2016 IEEE 27th International Symposium on Software Reliability Engineering

Fixing Resource Leaks in Android Apps with Light-weight Static Analysis and Low-overhead Instrumentation

Jierui Liu[1,3], Tianyong Wu[1,3], Jun Yan[†1,2,3], Jian Zhang[†1,3]
1. State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
2. Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
3. University of Chinese Academy of Sciences
{liujr, wuty, yanjun, zj}@ios.ac.cn

***Abstract*—Fixing bugs according to bug reports is a labor-intensive work for developers and automatic techniques can effectively decrease the manual efforts. A feasible solution is to fix specific bugs by static analysis and code instrumentation. In this paper, we present a light-weight approach to fixing the resource leak bugs that exist widely in Android apps while guaranteeing the safety that the patches should not interrupt normal execution of the original program. This approach first performs a light-weight static analysis and then carefully designs the concise patch code that will be inserted into the byte-code. When the program is running, the patches will trace the state of leaked resources and release them in a proper place. Our experiments on dozens of real-world apps show that our approach can effectively fix resource leaks in the apps with negligible extra execution time and less than 4% extra code in a few seconds.**

I. Introduction

Nowadays, Android smartphones are becoming more and more popular and a lot of new apps are deployed to the Android markets. The quality of the apps has attracted much attention in academic community and many research works focus on how to analyze the apps and find bugs. The resource leak problem is one of the common bugs for Android apps. It comes with the absence of the release operations of resources (such as Media Player, Camera and various kinds of Sensors) that have been acquired and should be manually released when they are no longer used. The resource leaks may cause some annoying effects, including battery shortage, memory waste and program crash.

Our previous work [11], [24] proposed a static analysis approach to automatically detecting potential resource leaks in Android apps. However, finding resource leaks is only the first step to debug the software. Fixing the reported bugs is also a labor-intensive work that developers and testers should design specific test cases to trigger the bugs, find the root cause, cautiously modify the incorrect parts of the source code and perform a regression testing until the program is completely repaired. Therefore, an automatic way to fix bugs can efficiently ease the burden of developers and testers. Furthermore, with the help of an accurate tool, we can even efficiently fix the bugs without the participation of the developers.

[†]Corresponding authors

In this paper, we try to automatically repair the Android apps to eliminate the reported resource leaks. Recently, some techniques [8], [14]–[16], [22] have been proposed to automatically fix the bugs by mutating software code heuristically until the software passes the regression testing. However, this kind of techniques is not efficient for fixing this specific type of bugs, since it needs a large number of test cases and repeated regression testing. Another type of approaches [10], [21], [25] aim to fix a specific type of bugs, based on precise static analysis. Compared with mutation based techniques, these approaches do not rely on a lot of testing and can patch a program in a short time. We believe that direct patching for a specific bug is more efficient. Thus we try to insert the missing resource release operations in proper places based on this thought.

According to Android reference, developers need to release resources in specific callbacks (e.g. `onDestroy()`), which makes the fixing of this bug different from some similar problems (e.g. memory leak). The effects of this requirement to bug fixing lie in two-folds. On one hand, it limits the scope of places to insert the resource release operations. Even so, we still need some static analysis techniques to figure out the precise release location, due to the complex dynamic features brought by Android system (e.g. multi-entrance, implicit invocations). On the other hand, it increases the difficulty to obtain the references to the leaked resource objects and their states, especially when the objects are not visible in those specific callbacks.

Another issue that should be considered is which level of code we should process. Most of existing specific fixing techniques focus on source code of C or Java programs [10], [21], [23]. However, for the apps running on Android platform, byte-code plays an important role because an app is released in the form of byte-code instead of source code. Therefore, byte-code is easier to be obtained and the software maintainers can perform a quick fixing even without accessing the source code, especially when the apps need to be deployed to the market in a short time.

This paper aims to find an efficient way to fix resource leaks in real-world Android apps. To our best knowledge,

2332-6549/16 $31.00 © 2016 IEEE
DOI 10.1109/ISSRE.2016.15

there is no dedicated approach that targets fixing resource leaks in Android apps. With the consideration of the above issues, we propose a fully-automated approach based on light-weight static analysis and byte-code instrumentation. Specifically, for each reported resource request operation, we statically figure out the places to insert the code, and design auxiliary variables to dynamically trace the states of the resources for preventing improper release. We have implemented these techniques into a prototype tool `RelFix` and validated our approach on dozens of real-world instances. The experimental results show our approach possesses the following characteristics.

- **Safe fixing.** Compared to the original app, the fixed APK preserves the same functionalities without introducing extra resource related bugs. All the processed apps passed the testing of `Monkey` [7].
- **Light-weight analysis.** The static analysis for the instances are all processed in less than one second.
- **Low overhead.** Our approach only inserts a few lines of additional code into byte-code to release leaked resources with an acceptable instrumentation overhead (less than 4% increase of code lines).

The rest of this paper is organized as follows. The next section introduces some background knowledge for this paper. Section III shows the overview and the details of our approach and Section IV shows some of our analysis on it. Experimental results on our self-designed benchmarks and real-world apps are shown in Section V. Then we have some discussions in Section VI. Section VII presents the related works and Section VIII concludes our works.

II. Background

This section provides the necessary background knowledge and a motivating example for further discussions.

A. Android Basics

Android apps are composed of four types of components: Activities (that provide an interface with which users can interact), Services (that handle long-running processes in background), Content Providers (that manage data switching between different apps) and Broadcast Receivers (that react to broadcast messages). According to the Android reference [4], the Activities are frequently paused and resumed, therefore the resource leaks in them will seriously affect the system, so we restrict our work to the Activities.

In an Android app, each Activity is required to follow a lifecycle defining how it should be created, used and destroyed. There are several callbacks in the lifecycle [1], where developers can override them to handle state transitions in the lifecycle. Fig. 1 shows the lifecycle of an Activity. We can see that an Activity begins its life with `onCreate()` and ends with `onDestroy()`, and a state change will make a corresponding callback be called. Note that there is only one active Activity at a time.

