceptible of causing runtime defects. Certainly, more specific analyses could be used afterwards for in-depth verification.

The question of investigating the correct application of programming guidelines is related to the problem of appropriately using APIs. However, works on the latter direction do not focus on analyzing program compliance with documentation guidelines, like ours. On the contrary, they seek improving API documentation in order to helping programmers using the API correctly. For instance, [14] studies programming obstacles derived from badly documented APIs, [15] seeks discovering API-usage patterns by mining client code, and [16] pursues the same purpose through the analysis of posts in developer forums.

To the best of our knowledge, only [12] attacks the same research question. The approach developed in [12] requires the Java source code of the application and relies on Android Lint [6]. This tool is an Android Studio utility that scans project sources checking for a number of common mistakes and structural issues. Android Lint checks for layout problems, unused resources, hard-coded strings, icon issues (e.g., duplications, wrong sizes), usability problems (e.g., untyped text fields), manifest errors, deprecated elements, etc.. The tool developed in [12] is limited to searching for some specific bad practices related to memory leakage and performance slowdowns due to inappropriate management of thread priorities and system objects.

In contrast, we devised a customizable approach based on formal path-sensitive program analysis. The cornerstone idea of our technique consists in relating bad practices with paths in the code which can be found by static taint analysis [13]. To evaluate the practical applicability of the idea, we developed the tool CheckDroid which steps on FlowDroid [1], a state-of-the-art static taint-analysis framework which analyzes the *.apk*, instead of the application source code, and takes into account the application life-cycle, providing higher precision than Android Lint.

Our approach does not rely on the source code, but on the *.apk*. Nevertheless, having the source code available, may be useful for eliminating false positives, that is, situations which are not actual violations of the guidelines, but reported

by CheckDroid as potential ones, as a consequence of the abstractions made by the underlying static analysis tool.

We carried out a preliminary experimental evaluation of CheckDroid on a set of Android applications drawn from different sources. To start up with, we run CheckDroid on applications developed by undergraduate students as their final project for a mobile computing course at the Department of Computer Science of University of Buenos Aires. Then, we analyzed applications published on Google Play. The experiments showed that recommendations related to performance and memory usage were the most commonly violated ones.

Outline. The rest of the paper is organized as follows. Section 2 discusses the research problem in more detail. Section 3 describes the general approach. Section 4 explains in how bad practices are mapped into paths in the code in order to enable taint analysis to be used for detecting violations. Section 5 presents CheckDroid. Section 6 discusses the experimental evaluation. Finally, Section 7 presents conclusions and future work.

2 Problem statement

An Android application which takes more than 200 milliseconds to respond to a user event is considered to be unresponsive [4]. The worst-case situation results in an Application Not Responding (ANR) dialog box, displayed by the Android runtime when the application does not respond to a key press within 5 seconds [7]. In such a case, Android offers the user an option to close the application.

Poor responsiveness and ANR messages are likely to motivate users to give low ratings and negative comments. Eventually users end up uninstalling unresponsive applications. Therefore, how to avoid ANR messages is an important issue thoroughly addressed by the Android API documentation. Indeed, an important part of the documentation is devoted to providing guidelines for developing responsive applications.

However, ANR messages are not the only cause of bad user experiences. In fact, these can be provoked by numerous reasons. For instance, execution lags due to periodic garbage collection are experienced by the user as small hiccups in the application behavior. Also, an application performing periodic network requests at a high rate will keep the radio on causing the battery to drain faster than expected.

Therefore, as a first step, we started up by analyzing the Android API guides with the aim of identifying programming recommendations intended to circumvent application slowdowns or unexpected crashes. We focused on two main categories of guidelines, namely performance and memory usage.

In this section, we describe several instances of these guidelines. The rest of the paper is devoted to addressing the problem of automatically assessing by static analysis whether the code of an application conforms to instances of these categories of good practices.

2.1 Performance

The execution of an Android application follows a single-threaded pattern in which the main thread of the application, also called the UI thread, handles all user-interface (UI) events. As a consequence, this thread should not perform heavy, long-running computations, such as network accesses, database operations, file I/O, bitmap processing, etc.. Otherwise, the user would actually feel the application to be unresponsive and most likely receive an ANR dialog box asking whether the application should be closed.

Android provides an execution policy, called *StrictMode*, which is meant to be used during the development cycle to detect accidental deviations from this expected use of the API. However, *StrictMode* is not guaranteed to find all misbehaviors. Moreover, it should never be left enabled in applications distributed on Google Play.

