IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 17, 2019, accepted January 3, 2020, date of publication January 14, 2020, date of current version January 21, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2966522

Do Memories Haunt You? An Automated Black Box Testing Approach for Detecting Memory Leaks in Android Apps

DOMENICO AMALFITANO[1], VINCENZO RICCIO[2], PORFIRIO TRAMONTANA[1], AND ANNA RITA FASOLINO[1]

[1]Department of Electrical Engineering and Information Technology, University of Naples Federico II, 80125 Naples, Italy
[2]Software Institute, Università della Svizzera italiana, 6904 Lugano, Switzerland

Corresponding author: Anna Rita Fasolino (fasolino@unina.it)

This work was supported in part by the Context of the Research Project OPL-APPS, IIoT Open Platform and Manufacturing Applications, funded by the Italian Ministry for University and Research (MIUR), under Grant ARS01_00615.

ABSTRACT Memory leaks represent a remarkable problem for mobile app developers since a waste of memory due to bad programming practices may reduce the available memory of the device, slow down the apps, reduce their responsiveness and, in the worst cases, they may cause the crash of the app. A common cause of memory leaks in the specific context of Android apps is the bad handling of the events tied to the Activity Lifecycle. In order to detect and characterize these memory leaks, we present FunesDroid, a tool-supported black box technique for the automatic detection of memory leaks tied to the Activity Lifecycle in Android apps. FunesDroid implements a testing approach that can find memory leaks by analyzing unnecessary heap object replications after the execution of three different sequences of Activity Lifecycle events. In the paper, we present an exploratory study that shows the capability of the proposed technique to detect memory leaks and to characterize them in terms of their size, persistence and growth trend. The study also illustrates how memory leak causes can be detected with the support of the information provided by the FunesDroid tool.

INDEX TERMS Android, Android activity lifecycle, memory leak, resource leak, Android testing.

I. INTRODUCTION

The number of users of mobile technology and smartphones is steadily growing and has surpassed 3 billion in november 2019.[1] Introduced by Google in 2007, Android is today the world's most popular mobile operating system. Suffice it to say that Android accounted for around 88 percent of all smartphone sales to end users worldwide in the second quarter of 2018.[2]

More and more people around the world rely on mobile software applications (*apps*) to carry out various daily tasks. And thus, the demand for quality to mobile apps has grown together with their spread. App users require them to be reliable, efficient, secure, usable. Therefore, failures exposed by an app may have a negative impact on the user experience and lead users to look for another application that offers the same features among about 2.47 million apps available on the official Google app store at September 2019.[3] As a consequence, mobile developers should give proper consideration to the quality of their applications by adopting suitable quality assurance techniques, such as testing [1]. Several techniques and tools are currently available for testing an Android app before it is published to the market [2], [3]. Test automation tools can facilitate software testing activities since they save humans from routine, time consuming and error prone manual tasks [4], [5].

Testing activities should focus both on functional and non-functional requirements of Android apps. Performance testing is a type of non-functional testing that intends to determine how a system performs in terms of responsiveness and stability under a certain load. Performance testing is particularly relevant to mobile apps, since they may be exposed

The associate editor coordinating the review of this manuscript and approving it for publication was Weizhi Meng.

[1]https://www.statista.com/statistics/330695/
[2]https://www.statista.com/statistics/266136/
[3]https://www.statista.com/statistics/276623/

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

to well-known problems of software aging [6], [7]. Software aging refers to the progressive performance degradation of long-time running software, which may by caused by the reduction of available memory [8], or the increasing use of virtual memory [9], and may cause worsening of application responsiveness [10], and application or system crashes [11]. Software aging is known to occur for Android smartphones and may greatly affect user's experience, especially after a long period of usage.

Memory leaks are one of the main causes of software aging [12] and performance degradation in Android apps [13], [14]. A memory leak is a type of resource leak [15] that occurs when a computer program incorrectly manages memory allocations in such a way that memory which is no longer needed is not released. Several examples of problems caused by memory leaks can be found in the Google's Android project.[4]

In Android a major source of issues, including memory leaks, is the mishandling of the Activity lifecycle, i.e. a mobile-specific feature that allows users to smoothly navigate through an app and switch between apps [1], [16]–[18].

Memory leaks impact a considerable number of real Android apps, as it has been shown in the recent work by Toffalini *et al.* [14], and therefore a number of solutions to detect them have been proposed in the literature. Since memory leaks are usually due to bad programming practices that negatively impact the app's memory usage, several source code static analysis approaches have been proposed in the literature to detect possible root causes of Android memory leaks [15], [19]–[22]. A key limitation of these approaches is that they focus only on subsets of bad practices. Moreover, they may report a considerable number of false positives [14].

Another family of approaches focuses instead on the effects produced by memory leaks and exploits dynamic analysis and memory monitoring to detect those effects, such as in the case of the LeakCanary library [23].

The effectiveness of these approaches depends on their capability to trigger program executions that cause memory leaks. However, often they require the source code instrumentation and, therefore, they are not applicable when the app code is not available or the Android version is not supported by the tool.

Other testing techniques have been proposed in the literature that are based on the execution of Android-specific events able to trigger memory object replications [24]. These testing techniques require models of the app to generate test cases and, therefore, their usage is not straightforward when such a model is not available.

To overcome some limitations of existing approaches, in this paper we propose a technique that automates the detection of the specific category of memory leaks in Android apps that are tied to the Android Activity lifecycle. Our technique combines automated testing with memory analysis and has been implemented in a tool called FunesDroid. The tool systematically tests each Activity composing an app by leveraging mobile-specific events able to exercise the Activity lifecycle and to expose possible memory leaks. Therefore, it detects object replications by automatically comparing the memory state before the event execution and after the event execution.

This technique is completely black-box, since it does not require the source code of the app, but just the Android Package (APK) file used for its distribution and installation. However, when the source code is available, the information provided by the tool can be also exploited to aid the developer in finding the root causes of the observed leaks.

We used this technique to analyze 283 real Android apps in order to assess the FunesDroid capability in detecting potential memory leaks in real Android apps. We found that about 37% of the considered apps showed object replications. Moreover, we carried out a study on a subset of these applications, aimed at characterizing the detected memory leaks in terms of their persistence, size, and growth trend. The results of the analysis allowed us to distinguish bad programming practices causing permanent or temporary memory leaks having constant or increasing size and to find the causes of the leaks.

The remainder of the paper is structured as follows. Section II provides the background and a motivating example. Section III describes the proposed testing technique and the implemented tool. In Sections IV and V, we describe the experiments that were performed. Section VI provides related work and a qualitative comparison between our approach and similar ones proposed in the literature. Finally, Section VII draws the conclusions and presents future work.

II. BACKGROUND ON MEMORY LEAKS DUE TO ANDROID ACTIVITY LIFECYCLE

A. MEMORY MANAGEMENT AND MEMORY LEAKS IN ANDROID

Each Android app runs in the context of a Virtual Machine such as Android RunTime (ART) or its predecessor Dalvik.[5] When an application starts, the Android system starts a new Linux process for the application with a single thread of execution and assigns it the required resources, including heap memory space. To maintain a functional multi-tasking environment, Android sets a hard limit on the heap size for each app. The exact heap size limit varies between devices based on how much RAM the device has available overall. If an app has reached the heap capacity and tries to allocate more memory, it can receive an OutOfMemoryError.[6] Moreover, when users switch between apps, Android keeps apps that are not foreground in a least-recently used (LRU) cache, thereby making the app switching faster. As the system runs low on memory, it kills processes in the LRU cache beginning with the process least recently used. The system

[4]https://issuetracker.google.com/issues?q=memory leak

[5]https://source.android.com/devices/tech/dalvik/

[6]https://developer.android.com/topic/performance/memory-overview#RestrictingMemory

also accounts for processes that hold onto the most memory and can terminate them to free up RAM.[7]

Android apps, like Java programs, do not explicitly destroy objects in the heap memory space. Instead, the heap memory is optimized by the Garbage Collector (GC). The purpose of the garbage collector is to free up memory space by destroying objects in the heap that are not directly or indirectly referenced by active objects. The garbage collector usually acts in two phases: in the labeling phase, it looks for objects that cannot be destroyed labeling them as *GC roots*, and recursively labels as *reachable objects* all the objects that are directly or indirectly referenced by the GC roots. An example of GC roots are static objects, that cannot be deallocated during the app execution. In the latter phase, the garbage collector frees heap memory space by deallocating all the unlabeled objects [25].

This mechanism can be the cause of a general problem called memory leak. According to Göhransson [13], a memory leak is defined as a portion of memory allocated by the application that is not used anymore but never identified by the Garbage Collector as freeable memory. This definition may include memory allocated for too long, such as memory occupied by objects referenced by a long running thread.

Memory leaks may cause the heap utilization ratio to continuously increase. As a result, garbage collection (GC) is frequently triggered, with the consequent increasing in the response time of the application [26].

An Android-specific triggering cause of memory leaks is represented by events that exercise the Activity lifecycle, whose influence on the garbage collector behavior will discussed in the next subsection.

B. ANDROID ACTIVITY LIFECYCLE

Activity classes are the essential building blocks of Android apps; an Activity can be seen as a single GUI through which the users can access the features offered by the app. An Activity is implemented as a subclass of the *Activity* class, defined in the Android Framework. The Activity instances exercised by the user are managed as an Activity stack by the Android OS. A user navigates through, out of, and back to an app but only the Activity at the top of the stack is active in the foreground of the screen. To ensure a smooth transition between the screens, the other Activities are kept in the stack. This allows the user to navigate to a previously exercised Activity without losing its progress and information. The system can decide to get rid of an Activity in background to free up memory space.