Similar to Java GUI programs, Android apps are usually driven by events and callbacks. Sometimes, app developers need to set a listener for a class (for example, the Button)

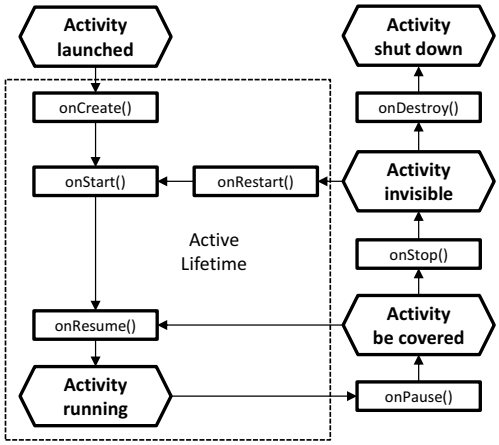

Fig. 1. Activity lifecycle

and when a specific condition is triggered (when the Button is clicked), the code in a specific method will be executed. We call the operation that sets the listener *registration operation* and the method that is invoked under a specific condition *callback method*.

Developers can use some resource components to make their apps attractive and interesting, but the improper use of these resources can lead to a resource leak problem, which leads to some negative effects including battery shortage, memory waste and even program crash. A resource leak happens when a resource is acquired in an Activity but is forgotten to be released before the execution of the appropriate callback method. Android reference suggests releasing the resource in one of the following callback methods: `onPause()`, `onStop()` and `onDestroy()`. In this paper, we try to automatically fix the potential resource leaks based on the bug report generated by `Relda2` [24].

In Android system, a *resource* that represents a real resource object at runtime is acquired by a *resource request operation* and released by a *resource release operation*, both of which are called *resource related operations* in this paper.

B. Motivating Example

In this section, we give a simple example to illustrate the motivation of this work. *SimpleRecorder* is a simple app to record and play audio data that uses the resource `AudioRecord`. A resource related code snippet of *SimpleRecorder* is shown in Fig. 2.

There are two Buttons (`mStartBt` and `mStopBt`) in the `RecordActivity`. It starts recording when we click the `mStartBt` Button and stops recording by clicking the `mStopBt` Button. When an `AudioRecord` instance is created, its `release()` method should be invoked to release the resource before the Activity is destroyed, i.e. in `onDestroy()` method. In this app, a new object (`mAR`) of `AudioRecord` will be created when the `mStartBt` is clicked and `mAR.release()` is only invoked when the

```
public class RecordActivity extends Activity {
  ......
  protected void onCreate(Bundle
      savedInstanceState) {
    ......
    initRecordArg();
    RecLis rl = new RecLis();
    mStartBt = (Button)findViewById(R.id.b1);
    mStartBt.setOnClickListener(rl);
    mStopBt = (Button)findViewById(R.id.b2);
    mStopBt.setOnClickListener(rl);
  }
}
class RecLis implements View.OnClickListener {
  private AudioRecord mAR = null;
  public void onClick(View view) {
    if (view.id == R.id.b1) startRecord();
    if (view.id == R.id.b2) stopRecord();
  }
  private void startRecord(){
    ......
    mAR = new AudioRecord(...);
  }
  private void stopRecord(){
    ......
    mAR.release();
  }
}
```

Fig. 2. Source code snippet of SimpleRecorder

mStopBt is clicked. As a result, a resource leak happens when we click the mStartBt and then quit the RecordActivity without clicking mStopBt Button.

To manually fix this leak without removing any parts of the original code, we may add another release() operation in the onDestroy() callback method. Besides, to avoid double release, we need some extra code to trace the release of the resource.

```
public class RecordActivity extends Activity {
  ......
  protected void onDestroy(){
    super.onDestroy();
    if (null != sAR) sAR.release();
  }
  public static AudioRecord sAR = null;
}
class RecLis implements View.OnClickListener {
  ......
  private void startRecord(){
    ......
    mAR = new AudioRecord(...);
    RecordActivity.sAR = mAR;
  }
  private void stopRecord(){
    ......
    mAR.release();
    RecordActivity.sAR = null;
  }
}
```

Fig. 3. The code to fix the resource leak

The patch code is shown in Fig. 3. We fix the resource leak in three steps. We find out the **location** to insert the target release operation. The RecLis.startRecord() method directly invokes the resource request operation AudioRecord.<init>() (in byte-code, the new operation invokes a <init> operation) and it is implicitly invoked in the method onCreate() of the Activity RecordActivity. So the target resource release operation should be inserted in the onDestroy() method of the RecordActivity class.

We find out the **release operation** that contains the name of the operation and the proper arguments the operation should accept. From the Android reference, we know that the AudioRecord.release() method is the corresponding release operation and it accepts one argument that is the reference to the resource object (mAR). However, the resource object is not visible in the location method (RecordActivity.onDestroy()), so we introduce an auxiliary variable (sAR) that is the snapshot of mAR when the resource is required to make the resource object become visible in the location method. Then we insert the sAR.release() as the release operation.

If we only insert the resource release operation, the resource may be double released. For example, when mStopBt Button is clicked, the original release operation mAR.release() will be executed and the inserted release operation will release the resource again. Thus we trace the state of the resource (released or not) and design the **guard** as the condition of the resource release operation that prevents this issue. In this simple case, we assign null to the auxiliary variable (sAR) to indicate the resource has been released, and the guard is to determine whether (sAR) is null. We will make use of more complicated data structures to record multiple unreleased resources during the implementation (see section III-F).

