Detecting Energy Bugs in Android Apps Using Static Analysis

Hao Jiang[1], Hongli Yang[1], Shengchao Qin[2], Zhendong Su[3], Jian Zhang[4] and Jun Yan[4]

[1]Beijing University of Technology [2]Teesside University [3]University of California, Davis
[4]Institute of Software Chinese Academy of Sciences

Abstract. Energy bugs in Android apps are defects that can make Android systems waste much energy as a whole. Energy bugs detection in Android apps has become an important issue since smartphones usually operate on a limited amount of battery capacity and the existence of energy bugs may lead to serious drain in the battery power. This paper focuses on detecting two types of energy bugs, namely resource leak and layout defect, in Android apps. A resource leak is a particular type of energy wasting phenomena where an app does not release its acquired resources such as a sensor and GPS. A layout defect refers to a poor layout structure causing more energy consumption for measuring and drawing the layout. In this paper, we present a static analysis technique called SAAD, that can automatically detect energy bugs in a context-sensitive way. SAAD detects the energy bugs by taking an inter-procedural anaysis of an app. For resource leak analysis, SAAD decompiles APK file into Dalvik bytecodes files and then performs resource leak analysis by taking components call relationship analysis, inter-procedure and intra-procedure analysis. For detecting layout defect, SAAD firstly employs $Lint$ to perform some traditional app analysis, then filters energy defects from reported issues. Our experimental result on 64 publicly-available Android apps shows that SAAD can detect energy bugs effectively. The accuracies of detecting resource leak and layout energy defect are 86.67% and 76.27% respectively.

1 Introduction

With the rapid development of mobile technology, smartphones, especially Android phones, provide people with convenient services. Android application markets like Google Play provide abundant apps for users. In order to enrich the user experience, Android systems are equipped with a wide range of hardware components, such as Sensors, WIFI, GPS, Camera and so on. Because of rich functionalities and convenient services, a majority of developers are attracted to develop apps on Android platforms.

Meanwhile, the usage time of a smartphone is constrained by its battery capacity. Since the existing techniques have not yet allowed the smartphones to be charged anywhere, and at anytime, services and functions will be constrained, and even forced to close. And as a consequence, the battery energy has great impacts on user experiences. The battery energy is mostly consumed by apps installed in smartphones, as the service and some resource intensive hardware components (such as screen, GPS, WIFI, and CPU) are usually invoked when apps are running [2].

Typical energy bugs [1, 3] that may be hidden in smartphone apps can be classified into either resource leak or layout defect. A resource leak refers to a case that an app does not release acquired resources such as sensor, GPS, wakelock, memory etc., and thus may hinder system from entering an idle state, making hardware reside at a continuous energy consumption situation. A layout defect can be caused by a poor layout structure (layout is too deep, too many or ineffective widgets, etc.) which leads to high energy consumption for measuring and drawing of this layout. Both types of bugs can result in unnecessary energy consumption.

There are some work related with energy bugs detection. However, they focus on detecting either background programs or foreground ones such as user interfaces. Comparatively, we mainly focus on detecting resource leaks and layout defects, which are more latent to the users.

This paper makes the following contributions:

- We propose a novel approach, called *SAAD* (Static Application Analysis Detector). It analyzes not only resource leak at background programs, but also layout defects at foreground. The generated reports of energy bugs help analyzers and developers improve their apps.
- *SAAD* detects resource leak by context sensitive analysis, which combines component call analysis, inter-procedural analysis and intra-procedural analysis. It considers the calling context when analyzing the target of a function call. In order to improve efficiency, It focuses on analyzing effective paths that are involved in resource applying/releasing operations. In particular, *SAAD* can detect more than eighty resources leak by automatically getting those resources API information from Android official website by Web crawler.
- We have implemented a tool to support *SAAD* and evaluated it on 64 free-available Android applications. Our results show that *SAAD* can detect energy bugs effectively. The accuracies of detecting resource leak and layout energy defect are 86.67% and 76.27%, respectively.

The rest of the paper is organised as follows. Section 2 introduces background about the classifications of resource leak and layout defect. Section 3 gives an overview of our energy bugs detection framework. Section 4 presents analysis approach of resource leak and layout defect. Section 5 demonstrates experimental results of 64 real practical Android apps for evaluating our approach *SAAD*. Section 6 presents related work, while Section 7 concludes.