To provide this rich user experience, Android Activity objects have a proper lifecycle, transitioning through different states. Figure 1 shows the Activity lifecycle as it is illustrated in the official Android Developer Guide.[8] The rounded rectangles represent all the states an Activity object can be in; the edges are labeled with the callback methods that are invoked by the Android platform when an Activity transits between states.

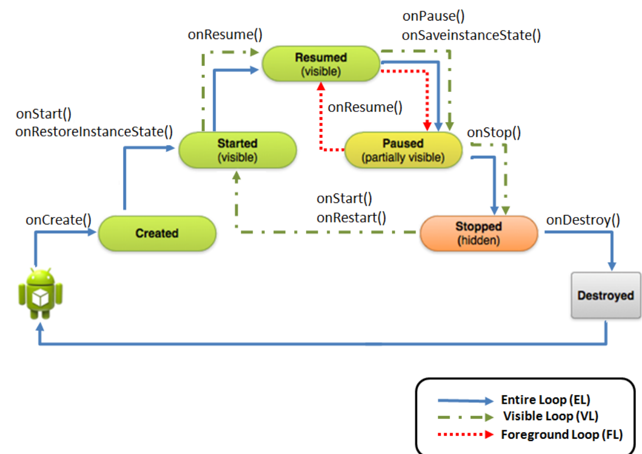

FIGURE 1. The android activity lifecycle key loops.

The Activity visible in the foreground of the screen and interacting with the user is in the *Resumed* state, either it is created for the first time or resumed from the *Paused* or *Stopped* states. When an Activity has lost focus but is still visible (e.g., a system modal dialog has focus on top of the Activity), it is in the *Paused* state; in this state the app maintains all the user progress and information. When the user navigates to a new Activity, the previous one is put in the *Stopped* state; it still retains all the user information but it is no longer visible to him. If an Activity is in *Paused* or *Stopped* states, the system can drop it from memory by either asking it to finish, or simply killing its process and it transits to the *Destroyed* state. When it is displayed again to the user, it should be completely restarted and its saved state must be restored.

Figure 1 also highlights three key loops of the Activity lifecycle. According to [27] we name these loops Entire Loop, Visible Loop, and Foreground Loop, respectively. In the following we describe these three loops and report event sequences able to exercise each of them:

1) The *Entire Loop (EL)* of an Activity consists in the Resumed-Paused-Stopped-Destroyed-Created-Started-Resumed sequence of states. This loop can be exercised by events that cause a configuration change, e.g. an orientation change of the screen, that destroys the Activity instance and then recreates it according to the new configuration;[9]
2) The *Visible Loop (VL)* corresponds to the Resumed-Paused-Stopped-Started-Resumed sequence of states during which the Activity is hidden and then made visible again. There are several event sequences able to stop and restart an Activity, e.g. turning off and on

[7] https://developer.android.com/topic/performance/memory-overview#SwitchingApps

[8] `https://developer.android.com/reference/android/app/Activity.html`

[9] https://developer.android.com/guide/components/activities/state-changes.html

the screen or putting the app in background and then in foreground again through the Overview or Home buttons;

3) The *Foreground Loop (FL)* of an Activity involves the Resumed-Paused-Resumed state sequence. The transition Resumed-Paused can be triggered by opening non full-sized elements such as modal dialogs or semi-transparent activities that occupy the foreground while the Activity is still visible in background. To trigger the transition Paused-Resumed the user should discard this element.

According to this model, the Activity classes represent entry points for Android applications but can be destroyed and recreated during the apps lifecycle. Therefore, they do not represent roots for the garbage collector and can be deallocated and re-allocated on heap memory. In the following subsection, a motivating example showing how this mechanism can cause leaks will be shown.

C. MOTIVATING EXAMPLE

The motivating example refers to an Android app that is designed to display a list of items whose values are read from a remote Firebase database. As an item in the database changes its state, the app GUI is automatically updated.

An excerpt of the app source code is shown in Listing 1.

```
public class MainActivity extends Activity {
  private ArrayAdapter<String> wishListAdapter;
  private MyWishlistModel wishlistModel;
  @Override
  protected void onCreate(Bundle b) {
    ...
    wishListAdapter =
        new ArrayAdapter<>(getActivity(),
        android.R.layout.simple_list_item_1);
    wishlistModel =
        new MyWishlistModel(wishListAdapter);
    ...
    }
}

public class MyWishlistModel {
  private DatabaseReference dbRef;
  private ArrayAdapter<String> wishListAdapter;
  private ChildEventListener childEventListener;
  public MyWishlistModel(ArrayAdapter<String> a){
    wishListAdapter=a;
    dbRef = FirebaseDatabase
        .getInstance().getReference();
    childEventListener = new ChildEventListener(){
      @Override
      public void onChildAdded(
        @NonNull DataSnapshot dataSnapshot,
            @Nullable String s) {
            wishListAdapter.add(
                dataSnapshot.getValue());
        }
      };
      dbRef.addChildEventListener(
        childEventListener);
}
```

Listing 1. An example of code causing leaks.

The entry point of the app is the class *MainActivity*. The *onCreate* method instantiates an *ArrayAdapter* object that will contain the data values that will be shown in the GUI and an object called *wishlistModel* of the class *MyWishlistModel* that is linked to the adapter.

The constructor of the *MyWishlistModel* class instantiates a listener of a Firebase Database (*childEventListener*) as an anonymous internal class. The *wishlistModel* object maintains an explicit reference to the *wishListAdapter* object, which in turn refers to the *MainActivity* object.

A memory leak problem can be observed after a screen orientation change, that causes the destruction of the current *MainActivity* object and the creation of a new one. In this case, the garbage collector cannot deallocate neither the object of the inner class implementing ChildEventListener (because the connection to the Firebase Database is still open), nor the object of the outer class *MyWishListModel*. Consequently, the *wishListAdapter* object (that is referenced by the *MyWishListModel* object) and the *MainActivity* object (that is referred by *wishListAdapter*) cannot be deallocated, together with all the other objects allocated by *MainActivity*.

In summary, after the orientation change we can observe memory leaks due to the replication of all these objects. A resource leak due to the replication of the connection to the Firebase Database may occur, too. Further rotations of the device will increase the number of leaked objects and the amount of leaked memory and could bring to an Out of Memory error.

In this case, the detected memory leak is *time persistent*, since the leaked objects will never be destroyed by the garbage collector until the connection is alive. The *size* of this memory leak will depend on the number and size of the objects included in the *MainActivity* and this size will grow with a constant *growth rate*, since each device rotation will cause the leaking of further objects. Due to the contemporary occurrence of these three characteristics, the found memory leak represents actually a severe issue.

In conclusion, a possible code fix consists of the explicit removal of the listener in the context of the *onDestroy* method of the *MainActivity* class, as shown in Listing 2.

```
@Override
public void onDestroy() {
  ...
  wishlistModel.removeEventListener();
  ...
}
```

Listing 2. Solution of the leak issue.

III. THE FUNESDROID APPROACH

In this section, we describe the FunesDroid testing technique and the tool that we have realized to implement it.

The goal of FunesDroid is to check whether the Activity classes of an analyzed Android app expose memory leaks tied to the Activity Lifecycle. Unlike other approaches, the proposed technique does not rely on the continuous soliciting of

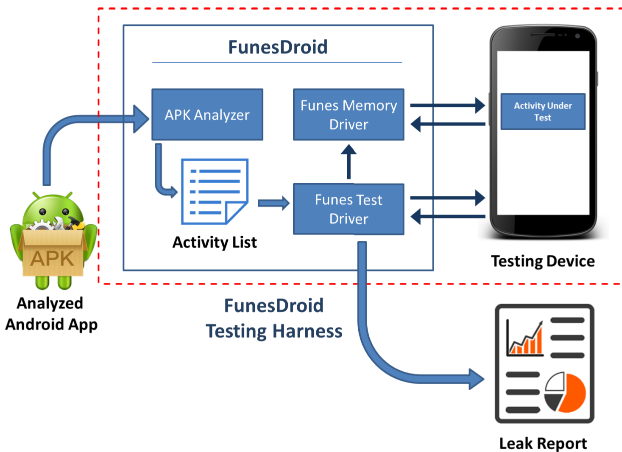

Figure 2. The FunesDroid leak detection technique.

the app under test and on the continuous monitoring of the whole memory. Moreover, it does not exploit a static analysis approach where the inspection and the instrumentation of the app source code is needed.

The FunesDroid technique is based on the detection of replicated objects in memory, after the execution of an input event sequence able to exercise one of the three key Activity Lifecycle loops introduced in Section II-B. Hereafter, we define Lifecycle Event Sequence (*LES*) a sequence of events able to trigger one of the key loops of the Activity Lifecycle [27].

Our technique requires that, after an Activity of the app under test is launched, a first dump of the memory is performed. Therefore, after the execution of the selected LES and the forced execution of the garbage collector, a second dump of the memory is performed. The comparison of the two memory dumps provides a list of replicated objects that cannot be deallocated by the garbage collector.

This technique is implemented in the *FunesDroid* tool, whose architecture is shown in Figure 2. FunesDroid relies on three components, i.e., the *APK Analyzer*, the *Funes Test Driver*, and the *Funes Memory Driver*.

