6-2018

Static analysis of context leaks in android applications

Flavio TOFFALINI

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Martín COHOA

Citation

TOFFALINI, Flavio; SUN, Jun; and COHOA, Martín. Static analysis of context leaks in android applications. (2018). *Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 2018 May 27 - June 3*. 215-224.

Available at: https://ink.library.smu.edu.sg/sis_research/4651

Static Analysis of Context Leaks in Android Applications

Flavio Toffalini, Jun Sun and Martín Ochoa
Singapore University of Technology and Design
flavio_toffalini@mymail.sutd.edu.sg
{sunjun,martin_ochoa}@sutd.edu.sg

ABSTRACT

Android native applications, written in Java and distributed in APK format, are widely used in mobile devices. Their specific pattern of use lets the operating system control the creation and destruction of key resources, such as activities and services (*contexts*). Programmers are not supposed to interfere with such lifecycle events. Otherwise contexts might be leaked, *i.e.* they will never be deallocated from memory, or be deallocated too late, leading to memory exhaustion and frozen applications. In practice, it is easy to write incorrect code, which hinders garbage collection of contexts and subsequently leads to context leakage.

In this work, we present a new static analysis method that finds context leaks in Android code. We apply this analysis to APKs translated into Java bytecode. We discuss the results of a large number of experiments with our analysis, which reveal context leaks in many widely used applications from the Android marketplace. This shows the practical usefulness of our technique and proves its superiority *w.r.t.* the well-known Lint static analysis tool. We then estimate the amount of memory saved by the collection of the leaks found and explain, experimentally, where programmers often go wrong and what the analysis is not yet able to find. Such lessons could be later leveraged for the definition of a sound or more powerful static analysis for Android leaks. This work can be considered as a practical application of software analysis techniques to solve practical problems.

KEYWORDS

Static analysis, memory leak, Android

ACM Reference Format:
Flavio Toffalini, Jun Sun and Martín Ochoa. 2018. Static Analysis of Context Leaks in Android Applications. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3183519.3183530

1 INTRODUCTION

Smartphones are standard technology nowadays. They have powerful computing capabilities, access databases, connect to remote servers, run games and heavy graphical operations. Their usage patterns are different from those of traditional desktop computers. In a smartphone, interaction screens, called *activities* in Android, and background services are fired on demand and never explicitly stopped by the user. Moreover, activities are not traditional windows, since phones have no windowing system that lets the user concurrently interact with multiple applications. Instead, activities are singularly brought to the foreground and back, depending on a given user's needs. The operating system controls the life cycle of activities and services and issues life cycle events to drive them, including destruction events. Programmers usually listen to such events to react accordingly. For instance, when an activity is brought to the background after another application is fired, programmers can intercept that event to turn off the sound, stop downloading data, or deallocate a large bitmap. When an activity is destroyed, more extensive clean-up operations could be run, *e.g.* so as to free up resources.

Android Apps thus require new design patterns which are different from traditional Java applications. Android programmers have often years of experience with Java, but are typically unaware of such patterns, as well as the related problems and solutions.

Activities and services are instances of Android *contexts*. The Android operating system controls the life cycle of contexts. Programmers are not supposed to interfere with the garbage collection mechanism in Android. In particular, contexts should not be made reachable from static fields or threads, that are roots of non-garbage collectable data. Otherwise they will not be garbage collected at the end of their life cycle. This rule however is easily violated in practice, because contexts are often contained in other objects, such as views (widgets) or fragments (portions of activities). For instance, the context in a fragment cannot be garbage collected until the fragment itself is garbage collected. This is furthered complicated by hidden references. For instance, the implementation of inner classes in Java entails that an inner class thread started from a context contains a hidden reference to that context. As a result, the context will never be garbage collected before the thread terminates.

These are examples of *memory leaks*, defined (page 89 of [15]) as *memory allocated by the application that is not used anymore but never identified by the garbage collector as memory that can be reclaimed. This also includes memory allocated for too long a time, essentially hogging memory.* Such leaks are so widespread. They even acclaimed Android applications crash or stop performing after a few minutes of usage. The effort dedicated for fixing memory leak problems is evident by the number of commits and issues dedicated to these problems in open source projects such as OsmAnd [25], Firefox for Android [24] and OwnCloud [28]. One particularly affected example is probably the dating application Tinder, that stops loading images of possible dates after 15 minutes of work [38].

In this work, we focus on context leaks in Android native applications written in Java, shipped in APK format containing Dalvik

ICSE-SEIP '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183530

bytecode. We develop, implement and experiment with a static analysis method that identifies context leaks by analyzing the bytecode of the APK. The method has been integrated in the industrial Julia analyzer [20]. Our method works as follows. Firstly, APKs are converted into Java bytecode automatically by using a custom version of Dex2Jar [7]. Secondly, the Java bytecode is analyzed with Julia to identify places where contexts might become reachable from static fields or threads. Lastly, the potential leaks are analyzed systematically *wrt.* their severity. For instance, if the execution of a method leaks a context, the issue is more dangerous if the method contains cycles or performs blocking I/O, hence extending the duration of the leak.

The actual solutions for coping with memory leaks in android applications are based on dynamic and static analysis. The most popular tool for dynamic analysis is Android Monitor [12], which allows developers to identify activities leaked. Another popular tool based on a dynamic approach is Canary [22], which is a library for tracing the allocated objects and detecting memory leaks. Since both approaches rely on dynamic analysis, developers need to have a deep understanding of the activities within the application in order to replicate and catch those errors. There are also tools based on static analysis. The most popular ones for android are Lint [23] and Infer [18]. However both tools only recognize syntactic code patterns for leakage and do not perform a semantic analysis.

