Relda2: An Effective Static Analysis Tool for Resource Leak Detection in Android Apps

Tianyong Wu[1,3], Jierui Liu[1,3], Xi Deng[2,3], Jun Yan[1,2], Jian Zhang[1,3]
[1] State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] Technology Center of Software Engineering
Institute of Software, Chinese Academy of Sciences, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China
{wuty, liujr, yanjun, zj}@ios.ac.cn, dengxi215@mails.ucas.ac.cn

ABSTRACT

Resource leak is a common bug in Android applications (apps for short). In general, it is caused by missing release operations of the resources provided by Android (like Camera, Media Player and Sensors) that require programmers to explicitly release them. It might lead to several serious problems for the app and system, such as performance degradation and system crash.

This paper presents Relda2, a light-weight, scalable and practical static analysis tool, for detecting resource leaks in the byte-code of Android apps automatically. It supports two analysis techniques (flow-insensitive for quick scanning and flow-sensitive for accurate scanning), and performs inter-procedural analysis to get more precise bug reports. In addition, our tool is practical to analyze real-world apps, and has been applied to 103 Android apps, including industry applications and open source programs. We have found 67 real resource leaks in these apps, which we confirmed manually. A demo video of our tool can be found at the website: https://www.youtube.com/watch?v=Mk-MFcHpTds.

CCS Concepts

•Software and its engineering → Automated static analysis;

Keywords

Android apps, resource leak, static analysis, byte-code

1. INTRODUCTION

To enrich user experience, Android phones contain a bunch of resources embedded in them, such as GPS, Camera, Media Player and different kinds of Sensors. These resources require explicit user management, that is, the programmers need to request and release them manually. Missing release operations may cause performance degradation (e.g. huge energy-consumption, memory-consumption), or even system crash. We call this kind of defects *resource leak* in our work [7, 12], which is a common type of bugs in Android apps, for the developers may be careless or may misunderstand the Android specification, or they tend to focus on the user friendliness and functionality of their apps.

Testing is a commonly used approach for detecting bugs in Android apps [11, 13]. However, the testing effectiveness depends on the fault detection capability of test suites. On the other hand, static analysis, a fully automatic approach, can detect some incorrect behaviors and performance degradations of Android apps without executing test cases. However, previous works on static analysis for Android apps [3, 8] have not systematically investigated the resource leak problem yet.

There are several challenges for static analysis on Android apps. The first one is about component-based and event-driven nature of Android framework. The implicit callback mechanism provided by Android system makes the generation of the precise Call Graph (CG, the essential part of static analysis) very difficult. The second one is how to perform inter-procedural analysis for real-world apps which may contain about dozens of thousands of methods. A straightforward approach is to inline each invoked method, however, it may result in path explosion, and is hard to be applied to large-scale apps.

This paper presents Relda2 (REsource Leak Detection for Android), a light-weight, scalable and practical static analysis tool for detecting resource leak in Android apps automatically. Our tool works on the byte-code of the apps directly. It provides an inter-procedural analysis framework that is implemented with the summary-based approach. To make a trade-off between scalability and precision, our tool supports two analysis techniques (flow-insensitive and flow-sensitive). The flow-insensitive analysis ignores the control flow information and can quickly analyze an app, while the flow-sensitive technique can eliminate a number of false negatives with more but acceptable time cost.

Relda2 is platform-independent, easily deployed, and practical to analyze real-world Android apps. It has been applied to 90 industry apps and 13 open source apps, and we have found and confirmed 67 real resource leaks in them.

2. RELDA2

In this section, we introduce the system architecture and techniques used in Relda2. It takes an Android app as in-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970278

put, and outputs the potential resource leaks in the apps. Note that there is a significant difference between our tool and other existing static analysis tools for Android apps. Most of existing tools (e.g. [3, 8]) do not analyze the Dalvik byte-code directly, and they are built on top of static analysis tools (like Soot, WALA) for Java programs. To analyze the Android apps, these tools have to translate the Dalvik byte-codes to some intermediate representation (like Jimple) or Java byte-codes. Several works have mentioned that the translation process may fail in some cases (e.g. [4]), especially in the deliberately modified apps. Besides, the Dalvik byte-code has a small instruction set and can be parsed by existing tools, such as Androguard [1] and Apktool [2]. The above issues motivate us to construct a pure Dalvik-byte-code-oriented analysis tool for Android apps.

