PhD-FSTC-2016-52
The Faculty of Science, Technology and Communication

DISSERTATION

Defense held on the 29th November 2016 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

LI LI
Born on 13th February 1989 in ChongQing (China)

BOOSTING STATIC SECURITY ANALYSIS OF ANDROID APPS THROUGH CODE INSTRUMENTATION

Dissertation Defense Committee

Abstract

Within a few years, Android has been established as a leading platform in the mobile market with over one billion monthly active Android users. To serve these users, the official market, Google Play, hosts around 2 million apps which have penetrated into a variety of user activities and have played an essential role in their daily life. However, this penetration has also opened doors for malicious apps, presenting big threats that can lead to severe damages.

To alleviate the security threats posed by Android apps, the literature has proposed a large body of works which propose static and dynamic approaches for identifying and managing security issues in the mobile ecosystem. Static analysis in particular, which does not require to actually execute code of Android apps, has been used extensively for market-scale analysis. In order to have a better understanding on how static analysis is applied, we conduct a systematic literature review (SLR) of related researches for Android. We studied influential research papers published in the last five years (from 2011 to 2015). Our in-depth examination on those papers reveals, among other findings, that static analysis is largely performed to uncover security and privacy issues.

The SLR also highlights that no single work has been proposed to tackle all the challenges for static analysis of Android apps. Existing approaches indeed fail to yield sound results in various analysis cases, given the different specificities of Android programming. Our objective is thus to reduce the analysis complexity of Android apps in a way that existing approaches can also succeed on their failed cases. To this end, we propose to instrument the app code for transforming a given hard problem to an easily-resolvable one (e.g., reducing an inter-app analysis problem to an intra-app analysis problem). As a result, our code instrumentation boosts existing static analyzers in a non-invasive manner (i.e., no need to modify those analyzers).

In this dissertation, we apply code instrumentation to solve three well-known challenges of static analysis of Android apps, allowing existing static security analyses to 1) be inter-component communication (ICC) aware; 2) be reflection aware; and 3) cut out common libraries.

- ICC is a challenge for static analysis. Indeed, the ICC mechanism is driven at the framework level rather than the app level, leaving it invisible to app-targeted static analyzers. As a consequence, static analyzers can only build an incomplete control-flow graph (CFG) which prevents a sound analysis. To support ICC-aware analysis, we devise an approach called IccTA, which instruments app code by adding glue code that directly connects components using traditional Java class access mechanism (e.g., explicit `new` instantiation of target components).
- Reflection is a challenge for static analysis as well, because it also confuses the analysis context. To support reflection-aware analysis, we provide DroidRA, a tool-based approach, which instruments Android apps to explicitly replace reflective calls with their corresponding traditional Java calls. The mapping from reflective calls to traditional Java calls is inferred through a solver, where the resolution of reflective calls is reduced to a composite constant propagation problem.
- Libraries are pervasively used in Android apps. On the one hand, their presence increases time/memory consumption of static analysis. On the other hand, they may lead to false positives and false negatives for static approaches (e.g., clone detection and machine learning-based

malware detection). To mitigate this, we propose to instrument Android apps to cut out a set of automatically identified common libraries from the app code, so as to improve static analyzer's performance in terms of time/memory as well as accuracy.

To sum up, in this dissertation, we leverage code instrumentation to boost existing static analyzers, allowing them to yield more sound results and to perform quicker analyses. Thanks to the aforementioned approaches, we are now able to automatically identify malicious apps. However, it is still unknown how malicious payloads are introduced into those malicious apps. As a perspective for our future research, we conduct a thorough dissection on piggybacked apps (whose malicious payloads are easily identifiable) in the end of this dissertation, in an attempt to understand how malicious apps are actually built.

Acknowledgements

This dissertation would not have been possible without the support of many people who in one way or another have contributed and extended their precious knowledge and experience in my PhD studies. It is my pleasure to express my gratitude to them.

First of all, I would like to express my deepest thanks to my professor Yves Le Traon who has given me this great opportunity to come across continents to pursue my doctoral degree. He has always trusted and supported me with his great kindness throughout my whole PhD journey.

Second, I am equally grateful to my daily supervisors, Dr. Jacques Klein and Dr. Tegawendé Bissyandé, for their patience, valuable advice and flawless guidance. They have taught me how to perform research, how to write technical papers and how to conduct fascinating presentations. Their dedicated guidance has made my PhD journey a fruitful and fulfilling experience. I am very happy for the friendship we have built up during the years.

Third, I am particularly grateful to Dr. Alexandre Bartel who has introduced me into the world of static analysis of Android apps. Since then, working in this field is just joyful for me. I would like to extend my thanks to all my co-authors including Prof. Eric Bodden, Prof. Patrick McDaniel, Prof. David Lo, Dr. Lorenzo Cavallaro, Dr. Damien Octeau, Steven Arzt, Siegfried Rasthofer, Jabier Martinez, Daoyuan Li, etc. for their helpful discussions and collaborations.