The APK Analyzer has the responsibility of analyzing the *.apk* file of the *Analyzed Android App* and extracting an *Activity List* which reports all the Activity classes providing potential entry points of the analyzed app. The Funes Test Driver is the component that can send LESs to the Activities in the Activity List. It interacts with a *Testing Device*, that can be either real or emulated, where the *Application Under Test* is running. The Funes Memory Driver is able to perform a memory dump by interacting with the memory of the Testing Device. It is also able to solicit garbage collector executions on the Testing Device. The Funes Test Driver component produces a *Leak Report* by comparing two memory dumps. The Leak Report contains the names of the replicated objects and for each of these objects the number of their instances and the corresponding heap memory occupation.

Figure 3 reports a sequence diagram showing how FunesDroid tests a single Activity of the application under test. The Funes Test Driver starts by launching, on the Testing Device, the Activity to be tested. After that, the Funes Test Driver asks the Funes Memory Driver to gather the state of the device memory. To this aim, the Funes Memory Driver first forces the execution of the Garbage Collector and subsequently gathers the dump of the heap memory. The dump is a data structure reporting the names of all the objects referenced by the activity under test along with their instances and heap memory occupation (measured in Bytes). We named this dump as *DBLES*, i.e., *Dump Before LES*.

Then, the Funes Test Driver executes *N* repetitions of a given Lifecycle Event Sequence (LES) that is able to exercise a specific Activity lifecycle key loop. The execution of more than one repetition of the same LES allows to observe whether the object replications grow with the number of executed events.

We leverage 3 Lifecycle Event Sequences, i.e., the Double Orientation Change (DOC), the Background Foreground (BF) and the Semi- Transparent Activity Intent (STAI) event sequences [27]. We chose these Lifecycle Event Sequences for 2 main reasons. The former reason is that each of these event sequences is able to exercise a different lifecycle key loop. The latter reason is that they represent *neutral event sequences*, i.e. events that should have neutral effects on resource consumption and should not lead to increases in resource usage [17]. Therefore, according

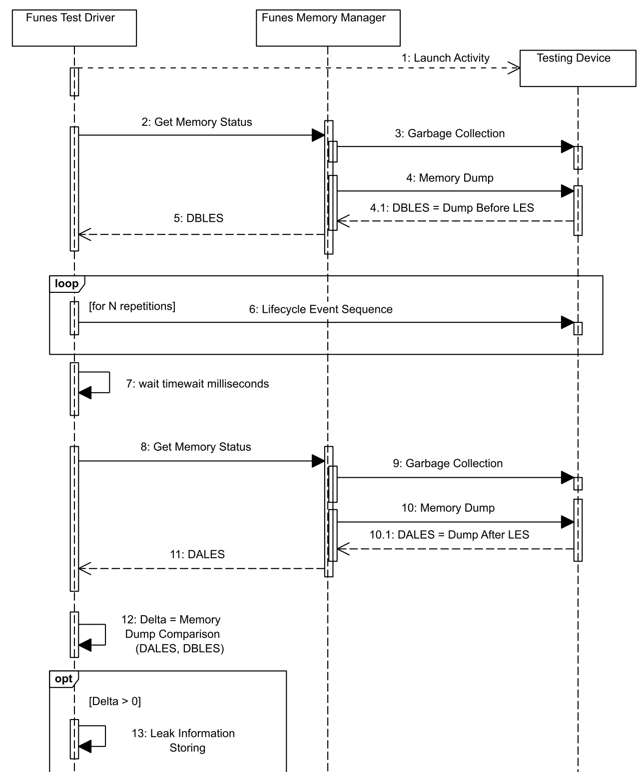

Figure 3. The FunesDroid detection technique.

with [24], replicated objects detected after the execution of a neutral event sequence may correspond to memory leaks.

1) **Double Orientation Change (DOC)** consists in a sequence of two consecutive device orientation change events, from landscape to portrait and back again to landscape or, equivalently, from portrait to landscape and, finally, to portrait. Each orientation change exercises the *EL* loop, since the running Activity is destroyed and recreated in order to adapt to the device layout.
2) **Background-Foreground (BF)**, corresponding to a sequence of events exercising the *VL* loop. This sequence puts the app in background through the tap of the Home button and then pushes the app again in foreground by selecting the same Activity from the list of the recently stopped ones. This last event causes the restarting of the Activity, without the need to redraw its graphical layout.
3) **Semi-Transparent Activity Intent and back (STAI)**, corresponding to a sequence of events exercising the *FL* loop. It consists in starting a semi-transparent Activity that pauses the current foreground Activity and then returning to it tapping the Back button.

A delay time of *tw* milliseconds is waited after the execution of the N Lifecycle Event Sequence repetitions. This delay represents the time interval after which we evaluate the existence of object replications. In other words, we detect only object replications that persist after this time interval.

After that *N* repetitions of a given LES have been triggered and after the waiting time, another memory dump, named *DALES* (*Dump After LES*), is gathered by the Funes Memory Driver.

Lastly, DALES and DBLES are compared to check the presence of undue replicated objects, that are added to the *Leak Report* output file.

A. FuneSDroid IMPLEMENTATION

FunesDroid is implemented in Python and its source code is freely available.[10] In the following, we report insights about the technological choices we adopted to realize the three FunesDroid components.

To implement the APK Analyzer we exploited *Apktool*.[11] It analyzes the *AndroidManifest.xml* of the app under test and extracts the list of all the Activity classes belonging to the analyzed app.

The Funes Test Driver component is responsible of installing the AUT, launching the Activities of the AUT, and firing the Lifecycle Event Sequences. In the following, we illustrate how we implemented these features.

- The AUT is installed by exploiting the command provided by the Android Debug Bridge (adb).[12]
- The Activity classes to be tested are launched by exploiting the Intent mechanism.
- The three Lifecycle Event Sequences are triggered by exploiting combinations of commands provided by the adb. For implementing STAI, FunesDroid preliminary installs a stub Android app we named *stai.apk* that shows a semitransparent Activity in the foreground when it is executed. To trigger the STAI LES, this stub app is first launched and then the semitransparent Activity is dismissed by triggering the key code event *KEYCODE_BACK*. The BF LES is obtained by sending to the device the sequence of key code events *KEYCODE_HOME*, *KEYCODE_APP_SWITCH*, and *KEYCODE_APP_SWITCH*. The DOC LES is implemented by setting the *user_rotation* parameter first to *1* and then to *0*.

The Funes Memory Driver is able to force the execution of the garbage collector and to return the current state of the heap memory. To trigger the GC we exploited the following adb command: *adb -s "+DEVICE+" shell kill −10 " + pid*, where *DEVICE* is the identifier of the real or virtual device executing the AUT and *pid* is the process ID of the AUT. The dump of the heap memory is obtained by exploiting the adb *dumpheap* command that returns an heap dump *.hprof* file.

A heap dump is a snapshot of the memory of a Java process at a certain point of time. The snapshot contains information about the Java objects and classes in the heap at the moment the snapshot was triggered. [13]

For comparing two heap dump files we exploited the Eclipse Memory Analyzer Tool (MAT).[14] MAT is widely used by Android developers [25] and is integrated in other memory leaks detection tools such as Android Studio Memory Profiler [28], Leak Canary [23] and LeakDAF [16].

In FunesDroid we used MAT for obtaining the list of all the class names, number of instances of each class and the total shallow heap. The comparison performed by MAT just matches on class names and then shows the delta of the number of instances of each class. This delta may represent a list of potential memory leaks and their retained heap size, measured in bytes.[15] Of course, the replicated objects may also correspond to the expected behaviour of the app. A further source code analysis would be necessary to confirm the existence of a memory leak.

[10] https://github.com/reverse-unina/FunesDroid
[11] https://ibotpeaches.github.io/Apktool/
[12] https://developer.android.com/studio/command-line/adb
[13] https://help.eclipse.org/2019-09/index.jsp?topic=/org.eclipse.mat.ui.help/welcome.html
[14] https://www.eclipse.org/mat/
[15] http://memoryanalyzer.blogspot.com/2010/01/heap-dump-analysis-with-memory-analyzer.html

TABLE 1. Results obtained by the exploratory study carried out on 1184 activity classes belonging to 283 different android applications.

Tested Event Sequence	Potentially Leaked Apps	Potentially Leaked Activities
DOC	90/283 (32%)	134/1184 (11%)
BF	49/283 (17%)	102/1184 (9%)
STAI	38/283 (13%)	73/1184 (6%)
Union	106/283 (37%)	201/1184 (17%)

TABLE 2. Number of activities showing a monotonous growing trend of heap memory allocation.

Triggered LES	Analyzed Activities
DOC	111/134 (82.8%)
BF	86/102 (84.3%)
STAI	61/73 (83.6%)

IV. EXPLORATORY STUDY: FINDING OBJECT REPLICATIONS

To evaluate our approach, we conducted an exploratory study with the aim to investigate the capability of FunesDroid to detect object replications tied to Activity Lifecycle events that represent potential Memory Leaks in real Android apps.

To carry out the study, we randomly selected a sample of 283 open source Android apps[16] including about 20% of the 1324 apps that were present in the online repository F-Droid[17] at the moment of the study. We restricted our selection to apps that were also published on Google Play[18] and were executable on devices equipped with Android 7.1.1 (API 25).

We tested all the apps by using FunesDroid. For each Activity class listed in the app manifest file, we triggered once each of the three Activity Lifecycle Event Sequences (LES) described in Section III (i.e. the DOC, BF and STAI event sequences). We set FunesDroid to have a waiting time $wt =$ 1 sec, in each test execution. For each Activity, FunesDroid returned the list of replicated objects and the total amount of heap memory allocation for each object. The study was performed on a desktop PC having an Intel(R) Core(TM) i7 4790@3.60GHz processor and 8 GB of RAM, running a standard Nexus S AVD equipped with Android 7.1.1 (API 25).