2.1 System Overview

The high-level overview of Relda2 is shown in Fig. 1. The resource table contains the target resources that we focus on, while the Callback Graph (CBG) stores the information of implicit callbacks provided by the Android framework. Currently, we collect them according to the Android reference, and the users can customize the resource table based on their requirements. More details about how to collect these information can be seen in our work [12].

Our tool Relda2 mainly consists of three parts as follows.

- The *Preprocess* module disassembles the APK file into DEX byte-code and constructs a Function Call Graph (FCG) containing the call relations of methods to perform inter-procedural analysis.

- The *Analysis* module employs the summary-based approach to record the resource request and release operations of each method in the app. The analysis process can be divided into two stages: *Resource summary* obtains all the resource operations in each method, and *Implicit callback order handling* refines the resource summaries with callback information.

- The *Bug report* module checks the resource summaries to determine whether the app has resource leaks and provides the trace information of potential defects to help developers locate the bugs.

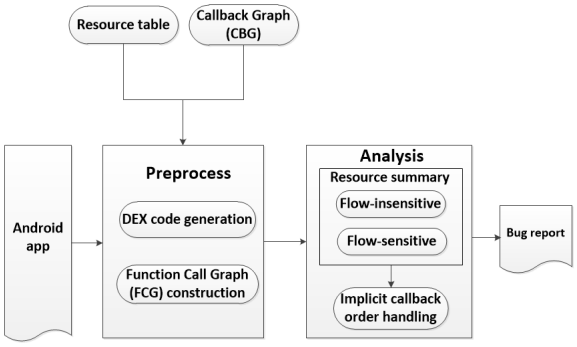


Figure 1: High-level Overview of Relda2

2.2 FCG Construction

In the analysis of an Android app, we distinguish between local resources and global resources. For a local resource (resource declared and only used within a method), we assume that developers want to release it when the method finishes. To detect the possible leaks, we can analyze each function separately if Android apps only use local resources. However, in real-world Android apps, the resources are usually requested in one function and released in another function. We call this kind of resources global resources that should be released in all the paths that go through its request point, or in the exit callbacks of the app. To address this issue, we construct the Function Call Graph (FCG) to assist inter-procedural analysis. We give the FCG construction process as follows. It first gets the main activities of the app, then obtains all the reachable methods and constructs nodes and edges in the FCG. Each node indicates a method and each edge represents a method call relation. For each method, we traverse its instructions one by one. When it comes to an "invoke" instruction, we collect all the invoked functions of this instruction to construct the edges. Specially, if the instruction invokes an implicit callback, we treat all of its invoked methods as the callees according to the CBG.

2.3 Analysis

The analysis module analyzes the methods in the FCG to detect whether there are resource leaks. To perform inter-procedural analysis, we leverage the function summary technique, which can collect side-effects as function summaries and reuse them in the function call sites. This module can be divided into two steps: resource summary and implicit callback order handling.

The first step generates the resource summary for each method. Here, we define the resource summary of a method as a set of resource operations invoked by it directly or indirectly. Depending on the analysis techniques, the resource summary can be generated with or without the control flow information in the method, a.k.a. flow-sensitive or flow-insensitive. The flow-insensitive technique carries out a simple and fast checking on byte-code in a sequential order. We traverse all the methods of the FCG in a bottom-up manner and construct their resource summaries. On the other hand, the flow-sensitive technique generates the resource summaries according to the Control Flow Graph (CFG). We use a model checker to systematically analyze all the methods from bottom to up, as we find that the property of resource leak (the resource that has been requested should be released eventually) can be easily defined by a temporal logic formula and verified by a model checker. For some of the methods are independent in FCG, we can distribute the analysis workloads into multiple threads (processes) to reduce the execution time further. In a word, we can employ the flow-insensitive method to process a quick checking and the flow-sensitive one to perform a more precise but time-consuming analysis.

The implicit callback order handling module rearranges the resource summaries with the calling orders from CBG. For each callback order in the CBG, we figure out its all possible execution sequences in the app. Then for each sequence, we sum up the resource summaries of all the methods in the sequence as the summary of the first method in the sequence.