I would like to thank all the members of my PhD defense committee, including chairman Prof. Lionel Briand, external reviewers Prof. Xiangyu Zhang and Prof. Michael Backes, my supervisor Prof. Yves Le Traon, and my daily advisers Dr. Jacques Klein, Dr. Tegawendé Bissyandé. It is my great honer to have them in my defense committee and I appreciate very much their efforts to examine my dissertation and evaluate my PhD work.

I would like to also express my great thanks to all the friends that I have made in the tiny Grand Duchy of Luxembourg for the memorable moments that we have had. More specifically, I would like to thank all the team members of SerVal at SnT for the great coffee breaks and interesting discussions.

Finally, and more personally, I would like to express my deepest appreciation to my dear wife Fengmei Chen, for her everlasting support, encouragement, and love. Also, I can never overemphasize the importance of my parents in shaping who I am and giving me the gift of education. Without their support, this dissertation would not have been possible.

Li LI
University of Luxembourg
November 2016

Contents

List of figures

List of tables

1 Introduction

In this chapter, we first introduce the motivation of conducting static security analysis for Android apps. Then, we summarize the challenges people face when conducting static security analysis of Android apps, and finally we present the contributions and roadmap of this dissertation .

Contents

1.1 Motivation for Static Security Analysis

In 2008, the first Android-based device, namely *T-Mobile G1* (or HTC Dream in areas outside of USA), was released. Since then, the number of Android-based devices varying from smart phones to TVs has never stopped increasing. Until now, there are over 1.5 million activated devices everyday and over 2 million apps available for users to choose[i]. Those apps have infiltrated into nearly every activity of users, including social networking, schedule planning, financial resource managing, etc. Positively, users' life has been much easier thanks to those apps. However, these phenomena, on the other hand, also threaten users' daily life, as the quality of Android apps are uneven, e.g., some of them may be malware [121, 125].

The current Android ecosystem does not provide a perfect way to prevent malicious apps from entering it [1]. Android malware passed into Google Play store could infect millions of users[ii]. As a result, the current Android ecosystem actually provides a hotbed for malicious apps to keep penetrating into our daily life. Moreover, because of its popularity, Android has become the primary target for attackers. There are actually many sweet points leveraged by attackers to exploit Android apps. We now summarize them as follows:

- Android apps are relatively easy to be reverse-engineered compared to other kinds of apps like iOS apps or PC executables. Indeed, there are many off-the-shelf tools (e.g., Soot, Apktool, dex2jar) that support the reverse engineering of Android apps. Thanks to that, it is also relatively easy to understand and instrument the code structures of Android apps, making app repackaging a popular task that injects malicious payloads or updates advertisement packages for popular Android apps.

- Android employs an all-or-nothing mechanism for its permission system (before Android 6.0, API level 23). At installation, users must grant all permissions requested by an app or simply terminate the installation. Because of this mechanism, users may install apps that declare more permissions than that their really need, resulting in a permission gap. This gap could be exploited by other (malicious) apps to achieve some activities that are not originally allowed (the so-called privilege escalation attack).

- Android apps are heavily dependent on libraries, including massive Java libraries and Android-specific advertisement libraries. Those libraries can only be shipped within Android apps and there is no simple way to separate or isolate them from the primary code [255]. However, the quality of those libraries cannot be guaranteed and thus the integration of libraries may increase the attack surface of Android apps. For example, third-party libraries can abuse the granted permissions of their host apps to leak sensitive user information.

- Android apps are written in Java and thus inherit the same reflection mechanism. With this feature, Android apps can reflectively access malicious payloads at runtime for their malicious behaviors. Furthermore, with the combination of dynamic code loading (DCL) feature, Android apps can even reflectively access remote malicious payloads at runtime (e.g., download $\rightarrow$ dynamically load $\rightarrow$ reflectively access).

- Android apps can invoke native code through Java Native Interface (JNI). This may induce security problems due to widely known memory corruption bugs in low-level languages (e.g., C or C++). Furthermore, as native code is not fully tamed yet by existing approaches, it could be intentionally adopted by malware writers to evade Java/Android bytecode-based analyses.

- Android adopts a special crossing-component communication mechanism, the so-called inter-component communication (ICC) mechanism, to facilitate the reuse of existing modules, resulting in a loose coupling architecture for speeding app implementation. As an example, every app can

[i]http://www.statista.com/topics/876/android/
[ii]http://www.ibtimes.co.uk/android-malware-discovered-google-play-store-1553341

leverage the dialing module of system to perform dialing service. However, this ICC mechanism also introduces security issues for Android apps. For instance, malicious apps can illegally access components of other apps through ICC, leading to *spoofing* attacks. Similarly, malicious apps can also hijack an ICC request, so as to steal information from the source component.