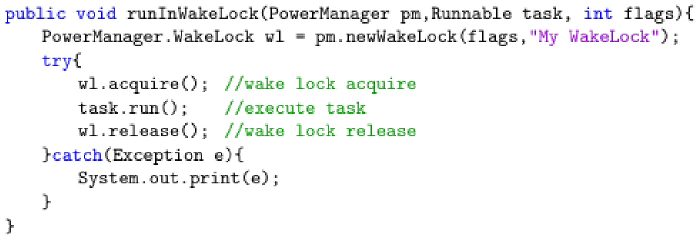
2 Background

2.1 Resource Leak Classification

Some typical energy bugs due to resource leak are listed as follows:

- Non sleep bug: An app applies a wakelock object to keep CPU and Screen to reside in an active state, and does not release the object in time. It results in that the CPU and the LCD component cannot enter a dormant state with sustainable energy consumption. An example of non sleep bug is illustrated in Figure 1. A *Wakelock* resource is applied in the *try* block, then after running a task, this resource is released using *release*() method. However, if an exception takes place when the task runs, the execution of the method *runInWakeLock*() will throw exceptions and enter the *catch* block. It means the release operation cannot be completed at the end, causing a resource leak.

- Sensor leak: Sensors (e.g., pressure sensors, direction sensors) are acquired, while the sensor object may not be released when the Android system enter its background state, making sensors stay active.
- Camera leak: The camera resource is occupied during an app's execution process, but fails to be released when the app switches into the background, leaving the camera driver stay in active state. In particular, since a camera resource in a smartphone is usually exclusive, if it is not released, other apps may not be able to access.
- Multimedia leak: A media player object or an audio manager object may be acquired by apps to play video or audio files. However, the corresponding object resource may not be released when an app enters its background state, and the leak makes the devices work continuously.
- Memory leak: The running system continues to allocate memory space for apps. Because of negligence or errors, it fails in releasing the corresponding memory space when apps are closed.



```
public void runInWakeLock(PowerManager pm,Runnable task, int flags){
    PowerManager.WakeLock wl = pm.newWakeLock(flags,"My WakeLock");
    try{
        wl.acquire();  //wake lock acquire
        task.run();    //execute task
        wl.release();  //wake lock release
    }catch(Exception e){
        System.out.print(e);
    }
}
```

Fig. 1. An Example of Non Sleep Energy Bug

Resource leaks are not limited to the above cases only. We use Web crawler to explorer as many resources as possible according to characteristics (such as their operation names containing key words *open/release*, *start/stop* and *register/unregister*). We get more than eighty resources API information, and store them in a configuration file to support energy bugs detection.

2.2 Layout Defect Classification

Unlike resource leaks, layout defects are mainly about bad designs of the layout structure, which may cause more unnecessary CPU time or memory spaces. Traditionally, each layout file uses an XML format to define and manage widgets. It is composed of several *View* objects and *ViewGroup* objects, organized in a tree hierarchy. A layout can be nested and referenced to other sub layout files. Under normal circumstances, each activity component is associated with a specific layout file. When an activity starts, it will load its layout file by invoking the *setContentView*() method. After finishing the steps of reading, parsing and measuring, the corresponding widgets in a layout are arranged to a coordinate position and the system begins to render and show them on the screen. As the number of widgets becomes bigger, or nesting level becomes deeper, the complexity of the layout file can be high, requiring much resources like CPU and memory space to be consumed. Compared with resource leak, layout defect may not increase energy consumption very obviously, and thus are often less concerned by developers, but it is surely a problem to energy inefficiency. The typical classification of layout defects are:

- Too many views: in a layout file, the default maximum widget number is 80 by default. When the number is greater than the default value, the system's running fluency can be decreased.
- Too deep layout: the default maximum nesting depth is 10. Similarly, the system may not run fluently when the depth is more than the default value.
- Compound drawables: it implies that a pair of widgets defined in a layout file can be replaced by one compound widget, such as a combination of an *ImageView* and a *TextView* can always be replaced by a *TextView* using a compound drawable.
- Useless leaf: if a widget does not have a child node or does not have a set of background properties, it is treated as a useless leaf node. A useless leaf node can be removed in order to reduce the complexity of the layout structure.
- Useless parent: if a widget has only one child node, and it is not a *scrollview* or a root node. Without the background properties, a useless parent can be removed so that the child node moves to its position.

The layout defects are common, and can also raise the complexity of layout structures, while they are less researched.

3 FRAMEWORK OVERVIEW

An overview of our bug-detection framework is shown in Figure 2. The input of the framework is an *APK* file, the outputs comprise a resource leak report and a layout defect report, and the modules of the dashed box perform energy bugs analysis and detection, using *Apktool* [4], *SAAF* [6], *Lint* [5], resource leak analysis and layout defect analysis, and a report generator.

The *Apktool* is a reverse-engineering tool that decompiles an *Apk* file to generate a manifest configuration file, several bytecode files named *Dalvik* [7] bytecode and layout files of an app. Generally, the components of an app are defined in its manifest file.