C. Automatic Bug Fixing

In this paper, we want to fix the resource leaks automatically. Some researches have focused on the specific bug fixing. Gao *et al.* [10] put forward an approach that fixes the memory leak problem for C programs and introduced the concept of "safe fixing". Inspired by this work, we try to safely fix the resource leak problem that the fixed app should keep the behavior of the original one without introducing new bugs. We give the definition of safe fixing on resource leaks as follows.

Definition 1: A resource leak fixing is safe, if all of the following properties hold for any execution path after we insert the corresponding resource release operations into the program.

1. Every resource that has been acquired, should be released.
2. Every resource that has not been acquired, should not be released.
3. Every resource that has been released, should not be released again.

D. Basic Concepts

For convenience and further discussion, we introduce some concepts that will be used in the rest of the paper.

- **OCA.** For a leaked resource request operation, the key part to determine the location where we should insert the

target release operation is to find the Activities that may invoke it. We call these Activities *Operation Carrying Activities* (*OCA* for short). Note that the OCA is a set of Activities.

- **RCA.** A leaked resource request operation may be invoked by more than one Activities statically, but each resource required by this operation only belongs to a unique Activity where the resource should be released. We call this Activity *Resource Carrying Activity* (*RCA* for short). Note that the RCA of a resource is in the OCA of the resource request operation that acquires the resource.
- **ID-params.** For a leaked resource request operation, a part of work of determining the target resource release operation that should be inserted is to figure out the arguments of the release operation. In fact, these arguments are all picked from the specific parameters (and the return-value) of the corresponding resource request operation. For a resource request operation, we call these specific parameters (and the return-value) *identity parameters* (*ID-params* for short). For each resource request operation, the ID-params can be figured out by Android reference. Note that resources are distinguished by the values of its ID-params.
- **FCG.** To discover the OCA of leaked resource request operations we construct the *Function Call Graph* (*FCG* for short) to assist inter-procedural analysis. An FCG is a directed graph where each node denotes a method that is defined by developers and each edge (*m, c*) denotes method *m* invokes method *c*.
- **AAG.** Since Android apps are usually driven by events and callbacks and asynchronous invocations usually occur in Android apps. So only the FCG is not enough to assist inter-procedural analysis in Android apps. To handle the asynchronous invocations, we construct the *Activity Asynchronous Graph* (*AAG* for short). An AAG is a directed bipartite graph. The head node of an edge in AAG is a callback method and the tail node of an edge is an Activity. Each edge (*a, c*) denotes the callback method *c* is indirectly invoked by Activity *a*.

III. Approach

In this section, we introduce an overview of our approach followed by the details of the proposed techniques.

A. Overview

Fig. 4 shows a high-level overview of our approach, which consists of several key modules. Graph construction module generates the FCG and AAG to support the OCA discovery module which tries to find the locations where the target resource release operations should be inserted; Resource identification module traces values of ID-params to figure out the operated resources; Unreleased resource list construction module maintains an unreleased resource list, according to which we setup guards to determine when the target resource release operations should be executed; Release operation insertion module finally patches the original Dalvik byte-code with the missing resource release operations.

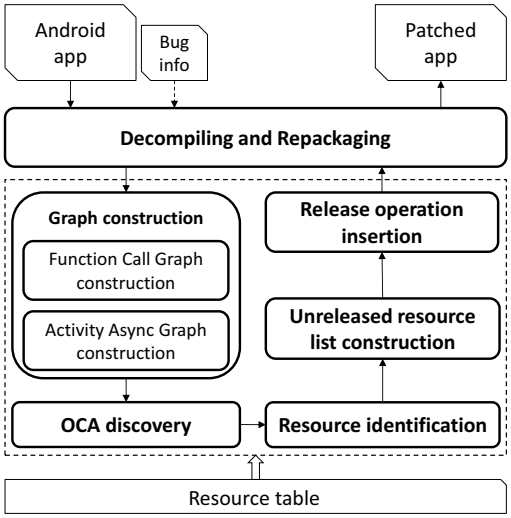

Fig. 4. High-level overview of our approach

We take an Android app with the bug report from `Relda2` as our inputs. A typical bug report usually contains the type and position of bugs. Specifically, for this resource leak issue, the bug report includes the suspicious resource request operations and the corresponding release operation, and their positions in the byte-code of the app. With the bug report, we can easily locate these resource related operations.

We save all the resource related operations, which the Android system contains, with some extra information in the *resource table* file that is used in the entire fixing procedures. Each line of the file represents a distinct resource request operation with the following three attributes: the corresponding resource release operation, the suggested release callback method and the ID-params. Note that the *resource table* is extendable and can be updated along with the Android system's upgrade.

The main functionality of each module is shown as follows.

Graph construction. Before the analysis, we statically construct the Function Call Graph (FCG) and the Activity Asynchronous Graph (AAG).

OCA Discovery. For each resource request operation that is reported to cause a resource leak, we need to insert the corresponding resource release operation into the Activities (OCA) that may invoke it to fix the leak. Therefore it is necessary to find out the OCA of all leaked resource request operations and we can achieve this goal by the FCG and AAG.

Resource identification. To determine the arguments that the inserted operation should accept, we need to keep track of the values of the ID-params. In our approach, for each resource related operation, we introduce some auxiliary variables to record the values of its ID-params.

Unreleased Resource list construction. To guarantee the fixing is safe, we should keep track of the resources that have been acquired but are not released. In our approach, we insert a piece of code for creating and maintaining a list to record all unreleased resources at runtime.

Release operation insertion. Finally, we insert the corresponding resource release operations under conditions for each reported resource request operation in the suggested callback method of OCA to complete the fixing. Based on the unreleased resource list, each condition is to determine whether the release operation should be executed.

In the following sections, we will discuss the technical details for byte-code analysis and instrumentation.

B. Graph Construction

Before starting fixing resource leak, we have done some necessary static analysis work that is light-weight and important for the following fixing procedures. The analysis work contains the FCG construction and AAG construction.