In the context of the 283 tested applications, 3,524 different Activity classes were included in the manifest files. Only 1,184 of these classes could be tested by FunesDroid. The other ones were not executed due to the need to set specific extra parameters to start them, that could not be deduced by the manifest analysis.

The obtained results are summarized in Table 1 that shows, for each LES, the number of apps exposing at least a potential memory leak (*Potentially Leaked Apps*) and the number of tested Activities exposing at least an object replication (*Potentially Leaked Activities*). We can observe that 106 out of 283 applications (37.5%) have revealed at least an object replication, while the testing of 201 out of 1,184 Activity classes revealed at least one object replication (17%). As the Table shows, the DOC event sequence is the sequence that triggered most of the object replications.

FunesDroid is able to detect object replications that represent potential memory leaks. A possible approach to confirm the observed leaks would require an expensive source code analysis of the apps to find the causes of these object replications. An alternative approach to validate memory leaks, which is widely adopted in the literature [9], [17], [24], consists of monitoring the heap memory usage during the app execution and of observing the possible performance degradation.

According to the latter family of approaches, we decided to investigate the trend of object replications as the number of executed event sequences grows. To this aim, we considered each of the 201 Activities that showed object replications and solicited them by sequences of 100 LESs of the same type of the ones that caused object replications. We monitored the trend of object replications by sampling the heap memory after each 10 LESs. Therefore we were able to distinguish Activities showing a monotonous growing of the heap, from Activities not showing this trend. Table 2 reports the percentage of Activities with monotonous trend.

This datum indicates that in more than 80% of cases, the object replications observed by FunesDroid caused a growing memory allocation and could likely be considered as true memory leaks. In the remaining cases, we could not conclude anything about the observed object replications. A source code analysis would be strictly needed to validate these latter leaks.

These results show (1) the capability of FunesDroid in finding memory leaks due to the triggering of Activity lifecycle events and (2) the large spread of memory leaks tied to Activity Lifecycle Events in real Android apps.

The latter finding is coherent with the experimental results obtained by Toffalini *et al.* [14], who found potential causes of memory leaks via static analysis in the majority of the applications they downloaded from Google Play. Moreover, our results are coherent with the ones reported in [16], which found memory leaks related to Activity classes in 37 out of the 99 tested Android apps in an experiment carried out in 2017.

V. SECOND EXPLORATORY STUDY: CHARACTERIZING MEMORY LEAKS

The exploratory study we presented in Section IV did not investigate some aspects of the observed leaks that may be useful for characterizing their severity and impact on the system performance, such as the size and the persistence of the memory leak. Moreover, it did not investigate the possible causes of the observed leaks. Therefore, we decided to conduct a further exploratory study that focused on these specific aspects.

[16]The list of the 283 tested applications is available at `https://github.com/reverse-unina/FunesDroid/blob/master/testedApps.txt`

[17]`https://f-droid.org`

[18]`http://play.google.com`

TABLE 3. Object apps.

	App Name	Version	#Activity classes
A1	aMetro	2.0.1.5	6
A2	Quasseldroid	0.11.7	7
A3	RadioDroid	0.37	2
A4	RadioDroid	0.47	2
A5	UHabits	1.7.9	7
A6	World Clock	1.8.6	4
A7	ConnectBot	1.9.5	12
A8	Equate	1.7	1
Total			41

The goal of this second study is to characterize a sample of the observed object replications in terms of their size, persistence and growth trend, as the type of event sequence, the number of repetitions of event sequence, and the waiting time of FunesDroid vary. Moreover, the study aimed at exploring the possible causes of the observed leaks.

A. OBJECTS

As objects of the study we considered a subset of 8 versions of 7 different apps, selected among the ones involved in the former study and that showed at least one object replication.

Table 3 reports for each considered object the app name, the number of the release we tested, and the total number of executable activity classes that are listed in the manifest file. For 6 out of 7 applications, only a single release available on F-Droid was tested. For the RadioDroid app, instead, we considered two releases, i.e. the version 0.37 and the version 0.47 published after the maintenance intervention that fixed a bug causing the leak we found in the previous version.

B. VARIABLES AND METRICS

The independent variables of this study are (1) the type of Lifecycle Event Sequence (*LES*) triggered on the application under test, (2) the number of times *N* the *LES* is sequentially triggered on it, (3) the *wt* time waited after the triggering of the last event of the sequence.

As regards the *LES* categories, the three types of sequence presented in the previous sections and called *DOC*, *BF* and *STAI* were considered.

As regards the N number of LES repetitions, we considered sequences including a single LES, a sequence of 2 identical LESs and a sequence of 10 identical LESs in order to evaluate if the amount of leaked memory varied with the number of executed events.

As regards the waiting time ***wt***, we considered three possible values, respectively of 1, 2 and 5 seconds. The reason to consider three different values is to evaluate if the memory leaks are only temporary, since their occurrence depends on the wt time, or persistent (when they occur for each considered wt time).

In summary, the three independent variables belong to the following sets:

$$LES \in \{DOC, BF, STAI\}, \tag{1}$$

$$N \in \{1\ sequence, 2\ sequences, 10\ sequences\} \tag{2}$$

$$wt \in \{1\ second, 2\ seconds, 5\ seconds\} \tag{3}$$

For each of the 27 combinations of the three independent variables, FunesDroid was executed by obtaining sets of replicated objects and by measuring the following metrics:

- *DHM*: total quantity of Heap Memory allocated to replicated objects (in bytes);
- *DC*: number of distinct classes for which at least a replicated object instance were found;
- *DI*: total number of replicated objects instances.

On the basis of the analysis of the measured values, we evaluated the following three characteristics:

- **Size**: we evaluated the *size* (in bytes) of a memory leak in terms of the amount of retained memory due to object replications (DHM).
- **Persistence**: the *persistence* of a memory leak can be observed by comparing the memory leak *Size* for different waiting times *wt*. A leak can be considered *persistent* if it can be always observed when varying the waiting time *wt*, *temporary* (or *transient*) if the leak can be observed in executions having given *wt* values, but not in executions with *wt* values greater than the former considered ones.
- **Growth Trend**: the *growth trend* of a memory leak can be observed by comparing the memory leak size for increasing values of the number *N* of executed sequences of events. The Growth Trend is considered *positive* if the memory leak size increases with the number of LES repetitions. Elsewhere, the growth trend is considered *null*, when the memory leak size is constant with the number of LES repetitions.

C. EXPERIMENTAL PROCEDURE

The testing process implemented by FunesDroid was repeated for each of the 27 combinations of the independent variable values and for each of the 8 considered objects (from A1 to A8). It was performed on the same testing infrastructure used in the exploratory study. In order to assure that each run was executed in the same conditions, virtual devices data were cleaned after each FunesDroid execution.

The FunesDroid version used to carry out these experiments and the apks of the tested apps are freely available on our github space.[19]

D. RESULTS

Table 4 lists the obtained results. Each row reports the name of the app, the type of the executed *LES*, the waiting time *wt* (different values have been merged in the same row when the corresponding experiments produced the same results), the Activity class whose execution caused object replications (if any) and the values of *DHM*, *DC* and *DI* for LES repetitions including N = 1, 2 or 10 consecutive sequences, respectively.

[19] https://github.com/reverse-unina/FunesDroid/tree/8a8e70b4f72ab32d2ff7eb6ec161cecbf9dc980a

TABLE 4. Results of the experiments carried out using FunesDroid.

		LES	wt (sec)	Activity	N = 1 sequence DHM (bytes)	N = 1 sequence DC	N = 1 sequence DI	N = 2 sequences DHM (bytes)	N = 2 sequences DC	N = 2 sequences DI	N = 10 sequences DHM (bytes)	N = 10 sequences DC	N = 10 sequences DI
A1	AMetro	STAI	(1,2,5)	-	0	0	0	0	0	0	0	0	0
		BF	(1,2,5)	-	0	0	0	0	0	0	0	0	0
		DOC	(1,2,5)	CityList	17268	8	502	17268	8	502	17268	8	502
		DOC	(1,2,5)	MapList	17180	8	494	17180	8	494	17180	8	494
A2	Quasseldroid	STAI	(1,2,5)	-	0	0	0	0	0	0	0	0	0
		BF	1	LoginActivity	112	3	3	112	3	3	112	3	3
		BF	1	MainActivity	112	3	3	112	3	3	112	3	3
		BF	(2,5)	-	0	0	0	0	0	0	0	0	0
		DOC	(1,2,5)	MainActivity	2908	35	35	2908	35	35	2908	35	35
A3	RadioDroid v0.37	STAI	(1,2,5)	ActivityMain	15028	13	251	21476	12	353	21476	12	353
		BF	(1,2,5)	ActivityMain	15012	12	250	21620	13	356	21668	13	359
		DOC	(1,2,5)	ActivityMain	33840	15	554	67680	15	1108	338400	15	5540
A4	RadioDroid v0.47	STAI	(1,2,5)		0	0	0	0	0	0	0	0	0
		BF	(1,2,5)	-	0	0	0	0	0	0	0	0	0
		DOC	(1,2,5)	ActivityMain	336	1	2	336	1	2	336	1	2
A5	UHabits	STAI	(1,2,5)	-	0	0	0	0	0	0	0	0	0
		BF	(1,2,5)	-	0	0	0	0	0	0	0	0	0
		DOC	(1,2,5)	AboutActivity	2368	17	34	4736	17	68	23680	17	340
		DOC	(1,2,5)	SettingsActivity	1440	12	32	2880	12	64	14400	12	320
		DOC	(1,2,5)	EditSettingActivity	2432	18	36	4864	18	72	24320	18	360
A6	Worldclock	STAI	(1,2,5)	-	0	0	0	0	0	0	0	0	0
		BF	(1,2,5)	-	0	0	0	0	0	0	0	0	0
		DOC	(1,2,5)	AddClock	1040	4	8	2080	4	16	10400	4	80
		DOC	(1,2,5)	WorldClock	1040	3	6	2080	3	12	10400	3	60
A7	ConnectBot	STAI	(1,2,5)		0	0	0	0	0	0	0	0	0
		BF	(1,2,5)	-	0	0	0	0	0	0	0	0	0
		DOC	(1,2,5)	ActivityMain	944	8	16	1888	8	32	9440	8	160
A8	Equate	STAI	(1,2,5)	CalcActivity	8696	5	30	8696	5	30	8696	5	30
		BF	(1,2,5)	CalcActivity	8696	5	30	8696	5	30	8696	5	30
		DOC	(1,2,5)	-	0	0	0	0	0	0	0	0	0