- Android embraces an open market model. In addition to the official Google market (called Google Play), there exist numerous alternative app markets such as AnZhi, Wandoujia, which impose no or limited app vetting process. Thus, malicious apps are easily delivered through those alternative markets. Besides, Android apps may be cloned from one market to another, opening the way for repackaging attacks.

Because of those aforementioned specificities, it is essential for the community to mitigate their threats. Fortunately, the literature has proposed a large body of works, including both static and dynamic analysis approaches, to secure Android apps. Because static analysis does not need to actually execute Android apps, it can be considered as more suitable to perform market-scale analysis. Thus, there is a strong need to perform static security analysis for Android apps.

1.2 Challenges of Static Security Analysis

In this section we introduce the technical challenges we face when conducting static security analysis of Android apps. More specifically, we introduce the challenges that are specific to Android and that are inherited from Java, respectively.

1.2.1 Android-specific Challenges

We now enumerate some challenges for static analysis that are mainly due to Android peculiarities.

Dalvik bytecode. Although Android apps are primarily developed in Java, they run in a Dalvik virtual machine (DVM), a register-based instruction model. Thus, all app packages (apks) are distributed on markets with Dalvik bytecode, and only a relatively few are distributed with source code in open source repositories. Consequently, a static analyzer for Android must be capable of directly tackling Dalvik bytecode, or at least of translating it to a supported format. Thus, most Java and Java bytecode analyzers, which could have been leveraged, are actually useless in the Android ecosystem. As an example, the mature FindBugs[iii] tool, which has demonstrated its capabilities to discover bugs in Java bytecode, cannot readily be exploited for Android programs.

Program entry point. Unlike programs in most general programming languages such as Java and C, Android apps do not have a `main` method. Instead, each app program contains several entry points which are called by the Android framework at runtime. Consequently, it is difficult for a static analyzer to build a global call graph of the app. Instead, the analyzer has to first search for all entry-points and build several call graphs with no assurance on how these graphs connect to each other, if ever.

Component Lifecycle. In Android, unlike in Java or C, different components of an application, have their own lifecycle. Each component indeed implements its lifecyle methods which are called by the Android system to start/stop/resume the component following environment needs. Figure 1.1 illustrates an example of a typical lifecycle of Android Activities, where at least six lifecycle methods (e.g., onPause) are involved in an Activity's life time. In practice, an application in the background (i.e., invisible lifetime), can first be stopped, when the system is under memory pressure, and later be restarted when the user attempts to put it in the foreground. Unfortunately, because these lifecycle

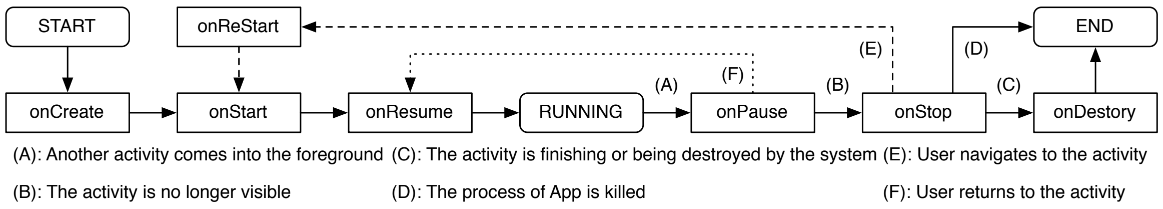

START
onReStart
END
onCreate
onStart
onResume
RUNNING
onPause
onStop
onDestory
(A)
(B)
(C)
(D)
(E)
(F)
(A): Another activity comes into the foreground
(B): The activity is no longer visible
(C): The activity is finishing or being destroyed by the system
(D): The process of App is killed
(E): User navigates to the activity
(F): User returns to the activity

Figure 1.1: The Lifecycle of Android Activities.

methods are not directly connected to the execution flow, they hinder the soundness of some analysis scenarios.

Callback Methods. With the user-centric nature of Android apps, a user can interact a lot with the apps (or system) through the touch screen. The management of user inputs is mainly done by handling specific callback methods such as the *onClick()* method which is called when users click a button. However, like lifecycle methods, callback methods are not directly connected as well (with the main context) and thus could also impact the soundness of some static analyses.

User/System Events Besides lifecycle methods, methods for handling user events (e.g., UI actions) and system events (e.g., sensor inputs [141]) constitute a challenge for static analysis of Android apps. Indeed, as such events can be fired at any time, static analysers cannot build and maintain a reliable model of the events [2]. It is further expensive to consider all the possible variants of a given model, due to limited resources [174, 242]. Nevertheless, not taking into account paths that include event-related methods may render some analysis scenarios unsound.