The *SAAF* is an open sourced static Android analysis framework which makes use of program slicing and static analysis techniques to uncover any suspicious behaviors. In the analysis, *SAAF* parses *Dalvik* bytecodes generated by *Apktool* and encapsulates them into data models provided by itself. There are different models such as the *Instruction* model, the *BasicBlock* model, the *Method* model, the *SmaliClass* model and so on. For example, a *SmaliClass* model encapsulates the current class's information including its name and method list, path and the type of its super class. *SAAF* also provides available APIs to retrieve such information.

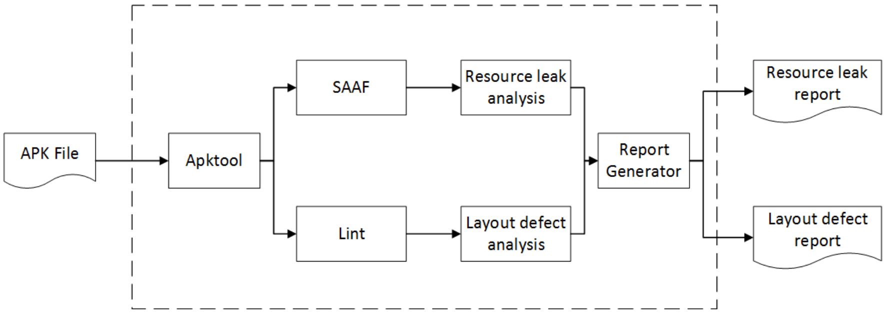


Fig. 2. The Framework of Energy Bug Detection

Lint is a static analysis tool for Android project source code, which detects potential bugs in the project and performs corresponding optimizations. The input of *Lint* contains two parts, the Android project source files (including java source files, configuration files, layout files and others), and an XML file named *lint.xml*, that defines severity levels of problems. *Lint* will detect performance problems of the code structure. For any problems detected, *Lint* gives an analysis report, and developers can fix these problems before releasing the apps.

The resource leak analysis module and the layout defect analysis module are the core parts in the framework. The resource leak analysis module judges whether the resource leak problems exist, and the layout defect analysis module further analyzes defects related to energy consumption based on the output of the *Lint* tool. We will present the two modules in Section 4 in detail.

4 ANALYSIS

4.1 Resource Leak Analysis

This section introduces resource leak analysis module in Fig. 2. It performs component call analysis, inter-procedural analysis, intra-procedural analysis and resource leak detection.

Components Call Analysis Usually, each app is composed of multiple components declared in its Manifest file. Each component can invoke methods such as *startActivity*() and *startService*() to call another component. The components relationship can be abstracted into a component call graph. In order to build the component call relationship, we find out an app's entry point, which is usually an activity targeted with *Android.intent.action.MAIN*. Then we search for *intent* objects, which are data objects for recording data that needs to be transmitted. A target component is defined as a parameter in an *intent* object.

After building the component call graph, we can extract a set of component call paths from the graph. Our framework can analyze each path whether there exists a resource leak or not.

An example component call graph is shown in Fig. 3. Here each node represents a component, and each arrow stands for the call relationship between the corresponding components.

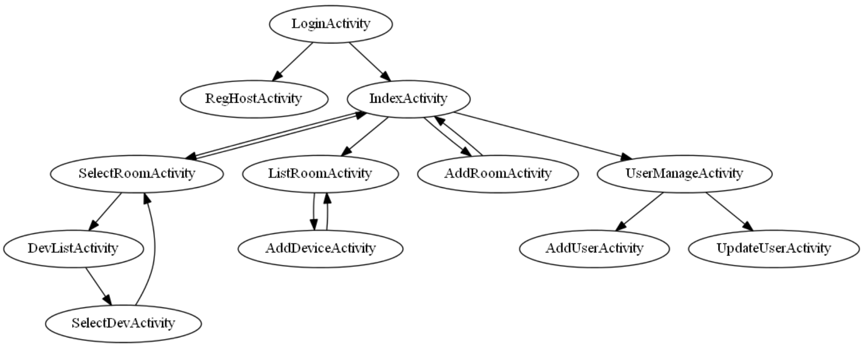


Fig. 3. An Example of Component Call Graph

Fig. 4 shows algorithm $generateCCGPaths$ for extracting component call paths. The input is a list of components cp_list. The result $result\_set$ stores a set of component call paths. Lines 3-6 traverse list of components and find out entry component, which is the first node of each path. Then sub-function $ccgTraverse$ is called to traverse the component call graph.

```
generateCCPaths($cp\_list$)

1  $create\ a\ list\ as\ path\_list$
2  $create\ a\ set\ as\ result\_set$
3  foreach $cp\ in\ cp\_list$ do
4    if $cp\ is\ EntryComponent$ then
5      $path\_list.add(cp)$
6      $ccgTraverse(cp, path\_list, result\_set)$
7  return $result\_set$
```