To construct the FCG, we first pick out all developer-defined methods that are the whole nodes of the FCG. For each method *m*, we collect all the invocation instructions in the method, and if the invoked method *c* is developer-defined we add (*m, c*) into the FCG. The FCG construction is completed when all methods have been dealt with.

To construct the AAG, we first collect all the frequently used registration operations and their corresponding callback methods in Android system, according to the Android reference and real-world apps. Then we scan whole code of the app to find all registration operations that the app invokes. For each registration operation, we get the method that directly invokes it and then track the method backwardly according to the FCG and the partial AAG that contains the asynchronous invocations we have handled until we find an Activity in each tracking path. After Activities tracking, for each Activity *a* tracked and each callback method *c* of the registration operation, we add (*a, c*) into the AAG. The AAG construction is completed when all the registration operations have been dealt with.

Fig. 5 shows the partial invocation information according to the FCG and AAG about the code snippet in Fig. 2. The solid lines and dash lines represent the edges in the FCG and AAG, respectively.

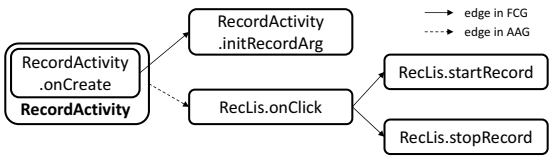

Fig. 5. Parts of graphs of SimpleRecorder

C. OCA Discovery

For each resource request operation, to locate where we should insert the corresponding resource release operation, we need to discover its OCA (see Section II-D) with the FCG and AAG. Algorithm 1 shows the details.

Algorithm 1 Discover OCA

Input: OP_{req}, G_{fcg}, G_{aag}, S_{act}
Output: S_{oca}

```
1:  m_i = the method directly invoking OP_req
2:  S_r = the root methods in G_fcg that can reach m_i
3:  for each method m_r in S_r do
4:    if m_r is a callback method of Activity c_a then
5:      add c_a to S_oca
6:    else if m_r in G_aag then
7:      for each Activity c_a that can reach m_r in G_aag do
8:        add c_a to S_oca
9:      end for
10:   end if
11: end for
12: if all Activities can not reach a method m_r ∈ S_r then
13:   for each Activity c_a in S_act do
14:     add c_a into S_oca
15:   end for
16: end if
```

In the algorithm, the input OP_{req} denotes the target resource request operation, G_{fcg} and G_{aag} denote the FCG and AAG constructed in Section III-B, S_{act} denotes the set of all Activities in the app and the output S_{oca} denotes the set OCA of the operation OP_{req}. For the resource request operation OP_{req}, we first find the method (variable m_i) that directly invokes it and the set of all root methods (variable S_r) in FCG that can reach m_i. For each method $m_r \in S_r$, if it is the lifecycle callback method, which is one of the methods `onCreate()`, `onStart()`, `onRestart()` and `onResume()`, of an Activity, we add the Activity into the result set S_{oca}. Otherwise, if m_r is a node in the AAG, we add all Activities that have edges to m_r in the AAG to the result set S_{oca}.

Considering that graphs constructed by our light-weight static analysis techniques may be incomplete, we may not be able to find an Activity that can reach a method $m_r \in S_r$. In this case (line 12-16), we make a conservative treatment for safe fixing by adding all Activities in the app to the result set S_{oca}.

Recall the example in Fig. 2 and its invocation information shown in Fig. 5, the method `startRecord()` directly invokes the resource request operation `AR.<init>()` (For convenience, we simply write `AudioRecord` as `AR` in the rest of the paper), and its root method in the FCG is the method `onClick()` of class `RecLis`. This `onClick()` method is not a callback method of all Activities, therefore we need search for this `onClick()` method in the AAG. The Activity `RecordActivity` has an edge directing to the `onClick()` method in the AAG, so we add `RecordActivity` into the OCA of the operation `AR.<init>()`.

D. Resource Identification

As mentioned in Section II-D, to determine the target resource release operation that we need to insert, not only do we need to find out the name of the release operation, but also the arguments that it accepts. In fact, these arguments are equal to the values of the ID-params of the corresponding leaked resource request operation. As a result, we need to identify the leaked resource request operation to figure out the values of all its ID-params. We introduce a set of variables to record the values of ID-params.

Recall the example shown in Fig. 2. When the resource request operation `AR.<init>()` is executed, a resource will be required. For this request operation, the corresponding resource release operation is `AR.release()` and the ID-params is the object it returns. As the returned object is stored in variable `mAR`, we can identify the required resource with the operation `AR.<init>()` and the set of the values of the ID-params $\{V_{mAR}\}$.

The identification information will be used to recognize resources and maintain the unreleased resource list built in the next section.

E. Unreleased Resource List Construction

As mentioned in Section II-C, we want to safely fix the resource leak problem. To guarantee the safety, we set guards in front of the inserted operations. The brief guard and flag put in Fig. 3 is not suitable for an automatic approach. In our approach, we introduce a global unreleased resource list, which is the testimony at all guards, to record the resources that have been acquired but are not released yet at runtime. This auxiliary list is declared as an `static` field of the launcher Activity. Each element in the list represents an unreleased resource r and it contains four attributes $\langle O_{req}, O_{rel}, A_{rc}, S_{IDv} \rangle$, where

O_{req} (Resource Request Operation) represents the name of resource request operation for the resource r, which we have obtained from the bug report of the static analysis tool `Relda2`.

O_{rel} (Resource Release Operation) represents the name of the corresponding release operation for the resource r, which can be obtained from the resource table file.

A_{rc} (Resource Carrying Activity, RCA) represents the Activity that is the unique active one when the resource r is acquired, in which the resource r should be released during the runtime. We make use of an Android API (`ActivityManager.getRunningTasks(1).get(0).topActivity`) to obtain the currently active Activity dynamically.

S_{IDv} represents the set of the values of all ID-params of the resource request operation that acquires the resource r.