For this reason, the documentation contains a number of performance-related guidelines which seek avoiding application slowdowns. In this paper, we consider the following examples of such kind of recommendations:

P1 Verbose logging level and *StrictMode* should never be left enabled in released applications.

P2 Long running tasks should execute in worker threads, such as `Service` threads, `AsyncTasks`, etc.

P3 Worker threads should have lower priority than the UI thread. The purpose of this recommendation is to enable the runtime to schedule the UI thread upon the reception of an UI event.

2.2 Memory usage

Memory is a scarce resource in mobile devices, specially the low-end ones. Application performance is significantly better if memory is managed efficiently. This entails releasing unused objects when they are no longer needed. Holding references to unused objects prevents the garbage collection to reclaim the associated memory which causes the application to "leak" memory. Memory leaks typically grow over time and are difficult to identify and to correct. Indeed, careless memory handling is an important cause of application crashes.

The documentation provides several guidelines in this respect. Two important ones considered in this paper are the following:

M1 References to objects associated with a `Context`, such as `Adapters`, should not be stored in static variables since they will leak all resources bound to the instance.

M2 Worker threads should be explicitly closed. Otherwise, their associated memory space will be leaked.

3 Approach

Recall that the purpose of this paper is to address the problem of automatically checking conformance with recommended Android programming practices. In

order to do it, we propose an approach based on static analysis of Android code. The originality of the approach relies on associating guidelines with data-flow paths in the code. This idea enables resorting to static taint analysis for verifying whether recommendations are indeed respected.

```
1  public class MyClass<T> {
2    public T data;
3    public MyClass() {
4      T src = source(); /* source */
5      this.data = src; /* this.data is tainted */
6    } /* this is tainted */
7    public T getData() { return this.data; }
8  }
9  public T foo() {
10   MyClass o = new MyClass(); /* o is tainted */
11   return o.getData();
12 }
13 void bar() {
14   T d = foo(); /* d is tainted */
15   sink(d); /* sink */
16 }
```

Fig. 1. Taint analysis.

3.1 Taint analysis

Taint analysis searches for information-flow between two specific points in the program, called *source* (or *origin*) and *sink* (or *target*), by applying data-flow analysis through its control-flow and call graphs. The idea consists in *tainting* all assignments and method calls along the path [13].

Consider the example in Figure 1. Here, method `bar()` calls method `foo()` which creates an object of class `MyClass`. In line 4, the constructor `MyClass()` calls method `source()`. The local variable `src` then becomes *tainted*. In line 5 the field `data` of the newly created object referenced by `this` is tainted because it is assigned to `src`, which is tainted. As a consequence, object `o` gets itself tainted in line 10 because it holds a reference to a tainted object of type `T`. Furthermore, calling `foo()` in line 14 ends up tainting variable `d` which is passed as argument to `sink()`. Hence, information flows from a source method `source()` in line 4 to a sink method `sink()` in line 15 when method `bar()` is called.

This flow is depicted in Figure 2. Nodes correspond to methods. Solid arrows represent the call graph. Numbers labeling solid arrows indicate the order in which the calls are executed in the execution path described above. Dotted arrows show how the tainted source reaches the sink through the data-flow path.

An information flow path from an origin o to a target t, denoted $o \rightsquigarrow t$, is a sequence of solid and dashed arrows. In our example the information flow path from `source` to `sink` which traces the path followed by the tainted data is: `source` ⇢`MyClass` ⇢`foo` →`getData` ⇢`bar` →`sink`.

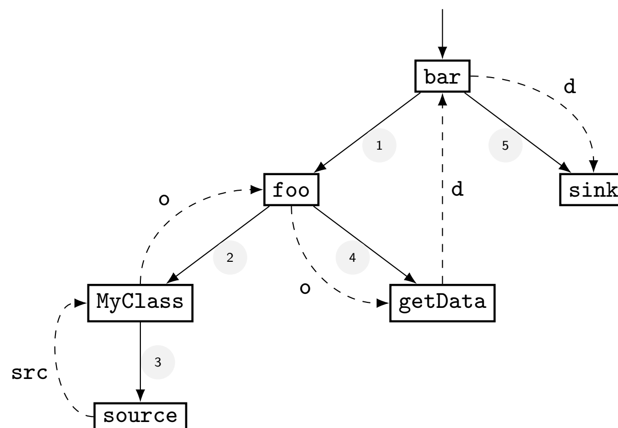

Fig. 2. Information Flow Graph.

A foremost usage of taint analysis is in looking for security vulnerabilities [3]. For instance, a typical use-case consists in finding whether some valuable asset, like a password, leaks from a secure origin, such as a login dialog box, to a dangerous destination, such as writing the password decrypted into a text file.