We conduct extensive experiments by applying our method to 500 third-party widely used APKs from the Android market. The experiment results suggest that context leaks are potentially widespread in real applications, with various severity degrees. In order to rule out false-positives (due to the limitation of static analysis techniques), we select 8 open source Android applications to manually verify the findings against their source code. We conclude that our method is fast (with an average analysis time of 1 minute and a maximum less than 10 minutes) and precise enough (with an overall accuracy of 71.5%). As a baseline comparison, we compare our method to the Lint tool [23], which is currently used by Android developers to find bugs through syntactical static analysis. The experiment results show that our method performs better than *Lint*, because our technique allows us to identify a larger number of cases than only few syntactic patterns. Also, our results are more precise because we exclude those objects which point only to *ApplicationContext*. In general, our analysis is unsound. Due to the heuristics we adopt for efficiency and avoiding false positives, there might be leaks that we do not identify. In short, the experiment results show that our technique is useful in practice. Lessons learned from its use can be used to study, understand the problem and pave the way to the development of a new commercial analyzer for Android Apps.

The paper is organized as follows. Section 2 describes context leaks in Android. Section 3 defines our static analysis method that systematically finds such context leaks. Section 4 shows how we classify the resulting warnings *w.r.t.* their severity. Section 5 reports experiments with the analysis of real Android applications. Section 6 reviews related work. Section 7 concludes.

2 CONTEXT LEAKS IN ANDROID

In this section, we present background on context leakage in Android Apps. An object has a *life cycle* when its behavior and usability window depend on invocation of methods on the object, explicitly and clearly marked as state transitions, often non-revertible. For instance, in Java, resources such as files or data streams end their life cycle with a call to `close()`. After the call, they cannot be used anymore and are expected to be eligible for garbage collection. Programmers are expected to explicitly call `close()` to help garbage collection. Android extends this idea to *contexts*, with a much larger set of life cycle methods (also known as *callbacks*). Moreover, the Android operating system fires such methods asynchronously, i.e., the programmer does not call them directly. Some events perform object bootstrap; some mark the end of life; in between, other events notify user interaction or the availability of data. Events cannot be rejected: one can only react in the appropriate callback.

If a context reaches its end of life, it is expected to be garbage collectable. However, if the context is reachable from a non-garbage-collectable root, it cannot be garbage collected and it is leaked. In Java, non-garbage-collectable roots include running threads or static fields. Hence, a *context leak occurs in Android if and only if a context has reached its life cycle end but is still reachable from a running thread or from a static field.* This includes the main and the user interface threads.

The severity of a leak depends on its duration. Notable examples of Android contexts are *activities* (screens interacting with the user) and *services* (background tasks). *Broadcast receivers* (background triggers) and *fragments* (portions of user interface) are not contexts but contain a context. Hence they are *context containers*. Programmers often define other context containers. Also, context containers can generate context leaks, since their context cannot be garbage collected if the container itself is not garbage collectable.

In the following, let us focus on activities. When an activity is invoked, Android pushes it on a stack and runs it in the foreground. The activity previously on top loses focus and goes down in the stack. The user interacts with the foreground activity only. Others are kept in the stack until they are invoked and moved onto the top again. This improves user experience by keeping activities in memory without repeated destructions and creations. Activities deep down in the stack are eligible for destruction. Figure 1 shows the activities life cycle. At activity creation, its `onCreate()` method is fired to initialize its state. The activity is now in memory, but invisible. When it becomes visible, Android calls `onStart()`, which may show pictures or start animations. When the Activity moves to the foreground and starts interacting with the user, Android calls `onResume()`, that typically initializes all resources that the activity needs (*e.g.* network sockets or database connections). There are closing methods like `onDestroy()`, `onStop()` and `onPause()`. After `onDestroy()`, the activity cannot be used anymore and is expected to be eligible for garbage collection.

In the literature [15] (including programming forums), expert Android programmers have shown many scenarios in which a leak might occur in Android, which provides a basis to build a tool that finds context leaks in Android code. Namely, the three typical origins of a context leak in Android are: 1) a thread that reaches a context; 2) a static field that reaches a context; 3) a system callback

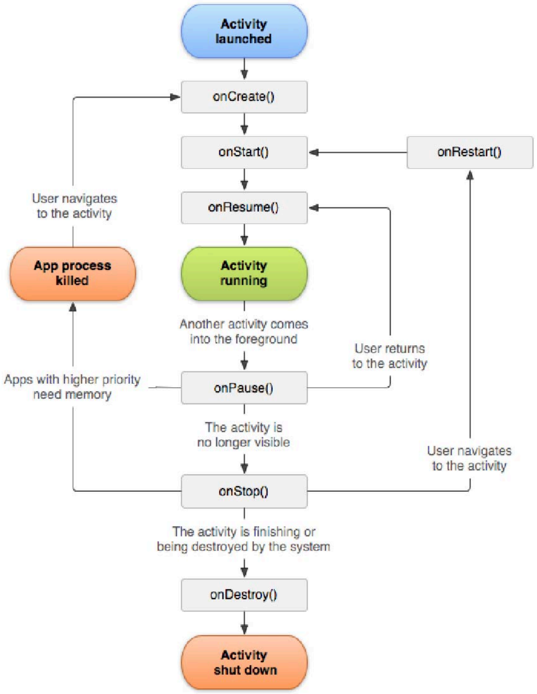
Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 1: The lifecycle of an Android activity.

```
public class TerminalBridge implements VDUDisplay {
  AbsTransport transport = ...;
  private TerminalView parent = null;

  protected void startConnection() {
    Thread connectionThread = new Thread(
      new Runnable() {
        public void run() {
          transport.connect();
        }
      });
   connectionThread.start();
  }

  public final synchronized void parentChanged(
                        TerminalView parent) {
    this.parent = parent;
  }
}
```

Figure 2: A memory leak due to a running thread.

that reaches a context. Our tool is built to identify all these three kinds of leaks. In the following, we report three concrete leaks that our tool found in third-party Android Apps accordingly.