Inter-Component Communication (ICC). Android has introduced a special mechanism for allowing an application's components to exchange messages through the system with components of the same application or of other applications. This communication is usually triggered by specific methods, hereafter referred to as ICC methods. ICC methods use a special parameter, containing all necessary information, to specify their target components and the action requested. Similar to the lifecycle methods, ICC methods are actually processed by the system who is in charge of resolving and brokering it at runtime. Consequently, static analyzer will find it hazardous to hypothesize on how components connect to one another unless using advanced heuristics. As an example, FlowDroid [10], one of the most-advanced static analyzers for Android, fails to take into account ICCs in its analysis.

Libraries. An Android apk is a standalone package containing a Dalvik bytecode consisting of the actual app code and all library suites, such as advertisement libraries and burdensome frameworks. These libraries may represent thousands of lines of code, leading to the size of actual app to be significantly smaller than the included libraries. This situation causes two major difficulties: (1) the analysis of an app may spend more time on vetting library code than on real code; (2) the analysis results may comprise too many false positives due to the analysis of library "dead code". As an example, analyzing all method calls in an apk file to discover the set of permissions required may lead to listing permissions which are not actually necessary for the actual app code.

1.2.2 Java-inherited Challenges

Since Android apps are mainly written in Java, developers of static analyzers for such apps are faced with the same challenges as with Java programs, including the issues of handling dynamic code loading, reflection, native code integration, multi-threading and the support of polymorphism.

Reflection. In the case of dynamic code loading and reflective calls, it is currently difficult to statically handle them. The classes that are loaded at runtime are often practically impossible to analyze since they often sit in remote locations, or may be generated on the fly.

[iii] http://findbugs.sourceforge.net

Native Code. Addressing the issue of native code is a different research adventure. Most of the time, such code comes in a compiled binary format, making it difficult to analyze.

Multi-threading. Analyzing multi-threaded programs is challenging as it is complicated to characterize the effect of the interactions between threads. Besides, to analyze all interleavings of statements from parallel threads usually result in exponential analysis times.

Polymorphism. Finally, polymorphic features also add extra difficulties for static analysis. As an example, let us assume that method m_1 of class A has been overridden in class B (B extends A). For statement $a.m_1()$, where a is an instance of A, a static analyzer by default will consider the body of $m_1()$ in A instead of the actual body of $m_1()$ in B, even if a was instantiated from B (e.g., with $A\ a\ =\ new\ B()$). This obvious situation is however tedious to resolve in practice by most static analyzers and thus leads to unsound results.

1.3 Contributions

We now summarize the contributions of this dissertation as follows:

- **An SLR on static analysis of Android Apps.** We provide a clear view of the state-of-the-art works that statically analyze Android apps, from which we highlight the trends of static analysis approaches, pinpoint where the focus has been put and enumerate the key aspects where future researches are still needed. In particular, we have performed a systematic literature review (SLR) which involves studying 124 research papers published in software engineering, programming languages and security venues in the last 5 years (January 2011 - December 2015). Our in-depth examination on those papers has led to several key findings: 1) Static analysis is largely performed to uncover security and privacy issues; 2) The Soot framework and the Jimple intermediate representation are the most adopted basic support tool and format, respectively; 3) Taint analysis remains the most applied technique in research approaches; 4) Most approaches support several analysis sensitivities, but very few approaches consider path-sensitivity; 5) There is no single work that has been proposed to tackle all challenges of static analysis that are related to Android programming; and 6) Only a small portion of state-of-the-art works have made their artifacts publicly available.

  This work has led to a technical report entitled "Static Analysis of Android Apps: A Systematic Literature Review", which is currently under submission to an international peer-reviewed journal.

- **A code instrumentation based approach that bridges ICC for static analysis.** We present IccTA, a code instrumentation based approach, to detect privacy leaks among components in Android apps, where different components are explicitly connected through instrumented glue code. IccTA goes beyond state-of-the-art approaches by supporting inter-component detection. By propagating context information among components, IccTA improves the precision of the analysis. IccTA outperforms existing tools on two benchmarks for ICC-leak detectors: DroidBench and ICC-Bench. Moreover, IccTA detects 534 ICC leaks in 108 apps from MalGenome and 2,395 ICC leaks in 337 apps in a set of 15,000 Google Play apps.

  This work has led to a research paper published to the 37th International Conference on Software Engineering (ICSE 2015).

- **A tool-based approach that reduces IAC problems to ICC problems.** We present a tool called ApkCombiner which aims at reducing an IAC problem to an intra-app ICC problem. In practice, ApkCombiner combines different apps into a single apk on which existing tools can indirectly perform inter-app analysis. We have evaluated ApkCombiner on a dataset of

3,000 real-world Android apps, to demonstrate its capability to support static context-aware inter-app analysis scenarios.

