Static Taint Analysis Applied to Detecting Bad Programming Practices in Android

Gonzalo Winniczuk
Universidad de Buenos Aires
Departamento de Computación
Buenos Aires, Argentina

Sergio Yovine
CONICET-Universidad de Buenos Aires
Instituto de Investigación en Ciencias de la Computación
Buenos Aires, Argentina
syovine@dc.uba.ar

***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, and at most, 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.**

***Index Terms*—Android; programming recommendations; static analysis; taint analysis.**

I. Introduction

Today, Android runs on more than 80% of smart-phones and tablets in the market. The documentation of the Android API exhibits a number of recommended practices which should be respected by the software developer in order to avoid bad user experiences, such as frozen screens and Application Not Responing (ANR) messages, poor performance caused by memory leaks, or even unexpected faults causing application crashes, lost data, etc.. Despite the existence of such recommendations, current IDEs do not enforced them. Besides, they are sometimes difficult to reproduce during testing.

There is a significant amount of research effort devoted to analyzing Android applications. Nevertheless, most of this work is addressed to providing tools for seeking specific defects, such as security vulnerabilities [1], resource (memory, energy, ...) leaks [2], and poor responsiveness [3], [4]. In fact, many of these issues 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 susceptible of causing runtime defects, while 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 are not concentrated in analyzing program compliance with documentation guidelines, like ours. On the contrary, they seek improving API documentation in order to helping programmers. For instance, [5] studies obstacles derived from badly documented APIs, [6] discovers usage patterns by mining client code, and [7] does the same by analyzing posts in developer forums.

To the best of our knowledge, only [8] attacks the same research question. That work requires the Java source code of the application and relies on Android Lint [9], which is an Android Studio utility that scans project sources checking for a number of common mistakes, such as layout problems, unused resources, hard-coded strings, icon issues (e.g., duplications, wrong sizes), usability problems (e.g., untyped text fields), manifest errors, etc. The approach developed in [8] is restricted 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 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. To validate the idea, we developed the tool CheckDroid which steps on FlowDroid [10], a state-of-the-art static taint-analysis framework which (a) analyzes the *.apk*, instead of the application source code, and (b) 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. Preliminary experimental evaluation shows that CheckDroid is able to find common bad practices incurred in a number of applications.

The rest of the paper is organized as follows. Section II discusses the research problem in more detail. Section III describes the approach, in particular, how bad practices are mapped into paths in the code. Section IV presents CheckDroid. Section V discusses the experimental evaluation. Finally, Section VI presents conclusions and future work.

II. Problem statement

An Android app which takes more than 200 ms to respond to a user event is considered to be unresponsive [11]. The worst-case situation results in an “Application Not Responding” (ANR) dialog, displayed by the runtime when the application does not respond to a key press within 5 secs [12], offering the user an option to close it. Poor responsiveness and ANR messages are likely to motivate users to give low ratings and

negative comments, and eventually to end up uninstalling the application. Thus, how to avoid ANRs is an important issue thoroughly addressed by the documentation which provides guidelines for developing responsive applications. However, ANRs are not the only cause of a bad user experience. Therefore, we started up by analyzing the available documentation with the aim of identifying programming recommendations intended to circumvent application slowdowns or unexpected crashes. In this paper we focus on three main categories of guidelines, namely performance and memory usage.

a) Performance: The execution of Android applications follow a single-threaded pattern in which the main or UI thread of the application handles all UI events. As a consequence, this thread should not perform heavy long-running computations, such as, networking, database operations, file I/O, bitmap processing, etc., since it would most likely result in an ANR. Android provides an execution policy, called *StrictMode*, which is meant to be used during the development cycle to detect accidental deviations of this expected usage. However, StrictMode is not guaranteed to find all misbehaviors, and it should never be left enabled in applications distributed on Google Play. Therefore, the documentation contains a number of performance-related guidelines which seek avoiding application slowdowns. Examples of such recommendations are the following:

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 UI thread, to enable the runtime to schedule the UI thread upon the reception of an UI event.

b) Memory usage: Memory is a scarce resource in mobile devices, specially the low-end ones. App performance is significantly better if it uses memory efficiently, which entails releasing it when it is no longer needed. 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. Some important ones 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.

c) User interface: Inappropriate manipulation of system-managed objects are subject to provoke application crashes, mainly due to the fact that the UI toolkit is not thread-safe. Thus, some good practices are defined in order to prevent against such misbehaviors. A critical example is the following:

U1 UI objects must not be manipulated by a worker thread.

