Remembrall: A Static Analysis Tool for Android Resource Leak Detection

Rabia Akhtar
Columbia University
ra2805@columbia.edu

Joshua Learn
Columbia University
jrl2196@columbia.edu

Abstract

In this paper we present our lightweight analysis tool called Remembrall, which can statically analyze an Android applications resources and locate the allocation path to the resource leaks. Remembrall builds and augments a function call graph construction that can incorporate links to callbacks. This allows our tool to give detailed leak information to developers and offers an improvement in runtime. Remembrall has been applied to real applications and has been able to successfully identify whether or not these applications have resource leaks without overhead in runtime.

I. Introduction

Android phones dominate over 80% of the global smartphone market share and have become significant players in the global market [1]. Software reliability in Android apps is vital as system crashes can have wide, long-lasting impacts. In particular, memory leaks in Android apps are unique because of the interaction between the Android life cycle and hardware components. Consequently, resource leaks in Android apps occur when allocated resources, like hardware components, are not explicitly released by programmers. This can lead to degradation in the performance of the application or even a system crash. In this paper we focus on resource leak problems in Android apps, and present our lightweight analysis tool called Remembrall[1], which can statically analyze an Android applications resources registered through callbacks and locate the path to the resource leaks.

II. Background

Android is an open source mobile operating system developed by Google. It is based on a modified version of Linux and is the best selling OS worldwide [1]. Android applications are written in Java and compiled to Dalvik bytecode. This is then packed into Android app package files like an APK file for easy installation. An Android app usually consists of four components: activities, services, broadcast receivers, and content providers [2]. In particular, activities are fundamental to the application. As a user navigates through the application, the activity instances move through different states in their lifecycle as shown in Figure 1 [3]. The Activity class provides callbacks that allow the activity to know that a state has changed like onPause, onStop, or onRestart. In addition, resources are registered in the main lifecyle of the activity. Take for example, the camera hardware component. Camera hardware is a shared resource that must be carefully managed so that an application does not block other applications that want to use it. The Camera API warns that a user should remember to release the Camera object by calling Camera.release() when the application is done using it. In the Android life cycle, as shown in Figure 1, this would be in the onPause method. If an application does not properly release the camera, all following attempts to access the camera will fail and may cause a system crash [4]. The main challenge in working with Android apps lies in the component based and event driven nature of the Android framework. The implicit callback mechanism provided by Android system makes the generation of a precise call graph very difficult [5].

[1]In the Harry Potter universe, a Remembrall is a large marble sized glass ball that contains smoke which turns red when its owner has forgotten something.

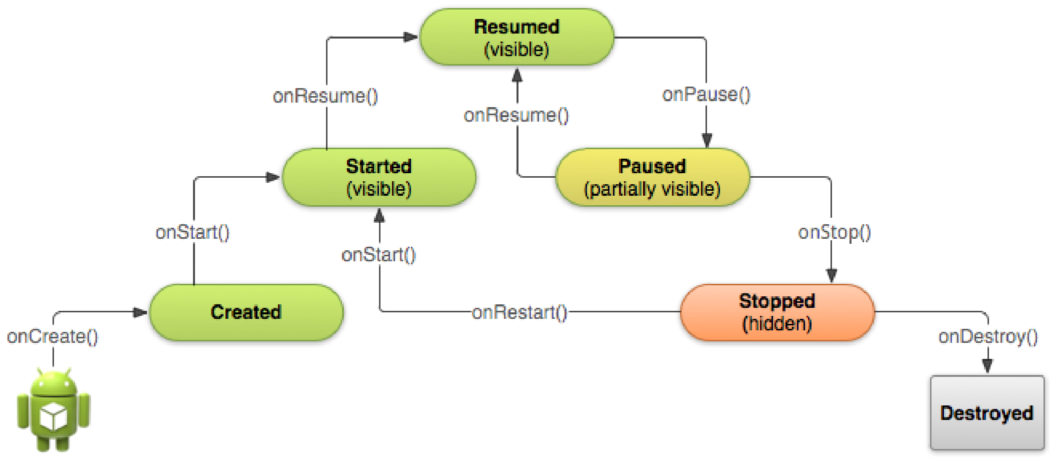

Fig. 1. Flow of the Android activity lifecyle [6]

Remembrall presents a way to build this call graph and incorporate the missing callback links into the graph.

III. Prior Work

As software reliability is critical for Android applications, there have been several studies on static analysis tools for Android memory leaks that we improve on with Remembrall.