3.2 FlowDroid

Our approach to solve the problem of detecting violations to programming guidelines for Android applications consists in encoding a recommendation as a path between a source and a sink in the code. This idea enables using taint analysis and the tool FlowDroid [1] as the underlying framework to actually perform the analysis on Android code. FlowDroid [1] implements a static taint analysis algorithm for Android applications. It achieves high precision and recall by relying on a context-, flow-, field-, and object-sensitive inter-procedural taint analysis, and creating a complete model of the application life-cycle, including callbacks. Since user interaction cannot be predicted statically, the model contains all possible combinations of callbacks to make sure no taint is lost.

4 Strategies

This section is devoted to putting the aforementioned approach in practice. We analyze in detail all listed recommendations in Section 2 and propose a concrete

strategy to cope with each one. A strategy consists in a path or set of paths starting at a source and leading to a sink. Capturing the path or set of paths with taint analysis provides evidence of the violation of the corresponding guideline. We define 3 different strategies which cover the complete set of recommendations identified in Section 2. We will use the application snippet shown in Figure 3 as a running example to illustrate the strategies.

```
1  /* class MyActivity */
2  private TextView tv;
3  private ImageView iv;
4  private static EmailAddressAdapter instance;
5  protected void onCreate(Bundle b) {
6    super.onCreate(b);
7    tv = (TextView) findViewById(R.id.tv);
8    iv = getView().getImageView();
9    EmailAddressAdapter.getInstance(this);
10 }
11 public void updTextView(String s) {
12   Log.v("MyActivity", "updTextView" + s);
13   Thread t = new Thread(new Runnable(){
14     public void run(){ tv.setText(s); }}).start();
15 }
16 public void updImgView(String s) {
17   HttpURLConnection h = null;
18   try {
19     h = (new URL(s)).openConnection();
20     Bitmap bmp = BitmapFactory
21       .decodeStream(h.getInputStream());
22     iv.setImageBitmap(bmp); }
23   catch (Exception e) { e.printStackTrace(); }
24   finally { if (h != null) h.disconnect(); }
25 }
26 public static EmailAddressAdapter
27 getInstance(Context cx) {
28   if (instance == null)
29     instance = new EmailAddressAdapter(cx);
30   return instance;
31 }
```

Fig. 3. Running example.

The schematic call and flow graph built by FlowDroid is depicted in Figure 4. The structure of the graph corresponds to the Android activity *lifecycle*. Rectangular nodes are methods of the Android `Activity` class. Solid ellipsoidal nodes correspond to user-defined methods in the application code. Dotted ellipsoidal

nodes are Android API methods other than `Activity` ones. Double head arrows of the form $\twoheadrightarrow$ model *callbacks*. It means that a method is called by the Android runtime during the application lifecycle. Those edges do not correspond to direct calls from callers to callees but to the order in which they are actually executed by the runtime. For instance, the edge from `onCreate` to `onStart` means that the Android runtime calls `onStart` *after* `onCreate`.

The node Activity Running is an abstraction of the activity being executed. Callbacks to `updImgView` and `updTextView` that occur while the activity is running are modeled by the edges from Activity Running to the corresponding nodes. The activity is removed from memory upon the call to `onDestroy`.

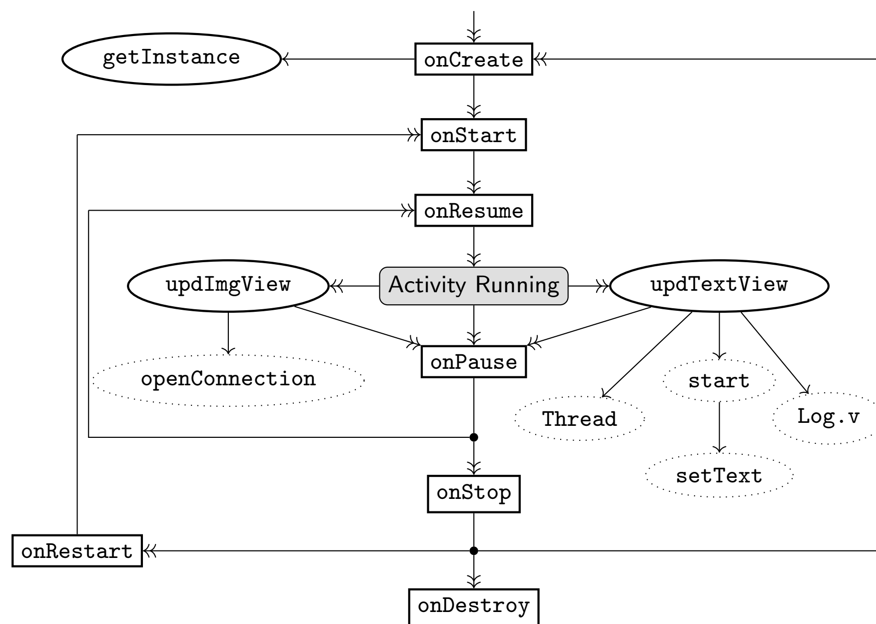

Fig. 4. Call Graph.