2.4 Bug Report

The bug report module summarizes the analysis results and generates the report to assist the users to locate and fix the potential resource leaks. For a potential resource leak, we do not report the method that requests the leaked resource, since there may exist multiple paths going through this method. Instead, a bug report generated by Relda2 contains the entry method that is the entry point of the path containing the resource leak, and the corresponding resource release operation for fixing the leak. Here, the entry method must be a callback defined in Android system (e.g. *onCreate*(), *onClick*) that can only be invoked by Android framework at runtime when some specific events are triggered. The user may also want to know the method that directly invokes the resource request operation so that our bug report also gives the FCG of the reported entry method (which contains all of its callee methods), and its CFG. With the above information, the user can locate and fix the resource leak more easily.

2.5 Implementation

We implemented our main techniques on top of `Androguard` [1] (an open-source tool that maps DEX/APK format into full Python objects) in Python language and developed a command-line tool Relda2. To enhance the user experience, we also developed a GUI program with Qt framework for user interaction.

Recall the system architecture of our tool. In the preprocess module, we leverage `Androguard` to decompress and disassemble the app into the DEX byte-code. In the flow-sensitive approach of the analysis module, we also make use of it to generate the CFG of each method in the app. The model checker used in Relda2 is NuSMV [6], which is a symbolic model checker for Computation Tree Logic (CTL). To implement our multi-threading technique for accelerating the analysis efficiency, we leverage the module *multiprocessing* provided by Python to distribute the analysis workload to multiple process.

Different with the bugs related to the functionalities of the software, the resource leaks can not be easily observed from the GUI. Thus, Relda2 leverages the program instrumentation technique on the byte-code of the app to record the resource-related operations when the app is running. Specifically, Relda2 first unpacks the app to get the smali (a format of DEX byte-code) files, and then inserts the monitoring instructions into these files. Finally, it repacks the smali files back to apk files. We only insert a few instructions following the relevant resource operations so that the overhead of instrumentation can be omitted.

To sum up, Relda2 is easy to be deployed on different operating systems, like Linux and Windows. In addition, the analysis process of the tool is fully automatic. Given an Android app, the user only needs to set the options and then run the tool to get the bug report.

2.6 Usage

In this section, we will introduce how to use Relda2 to analyze an Android app. When Relda2 is launched, the user can select the apk to be analyzed. Then it will first collect and show the basic information, such as size, the number of classes and methods (see Fig. 2). Next, the user needs to configure the options for analysis. The major options are shown in Fig. 3 that include four parts. The "Resource" part is used to configure which resources the user focuses on, while the "Activity" part decides which Activities in the app that the user wants to analyze. The "Analysis Strategy" part gives the analysis algorithms (flow-insensitive or flow-sensitive) to be used and whether the multi-thread technique is used. The "Output" part configures the output information including the FCG and CFG of the method that has resource leaks, and an instrumented apk modified from the analyzed apk by inserting a number of extra instructions into the original app for recording the resource-related operations.

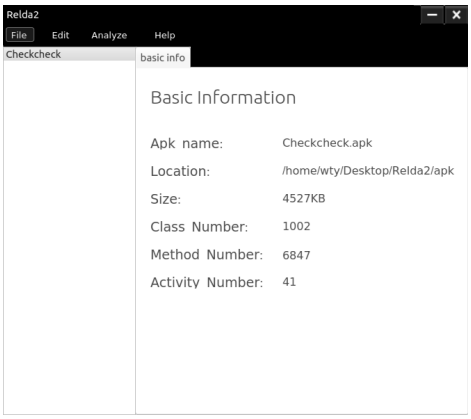


Figure 2: Basic information widget

Figure 3: Configuration widget

After the configuration, Relda2 analyzes and checks the resource summaries to generate the bug report (see Fig. 4). If the summary of the method only contains the request operations, that is, there may be resource leaks, then Relda2 records the locations where the leaks occur. Relda2 also provide the CFG and FCG of the entry method containing the resource leak and an instrumented app. The CFG and FCG help the user locate the resource leak, while the instrumented apk makes bug confirmation process easier.