In this paper, we address the problem of automatically assessing by static analysis whether the code of an application conforms to instances of these categories of good practices. In order to do it, we propose an approach based on binding guidelines with paths in the code.

III. Approach

Taint analysis searches for information-flow between two specific points in the program, called *source* and *sink*, 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]. A typical usage is in looking for security vulnerabilities [14], such as finding whether some valuable asset, e.g., a password, leaks to a dangerous destination, e.g., is written decrypted in a file, from a secure origin, e.g., a login dialog box.

FlowDroid [10] implements a static taint analysis algorithm for Android applications. It achieves high precision and recall by a) relying on a context-, flow-, field-, and object-sensitive inter-procedural taint analysis, and b) 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.

Our idea to solve the stated problem consists in encoding a recommendation as a path between a source and a sink. This approach enables using FlowDroid as the underlying tool to actually perform the checking.

To illustrate the idea, let us consider the example in Figure 1. The schematic control-flow graph (CFG) built by FlowDroid is depicted in Figure 2.

A. Strategy 1

Let us begin with **P1**. We can think of the call to method `Log.v()` in line 21 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 CFG. A path from o to t, denoted $o \rightarrow t$, 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, such as `StrictMode.ThreadPolicy.Builder` or `StrictMode.VmPolicy.Builder`.

A similar approach could be applied for **M1**. This case is illustrated as a path from the call in line 16, which is the source

```
1  /* class MyActivity */
2
3
4  private TextView tv;
5  private ImageView iv;
6  private static
7     EmailAddressAdapter instance;
8
9  protected void onCreate(Bundle b) {
10   super.onCreate(b);
11   tv = (TextView)
12         findViewById(R.id.tv);
13   iv = getView()
14         .getImageView();
15   EmailAddressAdapter
16         .getInstance(this);
17 }
18
19 /* callback method */
20 public void updTextView(String s) {
21   Log.v("MyActivity",
22         "updTextView␣"+s);
23   Thread t = new Thread(new Runnable()
24    {public void run(){tv.setText(s);}}
25    ).start();
26 }
27
28 /* callback method */
29 public void updImgView(String s) {
30   HttpURLConnection h = null;
31   try {
32     h = (new URL(s))
33           .openConnection();
34     Bitmap bmp = BitmapFactory
35       .decodeStream(h
36           .getInputStream());
37     iv.setImageBitmap(bmp);
38   } catch (Exception e) {
39     e.printStackTrace();
40   }
41   finally {
42     if (h != null) h.disconnect();
43   }
44 }
45
46 public static EmailAddressAdapter
47 getInstance(Context cx) {
48   if (instance == null)
49     instance = new
50       EmailAddressAdapter(cx);
51   return instance;
52 }
```

Fig. 1. Motivating example.

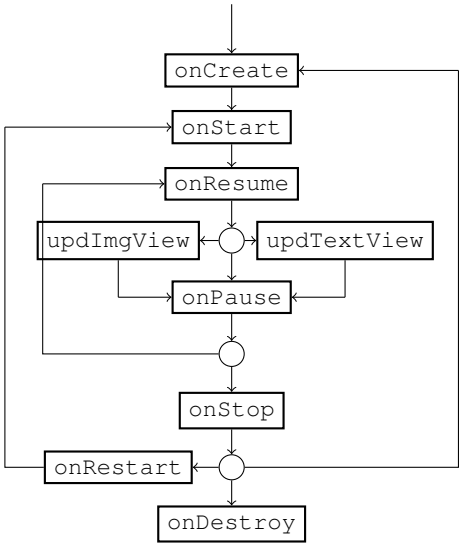

Fig. 2. Control Flow Graph.

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 instance is stored at the class (static) variable `instance` of `MyActivity`, thus violating recommendation **M1**.

This gives us a first *strategy* (S1):

> Find a path $o \to t$ where o and t belong to appropriately identified sets of sources and sinks.

Remark: Notice that $o \to t$ is not a path in the call graph, but a path in the data-flow graph. This means that information gathered in the origin o can reach the target t.

B. 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 map it into two requirements, yielding the two-path strategy S2:

> 1) Find a path from a thread creation (origin o) to `onStop()` (target t), and
> 2) Check there is *no* path from `interrupt()` or `join()` (origin o') to `onStop()` (target t').

This case is illustrated with the thread created in line 23 which is not closed afterwards.

It is not difficult to figure out that the same strategy could be used for **P3**.