Fig. 4. Generating Component Call Paths Algorithm

The algorithm $ccgTraverse$ in Fig. 5 takes component cp, $path\_list$ and $result\_set$ as parameters. Lines 1-2 extract one path and add it to the result set if cp has been visited and its calling target component list $cp.targetList$ is empty. Lines 3-7 traverse $cp.targetList$. Each target component is added into path list if it is not visited, and recursively traversed by calling $ccgTraverse$. Lines 8-9 process pop stack operations, which delete the last node of current path in order to traverse other target components of its source component.

```
ccgTraverse $(cp, path\_list, result\_set)$

if $cp\ is\ visited \wedge cp.targetList\ is\ Empty$ then
  $result\_set.add(path\_list)$
foreach $target\ in\ cp.targetList$ do
    if $target\ is\ not\ visited$ then
        $target.visited \leftarrow true$
        $path\_list.add(target)$
        $ccgTraverse(target, path\_list, result\_set)$
        $path\_list.remove(path\_list.size - 1)$
        $target.visited \leftarrow false$
return $result\_set$
```

Fig. 5. Traversing Component Call Graph Algorithm

Inter-Procedural Analysis Based on each component call path, we analyze each component in its own life cycle, taking into account its inter-procedural information such as the function call relations, in order to understand the comprehensive behavior and status of an app. Particularly, we explorer the function call path related with resource applying and releasing.

a) Resource APIs

Resource APIs are used for deciding whether a function call path is involved in applying or releasing system resource. Android resource APIs have been published as webpages at its official website. We use the Web crawler technique to automatically extract more than eighty resource APIs including bluetooth, wifi, camera, multimedia, GPS, sensor, memory etc. Each resource API is defined with both apply and release methods information such as class path, method name, parameter list and return type.

b) Function Abstraction

In order to build function call relationships, we perform function abstraction for simplifying function analysis. A function abstraction is a semantic abstraction of a function, which includes the name of a function, the class it belongs to, the parameter list, the type of its return value and an invoked functions list. It saves an XML format for further processing. We use function abstractions to construct function call relationship.

c) Effective Path

Before detecting resource leak, our framework filters function call paths obtained from step b), and only analyzes effective paths where resources are acquired or released. This preprocessing decreases the number of paths to be analyzed, making the analysis more efficient. Fig. 6 is an example of a function call graph, in which three paths can be extracted.

1. *onCreate*() → *init*() → *setContentView*()
2. *onCreate*() → *doCameraOpen*() → *open*()

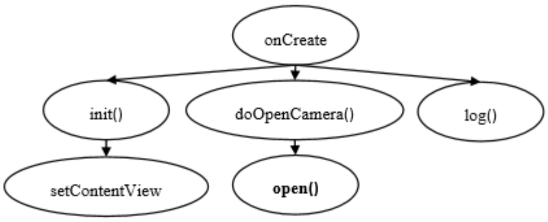


Fig. 6. An Example of A Function Call Graph

3. *onCreate*() → *log*()

However, only path 2 contains an instruction for opening a camera. Here the function *open*() is an API for opening camera resource. It is invoked to use the camera device. So this path is an effective path while the others are omitted in analysis. The following is the definition of effective path.

Definition 1. *An effective path is a 4-tuple* ⟨*com,path,res,op*⟩*. Here* com *is the current component,* $path$ *is a list of methods.* res *represents used system resource on this* $path$ *and* op *denotes the operation on corresponding resources. There are two kinds of resource operations, which is either* $apply$ *or* $release$.

```
extractEftPath $(cp, m, path\_list, result\_set)$

$path\_list.add(m)$
if $m.hasResource()$ is not null then
    create a path as $eftPath$
    $eftPath.com \leftarrow cp$
    $eftPath.path \leftarrow path\_list$
    $eftPath.res \leftarrow m.hasResource()$
    $result\_set.add(eftPath)$
if $m.abList$ is empty then
    $path\_list.remove(size - 1)$
else foreach $ab$ in $ablist$ do
    $nextMethod \leftarrow getMethod(ab)$
    $extractEftPath(nextMethod, path\_list, result\_set)$
    $path\_list.remove(size - 1)$
return $result\_set$
```

Fig. 7. Extracting Effective Paths Algorithm

Fig. 7 shows the algorithm $extractEftPath$, which takes three parameters: the current method m, a list of methods $path\_list$ and a set of effective paths $result\_set$.