A. *Relda and Relda2*

The first is Relda, a static analysis tool for Android apps that detects resource leaks by analyzing the apps byte code [7]. Relda's analysis focuses on three kinds of resources: exclusive resources, memory-consuming resources and energy consuming resources. These resources all need to be explicitly released. First, Relda inspects all the resource operations in the application to collect the potential resource leaks it contains. Relda identifies that for a global resource, it should be released on all the paths that can be reached from its request point or in at least one callback of all interfaces implemented by the classes. Relda does this by first decompiling an APK and generating standard dalvik bytecode of an app. The team takes a list of resource release and request methods they manually compiled to identify relevant methods in the program. Then the tool traverses the bytecode and for each resource request and release operation, it adds links into a function call graph. Relda resolves missing links in the call graph by using a Permission examining tool called Woodpecker. Woodpecker adds missing links in the call graph caused by callbacks. Then Relda's resource summary algorithm uses depth first search to analyze the resource operations and call graph in order to find unreleased resources. In the evaluation process, when Relda was tested on 43 non-trivial real world apps, it was found that 25 of them have one or more resource leaks, and there were 92 reported leaks in total. Overall, the teams research shows that Relda is effective in detecting various resource leaks in Android apps.

Relda2 builds upon the work done by Relda [5]. It provides an interprocedural analysis framework for resource leaks. The tool supports two analysis techniques, flow insensitive and flow sensitive. The flow insensitive analysis ignores the control flow information and can quickly analyze an app, while the flow-sensitive technique can eliminate a number of false negatives at the cost of speed. In particular, Relda2 handles the callback links in the call graph in an efficient manner. The mechanism figures out all possible execution paths in the application for each callback. Then each path is added to the resource path. Relda2 was applied to 103 apps where it found and confirmed 67 real resource leaks in them. Overall, we use the call back graph techniques that Relda and Relda2 delineate and improve them by adding the callback interface analyzer. We are able to improve runtime by our static approach to callback interface and resource allocation analysis.

B. *FlowDroid*

In order analyze callbacks, we incorporate the research of FlowDroid into Remembrall [8]. FlowDroid implements a static data flow tracking tool for Android apps that consists of a data flow tracker and platform-specific extensions. FlowDroid identifies the challenges of building a tool that can identify callback subscription methods. Such a tool would have to maintain a list of all methods in the Android SDK that register callbacks and this list would have to be updated as the Android system updates over time. Thus, FlowDroid presents an identification method that is based on the idea that a callback subscription method must accept the callback interface as one of its arguments. Thus, FlowDroid maintains a list of callback interface types. Therefore, if a method's argument has one of these types, FlowDroid assumes it to register that interface implementation as a callback. FlowDroid is able to confirm that with this method there are no false positives because this mechanism over approximates subscription methods. In Remembrall, we use the list of callback interfaces provided by FlowDroid's open source code in order to allow Remembrall to identify the callback subscription methods. Remembrall is then able to analyze each callback interface in the list and output their methods in order to link them back into the call graph.

IV. Materials

We use two open source tools in order to run Remembrall. The first is Androguard; Androguard is a python tool which provides reverse engineering for Android applications [9]. Androguard takes a raw APK file of an application and breaks it down to analyze. Androguard is capable of mapping APK and Dex format files. Androguard also provides access to the static analysis of the code, including features like basic blocks, instructions, and permissions. For example, the androguard.core.analysis.analysis.Analysis object contains information about the classes, methods, fields and strings inside one or multiple DEX files. This object also allows a user to build call graphs and links for each method, class, field and string. Thus, it is an ideal candidate to use to build a static analysis tool like Remembrall. Additionally, we use JavaParser to analyze each callback interface class in the callback list provided by FlowDroid and extract the methods of the interface [10]. JavaParser is able to parse Java code and build an Abstract Syntax Tree and it also provides analysis of the fields in the Java code. JavaParser is a key tool that allows Remembrall to incorporate method information from callback interfaces into its call graph.

V. Methods

The overall flow of Remembrall can be seen in Figure 2, where the APK is preprocessed, its callback interfaces analyzed, and then its resources analyzed. The results of the analysis tool are then outputted to the user.

A. *Preprocess*

Remembrall works by building an augmented call graph that links resource allocations to resource deallocations. First, it takes the user inputted APK file name and decompiles the APK into Dalvik bytecode. From the bytecode, it utilizes AndroGuard to build a simple call graph and abstract syntax tree.