2.1 A Thread Reaches a Context

It is recommended [33] to delegate long tasks (such as network or database operations) to *background threads*, to keep the main application thread reactive. This preserves the responsiveness of the application's user interface. Android provides several thread-related classes (*e.g.* Thread, Runnable, Handler, HandlerThread and AsyncTask [15]). From now on, we call them *thread-like* classes.

Fig. 2 shows an example in which using a thread leads to a leak. It is an example leak that our algorithm finds in OsmAnd, a navigation application based on OpenStreetMap [26]. Method startConnection() in Fig. 2 starts a thread, passing a Runnable implemented as a non-static anonymous inner class. Hence, the latter has a synthetic hidden field that references the parent TerminalBridge, which cannot be garbage-collected before the thread stops. A TerminalBridge holds a parent view that references an activity, as all views. Hence, this view and its activity are also kept in memory until the thread terminates, together with the whole view hierarchy and resources of the activity. It may take a while before the thread terminates since it performs a network operation.

In order to avoid such a leak, the inner class should be named and made static, if possible. Alternatively, the thread must be stopped in the onPause() method of the leaked activity, to keep their lifespan in sync.

2.2 A Static Field Reaches a Context

Static fields belong to a class and not to a specific instance. They become reachable as soon as the class is loaded and they remain so forever. Programmers tend to use static fields as global access points for shared data, such as contexts. In particular, if a context or context container is stored in a static field, possibly indirectly, then it gets leaked. One way to solve this problem is to reset static fields to null. However, this might happen too late, too early, or worse, be simply forgotten.

Figure 3 shows an example of leak due to a callback, that our algorithm finds in Telegram [36], a messaging application. Here, all MapActivitys share a static mapContextMenu field, *i.e.* a container of the last created MapActivity, set inside onCreate(). Since that static field remains reachable at the end of the activity's lifecycle, the activity remains reachable as well, until a new MapActivity is created. A solution here would be to implement onDestroy() to call setMapActivity(null).

2.3 A System Callback Reaches a Context

Android has a set of *managers* to interact with the OS. For instance, the *location manager* allows one to query the device location; the *sensor manager* allows one to access sensor data, such as acceleration; the *audio manager* allows one to intercept audio changes through an interface. The OS provides a Context.getSystemService() method that lazily creates such managers. As usual in Java, lazy creation is achieved through static fields, which comes with all problems shown in Sect. 2.2. Hence, once created, managers are not garbage collected anymore. If callback objects get attached to managers, in order to listen to specific events, they will remain allocated forever, or until they are explicitly detached. All data reachable from such callbacks, possibly a context, will also remain allocated. This is typically the case if the callback is implemented as a non-static inner class.

Figure 4 shows an example. A MediaController implements the callback interface OnAudioFocusChangeListener, that requestAudioFocus() passes to the audio manager. Hence the MediaController and its

```
public class MapActivity extends OsmandActionBarActivity implements... {
  private static MapContextMenu mapContextMenu = new MapContextMenu();
  public void onCreate(Bundle savedInstanceState) {
    mapContextMenu.setMapActivity(this);
  }
}

public class MapContextMenu extends MenuTitleController implements... {
  private MapActivity mapActivity;
  public void setMapActivity(MapActivity mapActivity) {
    this.mapActivity = mapActivity;
  }
}
```

Figure 3: A memory leak due to the store of an activity inside a static field.

```
public class MediaController
        implements OnAudioFocusChangeListener {

  public boolean playAudio(...) {
    NotificationsController.getInstance()
      .audioManager.requestAudioFocus(this, ...);
  }

  private ChatActivity raiseChat = ...;
}
```

Figure 4: A memory leak due to callback

raiseChat activity cannot be garbage-collected anymore. A solution here is to implement the activity's onDestroy() method in such a way to stop the audio and unregister the callback.

3 A NEW STATIC ANALYSIS FOR CONTEXT LEAK DETECTION

In the previous section, we show typical scenarios in which leaks may occur in practice, which provides us a basis to develop a static analysis algorithm that automatically detects such typical context leaks in an Android application. We remark that although these scenarios are not complete (*i.e.* there might be other subtle ways a leak might occur), they are common enough so that they must be properly handled. Namely, the algorithm aims to issue a warning about a potential leak for every field f of the application such that both the following conditions hold:

- f is a static field of some class, or is an instance (possibly synthetic) field of a thread-like class or of a callback implementation;
- f has a static type assignable to Context or to a context container.

These conditions are syntactical and hence easily implementable. The precision of the algorithm can be improved by observing that some contexts cannot induce a memory leak since, by definition, they cannot be garbage collected before the application stops. Namely, the running application itself is a context in Android, accessible through the getApplicationContext() method of each activity. That special context remains allocated in memory until the application stops and can be safely stored in static fields or made reachable from a thread, without inducing any leak. Hence, if f can *only* hold an application context at runtime, then the algorithm above need not issue a warning about a potential leak due to f.

Below we provide further details about the algorithm. Namely, we show how it is possible to identify context container classes and how it is possible to know that a field can *only* contain an application context.