Line 1 adds the current method m into list $path\_list$, and line 2 checks if current method has resource operation. If it has, lines 3-7 create an effective path $eftPath$, and sets its corresponding component, path and used resource, and add it to the result set. Otherwise lines 8-13 recursively traverse the next invoked method of m. Here the function $m.hasResource$ will check whether method m invokes resource APIs and returns resource type and operation in case of invoking. The function $getMethod(ab)$ takes the function ab as a parameter and returns the corresponding method.

d) Event Response and Callback Functions

Android apps are usually event driven. When an event response function calls a resource related API, it will be included in the corresponding effective path. Considering button events, our framework builds a hash table for mapping button objects into their monitoring objects. Thus it is easy to find out the event response functions defined in the monitoring class.

The callback mechanism is popular in Android system. For instance, the $Activity$ component's life cycle functions $onCreate$, $onStart$ etc. are system callback functions, which are automatically invoked by the system. A common example is the $Thread$ class. When a thread object executes the $start$ function, it actually executes the run function. However, the relation between $start$ and run functions are implicit. This situation causes some function call paths break in analysis. Our framework firstly tries to build a map between callback functions and real executed functions, and adds a callback function to the corresponding function call path.

Intra-Procedural Analysis The aim of the intra-procedural analysis is to analyze a single function. Based on basic blocks of a function, we build the control flow graph of the function. Our framework employs *SAAF* to generate a control flow graph of a function, and further extract a set of execution paths. The details are omitted here due to the page limitation.

Resource Leak Detection By combining the above analyses, Fig. 8 provides our algorithm for resource leak checking. The input $cpPath\_set$ is a set of component call paths obtained by component analysis. For each effective path $eftPath$ of the component cp, lines 4-6 add the resource into $apply\_list$ if the operation of the current path is an $apply$ operation. For $release$ operation, lines 7-16 traverse each method $method$ on effective path and make sure that: (1) its control flow paths $method.cfg$ must call the next method on the same path before getting to the last method; (2) when traversing the last method on the path, each of its control flow path must release the corresponding resource. If both conditions are satisfied, the resource res of current effective path is added into $release\_list$. Lines 17-19 compare $apply\_list$ and $release\_list$, and if they are matched, return $false$ for no release leak. Otherwise the algorithm returns $true$ for exiting release leak.

```
checkResourceLeak ($cpPath\_set$)

create an $apply\_list$ and a $release\_list$
foreach $cpPath$ in $cpPath\_set$ do
  foreach $cp$ in $cpPath$ do
    foreach $eftPath$ in $cp.getEftPath()$ do
      if $eftPath.op$ is apply then
        $apply\_list.add(eftPath.res)$
      else if $eftPath.op$ is release then
        for $i = 0;\ i \leq\ eftPath.size - 1;\ i++$ do
          $method \leftarrow eftPath.getMethod(i)$
          if $i \leq eftPath.size - 2$ then
            $next\_method \leftarrow eftPath.getMethod(i+1)$
            if $method.cfg$ do not invoke $next\_method$ then
              break
          else
            if $method.cfg$ do release then
              $release\_list.add(eftPath.res)$
if $apply\_list$ equals with $release\_list$ then
   return $false$
else return $true$
```

Fig. 8. Checking Resource Leak Algorithm

4.2 Layout Defect Detection

Fig. 9 shows the process of our layout defect analysis. The input is an *APK* file that needs to be decompiled by *Apktool*, and the output is a defect report. The analysis module and the filter module are explained as follows.

Analysis Module The analysis module mainly conducts an overall analysis of the layout files, including correctness, security, performance, usability and accessibility analysis. After receiving the paths of the layout files, that come from the result of decompiling of the *APK* file, the *Control Procedure* starts *Lint* to execute the layout file analysis. Finally, *Lint* will output an *XML* report about the issues for each layout file.

Filter Module Since the output of the analysis module includes different types of layout issues, which may or may not be energy consumption related layout defects. The Filter module extracts energy defects from the issues report of *Lint*. It is composed of a *Defect Table* and a *Filter Procedure*: the former acts as a set of filter rules, which are identified from layout defect classification in section 2.2, and the latter uses filter rules to find out layout energy defects.