We insert corresponding byte-code after each resource request and release operation to maintain the global unreleased resource list. The byte-code after a resource request operation adds the resource into the list if the resource has not been acquired by comparing the O_{req} and S_{IDv} of each resource in the list with the name and the values of ID-params of this resource request operation respectively. Similarly, the byte-code after a resource release operation removes the resource from the list if the resource has been acquired.

Let us recall the resource request operation `AR.<init>()` in Fig. 2. When the `mStartBt` button is clicked, the method `startRecord()` will be invoked and then the operation `AR.<init>()` will be executed. At this time, assume that the set of resources in the list is S_{ur}, for each resource r in S_{ur}, we compare `AR.<init>()` with O_{req} of the resource r, and if O_{req} of r is also `AR.<init>()` we then compare the values of ID-params $\{V_{mAR}\}$ with S_{IDv} of the resource r. If there is no unreleased resource of which O_{req} is `AR.<init>()` and S_{IDv} is equal to $\{V_{mAR}\}$, we add the newly acquired resource $\langle$ `AR.<init>()`, `AR.release()`, `RecordActivity`, $\{V_{mAR}\}$ $\rangle$ into the list. Besides, when the `mStopBt` button is clicked, we will remove the resource from the list.

F. Release Operation Insertion

The previous parts of this section pave the way for safe and concise release operations insertion. We find the OCA, for each reported resource request operation, that represents the Activities which possibly invoke the operation, and introduce a global unreleased resource list, which is the testimony at all guards, to record the resources that have been acquired but not released yet at runtime.

The key part about fixing resource leaks is inserting appropriate piece of release code in appropriate places. For each reported resource request operation, we need to insert a code snippet to fix the potential leaks into each Activity in OCA. Each code snippet corresponding to an Activity a in OCA is determined by a 4-tuple $\langle req, rel, act, met \rangle$, where req denotes the name of resource request operation; rel denotes the name of corresponding resource release operation; act denotes the name of Activity a; met denotes the name of suggested callback method to release the resource.

For a given 4-tuple $\langle req, rel, act, met \rangle$, we have the code snippet as shown in Fig. 6. To ease understanding, we express our idea in the form of Java code instead of the byte-code that we actually instrument.

```
class act{
 protected void met{
   ......
   for (int i = gList.size() - 1; i >= 0; --i){
    if (gList.get(i).get(2).equals(act)
         && gList.get(i).get(0).equals(req)){
      rel;
      gList.remove(i);
}}}}
```

Fig. 6. Form of the fixing code

The callback method $act.met$ is the location where we need to insert the resource release operation rel and the variable `gList` is the global unreleased resource list. The conditions of if-statement form the guard for this insertion location. Recall that each element in `gList` is also a list that has four elements ($\langle O_{req}, O_{rel}, A_{rc}, S_{IDv} \rangle$, see Section III-E). We

traverse gList to find the resources whose A_{rc} attribute (the 3rd element of gList[i]) is equal to act and O_{req} attribute (the 1st element of gList[i]) is equal to req, and these resources will be released by rel. Then we remove the released resources from gList.

Consider the motivating example shown in Fig. 2 again, the OCA of resource request operation AR.<init>() has one element RecordActivity, so there is one place where we insert a code snippet to fix the resource leak caused by the operation AR.<init>(), and for this insertion place, req is AR.<init>(), rel is AR.release(), act is RecordActivity and met is onDestroy().

IV. Effectiveness Analysis

In this section, we analyze our approach from two aspects: the safe fixing property and the number of inserted instructions.

A. Safe Fixing

Here, we briefly illustrate how our approach satisfies the safe fixing property (defined in Section II-C).

For each resource, if it is acquired, it will be added to the global resource list. When the Activity invoking the resource is inactive, our fixing code will check the global resource list and release this resource. So the first property holds.

If a resource is not acquired, it would not be added to the global resource list, so the inserted release operations will not release such an unrequested resource. The second property holds.

We address the last property from two aspects. First, for each acquired resource, once it is released, from either the original release operations or the inserted release operations, it will be removed from the global resource list. In addition, the inserted operations only release the resources in the global resource list. So the inserted release operations will not release a resource that has been released. Second, the release operations are inserted into the end of the lifecycle of an Activity, so the original release operations are always prior to them and will never release a resource that has been released by the inserted ones.

B. Complexity Estimation

In this section, let us estimate the upper bound on the number of instrumented instructions in our approach. For an Android app that contains C_{act} Activities, C_{req} reported resource request operations and C_{rel} related resource release operations, we give the estimated value as follows. Here the notations ω_i $(1 \leq i \leq 5)$ denote the coefficients.

- To identify resources, for each resource related operation, we will insert ω_1 instructions, so we insert $\omega_1(C_{req} + C_{rel})$ instructions in byte-code.
- To introduce the global unreleased resource list, we will insert ω_2 instructions in byte-code. To maintain the global unreleased resource list, for each resource request operation, we will insert ω_3 instructions, and for each resource release operation, we insert ω_4 instructions. So we insert $\omega_3 C_{req} + \omega_4 C_{rel}$ instructions in byte-code.
- To insert the target resource release operations, the worst occasion is that we need to insert the corresponding operation in every Activity for every reported resource request operation, therefore we insert at most $\omega_5 C_{act} C_{req}$ instructions in byte-code.

To sum up, the upper bound on the number of inserted instructions is $\omega_2+(\omega_1+\omega_3)C_{req}+(\omega_1+\omega_4)C_{rel}+\omega_5 C_{act}C_{req}$, where the values of coefficients ω_i $(1 \leq i \leq 5)$ depend on implementation of instrumentation. In our implementation, the values of these coefficients are $(\omega_1, \omega_2, \omega_3, \omega_4, \omega_5) = (6, 15, 112, 76, 80)$. The quadratic term $\omega_5 C_{act} C_{req}$ describes the worst case that we can not find the OCA in the FCG and AAG for every reported resource request operations. However our experiments show that this occasion rarely happens.