1) DATA ANALYSIS

Overall, FunesDroid found 17 object replications in the execution of 13 different Activities belonging to the apps under test. The type of LES causing the higher number of object replications was DOC, that caused replications in 11 different Activities belonging to 7 different applications. The execution of the BF sequence, instead, revealed object replications on 4 different Activities belonging to 4 different apps. Objects replications were found by the STAI sequence only on 2 Activity classes belonging to 2 different apps.

We analyzed the data reported in Table 4 in order to characterize the leaks in terms of *persistence* and *growth trend*.

As regards the *persistence*, we checked whether the object replications disappeared as the waiting time *wt* increased from 1 to 2, to 5 seconds. In Table 5 we labeled as *Persistent* the object replications that can be observed for each tested waiting time, and as *Temporary* the ones that tend to disappear with the increase of the waiting time. There were no cases of object replications that appeared only for waiting time values of 2 or 5 seconds.

We observed that for the 12 leaked activities the object replication phenomenon was independent of the waiting time *wt*, thus we hypothesized that those memory leaks were *persistent*. Of course, we cannot exclude that the memory leaks could disappear after longer testing times. For 2 activities belonging to QuasselDroid the object replications disappeared for *wt* values greater than 1 second, thus we labeled these memory leaks as *temporary*.

As regards the *growth trend*, we compared the size values for increasing values of the number of tested events *N*. In Table 5 we labeled the growth trend as *Positive* when the memory leaks size increased for sequences having an increasing number of events from 1 to 2 to 10 and as *Null* the growth trend when the memory leaks size was independent on the number of executed sequences. No cases in which the memory leak size decreased when the number of executed events increased were observed. In details, we found that in 4 applications (i.e. WorldClock, Uhabits, ConnectBot and version 0.37 of RadioDroid), the *DI* and *DHM* values increased with the number of executed events *N*, whereas in all the other cases they remained constant.

TABLE 5. Summary of the detected memory leaks.

		LES	Activity	Persistence	Growth Trend	Category
A1	AMetro	DOC	CityList	Persistent	Null	C2
		DOC	MapList	Persistent	Null	C2
A2	Quasseldroid	BF	LoginActivity	Temporary	Null	C4
		BF	MainActivity	Temporary	Null	C4
		DOC	MainActivity	Persistent	Null	C3
A3	RadioDroid v0.37	STAI	ActivityMain	Persistent	Positive	C1
		BF	ActivityMain	Persistent	Positive	C1
		DOC	ActivityMain	Persistent	Positive	C1
A4	RadioDroid v0.47	DOC	ActivityMain	Persistent	Null	C4
A5	UHabits	DOC	AboutActivity	Persistent	Positive	C1
		DOC	SettingsActivity	Persistent	Positive	C1
		DOC	EditSettingActivity	Persistent	Positive	C1
A6	Worldclock	DOC	AddClock	Persistent	Positive	C1
		DOC	WorldClock	Persistent	Positive	C1
A7	ConnectBot	DOC	HostListActivity	Persistent	Positive	C1
A8	Equate	BF	CalCActivity	Persistent	Null	C4
		STAI	CalCActivity	Persistent	Null	C4

We have summarized our findings in Table 5, that reports, for each combination of application under test, tested LES and tested Activity, their characteristics of Persistence and Growth Trend of the observed memory leaks.

In conclusion, the experimental data show that the execution of the selected LESs caused 17 potential memory leaks, involving 13 out of 41 tested activities. In the majority of these cases (15 out of 17), the leaks have been classified as

TABLE 6. FunesDroid report listing the replicated objects found after the execution of the ActivityMain class of A3 (RadioDroid v 0.37) and a sequence of 10 DOC events.

Class Name	DI	DHM
adapters.ItemAdapterStation$MyItem	151	3624
adapters.ItemAdapterStation$1	151	2416
FragmentCategories	114	20064
data.DataRadioStation	3800	243200
data.DataRadioStation[]	494	1976
FragmentStations$1	38	608
FragmentStations$2	38	608
MPDClient$2	19	304
ActivityMain	19	6080
ActivityMain$2	19	304
FragmentStations	190	31920
FragmentTabs	38	6080
data.DataCategory[]	114	456
adapters.ItemAdapterStation	38	3648
FragmentBase[]	38	152
Total	5261	321440

persistent, while in more than the half of the cases (9 out of 17), their size was increasing with the number of executed LES.

In the performed growth trend analysis we only reported the cumulative size of all the replicated objects and its growth trend. However, overlapping effects in the memory allocation of different types of objects (for example, when some objects occurrences increase whereas other ones decrease) may have caused the misclassification of the growth trends. In order to avoid such threat, we performed a finer grained analysis of the growth trend of replicated objects of single classes and a detailed source code analysis of the causes of the memory leak, which will be presented in the next subsection.

E. ANALYZING THE CAUSES OF THE MEMORY LEAKS

In order to investigate the possible causes of the observed leaks, we defined an approach for analyzing the app source code. The approach includes three steps. We first take into account the complete list of replicated objects produced by FunesDroid and build the class diagram representing all the corresponding classes and the relations among them. Therefore, among these classes, we look for the ones whose objects are GC roots, since they cannot be deallocated by the Garbage Collector. If none of the replicated objects can be considered as a GC root, we extend the search among other objects linked to the replicated ones. Eventually, we analyze the GC root code, looking for possible causes of the leak. As an example, we look for the existence of a reference from the GC root object to a Context object.

As an example, in the following we illustrate how we used this approach to detect the cause of the memory leaks found when executing 10 repetitions of the DOC sequence on the version 0.37 of RadioDroid (A3). Figure 4 shows the class diagram including all the classes of the objects reported by the FunesDroid output report illustrated in Table 6. For simplicity, this diagram does not report inner classes.

As Figure 4 shows, MPDClient class has a static StartDiscovery method. We observed that this method instantiates a Thread that is actually a GC root. Moreover, this class holds a reference to a context object (the ActivityMain starting class).

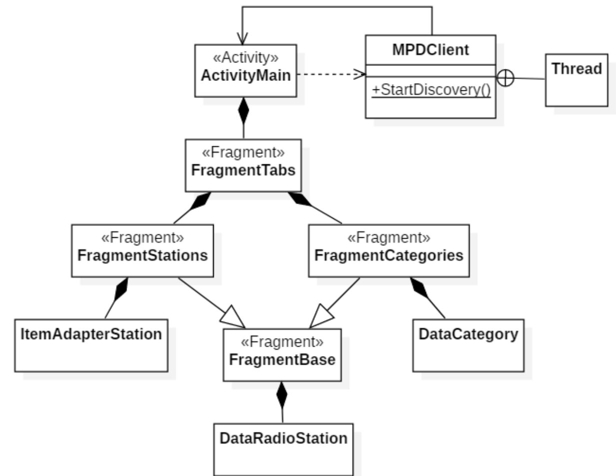

Figure 4. Class diagram reporting the classes listed in the FunesDroid report.

In addition, the ActivityMain class is linked to all the classes having replicated objects.

Therefore, we ascertained that this class was actually the cause of the leak. In fact, at each execution of a DOC event, a new ActivityMain object was instantiated and, consequently, new instances of all the other linked objects shown in Figure 4 were created. The previous instance of ActivityMain could not be deallocated by the Garbage Collector due to the presence of the reference from the previous MPDClient object that included the (previous, but still active) Thread instance.

Analysing the Thread source code, we observed that it executes a loop that could be ended only by the Activity instance that allocated it. Since this Activity was no more active, then the Thread could never be terminated. As a consequence, the number of active Threads and replicated objects increased as further DOC events were triggered. This behavior caused the linear increase of both the number of leaked objects *DI* and the amount of leaked memory *DHM*. This bug is very severe since it caused a leak of more than 300kBytes of memory after the execution of 10 DOC events.

To further confirm the results of our analysis, we opened an issue on the Github space of this application, by posting an excerpt of the list of replicated objects found by FunesDroid.[20] The developers accepted the issue and actually identified its cause in the MPDClient class.[21] They fixed the problem by refactoring the MPDClient class using a WeakReference to the Activity class, as shown by commit #0ed46d7.[22] We verified the validity of their solution by considering the successive RadioDroid v0.47 release, that does not present this memory leak problem.