This work has led to a research paper published to the 30th International Information Security and Privacy Conference (IFIP SEC 2015).

- **A code instrumentation based approach that tames reflection for static analysis.** We propose DroidRA, a code instrumentation based approach, to support reflection-aware static analysis in a non-invasive way. With DroidRA, we reduce the resolution of reflective calls to a composite constant propagation problem. We leverage the COAL solver to infer the values of reflection targets and app, and we eventually instrument this app to include the corresponding traditional Java call for each reflective call. DroidRA allows to boost an app so that it can be immediately analyzable, including such static analyzers that were not reflection-aware. We evaluate DroidRA on benchmark apps as well as on real-world apps, and demonstrate that it can allow state-of-the-art tools to provide more sound and complete analysis results.

  This work has led to a research paper published to the 2016 International Symposium on Software Testing and Analysis (ISSTA 2016) and a tool demonstration paper published to the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016).

- **A code instrumentation based approach that cuts off common libraries for static analysis.** We leverage a dataset of about 1.5 million apps from Google Play to harvest potential common libraries, including advertisement libraries. With several steps of refinements, we finally collect by far the largest set of 1,113 libraries supporting common functionality and 240 libraries for advertisement. We use the dataset to investigates several aspects of Android libraries, including their popularity and their proportion in Android app code. Based on these datasets, we have further performed several empirical investigations that demonstrate that (1) including common libraries will introduce both false positives and false negatives for pairwise app clone detection; (2) excluding common libraries can increase the accuracy of machine learning based Android malware detection approaches; and (3) excluding common libraries can significantly improve the time/memory performance of static security analysis tools.

  This work has led to a research paper published to the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016).

- **A thorough dissection on piggybacked apps for understanding malicious behaviors.** Malware writers can actually build on top of popular apps to ensure a wide diffusion of their malicious code within the Android ecosystem. In order to have a better understanding on how malware behaves, we choose piggybacked apps as the targets of investigation in this work because the malicious payload of piggybacked apps can be easily identified. Although recent research has produced approaches and tools to identify piggybacked apps, the literature lacks a comprehensive investigation into such phenomenon. We fill this gap by 1) systematically building a large set of piggybacked and benign apps pairs, which we release to the community, 2) empirically studying the characteristics of malicious piggybacked apps in comparison with their benign counterparts, and 3) providing insights on piggybacking processes. Experimental results show that piggybacking operations not only concern app code, but also extensively manipulate app resource files, largely contradicting common beliefs. We also find that piggybacking is done with little sophistication, in many cases automatically, and often via library code.

  This work has led to a technical report entitled "Understanding Android App Piggybacking", which is currently under submission to an international peer-reviewed journal.

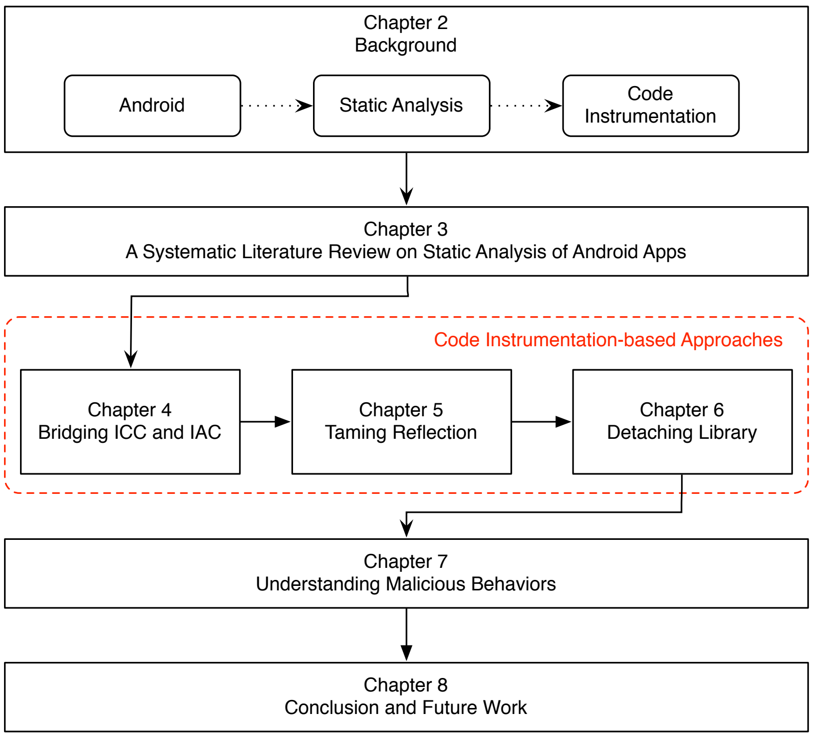


Figure 1.2: Roadmap of This Dissertation.