4.1 Strategy 1

Consider recommendation **P1**. We can think of the call to method `Log.v()` in line 12 as an origin o and the call to `onStop()` as a target t. Notice that this call is not explicit in the code, but it appears in the call graph shown in Figure 4.

A path from o to t, denoted $o \rightsquigarrow t$, is a violation of **P1**, since it means that method `Log.v()` may actually be executed. Of course, this is a very simple case, but only looking for occurrences of `Log.v()` in the code, though useful, could lead to a larger number of false positives.

StrictMode checking could be done exactly in the same way, except for the fact that there could be more than one source. Examples of possible origins are `StrictMode.ThreadPolicy.Builder` or `StrictMode.VmPolicy.Builder`. This case is not illustrated in our running example.

A similar approach could be applied for **M1**. This case is illustrated as a path from the call to `EmailAddressAdapter.getInstance()` in line 9, which is the source o, to method `onStop()`, which is the target t. Here, a reference to the activity instance making the call, e.g., `this`, ends up being passed to the constructor of `EmailAddressAdapter` while a reference to the created object of class `EmailAddressAdapter` is stored in the class (static) variable `instance` of `MyActivity`. Thus, this code violates recommendation **M1**.

Therefore, whether the code complies with recommendations **P1** and **M1** could be checked by the following strategy:

S1 Find a path $o \rightsquigarrow t$ where o belongs to appropriately identified sets of sources and target t is method `onStop()`.

4.2 Strategy 2

S1 is not enough for detecting all bad programming practices. For instance, a violation of **M2** requires *two* conditions to hold:

- a worker thread is created, and
- it is not explicitly closed.

Closing a thread in Android consists in interrupting it by invoking `interrupt()`, or waiting for it to terminate by calling `join()`.

We could therefore check whether there is a violation of recommendation **M2** with the following *two-path* strategy:

S2
1. Find a path $o \rightsquigarrow t$ from a thread creation (origin o) to method `onStop()` (target t), and
2. check there is *no* path $o' \rightsquigarrow t'$ from methods `interrupt()` or `join()` (origin o') to method `onStop()` (target t').

It is not difficult to figure out that the same strategy could be used for **P3**. In this case, the source o' corresponds to method `setThreadPriority()`. Notice, however, that calling `setThreadPriority()` does not ensure the worker thread priority will be set to a value lower than the one of the UI thread. That is, this strategy finds violations to **P3** only when the priority of worker threads is not set at all. This is reasonable, because it means the worker thread is assigned the same priority than the UI thread by default, which is undesireable.

An example of code violating recommendation **M2** is illustrated in Figure 3. The thread created in line 14 is never closed afterwards. Therefore, its memory is leaked when the activity is shut down. Besides, its priority is never set, so **P3** is also violated.

4.3 Strategy 3

Now, consider recommendation **P2**. A typical example is downloading an image from an URL whenever a button is clicked. This long running task should be executed in a worker thread or an instance of `AsyncTask`, which allows performing background operations and publish results on the UI thread without having to explicitly manipulate threads and/or handlers.

This situation is illustrated with callback method `updImgView()` in line 16. The guideline requires that method `doInBackground()` of `AsyncTask` should be a *caller* of `openConnection()` because the latter starts a network connection. That is, the long running operation should be executed by an `AsynTask` object.

If we view the recommendation from the standpoint of taint analysis, we have that `doInBackground()` should appear in any path from `openConnection()` to `onStop()`.

This idea gives us a new strategy that could be used to detect a violation of recommendations like **P2**:

S3 Find a path from a source o to a target t (method `onStop`), which does not go through a particular method i.

In our running example, the origin o is `openConnection()`, the target t is `onStop()`, and the intermediate method i is `doInBackground()`. Hence, recommendation **P2** is violated.