[20] https://github.com/segler-alex/RadioDroid/issues/306
[21] https://github.com/segler-alex/RadioDroid/pull/316
[22] https://github.com/segler-alex/RadioDroid/commit/0ed46d7da3867a2ecb654d3b6ac868aea932617f

F. CATEGORIZING LEAKS CAUSES

Using the approach presented in the previous section, we analyzed the causes of all the leaks we found. As a result of this analysis, we were able to classify these causes in four categories, that will be presented in the following.

1) CATEGORY C1: THREADS REFERRING CONTEXT OBJECTS

Threads are used in Android applications to concurrently execute some actions and improve application responsiveness. A Thread often hold a reference to the class that started it (for example an Activity class or a Fragment) in order to be able to notify it the results of its execution, or simply due to bad programming. This bad practice can cause leaks in Android applications. For example, if a Thread is instantiated and started by an Activity object and a double orientation change (DOC) is executed on it, then the Activity object is destroyed and replaced by another instance of the same class, which will start a new Thread instance. However, the old Thread instance cannot be destroyed by the garbage collector since it is considered a GC Root, and it holds a reference to the old Activity instance. Thus, the garbage collector cannot deallocate the old Activity instance and all the other objects instantiated by it. The size of this memory leak may increase with the number of executed DOC events, since new objects are instantiated each time the Entire Loop (EL) is repeated. This type of problem may be persistent or temporary depending on the duration of the execution of the Thread. In particular, in the case of Threads that are terminated only by user events on the original Activity object, the problem may be permanent. These problems are the ones with the maximum severity, since it can easily bring both to Out of Memory exceptions and to possible deadlocks.

These memory leaks may be long-running or persistent, depending on the expected duration of the Thread objects causing them, and may have a leaked memory size that increases with the number of executed test events.

Several instances of these problems were found in the applications under test. In detail, in the case of A3 (RadioDroid v0.37), we have already presented the leaks observed when executing DOC events. Similar problems were also observed executing STAI and BF events. In these cases the leaks are smaller in size since the Activity objects were not replicated. Other problems of this category, due to Thread objects maintaining implicit references to the Activity that instantiated them, were detected in A5 (Uhabits), A6 (WorldClock) and A7 (ConnectBot). An analogous case is the one described in the motivating example reported in Section II-C, where the ChildEventListener objects maintain references to the Activity objects instantiating them. A possible solution to these problems may consist in forcing the explicit termination of Threads instantiated by an Activity or Fragment when the creator object is destroyed (e.g. in the onDestroy method). Another possible solution is the use of WeakReferences instead of references in Threads. In fact, objects referred only through WeakReferences can be deallocated by the garbage collector. For example, an Activity that is referred only by a WeakReference included in a Thread object may be deallocated.

2) CATEGORY C2: TASKS OR SERVICES REFERRING CONTEXT OBJECTS

In the context of Android apps, there are several other constructs, other than Threads, that can be used to implement concurrency. As an example, the AsyncTask class provide an alternative to Threads for concurrent task execution. Differently from Threads, AsyncTasks are all executed in the same background Thread according to a FIFO scheduling. For this reason, two AsyncTasks cannot be executed concurrently.

A possible memory leak scenario involving AsyncTasks that we found in our study is the following: an Activity instantiates and starts an AsyncTask that keeps a reference to the Activity. When executing a DOC, the Activity is destroyed but it cannot be deallocated by the garbage collector because the AsyncTask cannot be destroyed, since it includes a Thread and holds a reference to it. In the meantime, another Activity is opened but it cannot start another AsyncTask until the first one terminates. In conclusion, the memory leak does not involve the AsyncTask but only the Activity and all the other objects instantiated by it. The size of the leak does not increase as the number of sequences increases due to the nature of the AsyncTask. These memory leaks are temporary but long running, as the ones due to Threads.

We have observed memory leaks of this type in A1 (AMetro), in which a Task is instantiated as an internal class of a Fragment. In this case an implicit reference is established from the inner class (AsyncTaskLoader) to the outer class (Fragment). This reference prevents the deallocation of the Fragment until the AsyncTaskLoader is alive.

A possible solution to prevent this leak is also in this case the use of WeakReferences instead of references to context objects such as Activity or Fragment objects. Only in the case of AsyncTask classes, this problem can be detected by the static analysis tool Lint [29]. Moreover, automatic fixes via refactoring have been proposed by Lin *et al.* [20] and Lin and Dig [30].

3) CATEGORY C3: STATIC FIELDS REFERRING CONTEXT OBJECTS

A static class always represents a GC root, i.e. an element that cannot be deallocated by the garbage collector. If such a class refers a context object (for example an Activity), then it prevents its possible deallocation.

We found a memory leak of this type in A2 (QuasselDroid), in which there is a static reference in the MainActivity class pointing to itself, which prevents the deallocation of the Activity after a DOC event, as well as that of all the objects it instantiates. We observed that the size of this memory leak is constant (since the static field cannot be replicated) and appeared to be persistent (since the leak did not disappear with larger waiting times *wt*). A possible solution to these problems consists in the explicit update of the references

provided by the static fields, in order to allow the GC to deallocate the referred Context Objects.

4) CATEGORY C4: TEMPORARY LEAKS DUE TO LATE RELEASE OF MEMORY

Some temporary memory leaks can be due to the relative slowness in releasing the memory caused by framework objects.

We found an example of this leak cause in the app A4 (RadioDroid v.0.47), in which the Fragments instantiated by the Activity are still in memory after performing the Garbage Collector. According to the results of our tool, this memory leak appears to be constant and persistent, but in reality it will be automatically solved successively, since this fragment does not in turn refer to other objects that cannot be deallocated.

We observed a similar problem in A8 (Equate), in which a dynamic Fragment is dynamically redrawn after each sequence of BF or STAI events, while the old Fragment instance is temporarily retained by the FragmentManager object.

Another example was found in A2 (QuasselDroid) when BF events were executed. In this case the memory leaks are due to a binding with a Service that does not terminate together with the Activity calling it. In this case, our experiments were able to demonstrate the temporary nature of this leak, since it was detected only with a waiting time of 1 second.

Temporary leaks generally do not represent severe problems, but they are a relevant cause of false positives for testing tools that are not able to measure the leak persistence.

G. STUDY CONCLUSIONS

The study showed us the feasibility of the FunesDroid technique in finding memory leaks tied to the Android Activity lifecycle and confirmed the suitability of the tool in supporting the characterization of the obtained leaks in terms of their size, persistence, and growth trend. The study also showed the usefulness of the information reported by FunesDroid about the observed memory replications for finding the causes of memory leaks. Of course, these results have to be considered as preliminary, due to the limitedness of the considered sample of apps involved in the study. Further experiments involving a wider set of apps, having different characteristics, will be needed in order to extend the validity of the obtained results.

VI. RELATED WORK

Several works have been proposed in the literature that are aimed at detecting and fixing memory leaks in managed runtime environments based on virtual machines. In the context of Java applications, several techniques based on the analysis of the trend of the size of the heap memory [31]–[33], or, more specifically, on the analysis of the objects stored in the heap memory [34]–[36] have been proposed. In particular, several techniques which analyze the *staleness* of the heap objects have been proposed [37]–[43]. In addition, there are some commercial tools for the automatic detection of memory leaks, such as Plumbr.[23]

Recently, the problem of memory leaks has become prominent in the context of mobile applications due to the limited resources of mobile devices. The Android OS commercial success along with its open-source nature, led the researchers to focus on this mobile platform.

The techniques found in literature for memory leak detection in the context of Android apps can be classified in two main categories: (1) techniques based on static analysis of the source code of the apps and (2) techniques based on dynamic analysis, monitoring and testing of the apps.

Techniques belonging to the first category can be carried out in a completely automatic way, but, differently from our technique, they require the source code of the apps under test and they can only find some potential memory leaks causes, without any estimation of their severity. For example, Guo *et al.* [15] proposed a static analysis approach to detect leaks due to resources that are not correctly released after their use. The technique proposed by Blackshear *et al.* [19] finds memory leak causes due to incorrectly managed references on the basis of a combination of static analysis and symbolic execution. Another specific analysis technique is the one proposed by Lin textitet al. [20] and Lin and Dig [30], that have focused their attention of the detection and fixing of memory leak problems due to an incorrect programming of asynchronous tasks. Qian and Zhou [18] proposed a technique to prioritize test cases of a test suite according to their likelihood to cause memory leaks. Their approach is based on a machine learning model that predicts for each input test case its likelihood to cause a memory leak given its code features.

More general contributions are the ones of Palomba *et al.* [21] and Toffalini *et al.* [14], who proposed lists of programming patterns (i.e. bad smells) that can cause memory leaks and techniques to automatically detect them. Palomba *et al.* [21] proposed aDoctor, a code smell detector that identifies 15 Android-specific code smells including some smells potentially able to cause leaks. ADoctor analyzes the Abstract Syntax Tree produced by the compiler in order to find these smells. More recently, Toffalini *et al.* [14] proposed a static analysis technique and a tool for the automatic detection of potential memory leaks in Android applications, due to three different causes. They demonstrated the commonality of memory leaks in open source apps in the context of a study involving 500 apps available both on F-Droid and on Google Play. They manually validated a subset of the potential memory leaks found by the proposed tool, by monitoring the heap memory using the Android Memory Profiler utility provided by the Android framework [28].