1.4 Roadmap

Figure 1.2 illustrates the roadmap of this dissertation. Chapter 2 gives a brief introduction on the necessary background information, including Android, static analysis and code instrumentation. In Chapter 3, we present an SLR on static analysis of Android apps, summarizing the state-of-the-art works that perform static security analysis for Android apps. In Chapters 4, 5, and 6, we present three different approaches, which bridge inter-component communication (ICC) and inter-app communication (IAC), tame reflection, and detach library respectively, for boosting static analysis of Android apps. It is worth to mention that all of these three approaches have leveraged code instrumentation to solve the challenges of statically analyzing Android apps. Thanks to these approaches, we are able to automatically identify malicious apps. However, it is still unknown how malicious payloads are introduced into those malicious apps. As a perspective for our future research, we then conduct an in-depth analysis on piggybacked apps (whose malicious payloads are easily identifiable) in Chapter 7. Finally, in Chapter 8, we conclude this dissertation and discuss some potential future works.

2 Background

In this chapter, we provide the preliminary details that are necessary to understand the purpose, techniques and key concerns of the various research works that we have conducted in this dissertation. Mainly, we revisit some details of the Android programming model, summarize the general aspect of static analysis, and delineate the idea of code instrumentation, respectively.

Contents

2.1 Android

Android is a software system that is presented for phones, tablets, and more generally for any kinds of smart devices. It was originally developed by a company called Android Inc. in the early 2000's. Google acquired it in 2005 and made a first public release in 2008, which was running Android 1.0. Since then, there has been over 20 versions released[i]. Each version has been specified with an API level to manage app compatibility between those different versions. Indeed, as the Android platform evolves to include new features and address new hardware requirements, new Android versions are released with each version being assigned a unique integer identifier, called *API level*. In practice, each release of Android is referred to by multiple names: (1) its version number (e.g., Android 4.4); (2) its API level (e.g., API level 19); and (3) a name of sweet (e.g., KitKat). Figure 2.1 overviews the adoption of API levels by millions of Android-powered devices using Google Play official store as of March 2016[ii].

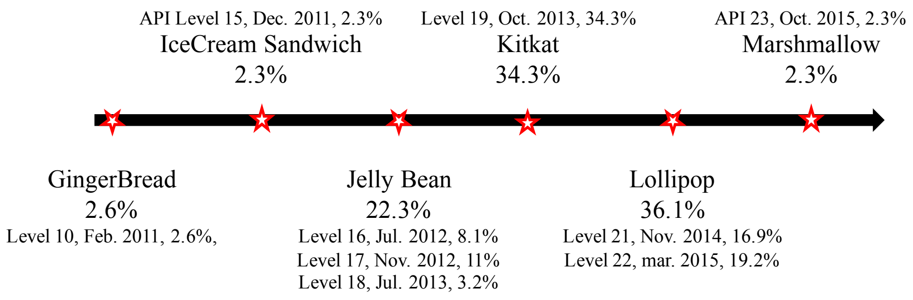


Figure 2.1: Distributions of API Levels Supported by Android-powered Devices (Versions with less than 1% are not Shown).

It is noteworthy that, approximately every 6 months, the API level is upgraded, suggesting substantial updates in the APIs set provided to Android developers. In any case, at a given time, an Android device supports exactly one API level which represents the version of the libraries that developer apps can invoke at runtime. The API level is even used to determine whether an application is compatible with an Android framework version prior to installing the application on the device.

Android is actually a software stack that features four software layers (cf. Figure 2.2). From bottom to top, they are

- Linux kernel layer. This layer provides not only operating system (OS) services, but also drivers that allow the OS to access the low-level hardware.
- Android runtime and native libraries layer. Android runtime features a Dalvik virtual machine (DVM) that executes Android apps' Dalvik bytecode, like Java virtual machine (JVM) for executing Java bytecode. Native libraries are shipped with native code that can be executed by the CPU directly and thus can provide fast solutions for time-intensive functionality.
- Application framework layer. This layer provides an interface for Android apps to access the system resources. As an example, *LocationManager* is presented in this layer to support Android apps retrieving GPS coordinates of a device. In development mode, this layer actually plays the role of the Android SDK, where Android apps are developed upon.
- Applications layer. This layer supports all the running apps, including system apps such as *Home*, *Contact*, etc. that are usually shipped with Android devices, and third-party apps such as *Facebook*, *Twitter*. that can be installed by Android users on-demand form Android markets (e.g., from Google Play, the official market maintained by Google).