C. 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 done 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 29. 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. Let `openConnection()` be the origin o, `onStop` be the target t, and `doInBackground()` be the intermediate method i. To detect a violation of **P2**, the strategy S3 is as follows:

> Find a path from a source o to a target t, which does not go through i.

Remark: The recommendation is violated because information flows from o to t without passing through i. In other words, the guideline requires method i should be a *caller* of o, but it is not.

D. Strategy 4

Checking **U1** is more involved. Take for instance the prototypical case where a text field is changed in a view. Here, the origin o is `setText()` and, again, the target t is `onStop()`. The bad situation occurs whenever method `setText()` is executed in a `run()` method of a thread, giving a first intermediate point i in the path, which is not the UI thread, that is, it is not executed inside a method `runOnUIThread()`, giving a second point i' not in the path.

This idea yields strategy S4:

> 1) Find a path from a thread creation (origin o) to `onStop()` (target t), and
> 2) Check it passes through some intermediate point i which is not followed by some other point i'.

Remark: **U1** is violated because information flows from o to t through i (a caller of o) but without passing through i'. That is, method i' should be a caller of i but it is not.

This case is illustrated with the callback method `updTextView()` in line 20.

Table I summarizes strategies and recommendation types.

one-path			two-path
S1	S3	S4	S2
$o \to t$	$o \to \neg i \to t$	$o \to i \to \neg i' \to t$	$o \to t \wedge \neg(o' \to t')$
P1, M1	**P2**	**U1**	**P3, M2**

TABLE I
STRATEGIES AND COVERED RECOMMENDATIONS

IV. TOOL

Figure 3 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

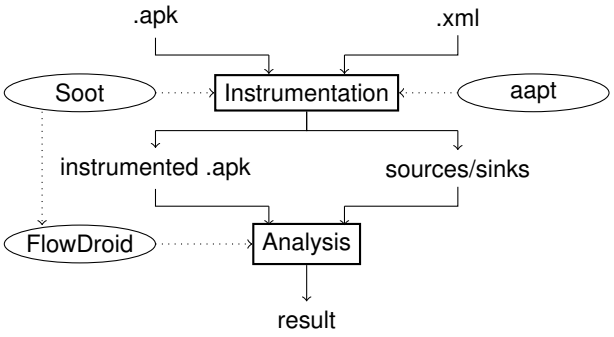

Fig. 3. CheckDroid architecture.

the bad practices to be checked (Figure 4). This allows extending the tool with further strategies and source/sink pairs.

The instrumentation phase is needed because taint analysis tracks information flowing through object links. So, the mere existence of a path in the CFG of the application from the call to `Log.v()` in line 21 to method `onStop()` is not enough to constitute a tainted 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 that data flow.

The instrumentation occurs at two places: a) at the application main activity class, and b) at every source. The Android Asset Packaging Tool (`aapt`) is used to avoid instrumenting sources which are not part of the application code in order to reduce false positives.

Figure 5 shows an example of instrumented main activity class. Our tool provides a class called `CheckDroidBinder` whose 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`. The static method `addBindingObjs()` is added to have a hook to the internal CheckDroid method. Method `onStop()` is instrumented to call `getBindingObjs()` to simulate a data read. `getBindingObjs()` becomes the only sink to be searched by FlowDroid.

Figure 6 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 12 makes the activity object to point to the `HttpURLConnection` object created in `updImgView()` through the added field `binder`. The added call to `getBindingObjs()` in `onStop()` creates the actual data path which is found by

```
<bad-practice>
  <name>
    Thread Priority Not Set
  </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. 4. Extract of bad practice declaration.

```
1  private
2    static CheckDroidBinder binder;
3
4  protected void onStart() {
5    super.onStart();
6    //.... (application code)
7    binder = new CheckDroidBinder();
8  }
9
10 protected void onStop() {
11   super.onStop();
12   //.... (application code)
13   List<Object[]> l =
14     binder.getBindingObjs();
15 }
16
17 public static void
18 addBindingObjs(Object[] objs) {
19   binder.addBindingObjs(objs);
20 }
```

Fig. 5. Example of instrumented sink.