3.1 Identification of Context Containers

A context container is an object that can reach a context, but not the context itself. Examples are objects of class android.view.View. The problem is now to identify all classes that might have an instance which is a context container.

A simple algorithm first selects the set I of classes that define a field of type Context, or supertype, or subtype. Then it expands I for all classes C that define a field whose type is in I, until fixpoint. This algorithm is sound but, in practice, it is extremely imprecise since it ends up classifying almost all classes as potential context containers. This is due to fields of type Object, used for instance in collection classes, that can potentially reach every other class.

Hence, we have decided to adopt the following two heuristics instead of the above-mentioned naive algorithm:

Heuristic over the constructors It first selects the set I of classes with a constructor that accepts a parameter of type Context or subtype. Then it expands I for all classes C such that C has a constructor with a parameter of type in I, until fixpoint. For instance, this heuristic spots views and fragments as context containers. Figure 5 depicts an example where this heuristic is more suitable. Compared to existing approaches adopted by popular tools like Lint, our heuristic is empirically shown to be at least as precise as Lint's one since all warnings raised by our method are also detected by Lint. Note that Lint adopts an exploration strategy which scans all fields of the classes,

Heuristic over the fields It first selects the set I of classes that implement an interface and have a field of type Context or subtype. Then it expands I for all classes C such that C has an instance field whose type is in I, until fixpoint. This heuristic is similar to the sound algorithm, but only considers subtypes and classes that implement an interface, such as callbacks. Indeed, this heuristic spots implementations of callbacks or listeners. Figure 6 depicts an example where it is more convenient analyzing fields than constructors.

```
public class MainActivity extends AppCompatActivity {
  private ArrayAdapter<String> mArrayAdapter;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
      mArrayAdapter = new ArrayAdapter<String>(this, ...);
    }
  }
```

Figure 5: Example of code suitable for heuristic by constructors

3.2 Identification of Application Contexts

Here we show how it is possible to know if a field can only contain an application context at runtime. This information can also be used to improve the heuristics shown in Sect. 3.1, by considering only constructors whose parameter is not necessarily an application context, or by considering only those fields whose values are not necessarily an application context.

We use a semantic approach that finds all *producers* of a value, based on the existing tool Julia, which a semantic Java analyzer that abstracts Android life-cycles and automatically recognizes entry points in Android applications [31, 32]. Note that our approach has been integrated into Julia. Julia also provides a data-flow algorithm that systematically yields the bytecode instructions that produce a value at a given program point. It is a traditional backwards data-flow reaching definitions analysis [1] that follows the flow of data across bytecode instructions, as long as it remains confined to the operand stack and local variables. If all producers of a value v are `null` or the return value of method `getApplicationContext()` or a subclass of `Application`, then v can only be an application context. Section 5 measures how much the availability of this extra information improves the precision of the leak analysis algorithm.

4 WARNING CLASSIFICATION

The algorithm in Section 3 finds fields that might induce a context leak. For each such field, the algorithm issues a warning. But warnings have different severity, depending on the type of the leaked object (*e.g.* an activity, a view…) and the duration of the leak. In this section, we show how to classify warnings *w.r.t.* these two dimensions.

Section 2 shows that a leak occurs when a context or context container is reachable from a static field f or from an instance field f of a thread-like class or callback. Hence, it is possible to classify a leak *w.r.t.* the class of the leaked object, by distinguishing leaks *w.r.t.* the static type of f. This classification captures an aspect of the danger of a leak: objects with lifecycle are in general larger than views; `ImageViews` are larger than a `View` since they contain a bitmap.

A leak can also be classified *w.r.t.* its duration: the longer, the more dangerous. Leaks due to contexts reachable from static fields or callbacks last *forever*, since the field is never garbage collected. Leaks due to thread-like classes last as long as the operations performed by the method m that implements the body of the thread (or by one of the methods that m invokes, possibly indirectly). Namely, we can classify the duration of m with the following grades, in increasing order of danger:

- *linear* methods contain no loops;
- *library interacting* methods call library functions;
- *looping* methods contain loops or recursion;
- *file system* methods call file system functions;
- *networking* methods call networking functions.

A method might fall in more classes, in which case the highest grade is taken. We have implemented an algorithm that computes the grade $g(m)$ of each method m in the program. Initially, it sets $g(m)$ to the grade resulting from the code of m, without looking at the methods that m invokes. Then, the grade is iteratively updated by setting $g(m)$ to the highest grade between $g(m)$ itself and that of the methods that m calls inside its body. This process is repeated until a fixpoint is reached.

5 EXPERIMENTS

Our method has been coded and integrated in the Julia static analyzer for Java bytecode [20], which is a rich framework for the development of new analyses. This is briefly how Julia works. Since Android applications are packaged into Dalvik bytecode [6] (dex), the tool Dex2Jar [7] has been used to translate Dalvik into Java bytecode. We used a modified version of that tool, in order to translate also the debug information, needed to recover the source line numbers of the leaks. In the following, we report experiments with the implementation of the leak analysis. All experiments were performed on a Windows 7 64bit machine with 16GB of RAM and an Intel i5 processor at 3.30MHz.

The experiments are conducted to answer the following research questions. Firstly, we would like to evaluate whether our target leaks are common in Android applications and whether our method scalable and efficient to identify them in real-world Android applications (RQ1). Secondly, because our method is based on static analysis, which as we all understand sometimes suffers from the issue of false positives, we would like to evaluate what is the likelihood of our method reporting false positives (RQ2). Lastly, we would like to evaluate whether the leaks in practice are severe enough so that our effort is justified (RQ3).