[i]https://en.wikipedia.org/wiki/Android_version_history
[ii]`http://developer.android.com/about/dashboards/index.html`

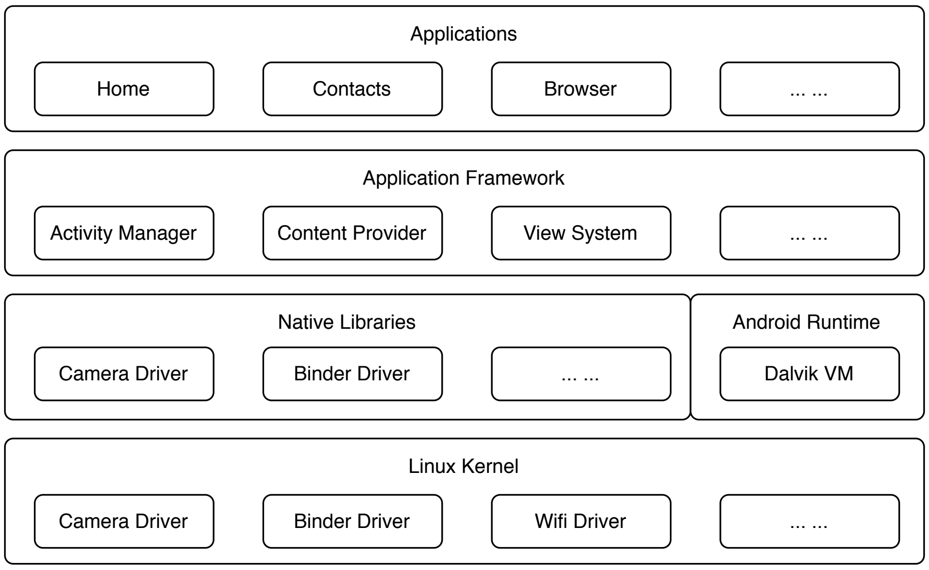


Figure 2.2: Android Architecture.

Our objective in this dissertation is to perform security analysis of Android apps. Therefore, our main focus is on the Android applications layer. In order to better explain this layer, in the next sub-section, we present more details on the structure of Android apps. Since we are going to solve the challenges caused by ICC and Reflection for static analysis, we give more details about them in Section 2.1.2 and Section 2.1.3 respectively.

2.1.1 App Structure

Android apps are written in the Java programming language on traditional desktop computing platforms. Before released, the Java code will be transformed to Dalvik bytecode and eventually assembled to an Android Application Package (APK) file, which in fact is a zip archive file. Table 2.1 enumerates the structured files of a given APK file. The most important files relating to static analysis of Android apps are the *classes.dex* file and the *AndroidManifest.xml* file. The file *classes.dex* contains the real app code, i.e., the implementation of app functionality, which is the main focus of this dissertation, we thus detail it independently in the next sub-section. We now briefly explain the structure of the *AndroidManifest.xml* file.

Table 2.1: The File Structures of an Android App.

Structured Files	Description
AndroidManifest.xml	App configuration like the declaration of components, permissions, etc.
classes.dex	Dalvik bytecode, generated from Java code
resources.arsc	Compressed resource file
META-INF/	Meta-data related to the APK file contents
res/	Resource directory, storing files like image, layout, etc.
assets/	Data directory, storing files that will be compiled into APK file

Listing 2.1 illustrates a simplified example of an Android manifest file, which is extracted from an app named *ActivityCommunication3* of DroidBench[iii]. There are several main attributes that are important for static analysis of Android apps. We now summarize them as follows:

- **package.** The *package* attribute (cf. line 3) depicts the name of a given Android app. Ideally, this name should be unique in the world of Android apps. Thus, by using this name, it is

[iii] `https://github.com/secure-software-engineering/DroidBench`

```
 1 <?xml version="1.0" encoding="utf-8"?>
 2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
 3 package="de.ecspride"
 4 android:versionCode="1"
 5 android:versionName="1.0" >
 6 
 7 <uses-sdk
 8 android:minSdkVersion="8"
 9 android:targetSdkVersion="17" />
10 <uses-permission android:name="android.permission.READ_PHONE_STATE" />
11 
12 <application
13 android:allowBackup="true"
14 android:icon="@drawable/ic_launcher"
15 android:label="@string/app_name"
16 android:theme="@style/AppTheme" >
17 <activity
18 android:name="edu.mit.icc_componentname_class_constant.OutFlowActivity"
19 android:label="@string/app_name" >
20 <intent-filter>
21 <action android:name="android.intent.action.MAIN" />
22 <category android:name="android.intent.category.LAUNCHER" />
23 </intent-filter>
24 </activity>
25 <activity
26 android:name="edu.mit.icc_componentname_class_constant.InFlowActivity"
27 android:label="@string/app_name" >
28 <intent-filter>
29 <action android:name="edu.mit.icc_component_class_constant.ACTION" />
30 <category android:name="android.intent.category.DEFAULT" />
31 </intent-filter>
32 </activity>
33 </application>
34 </manifest>
```

Listing 2.1: Example of an Android Manifest.

possible for static analyzers to localize same apps.