In addition, Android programmers often use Lint [29], a static analysis tool integrated in the Android Studio programming environment, for the detection of a large and extensible set of bad smells, including some smells related to memory leaks.

[23] https://plumbr.io/

TABLE 7. Qualitative comparison between techniques and tools for dynamic memory leaks detection.

Ref.	Name	Tool Avail.	Autom.	Test Typology	Observed Leak Effects	Leak Characterization
[9]	-	N	Y	Random Events	Performance Degradation	Growing Trend
[44]	-	N	N	Specific testing patterns	App/System crashes	-
[17]	LeakDroid	Y	N	Neutral Event Sequences	App/System crashes	Growing Trend
[24]	-	N	N	Neutral Event Sequences	App/System crashes	Growing Trend
[16]	LeakDaf	N	Y	Systematic Exploration and Neutral Event Sequences	Replicated Objects	Size
[23]	LeakCanary	Y	N	User Interactions	Replicated Objects	Size
	FunesDroid	Y	Y	Neutral Event Sequences	Replicated Objects	Size,Persistence,Growing Trend

As regards the second category of techniques, which are based on dynamic analysis, monitoring and testing of the apps, the oldest relevant contribution is the one of Araujo *et al.* [9]. These authors proposed in 2013 a leak detection technique based on monitoring the trend of heap memory size in an Android app in response to a stress testing session carried out with Monkey. In 2014, Shahriar *et al.* [44] compiled a list of some specific causes of memory leaks and proposed some possible test cases able to evaluate if they can cause crashes or exceptions.

Several works in the literature have pointed out that Android apps suffer from issues that can be attributed to Activity Lifecycle mishandling, including memory leaks.

Yan *et al.* [17] proposed in 2013 a technique and a freely available tool called LeakDroid for the automatic generation of test cases exposing memory leaks and threading issues in Android applications. Their technique adopts a GUI model-based testing approach to produce test cases that perform repeated executions of events that should have neutral effects on resource consumption and should not lead to increases in resource usage. Among these neutral sequences of events, it considers sequences that exercised the Activity lifecycle loops, i.e. the EL and VL loops and provides a report of the growth of the used resources. Zhang et al. refined in [24] the notion of neutral sequence of GUI events proposed in the prior work [17]. They exploited this notion to systematically and automatically generate test cases from a static control-flow model of Android apps to detect resource leaks. Test case execution and leak detection were carried out with less manual effort with respect to their prior work, but still need some app-specific knowledge to provide additional manual inputs to some events from the tester, e.g., to decide which item to click in a list, or which string to enter in a text field. With respect to these works, our approach does not need any knowledge of the app under test, it is able to test also the FL loop and it provides detailed reports of the leaked objects.

Jun *et al.* [16] designed LeakDAF, a fully automated testing approach for detecting leakages of Activity and Fragment components in Android apps. Their approach combines the automated app UI exploration with the analysis of Android-specific Heap Profiling (hprof) files to identify leaked activities and fragments. To this aim, their exploration strategy systematically tests each Activity encountered during the app exploration by exercising the EL loop. LeakDAF carries out a memory dump only when the app crashes or when a predefined number of lifecycle events has been executed. LeakDAF inspects the dumps by checking framework-specific properties that reveal whether memory leaks occurred. FunesDroid instead does not depend on any app exploration technique and is able to directly test the app activities. Moreover, FunesDroid has the objective to evaluate the size and growth trend of the detected memory leaks, whereas LeakDAF only reports the number of leaks observed in each specific testing session.

Finally, a tool that is freely available to developers is LeakCanary [23]. LeakCanary is a memory leak detection library that has to be included in the source code of the app under test. It dynamically investigates the heap memory during the app execution, by searching for replicated objects and reporting them in a hprof output file. LeakCanary helps developers to dinamically find the occurrence of memory leaks while they interact with the app under test. With respect to our approach, the main limitations of LeakCanary are that it can be used only for open source apps (since the app has to be rebuilt after the insertion of some testing parameters), and that it cannot be used in the context of a completely automated testing process.

In Table 7 we qualitatively compare FunesDroid against six techniques and tools for Android memory leak detection that are based on dynamic analysis, testing, or monitoring. The Table reports: the Reference to the paper describing the technique, the Name of the tool implementing the technique if it exists, the Tool free Availability, the fully automation property, the type of executed tests, the observed effects of the leaks, and the observed characteristics of the found leaks. The last line of the Table reports the characteristics of the FunesDroid approach proposed in this paper.

As the Table shows, FunesDroid presents the following characteristics: (1) it is implemented in a tool that is freely available and usable on current Android applications; (2) it implements a completely automated testing process; (3) it allows to observe memory leaks at a fine level of details (in terms of sets of replicated objects); (4) it is able to provide a characterization of the memory leaks in terms of their size, persistence and growing trend. Even if other approaches proposed in the literature present some of these same characteristics, none of these works provide all these characteristics at the same time. This finding may be considered a FunesDroid peculiar point of strength.

VII. CONCLUSION

In this paper we presented FunesDroid, a testing technique and a tool for the automatic black box detection of memory leaks tied to the Activity Lifecycle in Android apps. The

technique is based on the execution of repetitions of three different sequences of neutral events exercising the Activity Lifecycle.

Differently from the static analysis approaches proposed in the literature, which only detect causes of potential memory leaks, FunesDroid is able to trigger real memory leaks and characterize their severity. Unlike most existing dynamic analysis approaches for finding memory leaks, ours is not invasive to the source code of the apps. In addition, the FunesDroid tool is freely available.

The proposed tool was exploited in a study whose results confirmed the diffusion of this issue in real apps and showed that more than a third of 283 open source apps published in F-Droid are affected by these leaks. In the paper, we also reported the results of a second study where we used FunesDroid for characterizing the size, persistence and growth trend of the obtained leaks. Moreover, we exploited the information provided by the tool to find the causes of the observed leaks.

The proposed memory leak detection approach presents some limitations. The first limitation is that FunesDroid is only able to detect potential memory leaks and that a further source code analysis of the app is needed to confirm the actual presence of memory leaks. However, the experimental results that we collected in the second study were encouraging, since we were able to confirm all the leaks that were considered. Another limitation is that FunesDroid limits itself to directly launch an Activity by an Intent call, disregarding the cases in which the Activity is launched by other activities at runtime. Therefore, we cannot exclude that additional potential memory leaks may be reported in these other app execution scenarios. To overcome this limitation, in future work we plan to explore further app behaviors by integrating FunesDroid with tools for the automatic GUI exploration of the apps, such as the Android GUI Ripper we presented in [45]. Moreover, we intend to improve the FunesDroid ability to test Activities by adopting heuristic-based strategies to define specific input values in the Intent call.

In future work we also intend to carry out a study for investigating the proportion of memory leaks due to lifecycle changes in real Android apps by exploiting bug repositories mining and reported leaks classification. Finally, we want to further investigate the causes of observed memory leaks. To this aim, we plan to combine static and dynamic analysis approaches for automatically classifying the causes of memory leaks.

ACKNOWLEDGMENT

The authors would like to thank R. Sellitto and G. D'Alterio for their contributions to the prototyping of the FunesDroid tool and to the analysis of some of the found memory leaks.

REFERENCES

[1] D. Amalfitano, V. Riccio, A. C. R. Paiva, and A. R. Fasolino, "Why does the orientation change mess up my Android application? From GUI failures to code faults," *Softw. Test. Verification Reliab.*, vol. 28, no. 1, p. e1654, Jan. 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1654

[2] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyande, and J. Klein, "Automated testing of Android apps: A systematic literature review," *IEEE Trans. Rel.*, vol. 68, no. 1, pp. 45–66, Mar. 2019.

[3] P. Tramontana, D. Amalfitano, N. Amatucci, and A. R. Fasolino, "Automated functional testing of mobile applications: A systematic mapping study," *Softw. Qual. J.*, vol. 27, no. 1, pp. 149–201, Mar. 2019, doi: 10.1007/s11219-018-9418-6.

[4] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet? (E)," in *Proc. 30th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, Lincoln, NE, USA, Nov. 2015, pp. 429–440.

[5] D. Amalfitano, N. Amatucci, A. M. Memon, P. Tramontana, and A. R. Fasolino, "A general framework for comparing automatic testing techniques of Android mobile apps," *J. Syst. Softw.*, vol. 125, pp. 322–343, Mar. 2017.

[6] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: Analysis, module and applications," in *25th Int. Symp. Fault-Tolerant Comput. Dig. Papers*, Nov. 2002, pp. 381–390.

[7] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *Proc. IEEE Int. Conf. Softw. Rel. Eng. Workshops (ISSRE Wksp)*, Nov. 2008, pp. 1–6.

[8] C. Weng, J. Xiang, S. Xiong, D. Zhao, and C. Yang, "Analysis of software aging in Android," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Oct. 2016, pp. 78–83.

[9] J. Araujo, V. Alves, D. Oliveira, P. Dias, B. Silva, and P. Maciel, "An investigative approach to software aging in Android applications," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Oct. 2013, pp. 1229–1234.

[10] D. Cotroneo, F. Fucci, A. K. Iannillo, R. Natella, and R. Pietrantuono, "Software aging analysis of the Android mobile OS," in *Proc. IEEE 27th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2016, pp. 478–489.

[11] S. Garg, A. Puliafito, M. Telek, and K. Trivedi, "Analysis of preventive maintenance in transactions based software systems," *IEEE Trans. Comput.*, vol. 47, no. 1, pp. 96–107, 1998.