5 Tool

We developed the tool CheckDroid, which implements the strategies in Table 1.

one-path		two-path
S1	S3	S2
$o \rightsquigarrow t$	$o \rightsquigarrow \neg i \rightsquigarrow t$	$o \rightsquigarrow t \wedge \neg(o' \rightsquigarrow t')$
P1, **M1**	**P2**	**P3**, **M2**

Table 1. Strategies and covered recommendations

5.1 Architecture

Figure 5 depicts the schematic architecture of CheckDroid. Overall, CheckDroid comprises 20 classes and 1500 LOCS. It takes as inputs the application *.apk* and an XML file containing the bad practices to be checked.

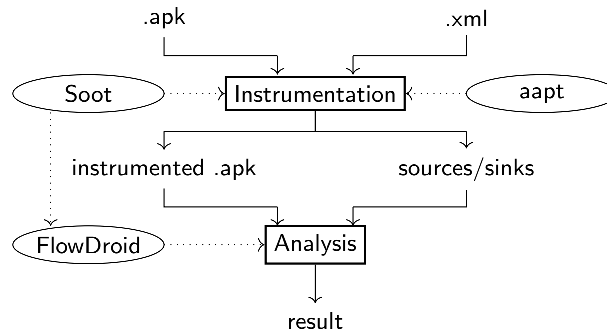

Fig. 5. CheckDroid architecture.

5.2 Strategies

Instances of the strategies are provided in an XML file. This allows extending the tool with further strategies and source/sink pairs. An example of such a strategy instance is shown in Figure 6. It corresponds to an instance of strategy `S2`. It is defined to search for violations to recommendation **P3**. It says that a path from source `start()` (line 5) of class `java.Lang.Thread` (line 6) to `onStop()` (target) should be present (line 2), while a path from `setPriorityThread` (line 13) to `onStop()` should not (line 10). Notice that target method `onStop()` is implicit. This is because in our approach, `onStop()` is always the target method.

5.3 Instrumentation

The instrumentation phase is needed because taint analysis tracks information flowing through object links. So, the existence of a path in the call graph from the call to `Log.v()` in line 12 to method `onStop()` is not enough to constitute a tainted path. That is, we need to create an information flow path. For this, we need to have some data value stored at the source and read at the sink.

CheckDroid uses Soot to instrument the application *.apk* in order to recreate information flow paths which could be found by FlowDroid. The instrumentation occurs at the application main activity class, and at every source.

The Android Asset Packaging Tool (`aapt`) is used to detect the application main activity class and to avoid instrumenting sources which are not part of the application code in order to reduce false positives.

Figure 7 sketches the instrumented `MyActivity` class. CheckDroid provides a class called `CheckDroidBinder`. Its role is to store references to objects which will be used to reconstruct the data-flow path during the analysis. The `onStart()` method is modified so as to create an instance of `CheckDroidBinder` (line 4). The static method `addBindingObjs()` (line 10) is added to have a hook to the internal CheckDroid method. Method `onStop()` is instrumented to call `getBindingObjs()` (line 8) to simulate a sink.

```
<bad-practice><name>P3</name>
  <path presence="true">
    <sources>
      <source>
        <method>void start()</method>
        <class>java.lang.Thread</class>
      </source>
    </sources>
  </path>
  <path presence="false">
    <sources>
      <source>
        <method>void setPriority(int)</method>
        <class>java.lang.Thread</class>
      </source>
    </sources>
  </path>
</bad-practice>
```

Fig. 6. Extract of bad practice declaration.

```
private static CheckDroidBinder binder;
protected void onStart() {
  .... /* application code */
  binder = new CheckDroidBinder();
}
protected void onStop() {
  .... /* application code */
  List<Object[]> l = binder.getBindingObjs(); /* sink */
}
public static void addBindingObjs(Object[] objs) {
  binder.addBindingObjs(objs);
}
```

Fig. 7. Example of instrumented sink.

Figure 8 shows an example of instrumented sources. At each source point, a call to `addBindingObjs()` is added with appropriate arguments. For instance, the call at line 11 makes the activity object to point to the `HttpURLConnection` object referenced by variable `h` and created in `updImgView()`. This is done by adding it to the list of binding objects kept by `MyActivity binder`. The same approach is followed with the call to `Log.v()` in `updTextView`. The added call to `getBindingObjs()` in `onStop()` creates the actual data path which is found by taint analysis with FlowDroid.

```
1  public void updTextView(String s)
2    Log.v("MyActivity", "updTextView" + s);
3    MyActivity
4    .addBindingObjs(new Object[]{"MyActivity","updTextView"+s});
5    ...
6  }
7  public void updImgView(String s) {
8    HttpURLConnection h = null;
9    try {
10     h = (new URL(s)).openConnection();
11     MyActivity.addBindingObjs(new Object[]{new URL(s), h});
12   ...
13 }
```

Fig. 8. Example of instrumented source.