B. *Callback Interface Analyzer*

This call graph does not maintain links to methods invoked in callbacks and so Remembrall utilizes the following process to augment the call graph. Remembrall traverses the AST and finds all registered callbacks, as shown in Algorithm 1. As shown by FlowDroid, we assume that every callback subscription method must at least accept the callback interface as one of its parameters and we use this to maintain a list of registered callback interfaces. Thus, by comparing the argument type to the list of callback interfaces

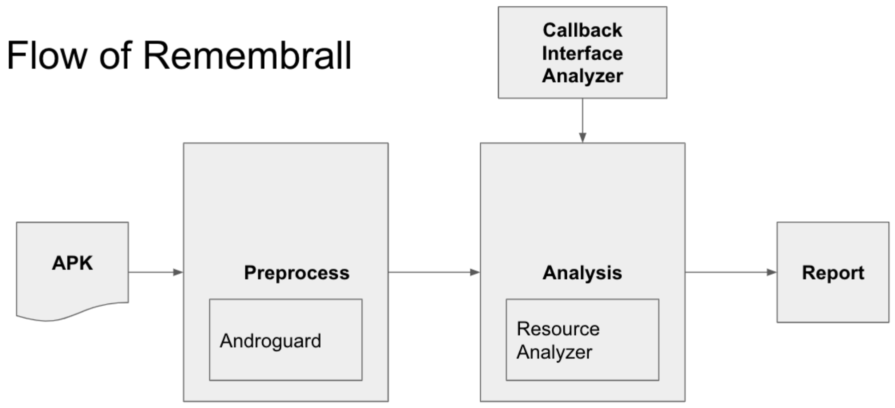

Fig. 2. Flow Diagram of Remembrall

provided by FlowDroid, we are able to build a list of registered callbacks. After obtaining this list of registered callbacks, the callback interface analyzes each of these callback interfaces using JavaParser. It then links each subscription method in the call graph to all methods in the callback interface. After this, the callback interface analyzer has finished augmenting the graph with the callbacks registered and its methods.

Algorithm 1: Find all registered callback interfaces.

```
Data: AST, CallbacksList
registeredCallbacks = []
while not at end of AST do
    read methods
    for method $\in methods$ do
        read method invocations
        for invocation $\in invocations$ do
            read argument types
            for type $\in types$ do
                if instanceOfClass(type) and type $\in CallbacksList$ then
                    registeredCallbacks.append(type)
```

C. Resource Analyzer

With this augmented call graph in place, the resource analyzer is able to link each resource allocation to its deallocation. First, the analyzer builds a list of potential methods that are resource openers and closers using the key words shown in Table 1, augmenting the keywords identified by Relda. Then, for each activity entry point, where an entry point is a method that does not have a method invoking it, the resource analyzer builds all simple paths from the entry points to each resource opener using the shortest simple paths method from the networkx package. This is maintained as the allocation path to the resource opener method. Then for each resource opener, it builds deallocation paths that link the function invoking the resource opener to any corresponding resource closer. If at the end of this process, a resource opener has no deallocation path associated with it, this is outputted as a resource leak and its allocation path is printed. Otherwise, the program outputs no leak.

Resource Request	Resource Release
start	end
request	abandon
lock	cancel
open	clear
register	close
acquire	disable
vibrate	finish
enable	recycle
	release
	remove
	stop
	unload
	unlock
	unmount
	unregister

TABLE I
RESOURCE REQUEST AND RELEASE KEYWORDS

VI. RESULTS

Our results showed that Remembrall is able to detect resource leaks in an Android app or confirm that a program is reliable.

A. Toy Example APK

In order to inspect our results, we present the application in Figure 3. This simple application allocates a Camera resource and forgets to release it on an onPause change. Remembrall builds the allocation path for this camera allocation and outputs this allocation path when a deallocation path for it in the call graph. The report given by Remembrall is visualized in Figure 4. Here the application follows the allocation in getCameraInstance to the Camera open invocation. Since this was never released, after generating all deallocation paths, Remembrall finds none for the Camera allocation and outputs that the resource has been leaked.

```
public class MainActivity extends AppCompatActivity {
    private Camera mCamera = null;
    private CameraPreview mPreview = null; ...
    protected void onCreate(Bundle savedInstanceState) { ...
        mCamera = getCameraInstance(); ...
    }
    public static Camera getCameraInstance(){ ...
            c = Camera.open(); // get a Camera instance
            ...
    }
    @Override
    protected void onPause() { ...
            //missing release operation: mCamera.release()
    }
```

Fig. 3. Snippet of Application Allocating and Not Deallocating a Camera Resource.

B. Real World APK

In order to test Remembrall, we also test it on a real world application, Spotify. Spotify is a Swedish audio streaming platform that provides music and podcasts from record labels and media companies [11].

```
Decompiling APK...
Getting syntax tree...
Analyzing callbacks...
...
Resource path not closed:
PATH:
onCreate -> getCameraInstance ->
Landroid/hardware/Camera; -> open()Landroid/hardware/Camera;
```

Fig. 4. Output Report by Remembrall for Application in Figure 3.