RQ1. In order to answer RQ1, we systematically downloaded 500 APKs from Google Play [13], and applied our method to analyze them. The applications were chosen by ordering them according to the number of downloads.

For all APKs, each analysis never lasted more than 10 minutes with an average analysis time of 1 minute and a standard deviation of 52, this gap depends on the nature of the APKs. Furthermore, each analysis is run both with and without the optimization that lets application contexts be safely leaked (Section 3.2). This idea is to evaluate how much this optimization actually improves the precision of the results.

Figure 7 summarizes the experiment results. We put apps in buckets *w.r.t.* the number of warnings that the analysis issues. The figure shows that the analysis issues no more than 30 warnings for 336 apps. Without the optimization in Section 3.2 about application contexts, the same number reduces to 331, *i.e.* a few more warnings are generated per application. This suggests that leaks are widespread in apps currently downloaded by Android users.

```
public class MainActivity extends AppCompatActivity {
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    WifiManager wifiMgr = (WifiManager) this.getSystemService(Context.WIFI_SERVICE);
    wifiMgr.startWps(.., new MyWifiCallback());
  }

  public class MyWifiCallback extends WifiManager.WpsCallback {
    /* ... */
  }
}
```

Figure 6: Example of code suitable for heuristic by fields

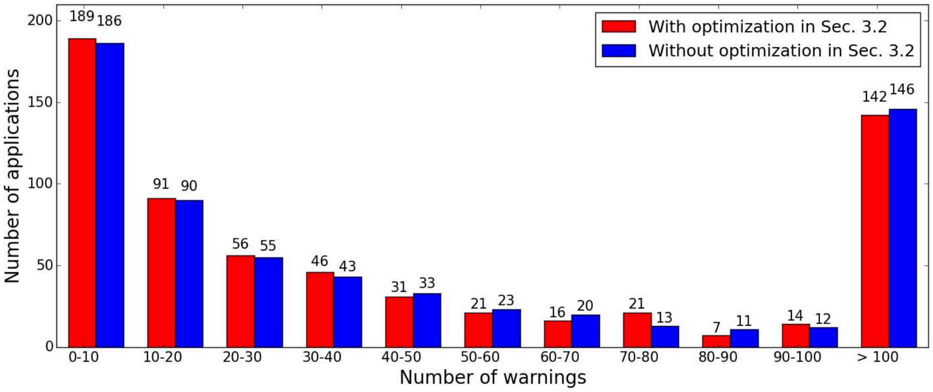

Figure 7: Number of warnings per application.

	LoC	version
webTube [39]	163854	741620d8
aSQLiteManager [3]	127186	29606afb
ConnectBot [5]	204813	fa590d1a
OwnCloud [29]	218542	a82bcce2
Kiwix [21]	179855	559eaee6
Firefox [10]	261601	a4071341
OsmAnd [27]	365205	e9486ab1
Telegram [37]	311004	cc7f3116

Figure 8: The 8 open source Android applications that we have analyzed

Furthermore, we observe that there are a large number of apps which have a large number of warnings and a large number of apps which have few warnings. This pattern can be explained by the nature of Google Play, on one hand, the most popular applications receive more attention by the developers because they are services provided by companies. On the other hand, Android market is also full of applications which can be considered attempts. They were uploaded but they do not receive many attentions by their developers. The quality of Google Play's applications is often matter of discussion by technical magazines [14].

RQ2. The previous experiment shows that our method can find many leaks in real-world apps. The question is then: how many of them are actual leaks and how many of them are false positives? Answering this question requires us to investigate the source code of these APKs. Unfortunately, the source code of these APKs is not available. We thus instead analyze 8 open source apps in order to answer RQ2, with the hope that the results drawn from these 8 apps are representative of real-world apps in general. According to the nature of memory leaks discuss in Section 2, we opted for those apps which are related to asynchronous operations with the file system or network activities, or else they provide a multimedia interaction. Figure 8 shows the details of the 8 apps, *i.e.* their number of lines of source code (**LoC**) and commit id (**version**).

Figure 9 shows the experiment results. For a baseline comparison, we present the results obtained using the popular Lint tool as well. Lint is the standard static analyzer integrated in Android Studio, it recognizes syntactical patterns in the source code that might bring to memory leaks. The main difference between Lint and our approach leans in our semantic analysis. In Lint approach, it might happen that even if a pointer refers only to *ApplicationContext*, it is still reported as dangerous. Lint does not inspect the possible values of that field. Also, Lint does not consider dangerous those pointers which might be cast to context, for instance, collections or *Objects* fields. On the other hand, our approach analyses all *producers* of a field, and so we can filter out those pointers which refer to only safe contexts (*i.e. ApplicationContext*). In our experiment we split warnings *w.r.t.* the origin of the leak: threads (Section 2.1), static fields (Section 2.2) or callbacks (Section 2.3). Experiments are performed

	Julia						Lint			
	threads		static		callbacks		threads		static	
	true	false	true	false	true	false	true	false	true	false
webTube	0	0(0)	1	0	0	0(0)	0	0(0)	1	0
aSQLiteManager	7	0(0)	1	0	0	1(0)	0	0(0)	1	0
ConnectBot	29	0(0)	0	3	0	3(0)	1	0(0)	0	1
OwnCloud	24	31(19)	0	6	1	2(0)	2	0(0)	0	1
Kiwix	7	0(0)	1	0	1	2(1)	1	0(0)	1	0
Firefox	213	70(12)	16	42	4	26(24)	0	0(0)	6	13
OsmAnd	260	29(2)	13	11	4	13(0)	2	0(0)	0	0
Telegram	237	9(0)	6	12	1	9(5)	0	0(0)	5	0

Figure 9: True and false positives for the analysis of 8 open source Android applications. Warnings are distinguished *w.r.t.* the categories identified in Sections 2.1, 2.2 and 2.3.

with Julia and with Lint. For the latter, we have considered checkers *StaticFieldLeak*. Note that Lint has no checkers for leaks through callbacks nor through thread-like classes other than Handler. It does not perform any analysis to assess that a leak is benign since it involves the application context, as we do instead (Section 3.2).