Figure 9 depicts the call graph of the instrumented code. Gray-colored, dotted rectangles correspond to methods added through instrumentation. For the sake of readability, data flows (dashed arrows ⇢ in Figure 2) are omitted.

The instrumented code is fed into the analysis phase, together with a representation of the set of source/sinks to be searched by FlowDroid. The analysis implements the strategies summarized in Table 1.

The first one (S1) consists in calling FlowDroid once, resulting in a yes/no answer depending on whether FlowDroid finds or not a path. The second strategy (S2) requires calling FlowDroid a second time in case a path is found during the first pass. The third one (S3) requires a post-processing of FlowDroid output whenever a path is found. This post-processing consists in traversing the paths generated by FlowDroid to check whether the intermediate conditions are met.

To illustrate the analysis, let us consider the example in Figure 8. The violation of recommendation **P1** is verified with a single run of FlowDroid (S1). The path is: `addBindingObjs()` ⇢`updTextView()` ↠`onPause()` ↠`onStop()` →`getBindingObjs()`. Tainted data flows from `updTextView()` to `onStop()` through the `MyActivity` static object `binder`.

The violation of **P2** needs analyzing the path produced by FlowDroid during the first pass from line 11 in the instrumented method `updImgView()` to line 8 in the instrumented class `MyActivity`. In this case, the path found by FlowDroid is: `addBindingObjs()` ⇢`updImgView()` ↠`onPause()` ↠`onStop()` →`getBindingObjs()`. Clearly, a traversal of this path finds no occurrence of `doInBackground()` (nor any call to a run method of a worker thread). Therefore, a violation of **P2** is reported.

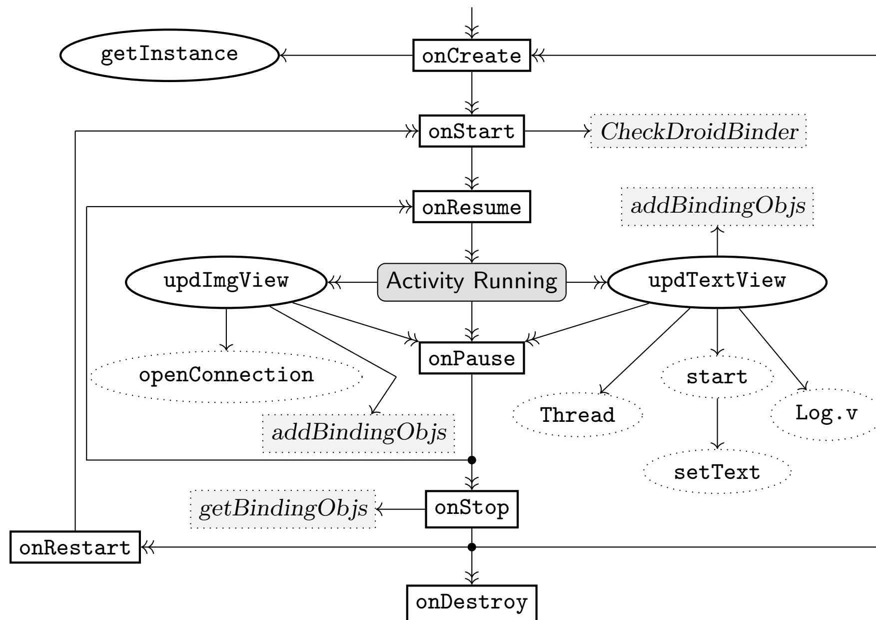

Fig. 9. Call Graph of the instrumented application.

Remark CheckDroid relies on static taint analysis which is a kind of *may* static analysis. This means that, if no violation is found, then the application conforms to the guidelines. Otherwise, the application may not. Thus, even if FlowDroid has proven to be quite accurate, it may result in false positives. Therefore, whenever a path is found, a deeper analysis (verification, testing, debugging, etc.) must be performed to verify whether the path is indeed executable, that is, to determine whether it is a false positive. For this, it may be useful or even required to have the source code of the application, although CheckDroid does not need it.

6 Experimental evaluation

6.1 Undergraduate apps

We first experimented CheckDroid on the *apks* of 18 applications developed by undergraduate students as their final project for a mobile computing course at the Department of Computer Science of University of Buenos Aires. Figure 10 shows the distribution of reported bad practices.