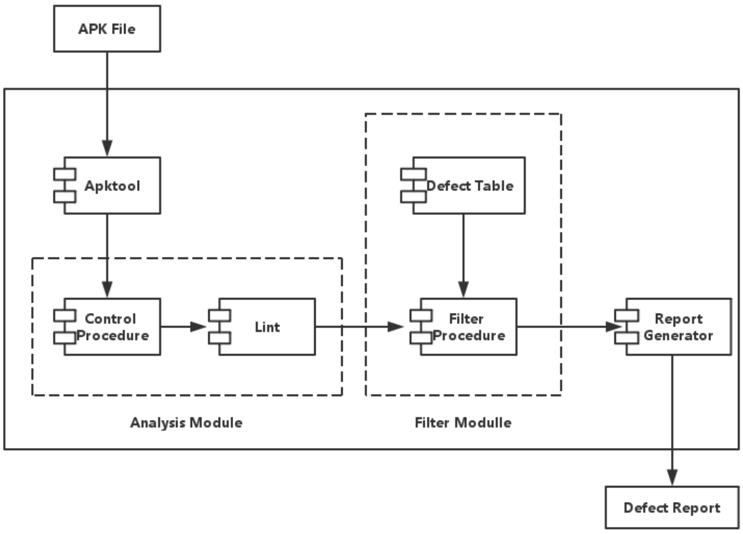


Fig. 9. Layout Defect Analysis Process

5 EVALUATION

We have implemented the proposed analysis as a prototype tool called *SAAD*. In order to evaluate our tool, we have conducted experiments on 64 real *APK* files, with 28 of them from well-known markets, and the other 36 are open sourced apps. In order to complete a more comprehensive experiment, we select these apps belonging to different classifications. In the process of the experiment, we collect the statistics of apps based on characteristics and scale, some of them are shown in Table 1.

Table 1. Scale Statistics of Apps(part)

APP	File Size	Component Number	Layout Number
Agenda Plus	1.91M	3	9
Heart Rate Runtastic	6.75M	24	101
Duomi Radio Station	8.79M	14	25
Drifting Bottle	7.29M	23	49
Constellation Camera	9.04M	42	93

5.1 Result of Resource Leak Detection

With code confirmation, we have detected that 52 of them have resource leaks, 8 of them are false positives; 4 apps are leak free. The accuracy is thus 86.67%, while the false positive rate is 13.33%. Among the 52 apps that have resource leaks, three kinds of leaks can be detected after we review their source code.

- no release operation. The current component in an app does not take initiatives to release resources. For example, the *Drifting Bottle* application in Table 1, uses the *SoundPool* class without releasing the obtained resources. Table 2 shows the details of the invoking path. It appears in an activity named *HrLoginSelectionActivity*, which is invoked by *SplashScreenActivity*.
- The path of existing release operation may blocked, e.g., by exception handlings.
- The release operation has not been activated by an event. In this situation, an app has a release operation, while it releases only when a specific event such as *onClick*, *onKeyDown* and so on occurs. If the user cannot trigger any of these events, the related resources cannot be released.

Table 2. Report Fragment of Resource Leak

```
<class name=” HrLoginSelectionActivity”>
  <path>
    SplashScreenActivity,HrLoginSelectionActivity
  </path>
  <leak>[Landroid/media/SoundPool;]</leak>
</class>
```

5.2 Result of Layout Defect Detection

We have detected that 45 apps have layout defects, 14 apps are false positives and 5 apps are defect free. The accuracy is 76.27% and the false positive rate is 23.73%. To validate the experimental results, we employ a view hierarchy tool called *HierarchyViewer* [8] to visualize the nested structure of layout files when running applications.

There are two types of false positives: *UselessParent* or *UselessLeaf*, which are raised by the static analysis of *Lint*. Since the widgets of a layout can be loaded only during an app's execution, we monitor the behavior of the layout by *HierarchyViewer*, and identify all of the 14 false positives. In addition, *HierarchyViewer* can report the start time of each widget and its drawing time, which helps to confirm that layout defects consume system resources and time.

Fig. 10 summarizes 45 apps that have layout defects. The x-axis denotes the number of layout defects. Through the experimental results, we can see some defects appear more frequently, including *Useless Parent*, *Inefficient Weight*, *Compound Drawables* etc. It indicates that some developers may not design the layout structure rigorously, and thus create some useless widget and useless properties, resulting in a more complex layout structure and causing unnecessary consumption of CPU resources and memory.