3. EVALUATION

To evaluate the effectiveness of our tool Relda2, we collected 103 real apps that can be disassembled by Androguard

for our evaluation, where 90 apps are from two famous official app markets, Google Play and Wandoujia (a popular market in China), and 13 apps are from an open-source Android app repository[1]. The functionality of these apps are varied (like sociality, media, game), and they all use several resources. Due to the limit of the space, we only show 15 apps in Table 1. In this table, the first column shows the origin of the apps (open-source community or app markets), and the second column gives the name of the apps. The third column shows the size of the APK file (MB). The size of our experimental apps range from tens of KB to dozens of MB. The last two columns indicate the number of classes and methods in an app. In our experiments, we run the command-line version of Relda2 on an Intel Xeon 2.40GHz (16 cores) machine, with 32GB memory and CentOS 6.5 operating system.

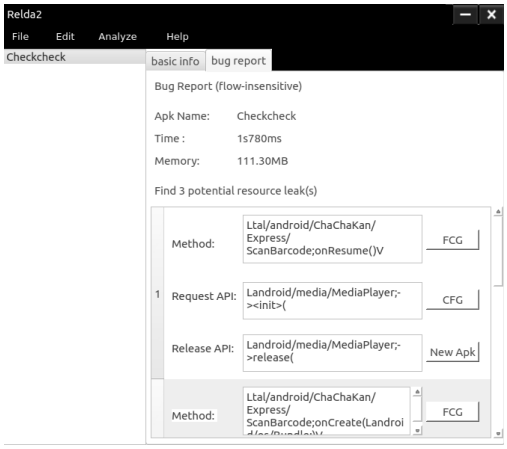


Figure 4: Bug report widget

Table 1: The statistics of experimental applications

origin	*app*	*size*	*#class*	*#method*
open-source community	Andless	0.3	69	399
	Bluechat	0.1	47	154
	Foocam	0.6	507	3347
	Getbackgps	0.2	69	442
	Impeller	4.5	3858	28178
app markets	Anydo	10.8	6926	45985
	BaibianTorch	2.6	736	4747
	BubeiListen	3.8	2804	15966
	BudingTicket	3.8	1948	14727
	Checkcheck	4.5	1002	6847
	Compass	1.4	472	3449
	HikiPlayer	0.2	164	1062
	KuaipaiQR	5.5	3385	20113
	PicsArt	18.3	7052	43838
	Yelp	14.3	6024	35809

3.1 Bugs Detected

Relda2 reported that 76 apps (68 closed-source apps and 8 open-source apps) might have resource leaks, and we tried to confirm these bug reports. However, there are currently no specific tools publicly available to confirm whether the resource leaks in our bug reports are real bugs. Here, we mainly used error-guessing testing[2] to design appropriate test cases that can exactly trigger the resource leaks reported by our tool (note that this technique can confirm a part of the reports, the rest ones may also be real resource leaks).

Totally, Relda2 detected 69 potential resource leaks with flow-insensitive technique, and 121 leaks with flow-sensitive technique. The precision rates of two analysis techniques are 47/69 (68.1%) and 67/121 (55.4%), respectively. In fact, the resource leaks detected by the flow-insensitive approach are also found by the flow-sensitive approach. Since our error-guessing testing is incomplete, the real precision rate should be greater. Table 2 gives the detailed results of the apps in Table 1, where #REP is the number of leaks reported by the tool and #TP is the number of true positives we have confirmed. Here, we count the values of #TP and #REP with merging the redundant reports that are caused by the same resource request statements if they can be fixed by inserting release operations in the same exit points.

Table 2: Detected bugs

App	Flow-insensitive		Flow-sensitive	
	#TP	#REP	#TP	#REP
Andless	1	2	1	3
Bluechat	1	1	1	1
Foocam	1	1	1	2
Getbackgps	1	3	1	3
Impeller	2	2	2	2
Anydo	1	1	1	1
BaibianTorch	4	4	5	5
BubeiListen	3	4	4	5
BudingTicket	0	1	1	2
Checkcheck	3	3	4	6
Compass	0	0	1	1
HikiPlayer	0	0	2	4
KuaipaiQR	2	4	3	6
PicsArt	2	2	2	2
Yelp	1	1	2	4

3.2 Case Study

In this section, we present a case study to show how our tool helps the users to detect, confirm, and locate the resource leaks in their apps. **Bluechat** is an open-source and light-weight P2P chat app that uses the energy-consuming resource *Bluetooth*. Two devices connect each other with bluetooth and one can send messages to another. Fig. 5 shows a simplified code snippet of the process. As soon as *ClientActivity* is activated, it first obtains an instance of BluetoothAdapter (the variable *mBluetoothAdapter*) in the method *onCreate*(), which is used to perform fundamental Bluetooth tasks. Then it invokes the method *startDeviceSearch*() to search for the nearby devices. At the beginning of the method *startDeviceSearch*(), it turns on the local Bluetooth adaptor by calling method *mBluetoothAdapter.enable*(). However, the developers forgot to turn off the Bluetooth adaptor by calling the method *mBluetoothAdapter.disable*() in the *onDestroy*() callback.