V. Evaluation

To evaluate the usefulness of our method, we developed an automated fixing tool called RelFix, and applied it to real Android apps. Specifically, RelFix is implemented on top of Apktool [2] that is a reverse engineering tool for Android apps. All code in RelFix is written in Python language, so it is easy to be deployed on different operating systems, such as Linux and Windows. We make use of RelFix to fix resource leaks on selected apps, and evaluate the patch overhead, effectiveness and safety of our approach. At last we report the execution time of our tool.

A. Experiments Setup

To evaluate the applicability of our approach, we developed a set of benchmarks, each of which is a small Android app and contains one specific type of resource leak bug. We list a part of the self-designed benchmarks in table I. The first column shows the resource request operation that the app uses. The second and third columns represent the locations of the resource request and release operations respectively. Among these apps, the first seven apps always leak the resources and the last three apps leak the resources under special conditions. Specifically, the eighth app needs the users to manually release the resource by clicking a Button; the ninth app will not release the resource when the Activity mode is set as SINGLE_INSTANCE at runtime; the last app will leak the resource when the condition of the if-statement evaluates to true. These conditional resource leaks should be fixed carefully with the elaborative guards.

In addition, we apply our tool to fix some real-world apps that have been reported with unreleased resources. We collect 427 real-world apps for our evaluation, where 403 apps are from famous Android markets and 24 apps are from an open-source Android community. We statically analyze these apps and find 165 apps (162 close-source and 3 open-source apps) are reported with the resource leaks. Then we manually check these suspicious apps and confirm 12 apps that indeed have resource leak problem. For each resource leak in confirmed app, we construct the corresponding event sequence that can

TABLE I
SOME SELF-DESIGNED BENCHMARKS

Apps	$Resource_{use}$	$Location_{req}$	$Location_{rel}$
1	Camera.open	Activity.onCreate	-
2	BluetoothAdapter.startDiscovery	Activity.onStart	-
3	SensorManager.registerListener	Activity.onResume	-
4	AudioManager.loadSoundEffects	OnClickListener.onClick	-
5	URL.openConnection	BroadcastReceiver.onReceive	-
6	VelocityTracker.obtain	SensorEvenListener.onSensorChanged	-
7	WakeLock.acquire	LocationListener.onLocationChanged	-
8	Chronometer.start	OnClickListener.onClick	OnClickListener.onClick
9	PresetReverb.setPreset	Activity.onCreate	Activity.onActivityResult
10	NfcAdapter.enableForegroundDispatch	Activity.onResume	under If-statements

trigger the bug. We have not confirmed all the reported resource leaks. Possible reasons include the false positive of static analysis and failing to explore all execution paths of the app. Table II shows the information of these apps, including the name, the size of the APK and DEX file (KB), the number of Dalvik byte-code instructions (*#Ins*), Activities (*#A*) and the information from the bug reports ($R_1/R_2/C$). The last column includes three items, where R_1 and R_2 denote the number of resource request and release operations in the bug reports respectively, and *C* denotes the number of resource leaks that we have confirmed.

TABLE II
INFORMATION OF BUGGY APPS

Apps	S_{apk}/S_{dex}	*#Ins*	*#A*	$R_1/R_2/C$
Bluechat	62.5/38.3	4218	3	1/0/1
GetBackGPS	249.5/104.3	13177	4	5/5/4
FooCam	290.8/574.1	11366	1	1/1/1
ErWeiMaL	2025/1443	92329	3	1/0/1
QiCaiScan	1176/1192	228205	25	4/2/1
CheDengWo	3949/1240	200493	5	1/0/1
WebPCSuite	3380/2764	321713	23	2/2/1
FromCat	2112/1023	92236	3	12/0/12
MaMa	1077/576.0	95730	4	1/0/1
XiaoMiWiFi	859.8/488.6	56554	8	1/0/1
SuperTorch	1062/1264	177185	12	3/2/1
FontMaster	5400/932.4	68729	20	1/0/1

B. Overhead of Patches

Our fixing strategy is inserting additional instructions into Dalvik byte-code instead of modifying or deleting instructions. The additional byte-code will inevitably bring overhead to the target app. We first make use of the self-designed benchmarks to study the size of patch for specific types of resource leak bug patterns. Table III lists the number of inserted instructions ($\#I_i$) and the upper bound value ($\#I_e$).

From Table III, we can see that for each app in self-designed benchmarks, the number of inserted instructions by `RelFix` conforms to the estimation value. Since all the instances in self-designed benchmarks are small scale programs, we also apply our tool on real-world apps and list the results in Table IV. The second column shows the reported resource request operations. The third to fifth columns list the increments of instructions, where the third and fourth columns list the number of instructions before and after instrumentations respectively, and the fifth column shows the relative increments per reported resource request operation. Similarly, the last three columns list the increments of DEX file size. Different from the app in self-designed benchmarks that contains one Activity, and only one resource request operation with at most one corresponding resource release operation, the real-world apps are more complicated that for an item in bug report, `RelFix` may insert byte-code into multiple places.

TABLE III
OVERHEAD OF INSTRUMENTATION FOR SELF-DESIGNED BENCHMARKS

Apps	C_{act}	C_{req}	C_{rel}	$\#I_i$	$\#I_e$
1	1	1	0	170	213
2	1	1	0	169	213
3	1	3	0	577	609
4	1	1	0	170	213
5	1	1	0	169	213
6	1	1	0	173	213
7	1	1	0	170	213
8	1	1	1	222	295
9	2	1	1	221	375
10	1	1	1	258	295

For these apps, `RelFix` inserted less than 4% of instructions per item in the report and as a result, the size of DEX file increased no more than 4%. In general, a buggy real-world Android app has a small number of reported resource request operations, thus the instrumentation overhead is acceptable.