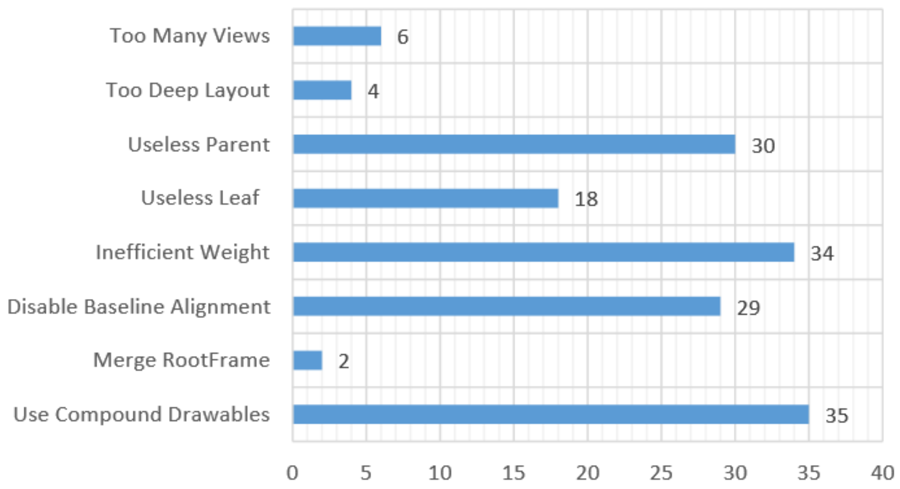


Fig. 10. Summary of Layout Defect

6 RELATED WORK

We present related work in the following three aspects: (1) detecting and testing energy bugs; (2) estimating energy consumption; (3) optimizing energy.

6.1 Detecting and Testing Energy Bugs

The researches in [22, 24, 27] are strongly relevant with our work. Guo et al [27] aim to detect resource leak of Android apps. However, the provided approaches are not context sensitive. For instance, considering resource leak, if there exists one releasing resource path, no resource leak report is given, causing false negatives. Wu et al [22, 24] present Relda2, a light-weight static resource leak detection tool, which takes model checking technology to detect resource leak. Comparatively, our approach combines function call analysis with control flow analysis to locate the real paths related to energy bugs.

There are some other work related with detecting and testing energy bugs. Wu et al. [13] focus on detecting energy-related defects in the UI logic of an Android application. The authors identify two patterns of behavior in which location listeners are leaking. Liu et al. [12, 19] implement a static code analyzer, *PerfChecker*, to detect identified performance bug patterns. Moreover, they build a tool called *GreenDroid* for automatically analyzing an app's sensory data utilization at different states and reporting actionable information to help developers locate energy inefficiency problems and identify their root causes [11]. Wan et al. [20] present a technique for detecting display energy hotspots and prioritizing them for developers. Abhik et al. [25] present an automated test generation framework, which systematically generates test inputs that are likely to capture energy hotspots/bugs.

6.2 Estimating Energy Consumption

Energy is a critical resource for smartphones. However, developers usually lack quantitative information about the behavior of apps with respect to energy consumption. Lu et al. [23] propose a lightweight and automatic approach to estimating the method-level energy consumption for Android apps. Li et al. [15] provide code-level estimates of energy consumption using program analysis and per-instruction energy modeling. [16] presents an approach to calculating source line level energy consumption information by combining hardware-based power measurements with program analysis and statistical modeling. Mario et al. [21] present a quantitative and qualitative empirical investigation by measuring energy consumption of method calls when executing typical usage scenarios in 55 mobile apps from different domains. Ferrari et al. [9] present the design and implementation of a Portable Open Source Energy Monitor (POEM) to enable developers to test and measure the energy consumption of the basic blocks, the call graph, and the Android API calls, allowing developers to locate energy leaks with high accuracy. [26] presents the design, implementation and evaluation of eprof, the first fine-grained energy profiler for smartphone apps. Eprof also reveals several wakelock bugs, a family of energy bugs in smartphone apps.

6.3 Optimizing Energy

A smartphone's display is usually one of its most energy consuming components. There are several researches focusing on optimization issues. Li et al. [17] develop an approach for automatically rewriting web applications so that they can generate more energy efficient web pages. Kim et al. [14] propose a novel static optimization technique for eliminating drawing commands to produce energy-efficient apps.

Making HTTP requests is one of the most energy consuming activities in Android apps, Li et al. [18] propose an approach to reducing the energy consumption of HTTP requests by automatically detecting and then bundling multiple HTTP requests.

7 CONCLUSION AND FUTURE WORK

Due to the limited capacity of the battery power in (Android) smartphones, energy bugs presenting in Android apps may cause serious battery drain. In this paper, we have proposed a static analysis approach to detect both resource leak and layout defect in Android applications. We have implemented our analysis in the *SAAD* tool and have used it to analyze 64 real applications. In our experiment, we have found that 52 apps are resource leakage, and 45 apps layout defects. The corresponding accuracies are 86.67% and 76.27%, with false alarm rates at 13.33% and 23.73% respectively. The results show that our *SAAD* tool can effectively analyze energy bugs of Android apps.