```
1  public void updImgView(String s) {
2    Log.v("MyActivity",
3          "updImgView␣" + s);
4    MyActivity.addBindingObjs(
5      new Object[]{"MyActivity",
6      "updImgView " + s});
7    HttpURLConnection h = null;
8    try {
9      h = (new URL(s))
10       .openConnection();
11     MyActivity.addBindingObjs(
12       new Object[]{new URL(s), h});
13     Bitmap bmp = BitmapFactory
14       .decodeStream(
15         h.getInputStream());
16     iv.setImageBitmap(bmp);
17   } catch (Exception e) {
18     e.printStackTrace();
19   }
20   finally {
21     if (h != null) h.disconnect();
22   }
23 }
```

Fig. 6. Example of instrumented source.

the taint analysis implemented by FlowDroid.

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 I. The first one consists in calling FlowDroid once, resulting in a yes/no answer depending on whether FlowDroid finds or not a path. The second strategy requires calling FlowDroid a second time in case a path is found during the first pass. The third and fourth strategies require 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 6. The violation of recommendation **P1**, is verified with a single run of FlowDroid. The violation of **P2** needs analyzing the path produced by FlowDroid during the first pass from line 12 in the instrumented method `updImgView()` to line 14 in the instrumented class `MyActivity`. A traversal of this path finds no occurrence of `doInBackground()` (nor any call to a run method of a worker thread), thus confirming that **P2** is not respected.

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.

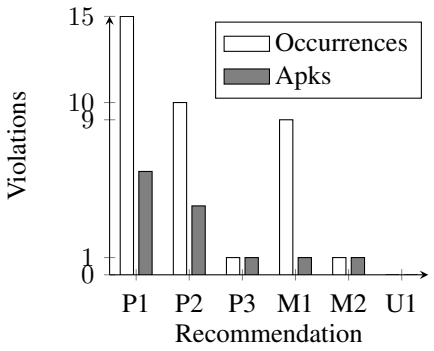

Fig. 7. Distribution of reported bad practices in student applications.

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 it it is a false positive or not. For this, it may be useful or even required to have the source code of the application, although CheckDroid does not need it.

V. Experimental evaluation

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.

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.
- No violations of **U1** were reported.

Figure 7 shows the distribution of reported bad practices.

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.

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, with the exception of **U1**.
- 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.
- No violation was observed for **U1**. Clearly, this fact suggests that it is necessary to revisit the definition of this class of recommendations and the associated strategy.

VI. Conclusions

This paper briefly presents the preliminary results obtained in the context of the ongoing final project of the first author towards obtaining the degree in Computer Science at Universidad de Buenos Aires.

The main originality of the approach resides in applying taint analysis to a different problem, namely, checking bad programming practices instead of information leaks.

For this purpose, a bad practice is modeled as a path of information flow through an appropriate instrumentation of the code.

We have successfully applied the tool to a number of Android applications developed by newbie programmers and others freely available in the Play store.

The low execution time rates of both instrumentation and analysis phases let us envisage that CheckDroid could be profitable integrated in an IDE for helping detecting such bad practices 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 revise the approach for recommendations related to user interface, and (d) to perform a broader experimental evaluation.

References

[1] A. Sadeghi, H. Bagheri, J. Garcia *et al.*, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software," *IEEE TSE*, 2016.

[2] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in android applications," in *Proc. IEEE/ACM 28th Int. Conf. ASE*, 2013, pp. 389–398.

[3] S. Yang, D. Yan, and A. Rountev, "Testing for poor responsiveness in android applications," in *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the*. IEEE, 2013, pp. 1–6.

[4] T. Ongkosit and S. Takada, "Responsiveness analysis tool for android application," in *Proceedings of the 2nd International Workshop on Software Development Lifecycle for Mobile*. ACM, 2014, pp. 1–4.

[5] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[6] J. Wang *et al.*, "Mining succinct and high-coverage api usage patterns from source code," in *Proc. 10th Work. Conf. Mining Software Repositories*. IEEE Press, 2013, pp. 319–328.

[7] W. Wang and M. Godfrey, "Detecting API usage obstacles: A study of iOS and Android developer questions," in *Proc. 10th Work. Conf. Mining Soft. Rep.* IEEE Press, 2013, pp. 61–64.

[8] I. A. Saglam, "Measuring and assesment of well known bad practices in Android application developments," Master's thesis, Middle East Tech. Univ., Turkey, 2014.

[9] Google, "Android lint," http://tools.android.com/tips/lint.

[10] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, Jun. 2014.

[11] B. Fitzpatrick, "Writing zippy android apps," in *Google I/O Developers Conference*, 2010.

[12] Google, "Keeping your app responsive," https://developer.android.com/training/articles/perf-anr.html.

[13] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *Proc. 12th Int. Conf. SAS*, 2005, pp. 352–367.

[14] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.