[1]https://f-droid.org/

[2]https://en.wikipedia.org/wiki/Error_guessing

Relda2 can detect this resource leak, and the bug report generated by it is shown in Fig. 6. It consists of three parts: the entry method (system callback) that contains the resource leak, the system APIs to request and release the resource. To assist the developer locate the resource leak, Relda2 also provides an FCG that contains all callee methods of the reported entry method. Fig. 7 shows a part of the FCG that helps the user find the method *ClientActivity.startDeviceSearch*() that directly requests the resource.

```
public class ClientActivity extends Activity {
  protected void onCreate(Bundle
              savedInstanceState) {
    ......
    mChatManager = new ChatManager(this, false);
    mBluetoothAdapter =
    BluetoothAdapter.getDefaultAdapter();
    ......
    startDeviceSearch();
  }
  private void startDeviceSearch() {
    mBluetoothAdapter.enable();
    ......
  }
  public void onDestroy() {
    ......
    mBluetoothAdapter.cancelDiscovery();
  }
}
```

Figure 5: Bluechat example

```
method:      ClientActivity.onCreate
request API: BluetoothAdapter.enable
release API: BluetoothAdapter.disable
```

Figure 6: Bug report of Bluechat

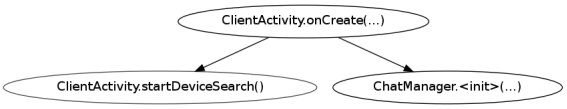


Figure 7: FCG of the method *ClientActivity.onCreate*()

As mentioned before, the resource leaks can not be easily observed from runtime information, so we leverage the program instrumentation technique on the byte-code of the app to record the resource-related operations when the app is running. In this case, there is only one statement to request the resource (*mBluetoothAdapter.enable*()), thus we insert extra statements following this statement to log some tipping information when the resource-related operation is executed. Fig. 8 shows a part of byte-code after the instrumentation, where Line 2 to 4 are the instrumented statements. When the users execute the instrumented app, they can observe the logging information about the resource operations to confirm whether the resource leak occurs.

```
......
1. invoke-virtual v1, Landroid/bluetooth/
    BluetoothAdapter;->enable()Z
2. const-string v2, "Relda2"
3. const-string v3, "BluetoothAdapter.enable"
4. invoke-static/range {v2 .. v3},
    Landroid/util/Log;->d(Ljava/lang/String;
    Ljava/lang/String;)I
......
```

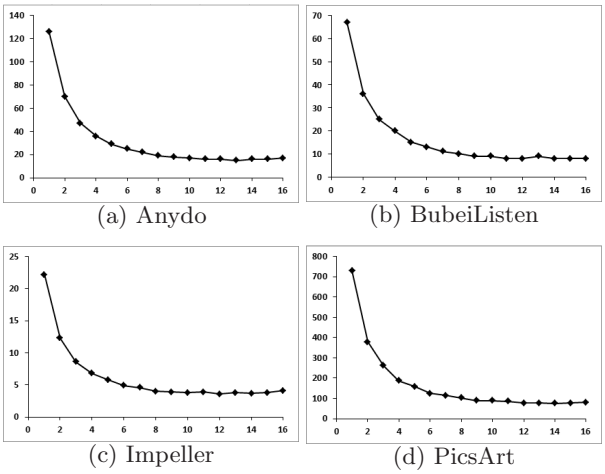
Figure 8: Instrumented byte-code of Bluechat

3.3 Performance

The analysis time of flow-insensitive technique is within one minute for each app, and it costs 6.7 seconds on average. The flow-sensitive technique costs at most 728.3 seconds (about 14 minutes), and 79 seconds (about 1.3 minutes) on average. Table 3 indicates the analysis time (second) and memory (MB) for the apps in Table 1. In our experiments, our tool requires at most 1GB memory.