For the future work, due to the limitations of static analysis, we will combine dynamic analysis with current framework. Moreover, although a number of callback functions and event response functions are available in Android system, our framework only analyzes some common functions. As future work, we will include more functions analysis into the framework.

References

1. A. Pathak, Y. C. Hu and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. HotNets-X Proceeding of The 10th ACM Workshop on Hot Topics in Networks. 2011.
2. Banerjee A, Chong L K, Chattopadhyay S, et al. Detecting energy bugs and hotspots in mobile apps. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014:588-598.
3. Zhang J, Musa A and Le W. A comparison of energy bugs for smartphone platforms. Engineering of Mobile-Enabled Systems (MOBS), IEEE, 2013:25-30.
4. APKTool. https://code.google.com/p/android-apktool/.
5. Lint. http://tools.android.com/tips/lint.
6. Hoffmann J, Ussath M, Holz T, et al. Slicing droids: program slicing for smali code. Automated Software Engineering (ASE), March 18-22, 2013, Coimbra, Portugal. IEEE, 2013:1844-1851.
7. Dalvik. https://en.wikipedia.org/wiki/Dalvik.
8. Hierarchy Viewer. http://developer.android.com/tools/help/hierarchy-viewer.html.
9. Ferrari A, Gallucci D, Puccinelli D, et al. Detecting energy leaks in Android app with POEM. Pervasive Computing and Communication Workshops (PerCom Workshops), IEEE, 2015.
10. Java Path Finder. http://babelfish.arc.nasa.gov/trac/jpf/.
11. Yepang Liu, Chang Xu and S.C. Cheung. Where Has My Battery Gone? Finding Sensor Related Energy Black Holes in Smartphone Applications. Pervasive Computing and Communications (PerCom), IEEE, 2013:2-10.
12. Yepang Liu, Chang Xu and S.C. Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. Proceedings of the 36th International Conference on Software Engineering, 2014:1013-1024.
13. Haowei Wu, Shengqian Yang and Atanas Rountev. Static Detection of Energy Defect Patterns in Android Applications. Proceedings of the 25th International Conference on Compiler Construction, ACM, 2016:185-195.
14. P. Kim, D. Kroening and M. Kwiatkowska. Static Program Analysis for Identifying Energy Bugs in Graphics-Intensive Mobile Apps. In Proc. 24th IEEE International Conference on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2016), IEEE CS Press.
15. Shuai Hao, Ding Li, William G. J. Halfond and Ramesh Govindan. Estimating Mobile Application Energy Consumption using Program Analysis. In Proceedings of the 35th International Conference on Software Engineering (ICSE), May 2013.
16. D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating Source Line Level Energy Information for Android Applications. In ISSTA, 2013.
17. Ding Li, Angelica Huyen Tran and William G. J. Halfond. Making Web Applications More Energy Efficient for OLED Smartphones. In Proceedings of the International Conference on Software Engineering (ICSE), June 2014.
18. Ding Li, Yingjun Lyu, Jiaping Gui and William G.J. Halfond. Automated Energy Optimization of HTTP Requests for Mobile Applications. In Proceedings of the 38th International Conference on Software Engineering (ICSE), May 2016.
19. Yepang Liu, Chang Xu, S.C. Cheung, and Jian Lu. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. In IEEE Transactions on Software Engineering, Vol. 40, No. 9, pp.911-940, September 2014.
20. Mian Wan, Yuchen Jin, Ding Li and William G. J. Halfond. Detecting Display Energy Hotspots in Android Apps. In Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST), April 2015.

21. Mario Linares Vsquez, Gabriele Bavota, Carlos Bernal-Crdenas, et al. Mining energy-greedy API usage patterns in Android apps: an empirical study. In 11th Working Conference on Mining Software Repositories, MSR 2014:2-11.
22. Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan and Jian Zhang. Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps. IEEE Trans. Software Eng. 42(11): 1054-1076 (2016).
23. Qiong Lu, Tianyong Wu, Jiwei Yan, Jun Yan, Feifei Ma and Fan Zhang. Lightweight Method-Level Energy Consumption Estimation for Android Applications. TASE 2016: 144-151.
24. Tianyong Wu, Jierui Liu, Xi Deng, Jun Yan and Jian Zhang. Relda2: an effective static analysis tool for resource leak detection in Android apps. ASE2016: 762-767.
25. Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. SIGSOFT FSE 2014: 588-598.
26. Abhinav Pathak, Y. Charlie Hu and Ming Zhang: Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof. Proceedings of the 7th ACM european conference on Computer Systems, EuroSys 2012: 29-42.
27. Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang and Yanli Zhang. Characterizing and detecting resource leaks in Android applications. IEEE/ACM 28th International Conference on Automated Software Engineering, ASE2013: 389-398.