For each category of leak, Figure 9 counts true and false alarms and reports, under parentheses, the number of false alarms due to the fact that a thread is stopped or a callback is unregistered as soon as an object terminates its life cycle, so that it cannot be leaked. This is a typical Android programming pattern, that our algorithm does not handle. According to Figure 9, our algorithm performs better with memory leak warnings related to threads (81.95%), while static fields warnings result less precise (33.93%). The worst performances are presented for callback’s memory leak (11.7%). For instance, for Firefox our algorithm reports 213 true positives and 70 false positives about objects leaked through threads; 12 are due to threads stopped when an object ends its life cycle. Figure 9 clearly shows the strict superiority of our technique *w.r.t.* Lint, i.e., we find many more true positives and also leaks due to callbacks implementations. Furthermore, we find all true positives found by Lint.

We report below further examples of context leaks that our algorithm finds in some notable apps that we have analyzed along with some example of false positive, these latters help to understand the limitation of our approach.

Example of false positive. The more recurrent cause of false positive in our approach depends by a correct handling of the objects within the applications. In this sense, Figure 10 shows an example of possible leak through thread in OsmAnd. In this case the static field `mThread` is a thread instance of a inner class of `ContributionVersionActivity`. Therefore that thread might leak its own outer class. However, the activity handles the object properly since it starts `mThread` in `onCreate()` method, and also stops the thread in `onDestroy()` method. We plan to deepen these cases in a future release of our algorithm.

```java
public class ContributionVersionActivity
 extends OsmandListActivity {

  private static ContributionVersionActivityThread mThread
   = new ContributionVersionActivityThread();

  @Override
  protected void onCreate(Bundle savedState) {
    super.onCreate(savedState);
    /* ... */
    // this line starts the mThread field above
    startThreadOperation(/* ... */);
  }

  @Override
  protected void onDestroy() {
    super.onDestroy();
    // this line stops the mThread field above
    mThread.setActivity(null);
    /* ... */
  }
}
```

Figure 10: Example of false positive in OsmAnd

```java
public class BrowserApp extends GeckoApp ... {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    Permissions.from(this).withPermissions(Manifest
      .permission.WRITE_EXTERNAL_STORAGE)
      .doNotPrompt()
      .andFallback(new Runnable() {
        @Override
        public void run() {
          showUpdaterPermissionSnackbar();
        }
      })
      .run();
  }
}
```

Figure 11: Example of memory leak in Firefox App.

Context Leaks Found in the Firefox App. Class `BrowserApp`, Figure 11 extends `Activity` (and not `Application`, as its name might suggest) and the anonymous implementation of `Runnable` references that activity and hinders its garbage collection also after its destruction. This is particularly bad since the implementation of `showUpdaterPermissionSnackbar()` (not shown below) performs networking:

Another example of context leak in the Firefox app is depicted in Figure 12. Class `HomeFragment` is a context container. The anonymous inner `Runnable` class leaks that context until the execution of

```
public class BrowserSearch extends HomeFragment ... {
  private void setSuggestionsEnabled (
                        final boolean enabled) {
    /* ... */
    // Make suggestions appear immediately
    // after the user opts in
    ThreadUtils.postToBackgroundThread(
       new Runnable() {
         @Override
         public void run() {
           SuggestClient client = mSuggestClient;
           if (client != null)
             client.query(mSearchTerm);
         }
       }
    );
  }
}

public class SuggestClient {
  public ArrayList<String> query(String query) {
    /* ... */
    if (!NetworkUtils.isConnected(mContext) &&
        mCheckNetwork) {
      Log.i(LOGTAG, "Not␣connected␣to␣network");
      return suggestions;
    }

    String encoded =
         URLEncoder.encode(query,"UTF-8");
    String suggestUri = mSuggestTemplate.replace(
         "__searchTerms__", encoded);

    URL url = new URL(suggestUri);
    String json = null;
    HttpURLConnection urlConnection = null;
    InputStream in = null;
    try {
      urlConnection =
         (HttpURLConnection) url.openConnection();
      urlConnection.setConnectTimeout(mTimeout);
      urlConnection.setRequestProperty(
         "User-Agent", USER_AGENT);
      in = new BufferedInputStream(
         urlConnection.getInputStream());
      json = convertStreamToString(in);
    }
    finally {  /* close connection (omitted) */  }
    /* ... */
  }
}
```

Figure 12: Example of memory leak in Firefox App.

its `run()` method completes. But that execution actually performs a potentially long network connection and string conversion, inside class `SuggestClient`:

Context Leak Found in the Telegram App. Class `GcmInstanceIDListenerService` (Figure 13) extends `Service` and the anonymous implementation of `Runnable` hinders its garbage collection also after its destruction. Also in this case, it can be verified that `postInitApplication()` performs a check on the Internet connection, which might take a while. This increases the severity of the warning:

```
public class GcmInstanceIDListenerService
         extends InstanceIDListenerService {

  @Override
  public void onTokenRefresh() {
    AndroidUtilities.runOnUIThread(new Runnable() {
      @Override
      public void run() {
        ApplicationLoader.postInitApplication();
        Intent intent = new Intent(ApplicationLoader.
             applicationContext,
          GcmRegistrationIntentService.class);
        startService(intent);
      }
    });
  }
}

public class ApplicationLoader extends Application {
  public static void postInitApplication() {
    /* ... */
    ConnectionsManager.getInstance().init(BuildVars.BUILD_VERSION,
      TLRPC.LAYER, BuildVars.APP_ID, deviceModel, systemVersion,
      appVersion, langCode, configPath, FileLog.getNetworkLogPath
           (),
      UserConfig.getClientUserId());
    /* ... */
  }
}

public class ConnectionsManager {
  public void init(/* ... */) {
    native_init(/* ... */);
    checkConnection();   // THIS PERFORMS AN INTERNET CONNECTION
    /* ... */
  }
}
```

Figure 13: Example of memory leak in Telegram

	Sum	Min	Max	Mean	SD
webTube	3	3	3	3	NA
aSQLiteManager	10	1	2	1	<1
ConnectBot	87	<1	9	3	3
OwnCloud	97	<1	37	4	8
Kiwix	501	<1	156	56	76
Firefox	668	<1	50	3	8

Figure 14: Statistics about memory saving in open source projects, measured in KB. SD stands for standard deviation. WebTube has only one true positive warning and thus no standard deviation

RQ3. In order to answer RQ3, we estimate the severity of each leak according to the size of the leak and the duration of the leak. For the leaks due to threads, it is possible to estimate the duration of the `run()` method of the thread. For all leaks, it is also possible to identify the kind of object that is leaked and estimate the amount of memory loose. This information is important for evaluating the severity of the leak (its duration and the size of memory that is leaked) and for collecting statistics about the most frequent leak scenarios. We did this during the analysis of the 500 APKs, by performing a further analysis of the warning found in the open-source projects. Figure 15 classifies the `run()` methods (or equivalent) of the threads at the origin of a leak by means of the heuristic of increasing duration and danger from Section 4. Figure 16 classifies instead the objects potentially involved in leaks. The most frequently leaked objects are activities (23.21%), context containers created by the programmer (22.41%) or from the standard Android

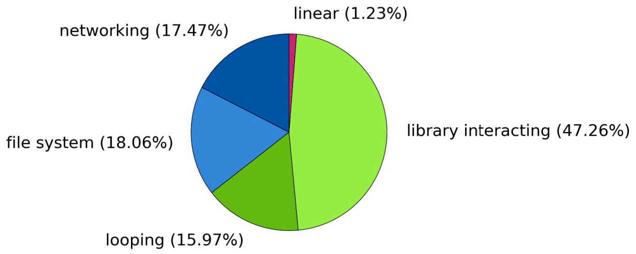

Figure 15: Classification of the `run()` method of the threads at the origin of leaks.

Type of leaked object	#	%
BroadcastReceiver	223	0.47%
Fragment	469	0.98%
Adapter	686	1.44%
ImageView	709	1.49%
Dialog	983	2.06%
Service	1100	2.31%
Context	1762	3.69%
Collections	1799	3.77%
Other interfaces	3574	7.49%
View	6851	14.37%
Android context container	7778	16.31%
Custom context container	10689	22.41%
Activity	11068	23.21%
Total:	47691	

Figure 16: Type of objects involved in context leaks.

library (16.31%). We distinguish between `View` and `ImageView` since the latter contains a bitmap and its leakage is consequently more severe.

We then precisely estimate the amount of memory involved by a memory leak based on the open-source projects. That is, we analyzed the true positive warnings detected in the open-source projects and estimate the size of the memory leak with the following steps.

(1) we debugged the project;
(2) we triggered the warning after inspecting the code;
(3) we dumped the memory by using Android Monitor tool [2].

From the memory dump, we measured the retained size for each object involved in the memory leak. We considered retained size because it is the amount of memory that might be collected by the garbage collector if that warning was solved [34]. The results of this experiment are shown in Figure 14, where we present the sum, minimum, maximum and the average of memory lost for each open-source project. We did not manage to debug OsaAnd and Telegram because of technical issues with those projects and Android Studio. Based on the results, we observe that on average each warning solved can release around 2KB, except for Kiwix project which has more the 50KB on average. Also, from this experiment, we can state that in some case the minimum amount of memory leaked can be few byes, for instance for Firefox and ConnectBot. Finally, the sum column shows that solving all warnings could result in memory saving of more than 100KB for Kiwix, Firefox and OwnCloud.

6 RELATED WORK

Our technique is static, while most techniques for finding leaks are dynamic leak detectors. Static analysis has been applied to resource leaks. A notable example is [16], that considers Android components with lifecycle as well. In the case of context leaks, only Lint applied static analysis in the past. Our experiments have shown the superiority of our technique *w.r.t.* Lint. Blackshear et al. [4] proposed a solution based on symbolic for finding leaks in heap, their work is not comparable with ours because they aim to detect different kind of leaked objects than context-leak ones.