[12] F. Machida and N. Miyoshi, "Analysis of an optimal stopping problem for software rejuvenation in a deteriorating job processing system," *Rel. Eng. Syst. Saf.*, vol. 168, pp. 128–135, Dec. 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0951832016305841

[13] A. Göransson, *Efficient Android Threading: Asynchronous Processing Techniques for Android Applications* (Programming/Android). Newton, MA, USA: O'Reilly, 2014. [Online]. Available: https://books.google.it/books?id=T141ngEACAAJ

[14] F. Toffalini, J. Sun, and M. Ochoa, "Practical static analysis of context leaks in Android applications," *Softw., Pract. Exper.*, vol. 49, no. 2, pp. 233–251, Feb. 2019. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85056271576&doi=10.10022fspe.2659&partnerID=40&md5=0e0803fd929feae0044c6f76fc73e54f

[15] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in Android applications," in *Proc. 28th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, Nov. 2013, pp. 389–398.

[16] M. Jun, L. Sheng, Y. Shengtao, T. Xianping, and L. Jian, "LeakDAF: An automated tool for detecting leaked activities and fragments of Android applications," in *Proc. IEEE 41st Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Jul. 2017, pp. 23–32.

[17] D. Yan, S. Yang, and A. Rountev, "Systematic testing for resource leaks in Android applications," in *Proc. IEEE 24th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2013, pp. 411–420.

[18] J. Qian and D. Zhou, "Prioritizing test cases for memory leaks in Android applications," *J. Comput. Sci. Technol.*, vol. 31, no. 5, pp. 869–882, Sep. 2016, doi: 10.1007/s11390-016-1670-2.

[19] S. Blackshear, B.-Y. E. Chang, and M. Sridharan, "Thresher: Precise refutations for heap reachability," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2013, pp. 275–286. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462186

[20] Y. Lin, S. Okur, and D. Dig, "Study and refactoring of Android asynchronous programming (t)," in *Proc. 30th IEEE/ACM Int. Conf. Automat. Softw. Eng.*, 2016, pp. 224–235. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84963852358&doi=10.1109%2fASE.2015.50&partnerID=40&md5=6f8c7c0067245374ff3b1dcf33ced077

[21] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of Android-specific code smells: The aDoctor project," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2017, pp. 487–491.

[22] F. Toffalini, J. Sun, and M. Ochoa, "Static analysis of context leaks in Android applications," in *Proc. 40th Int. Conf. Softw. Eng., Softw. Eng. Pract.*, 2018, pp. 215–224. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85049689380&doi=10.1145%2f3183519.3183530&partnerID=40&md5=7e9677446a7fc409edf6f3ec1ca4e99c

[23] *LeakCanary. A Memory Leak Detection Library for Android*. Accessed: Dec. 15, 2019. [Online]. Available: https://github.com/square/leakcanary

[24] H. Zhang, H. Wu, and A. Rountev, "Automated test generation for detection of leaks in Android applications," in *Proc. IEEE/ACM 11th Int. Workshop Autom. Softw. Test (AST)*, May 2016, pp. 64–70.

[25] P. Dubroy. (May 2011). *Memory Management for Android Apps*. [Online]. Available: https://dubroy.com/memory_management_for_android_apps.pdf

[26] Y. Qiao, Z. Zheng, Y. Fang, F. Qin, K. S. Trivedi, and K.-Y. Cai, "Two-level rejuvenation for Android smartphones and its optimization," *IEEE Trans. Rel.*, vol. 68, no. 2, pp. 633–652, Jun. 2019.

[27] V. Riccio, D. Amalfitano, and A. R. Fasolino, "Is this the lifecycle we really want?: An automated black-box testing approach for Android activities," in *Proc. Companion ISSTA/ECOOP Workshops (ISSTA)*, 2018, pp. 68–77, doi: 10.1145/3236454.3236490.

[28] *Android Studio Memory Profiler*. Accessed: Dec. 15, 2019. [Online]. Available: https://developer.android.com/studio/profile/memory-profiler Last

[29] *Android Lint*. [Online]. Available: http://tools.android.com/lint/overview

[30] Y. Lin and D. Dig, "Refactorings for Android asynchronous programming," in *Proc. 30th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, Nov. 2015, pp. 836–841. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84963863713&doi=10.1109%2fASE.2015.100&partnerID=40&md5=a1ec26127198b21fd6695b67cb66a917

[31] V. Sor, P. Ou, T. Treier, and S. N. Srirama, "Improving statistical approach for memory leak detection using machine learning," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2013, pp. 544–547.

[32] M. Jump and K. S. Mckinley, "Cork: Dynamic memory leak detection for garbage-collected languages," *SIGPLAN Not.*, vol. 42, no. 1, pp. 31–38, Jan. 2007, doi: 10.1145/1190215.1190224.

[33] H. Yu, X. Shi, and W. Feng, "LeakTracer: Tracing leaks along the way," in *Proc. IEEE 15th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2015, pp. 181–190.

[34] K. Chen and J.-B. Chen, "Aspect-based instrumentation for locating memory leaks in java programs," in *Proc. 31st Annu. Int. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2, Jul. 2007, pp. 23–28.

[35] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, May 2008, pp. 151–160.

[36] E. K. Maxwell, G. Back, and N. Ramakrishnan, "Diagnosing memory leaks using graph mining on heap dumps," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2010, pp. 115–124. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-77956202108&doi=10.1145%2f1835804.1835822&partnerID=40&md5=25600d6db76d467b2b105fc11f3a656a

[37] D. Rayside and L. Mendel, "Object ownership profiling: A technique for finding and fixing memory leaks," in *Proc. 22nd IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, 2007, pp. 194–203, doi: 10.1145/1321631.1321661.

[38] M. D. Bond and K. S. Mckinley, "Tolerating memory leaks," *SIGPLAN Notices*, vol. 43, no. 10, pp. 109–126, Oct. 2008, doi: 10.1145/1449955.1449774.

[39] M. D. Bond and K. S. Mckinley, "Leak pruning," *SIGPLAN Notices*, vol. 44, no. 3, pp. 277–288, Feb. 2009, doi: 10.1145/1508284.1508277.

[40] G. Novark, E. D. Berger, and B. G. Zorn, "Efficiently and precisely locating memory leaks and bloat," *SIGPLAN Notices*, vol. 44, no. 6, pp. 397–407, May 2009, doi: 10.1145/1543135.1542521.

[41] G. Xu, M. D. Bond, F. Qin, and A. Rountev, "Leakchaser: Helping programmers narrow down causes of memory leaks," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2011, pp. 270–282. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993530

[42] S. Lee, C. Jung, and S. Pande, "Detecting memory leaks through introspective dynamic behavior modelling using machine learning," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, 2014, pp. 814–824, doi: 10.1145/2568225.2568307.

[43] C. Jung, S. Lee, E. Raman, and S. Pande, "Automated memory leak detection for production use," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, 2014, pp. 825–836, doi: 10.1145/2568225.2568311.

[44] H. Shahriar, S. North, and E. Mawangi, "Testing of memory leak in Android applications," in *Proc. IEEE 15th Int. Symp. High-Assurance Syst. Eng.*, Jan. 2014, pp. 176–183.

[45] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proc. 27th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE)*, 2012, pp. 258–261, doi: 10.1145/2351676.2351717.

DOMENICO AMALFITANO received the Ph.D. degree in computer engineering and automation from the University of Naples Federico II, in 2011.

He is currently an Assistant Professor with the Department of Electrical Engineering and Information Technology (DIETI), University of Naples Federico II, Italy. His main research interests include in the areas of software engineering, software testing, software testing automation, reverse engineering, software maintenance, program comprehension, and software development processes improvement. He applied his research activity in the contexts of Mobile Apps, Web Applications, and Automotive Embedded Software.

VINCENZO RICCIO received the degree in computer engineering and the Ph.D. degree from the University of Naples Federico II, in 2015 and 2019, respectively. He is currently a Postdoctoral Researcher with the Software Institute of Università della Svizzera Italiana, Lugano, Switzerland. His research is focused on software engineering applied to mobile and machine learning-based applications. His research fields include reverse engineering, testing, comprehension, and software quality.

PORFIRIO TRAMONTANA received the degree in computer engineering and the Ph.D. degree from the University of Naples Federico II, in 2001 and 2005, respectively. He is currently an Assistant Professor with the University of Naples Federico II. His research is focused on software engineering applied to mobile and Web applications. His research fields include reverse engineering, testing, maintenance, comprehension, migration of legacy systems, and software quality.

ANNA RITA FASOLINO received the M.S. Laurea degree in electronic engineering and the Ph.D. degree in electronic and computer engineering from the University of Naples Federico II.

She was an Assistant Professor with the University of Bari, Italy. She is currently an Associate Professor with the Department of Electrical Engineering and Information Technology, University of Naples Federico II, Italy. She has coauthored more than 100 articles in peer-reviewed international journals, books, and proceedings of conferences and workshops. Her research interests are in the area of software engineering with a focus on software testing, mobile app testing, reverse engineering, Web engineering, and embedded software engineering. In such fields, she developed and participated in numerous R&D projects. She serves as a member of the program committee for several conferences in software engineering. She has won distinguished paper awards at ICSM 2002 and ICSM 2012 for her work on Web testing and automated mobile app GUI testing. She is also a Co-Organizer of special issues and workshops related to testing of event-based software. She co-chaired the Doctoral Symposium at the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017). She is an Academic Editor of the *Journal of Systems and Software*, *PeerJ Computer Science*, open access journal, and *Computers Journal MDPI*.