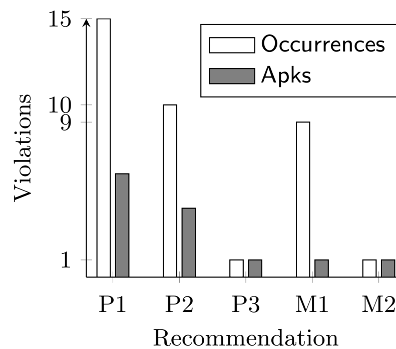

Fig. 10. Distribution of reported bad practices in student applications.

The mean size of the *apk* was 1MB. In average, the execution time of the instrumentation phase was 17 secs, yielding an average rate of 61.5KB/sec. The mean analysis time was 24 secs for one-path strategies, doubling for two-path ones. Overall, the mean total analysis time for the complete set of bad practices was 190 secs.

- CheckDroid reported a total of 32 occurrences of bad practices distributed in 9 of the applications. That is, 50% of them was not conforming with at least one guideline.
- The most common violated recommendation was **P1**, with 15 occurrences, spanning 67% (6/9) of the non-conforming applications.
- The second in importance was **P2** with 10 occurrences, spanned over 44% (4/9) of the applications with bad practices.
- The third in importance was **M1**, with 9 violations occurring in a single application, a network intensive one.
- The 9 bad applications violated at least one of these three, with **P1** and **P2** representing 78% (25/32) of the bad practices and 89% (8/9) of the violating applications.
- The 67% of those 9 case studies (6/9) had only 1 bad practice (with eventually more than one occurrence).

- A connection-intensive application implementing an Android-based client for an Enterprise Resource Planning (ERP) system incurred in 3 different bad practices.
- Two applications were reported having more than 10 deviations with respect to the guidelines. A social network-centric application had a total of 14 violations, spanning **P1** and **P2**, while a network intensive totaled 11, 9 of which correspond to **M1**.
- Taking out these outliers, the average was 2 reports per application.

Furthermore, we investigated whether the reported bad practices were false positives by inspecting 5 of the 18 applications for which we had the source code. This subset includes the network-centric application with 11 reported violations. We verified that all reported deviations with respect to the guidelines were actual violations. That is, there were no false positives in this subset of applications.

6.2 Google Play apps

After this preliminary experimental evaluation, we run CheckDroid on *apks* from Google Play.

To start up with, CheckDroid reported 3 guidelines violated by *BA Subte*, an application to query the status of the Subway of Buenos Aires City. This is a free app which registers between 100K to 500K installs, and a review score of 3.8 (4 of 5 stars). The reported bad practices reported by CheckDroid were the following:

- An occurrence of **M1** (thread leak)
- An occurrence of **P3** (thread priority not set)
- An occurrence of **P2** (long running task inside UIThread)

Of course, as remarked before, they could actually be false positives because we do not have the source code to perform a deeper analysis.

Besides *BA Subte*, we applied CheckDroid to two sets of applications from Google Play with *apk* sizes between 300KB and 3MB:

- A consists in 20 applications with more than 50K downloads and a review score less than or equal to 2, that is, a poor evaluation of 1 or 2 stars;
- B consists in 5 applications with more than 500K downloads and a review score greater than or equal to 4.7, that is, a very good evaluation of 4 or 5 stars.

The analysis of set A, gave the following results:

- No violations were reported on 10 of the 20 applications.
- Every guideline was violated at least once.
- Two applications were reported to incur respectively in 21 and 266 violations, with **M1** being the most violated one (10 and 242 occurrences, respectively).
- The most violated guideline was **M1**, with a median of 3.5, but a very large standard deviation due to the two possible outliers.

– The overall median for the subset of A having been reported to violate at least 1 recommendation was 4.5.

The analysis of set B, gave the following results:

– No violations were reported on 1 of the 5 applications.
– Only deviations with respect to **P1** and **P2** were reported, with almost equal numbers of occurrences: 13 and 15, respectively.
– One application was found to incur in 16 violations, with 7 of **P1** and 9 of **P2**.
– The overall median for the subset of B having been reported to violate at least 1 recommendation was 7.

We draw the following conclusions:

– The rationale behind separating the applications in two sets with low and high review scores was to try establishing a relationship between violating guidelines and bad user experiences manifested as low scores. However, we found high score applications with large numbers of reported violations and low score ones with no deviations with respect to the guidelines.
– Recommendations regarding thread manipulation, namely **P3** and **M2**, were the ones with the least reported violations overall.
– By far, the guidelines with the highest numbers of reported violations were **P1**, **P2** and **M1**. Besides they were very frequently reported to happen together. On one hand, it follows that strategies S1 and S3 revealed to be the most productive ones. On the other, even though we have found no false positives so far, it may be an indication that these strategies are too loose. Therefor, further investigation is needed to evaluate the actual occurrence of the reported deviations, even in the absence of source code.