For a fixing work, overhead should mainly be evaluated as the actually added execution time of the fixed apps over the original apps. We make use of an Android testing framework `Robotium` [3] to drive the apps to run under a given test sequence for measuring the execution time of the apps. Our experiments show that there is no significant difference in execution time between the apps with and without our patch. For example, the average time of 10 executions of `Bluechat` before and after patching under the given test sequence is 5.881s and 5.894s respectively. The added time is 0.013s that can be regarded as extra execution time introduced by our patch. However, the standard deviation of observed values is 0.154s that is far more than 0.013s, which implies that the runtime overhead caused by our patch can be negligible.

TABLE IV
OVERHEAD OF INSTRUMENTATION FOR REAL-WORLD APPS

Apps	C_{req}	*#Instructions*			*DEX Size*		
		BP	*AP*	$O_{avg}(\%)$	*BP*	*AP*	$O_{avg}(\%)$
Bluechat	1	4218	4380	3.8	38.3	39.8	3.9
GetBackGPS	5	13177	15320	3.2	104.3	115.0	2.0
FooCam	1	11366	11818	3.9	574.1	576.9	0.4
ErWeiMaL	1	92329	92494	0.2	1443	1444	0.1
QiCaiScan	4	228205	233168	0.5	1192	1202	0.2
CheDengWo	1	200493	200655	0.1	1240	1241	0.1
WebPCSuite	2	321713	323619	0.3	2764	2769	0.1
FromCat	12	92236	93691	0.1	1023	1030	0.1
MaMa	1	95730	95895	0.1	576.0	577.3	0.2
XiaoMiWiFi	1	56554	57178	1.1	488.6	492.3	0.7
SuperTorch	3	177185	179103	0.3	1264	1265	0.1
FontMaster	1	68729	70243	2.1	932.4	940.3	0.8

C. Effectiveness and Safety

A straightforward way to evaluate whether we have fixed the reported resource leaks is to employ `Relda2` to scan the fixed code again. However, the static analysis may contain false positives and false negatives, therefore this is not a reasonable validation approach and we choose another dynamic way to check the effectiveness. Specifically, we add extra instructions to profile the resource operations at runtime into log file and then run these patched apps under the event sequences that can trigger leaks of original apps. Finally, we review the log information and find that each resource request operation is followed by a matching release operation.

We also check that our patches do not crash a program via dynamical way. Here, we make use of `Monkey`, which is an automatic test generation tool provided by Android system, to randomly generate event sequences with its default option that includes all events it can generate and run all patched apps. We set the number of event sequences for `Monkey` as 10,000 with a delay time 100ms between events. At last, all instances pass the test.

D. Execution Time

Table V shows the execution time of `RelFix`. The second column shows the time (seconds) spent in building the FCG and AAG. The third column shows the time (seconds) spent in inserting target instructions, and the last column presents the whole time (seconds) to fix the apps. Our experiments were carried out on a desktop with Intel Core i3-4170 3.7GHz and 4GB memory in Ubuntu 14.04.

According to the results, the execution time of our tool for each instance is less than one second and building graph costs most of the time. In summary, the overall time is acceptable.

TABLE V
EXECUTION TIME

Apps	*Graph Build*	*Leak Fix*	*Total*
Bluechat	0.026	0.016	0.042
GetBackGPS	0.083	0.009	0.092
FooCam	0.164	0.070	0.234
ErWeiMaL	0.647	0.012	0.659
QiCaiScan	0.590	0.019	0.609
CheDengWo	0.760	0.034	0.794
WebPCSuite	0.535	0.004	0.539
FromCat	0.681	0.046	0.727
MaMa	0.515	0.015	0.530
XiaoMiWiFi	0.347	0.021	0.368
SuperTorch	0.976	0.007	0.983
FontMaster	0.637	0.013	0.650

VI. DISCUSSION

In this section, we will discuss some issues related to the realization of our approach.

A. Fixing Strategy

There are several strategies for fixing the resource leaks. The most conservative way is to release all leaked resources when the app exits, i.e., inserting all resource release operations at the end of the callback method `onDestroy()` of the Main Activity. This approach can guarantee the safety with a very low overhead. However, it has not resolved the adverse effects including high energy and memory consumption caused by resource leaks, since it releases the leaked resources too late and the app still occupies the resources for a long time.

The main idea of our strategy is to release leaked resources as early as possible with consideration of safety. Compared with the most conservative strategy, our approach increases a little but acceptable instrumentation overhead.

B. Intention of Developers

For some resource request operations, their corresponding release operations should be put in `onPause()` or `onStop()` method according to Android reference. A specific scenario is that the developers want to occupy the resource during the visible states of Activity lifecycle, but they implement it in an improper way that the resource is neither released in `onPause()` or `onStop()` method nor required in `onResume()` or `onStart()` method. We need to insert extra resource request operation in `onResume()` or `onStart()` method to fulfill the developers' intention. However, the intention of the developers usually can not be inferred from the code, thus we just report some warnings after the fixing.

C. Threat to Validity

There are some practical threats to validity. First, we employ `Apktool` to decompile and repackage the APK file, but it may not work on the APKs that have been put under some protections by the developers. Second, our approach does not support all the characteristics of the Android system, such as multi-thread and native code. Third, we make use of the resource table based on Android 5.0 and it may be incomplete when the system is updated and introduces new resources.

VII. Related Work

Bug fixing is a hot topic of software engineering in recent years. In this section, we discuss some related works on bug fixing. In addition, since we focus on modifying Android apps, we also include some works on instrumentation techniques in this section.