Google has best practices for leak prevention [11] and tools to analyze and debug memory at runtime [12]. Eclipse MAT [8] helps analyze memory at runtime. Leak Canary [22] inserts memory leak detection code in applications, that triggers at runtime. Aspect-oriented programming can spot memory leaks at runtime [19], but that work has not been extended from Java to Android yet. Testing has been used to find leaks [40]. They identify a sequence of dangerous user actions, that often lead to a leak, and replay them automatically to simulate dangerous behaviors. Testing has been applied to some common memory leak patterns [35], similar to those in Section 2. Testing can also find where the garbage collector cannot correctly deallocate some resource [30], a problem somehow similar to context leaks. Both [41] and [17] build unit tests that find memory leaks by stressing the application, raising an exception at runtime. In [9], the author analyses how callbacks are register/deregister in a framework. In particular, he suggests some simple static analysis which helps the developer to identity callbacks not deregistered properly. This solution is substantially different than our because it does not consider Android framework issues, *i.e.* , it does not consider contexts and their origins.

7 CONCLUSION

This article proposes a static leak analysis that is able to find context leaks in Android applications. The algorithm proposed is not sound, but it may give insights into why such leaks occur (threads, static fields, callbacks implementations) and how frequent we can expect this to happen in practice. Moreover, manual analysis of the analysis results allows us to gather lessons about possible, future improvements to the analysis and gives a direction for this, related to the identification of contexts that are made unreachable at the end of the lifecycle of an object (Fig. 9). Finally, we have analyzed the duration of the leaks (Fig. 15) and the typical kinds of the leaked object (Fig. 16). Also, these results are invaluable information for the future definition of a more precise and sound static analysis.

ACKNOWLEDGEMENTS

We thank Fausto Spoto (University of Verona), Ètienne Payet (Universitè de La Rèunion) and the entire team of Julia S.r.l. for their collaboration. Their help was fundamental to achieve the results presented in this paper.

REFERENCES

[1] R. Aho, A. V. Sethi and J. D. Ullman. *Compilers, Principles Techniques and Tools*. Addison Wesley Publishing Company, 1986.

[2] https://developer.android.com/studio/profile/android-monitor.html.

[3] https://sourceforge.net/p/asqlitemanager/code.

[4] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: Precise refutations for heap reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 275–286, New York, NY, USA, 2013. ACM.

[5] https://github.com/connectbot/connectbot.git.

[6] https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[7] https://github.com/pxb1988/dex2jar.

[8] http://www.eclipse.org/mat/.

[9] George Fairbanks. *Design Fragments*. PhD thesis, School of Computer Science Carnegie Mellon University Pittsburgh, 2007.

[10] https://hg.mozilla.org/mozilla-central.

[11] http://developer.android.com/tools/debugging/debugging-memory.html.

[12] http://developer.android.com/tools/performance/memory-monitor/index.html.

[13] https://play.google.com.

[14] http://www.netimperative.com/2017/08/google-play-starts-downranking-poor-quality-apps/.

[15] A. Göransson. *Efficient Android Threading*. O'Reilly Media, June 2014.

[16] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and Detecting Resource Leaks in Android Applications. In *Proc. of Automated Software Engineering (ASE)*, pages 389–398, Nov 2013.

[17] L. Hong, J. Qian, and J. Cui. Automated Unit-level Testing of Java Memory Leaks. *Computer Modelling and New Technologies*, 18(11), 2014.

[18] http://fbinfer.com/.

[19] M. Jain and D. Gopalani. Memory Leakage Testing using Aspects. In *Proc. of Applied and Theoretical Computing and Communication Technology (iCATccT)*, pages 436–440, Oct 2015.

[20] http://www.juliasoft.com.

[21] https://github.com/kiwix/kiwix.git.

[22] https://github.com/square/leakcanary/.

[23] http://tools.android.com/tips/lint.

[24] Mozilla. Commits related to memory leak in mozilla firefox for android, October 2016. https://hg.mozilla.org/mozilla-central/log?rev=memory+leak.

[25] Open Street Map. Issues related to memory leak in osmand, October 2017. https://github.com/osmandapp/Osmand/issues?utf8=%E2%9C%93&q=memory%20leak%20.

[26] http://osmand.net/.

[27] https://github.com/osmandapp/Osmand.git.

[28] OwnCloud. Issues related to memory leak in owncloud android, October 2017. https://github.com/owncloud/android/issues?utf8=%E2%9C%93&q=memory%20leak.

[29] https://github.com/owncloud/android.

[30] J. Park and B. Choi. Automated Memory Leakage Detection in Android-Based Systems. *International Journal of Control and Automation*, 5(2):35–42, 2012.

[31] ÃĽ. Payet and F. Spoto. Static analysis of Android programs. *Information and Software Technology*, 54(11):1192 – 1201, 2012.

[32] Étienne Payet and Fausto Spoto. *Static Analysis of Android Programs*, pages 439–445. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[33] https://developer.android.com/guide/components/processes-and-threads.html.

[34] https://www.ibm.com/support/knowledgecenter/en/SS3KLZ/com.ibm.java.diagnostics.memory.analyzer.doc/shallowretainedheap.html.

[35] H. Shahriar, S. North, and E. Mawangi. Testing of Memory Leak in Android Applications. In *Proc. of High-Assurance Systems Engineering (HASE)*, pages 176–183, Jan 2014.

[36] https://telegram.org/.

[37] https://github.com/DrKLO/Telegram.git.

[38] https://m.reddit.com/r/Tinder/comments/42bfa8/i_swear_to_god_tinder_is_doing_this_shit_on/, 2016.

[39] https://github.com/martykan/webTube.git.

[40] D. Yan, S. Yang, and A. Rountev. Systematic Testing for Resource Leaks in Android Applications. In *Proc. of Software Reliability Engineering (ISSRE)*, pages 411–420, Nov 2013.

[41] H. Zhang, H. Wu, and A. Rountev. Automated Test Generation for Detection of Leaks in Android Applications. In *Proc. of Automation of Software Test (AST)*, pages 64–70. ACM, 2016.