Table 3: Analysis time and memory

App	Flow-insensitive		Flow-sensitive	
	time	*mem*	*time*	*mem*
Andless	0.03	24	5.8	37
Bluechat	0.007	19	0.01	22
Foocam	0.001	59	0.002	61
Getbackgps	0.02	24	3.9	30
Impeller	0.13	381	22.2	393
Anydo	0.09	583	126	650
BaibianTorch	0.01	81	20.7	118
BubeiListen	1.87	211	91	300
BudingTicket	1	185	67	247
Checkcheck	0.97	111	20	173
Compass	0.07	62	10.3	77
HikiPlayer	0.01	30	7.2	45
KuaipaiQR	4.2	283	140	400
PicsArt	28.5	598	728.3	935
Yelp	3.4	429	181.8	527



(a) Anydo (b) BubeiListen (c) Impeller (d) PicsArt

Figure 9: Analysis time under different number of threads

Relda2 also uses the multi-thread technique to reduce the analysis time. In our experiments, we set the number of threads from 1 to 16 (our machine has 16 CPU cores). Fig. 9 shows the analysis time of the apps in Table 1 under different number of threads. From this figure, we can easily observe that the analysis time is significantly reduced, and we give the empirical value for the number of threads as 8, for there is little improvement when the number exceeds 8. The analysis time of the single thread method is often five to seven times as much as that of the multiple thread method.

4. RELATED WORK

There are several researchers focusing on the resource leak problem in Android apps. D. Yan et al. [13] proposed a model-based approach to generate test cases for detecting resource leaks. Note that the resources they focused on are memory, thread and binder. Y. Liu et al. [10] aimed to find sensor-related energy inefficiency problems in Android apps by runtime verification. In fact, these problems are caused by the resource leaks of sensors. K. Kim and H. Cha [9] focused on no-sleep energy bug, which is also caused by the resource leaks. They first analyzed the WakeLock request and release mechanism in Android platform and then monitored the kernel functions to detect the WakeLock behavior. Each of these works only considers the leaks of one or two kinds of resources, while we systematically collected the resources that require developers to release them manually and proposed a general approach to detect the leaks.

The event-driven nature of Android framework makes the generation of the precise control flow graph very difficult. Y. Cao et al. [5] implemented EdgeMiner, which can statically generate API summaries that describe implicit control flow transitions through the Android framework. S. Yang et al. [14] considered user-event-driven components and the related sequences of callbacks from the Android framework to the application code. They proposed a context-sensitive analysis method to capture such callback methods.

5. CONCLUSION

This paper describes a light-weight, scalable and practical static analysis tool for detecting resource leaks in Android apps automatically. It provides two analysis techniques and supports inter-procedural analysis. Tens of real resource leaks from popular commercial apps and open source programs have been found by our tool. In the future, we would like to design some techniques to support continuous upgrading Android specifics and effective analysis. In addition, without the significant reduction of the efficiency, we will implement more accurate analysis techniques (e.g. symbolic execution) to eliminate a number of false positives.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] Androguard. http://code.google.com/p/androguard/.

[2] Apktool - a tool for reverse engineering android apk files. http://ibotpeaches.github.io/Apktool/, 2016.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, pages 259–269, 2014.

[4] A. Bartel, J. Klein, Y. L. Traon, and M. Monperrus. Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *SOAP*, pages 27–38, 2012.

[5] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[6] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *CAV*, pages 359–364, 2002.

[7] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and detecting resource leaks in Android applications. In *ASE*, pages 389–398, 2013.

[8] W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for Android. In *ISSTA*, pages 106–117, 2015.

[9] K. Kim and H. Cha. Wakescope: Runtime wakelock anomaly management scheme for Android platform. In *EMSOFT*, pages 27:1–27:10, 2013.

[10] Y. Liu, C. Xu, and S. C. Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *PERCOM*, pages 2–10, 2013.

[11] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for Android apps. In *FSE*, pages 224–234, 2013.

[12] 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. *TSE*, 2016. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7442579, Accepted.

[13] D. Yan, S. Yang, and A. Rountev. Systematic testing for resource leaks in Android applications. In *ISSRE*, pages 411–420, 2013.

[14] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *ICSE*, pages 89–99, 2015.