Generic Bug Fixing. Recently, some works [8], [16], [22] focus on general bug fixing to automatically repair general types of bugs in programs. These approaches try to mutate the source code that violate correctness and check whether the condition is satisfied from either test cases or assertions. However, these types of approaches face some fundamental difficulty [10]. First, limited test cases or assertions are usually inadequate to ensure the correctness of modifications. Second, for a real program, the search space is often very large and it may take a long time to analyze and repair the program. To reduce the search space, some new techniques are proposed, such as Ke *et al.* [14] presented an approach to fixing programs by searching the potential patches with SMT solvers. There are also some approaches that design some bug templates to guide the mutations. Zhong el al. [26] conducts an empirical study on thousands of real-world bugs to pick out some findings and insights and summarize them into templates. Kim *et al.* [15] proposed a novel patch generation technique based on the templates that are learned from dozens of thousands of human-written patches of open source projects.

Specific Bug Fixing. There are some dedicated approaches to fix a specific type of bugs in C and Java programs. Gao *et al.* [10] proposed an approach to automatically fixing memory leak problems in C programs. They perform a precise static analysis to trace pointers and find release locations. Nistor *et al.* [21] did works on program performance and presented a novel static technique to detect and fix performance bugs that have non-intrusive fixes likely to be adopted by developers. There are also other types of bugs to be fixed, such as buffer overflow in C programs [23]. Monperrus *et al.* [9], [20] have presented a series of works on fixing specific bugs, by employing SMT technique to check whether a change to program is suitable and generate the proper patch.

A few works have been done on the Android system. Lin *et al.* [18] aimed at performance problem in Android apps and developed an automated refactoring tool that enables developers to extract long-running operations into AsyncTask by modifying Android source code. This approach makes use of a pure static technique and does not need dynamic trace. Zhang and Yin [25] proposed a technique for automatically generating patches to prevent component hijacking attacks in Android apps, and completed analysis and fixing procedures on Jimple IR that is translated from Dalvik byte-code using dex2jar [6] and Soot [5]. Their main idea is to track the propagation of private information and the techniques are similar to dynamic taint analysis, which may cause a high instrumentation overhead.

There are some other works which address characteristics that our work does not refer to, such as concurrency bugs. Jin *et al.* [12], [13] came up with an approach to automating the whole process of fixing a variety of concurrency bugs. Liu *et al.* [19] presented a context-aware algorithm to fix concurrency bugs while guaranteeing the deadlock freedom. These techniques may be useful for fixing concurrency bugs in Android system.

Instrumentation on Android apps. The instrumentation technique is widely used for program analysis. Li *et al.* [17] proposed a static taint analyzer to detect privacy leaks among components in Android apps via inserting instructions into Jimple IR. Some approaches also use the instrumentation technique to automatically fix bugs in Android apps, such as the work of Zhang and Yin [25], which prevents component hijacking attacks in Android apps. They also instrument on the Jimple IR. To our best knowledge, there is no approach that directly instruments on Dalvik byte-code to fix bugs in Android apps and we are the first ones to make this attempt.

VIII. Conclusion

In this paper, we propose an approach to automatically fixing resource leaks in Android apps. We give the definition of safe fixing, under which we insert necessary auxiliary variables and resource release instructions as well as some conditions to fix resource leaks that are reported by our static analysis tool `Relda2`. These instructions, with the help of the guard conditions, can effectively recognize the unreleased resources and safely release them at runtime. This approach comes with a light-weight static analysis that makes the whole patching process be efficient. Our experiments on real-world apps show that our approach can effectively fix the real resource leaks in apps with low instrumentation overhead in a few seconds. This work indicates that fixing on Android byte-code for specific types of bugs is a valuable attempt to decrease the labor-intensive debugging for developers and testers.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions.

This work is supported in part by National Natural Science Foundation of China (Grant No. 91418206) and the National Key Basic Research (973) Program of China (Grant No. 2014CB340701).

References

[1] Activity lifetime. http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle.

[2] Apktool - a tool for reverse engineering. http://ibotpeaches.github.io/Apktool/.

[3] Google code. Robotium. http://code.google.com/p/robotium/.
[4] Package index | Android developer ˙ http://developer.android.com/reference/packages.html.
[5] Soot - a framework for analyzing and transforming Java and Android applications. http://www.bodden.de/2008/09/22/soot-intra.
[6] Tools to work with Android .dex and java .class files - google project hosting. http://code.google.com/p/dex2jar/.
[7] UI/application exerciser monkey. http://developer.android.com/tools/help/monkey.html.
[8] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2008*, pages 162–168, 2008.
[9] F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA 2014*, pages 30–39, 2014.
[10] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for C programs. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, pages 459–470, 2015.
[11] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and detecting resource leaks in Android applications. In *28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*, pages 389–398, 2013.
[12] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 389–400, 2011.
[13] G. Jin, W. Zhang, and D. Deng. Automated concurrency-bug fixing. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*, pages 221–236, 2012.
[14] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 295–306, 2015.
[15] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering, ICSE 2013*, pages 802–811, 2013.
[16] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.
[17] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, pages 280–291, 2015.
[18] Y. Lin, C. Radoi, and D. Dig. Retrofitting concurrency for Android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*, pages 341–352, 2014.
[19] P. Liu, O. Tripp, and C. Zhang. Grail: context-aware fixing of concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*, pages 318–329, 2014.
[20] S. R. L. Marcote and M. Monperrus. Automatic repair of infinite loops. *CoRR*, abs/1504.05078, 2015.
[21] A. Nistor, P. Chang, C. Radoi, and S. Lu. CARAMEL: detecting and fixing performance problems that have non-intrusive fixes. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, pages 902–912, 2015.
[22] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Trans. Software Eng.*, 40(5):427–449, 2014.
[23] A. Shaw. Program transformations to fix C buffer overflows. In *36th International Conference on Software Engineering, ICSE 2014*, pages 733–735, 2014.
[24] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. *IEEE Transactions on Software Engineering*, 2016.
[25] M. Zhang and H. Yin. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014*, 2014.
[26] H. Zhong and Z. Su. An empirical study on real bug fixes. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, pages 913–923, 2015.