Spotify is available in 79 countries and has over 500 million downloads on the Google Play store. When testing Remembrall on the Spotify app we are able to test its performance, which can be seen in Figure 5. The test was run on an Ubuntu 18.04 computer with i5 consumer grade 4gb ram. Additionally, the Spotify APK file was 33 MB. As you can see, preprocessing takes around 202 seconds, or 3 minutes. Preprocessing includes decompiling the APK and building the call graph. The resource analyzer also only takes around 2.3 seconds. This fast runtime shows that Remembrall is able to build a function call graph for a large real world app in a short amount of time. In its output, Remembrall reports that startActivity, startForegroundService, and startService are leaked. Because of the obfuscation of the Spotify source code, confirmation of this leak is not a trivial task. If these leaks are false positives, we speculate that Remembrall may not be finding entry points caused from clicking on the user interface. When a user stops music, Remembrall may not be able to recognize the exit points.

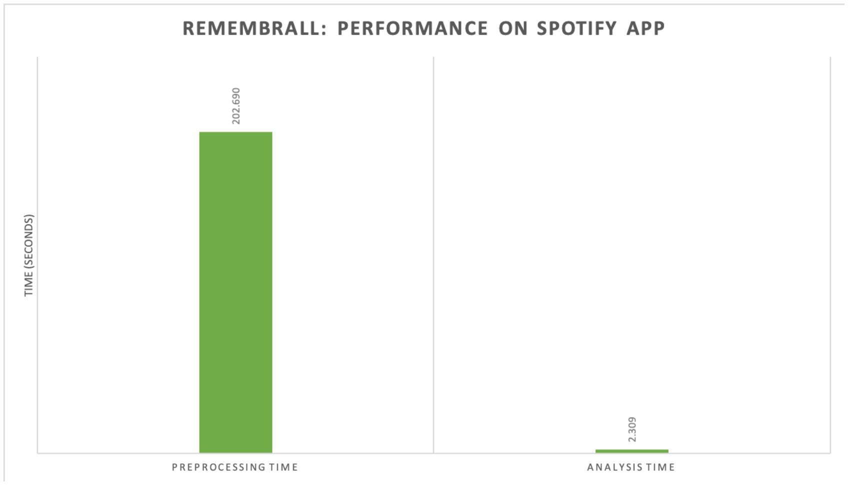

Fig. 5. Performance Metrics of Remembrall on the Spotfiy App.

VII. Conclusions and Future Work

Remembrall is shown to be an effective static analysis tool for Android applications. Remembrall can build an augmented function call graph that can incorporate callback interface methods. Remembrall is able to locate the allocation path to resource leaks. Remembrall has been applied to toy and real world applications and has been able to successfully build a call graph and identify whether or not these applications have resource leaks. However, in order for Remembrall to be released as an open source tool, it will need a comprehensive test suite. The test suite should be able to verify reported resource leaks, like the leaks in the Spotify app. Remembrall will also need to be expanded so that false positives and other noise is ruled out to make for a better report to users. Enterprise Android applications are complex, often obfuscated programs and Remembrall needs to be adapted to handle these cases. Finally, Remembrall can be improved to incorporate other memory leaks in Android like a static context leak. Overall, Remembrall can currently detect resource leaks in Android apps and has the potential to be released as a open source tool in the future.

References

[1] "Smartphone market share - os." [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os

[2] Y. Liu, L. Wei, C. Xu, and S.-C. Cheung, "Droidleaks: Benchmarking resource leak bugs for android applications," 11 2016.

[3] "Activity." [Online]. Available: https://developer.android.com/reference/android/app/Activity

[4] "Camera." [Online]. Available: https://developer.android.com/guide/topics/media/camera

[5] T. Wu, J. Liu, X. Deng, J. Yan, and J. Zhang, "Relda2: An effective static analysis tool for resource leak detection in android apps," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2016, pp. 762–767.

[6] S. Park and S. Park, "Activity lifecycle in android applications," Mar 2017. [Online]. Available: https://medium.com/sketchware/activity-lifecycle-in-android-applications-1b48a7bb584c

[7] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in android applications," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 389–398.

[8] S. Arzt, "Static data flow analysis for android applications," Ph.D. dissertation, Technische Universität, Darmstadt, 2017. [Online]. Available: http://tuprints.ulb.tu-darmstadt.de/5937/

[9] "Androguard." [Online]. Available: https://androguard.readthedocs.io/en/latest/intro/index.html

[10] "Javaparser." [Online]. Available: https://javaparser.org/

[11] "Spotify. music for everyone." [Online]. Available: https://www.spotify.com/us/