7 Conclusions

This work investigated how static analysis could be applied to seek unrecommended programming practices in Android applications. The originality of the proposed approach resides in the modeling of bad coding patterns as information flows and using taint analysis for checking their existence in application code. To achieve this in practice, we developed several concrete strategies to be applied to detect different kinds of misbehaviors with respect to the guidelines.

To validate the approach, we implemented a tool, called CheckDroid, which proceeds in three steps. First, it appropriately instruments the code with Soot using as input the *apk* of the application and the bad practices to be checked which are provided as an XML file. Second, CheckDroid runs FlowDroid to search for tainted paths. Third, CheckDroid analyzes the paths found by FlowDroid according to the strategy.

It is important to remark that relying on FlowDroid, rather on Android Lint, as done in [12], is esential for detecting violations of guidelines which require strategy S3, such as **P2**.

We successfully applied CheckDroid to a number of Android applications. First we experimented the tool on a set of applications developed by newbie programmers. Second, we applied CheckDroid on a set of freely available applications in the Play store. In both experiments, we found that strategies S1 and S3 were the most effective ones by reporting the vast majority of violations. In terms of concrete instances of non-respected guidelines, the most reported ones were **P1**, **P2**, and **M1**.

In all experiments, CheckDroid exhibited low execution time rates of both instrumentation and analysis phases. This observation let us envisage that CheckDroid could be profitable integrated in IDEs for helping detecting deviations with respect to Android guidelines early in the development cycle.

Further investigation is required to extend the tool so as (a) to detect a larger set of bad practices, (b) to fine tune the definition of the strategies to avoid false positives, (c) to study the applicability of the approach for recommendations related to user interface, and (d) to perform a broader experimental evaluation.

References

1. Arzt, S., et al.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. SIGPLAN Not. 49(6), 259–269 (Jun 2014)
2. Braberman, V.A., Garbervetsky, D., Hym, S., Yovine, S.: Summary-based inference of quantitative bounds of live heap objects. Sci. Comput. Program. 92, 56–84 (2014), https://doi.org/10.1016/j.scico.2013.11.036
3. Chess, B., McGraw, G.: Static analysis for security. IEEE Security & Privacy 2(6), 76–79 (2004)
4. Fitzpatrick, B.: Writing zippy android apps. In: Google I/O Developers Conference (2010)
5. Garbervetsky, D., Yovine, S., Braberman, V.A., Rouaux, M., Taboada, A.: Quantitative dynamic-memory analysis for java. Concurrency and Computation: Practice and Experience 23(14), 1665–1678 (2011), https://doi.org/10.1002/cpe.1656
6. Google: Android lint. http://tools.android.com/tips/lint
7. Google: Keeping your app responsive. https://developer.android.com/training/-articles/perf-anr.html
8. Guo, C., Zhang, J., Yan, J., Zhang, Z., Zhang, Y.: Characterizing and detecting resource leaks in android applications. In: Proc. IEEE/ACM 28th Int. Conf. ASE. pp. 389–398 (2013)
9. Li, L., Bissyand, T.F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., Traon, L.: Static analysis of android apps: A systematic literature review. Information and Software Technology 88(Supplement C), 67 – 95 (2017), http://www.sciencedirect.com/science/article/pii/S0950584917302987
10. Ongkosit, T., Takada, S.: Responsiveness analysis tool for android application. In: Proceedings of the 2nd International Workshop on Software Development Lifecycle for Mobile. pp. 1–4. ACM (2014)
11. Sadeghi, A., Bagheri, H., Garcia, J., et al.: A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software. IEEE TSE (2016)
12. Saglam, I.A.: Measuring And Assesment of well known Bad Practices in Android Application Developments. Master's thesis, Middle East Tech. Univ., Turkey (2014)

13. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Proc. 12th Int. Conf. SAS. pp. 352–367 (2005)
14. Uddin, G., Robillard, M.P.: How api documentation fails. IEEE Software 32(4), 68–75 (2015)
15. Wang, J., et al.: Mining succinct and high-coverage api usage patterns from source code. In: Proc. 10th Work. Conf. Mining Software Repositories. pp. 319–328. IEEE Press (2013)
16. Wang, W., Godfrey, M.: Detecting API usage obstacles: A study of iOS and Android developer questions. In: Proc. 10th Work. Conf. Mining Soft. Rep. pp. 61–64. IEEE Press (2013)
17. Yang, S., Yan, D., Rountev, A.: Testing for poor responsiveness in android applications. In: Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the. pp. 1–6. IEEE (2013)