- **sdkVersion.** The manifest leverages the *uses-sdk* element to express an app's compatibility with one or more versions of Android platform. The *uses-sdk* element features three attributes: *minSdkVersion*, *targetSdkVersion*, and *maxSdkVersion*. Attributes *minSdkVersion* and *maxSdkVersion* are designated to specify the minimum, and maximum API level required for a given app to run. Attribute *targetSdkVersion* specifies the targeted version that a given app targets, which informs the Android system that the app is tested against the target version and the system should not enable any compatibility features to maintain forward-compatibility. Therefore, when developers build their apps, they can specify in the Manifest file included in the Android package (apk) the *target* API level that the app is built to run on as well as the *minimum* API that is required to run the app and the *maximum* API level on which the app can run (we will give more details later.). These values help the SDK tools warn developers on the use of APIs. The flexibility of supporting a range of API levels is further put in place to ensure a larger compatibility of apps with devices in the wild.

- **permission.** A permission is specified in the manifest through the *uses-permission* element. Each declared permission indicates that the app is allowed to perform some permission-protected actions. As an example, permission *READ_PHONE_STATE*, declared in line 11, allows the app to read the phone state information like device id, which otherwise is forbidden.

- **components.** Android apps are made up of components, and those components should normally be declared in the manifest, except the rare cases that components are registered dynamically (in app code). Figure 2.3 illustrates the four different types of components: 1) *Activity*, which represents the visible part of Android apps, the user interfaces. Listing 2.1 has actually declared two Activities: *OutFlowActivity* (line 19) and *InFlowActivity* (line 27); 2) *Service*, which is dedicated to execute (time-intensive) tasks in the background; 3) *Broadcast Receiver*, which waits to receive user-specific events as well as system events (e.g., the phone is rebooted); 4) *Content Provider*, which acts as a standard interface for other components/apps to access structured data.

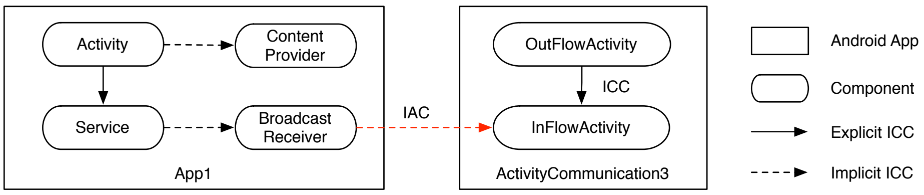


Figure 2.3: Overview of Basic Concepts of Android Apps.

2.1.2 Inter-Component Communication (ICC)

Android components communicate with one another through specific methods, such as *startActivity()*, which are used to trigger inter-component communications (ICC). ICC methods take an Intent object as a parameter which includes information about the target component that the source component wants to communicate with. There are two types of ICC interactions:

- *explicit* ICC where the intent object contains the name of the target component (e.g., explicit ICC example in Listing 2.2)
- *implicit* ICC where the intent object specifies the capability (or action) that the target component must have (e.g., implicit ICC example in Listing 2.2).

```
//Explicit ICC, ActivityCommunication3 (OutFlowActivity)
ComponentName comp = new ComponentName(getPackageName(),
    InFlowActivity.class.getName());
Intent i = new Intent().setComponent(comp);
startActivity(i);

//Implicit ICC, App1 (Broadcast Receiver)
Intent i = new Intent();
i.setAction("edu.mit.icc_component_class_constant.ACTION");
startActivity(i);
```

Listing 2.2: Example Snippet for Triggering ICC Calls.

In the case of implicit ICC, it must specify an *Intent Filter* in its Manifest configuration file, declaring what kind of Intents it is capable of handling, i.e., what kind of actions it can perform or what kind of data it can process. Let us take Listing 2.1 again as an example, there are two *Intent Filter*s defined: one at lines 21-24, which is declared for Activity *OutFlowActivity* and another at lines 29-32, which is declared for Activity *InFlowActivity*. *Intent Filter*s can declare either a system action predefined by the Android system or a user action specifically defined by developers. Action *action.MAIN* (line 22) is actually a system action, which states that *OutFlowActivity* is able to handle the *app launch* event, i.e., *OutFlowActivity* is the entry point of the app. In contrast, *icc_component_class_constant.ACTION* (line 30) is a user-defined action.

Inter-App Communication (IAC) in Android shares the same mechanism of ICC. The only difference is that components which communicate with one another are now in different apps. As an example, the red dot line between *App1* and *ActivityCommunication3* in Figure 2.3 illustrates an IAC. The source code that triggers this IAC is presented in Listing 2.2.

2.1.3 Reflection

Reflection is a property that, in some modern programming languages, enables a running program to examine itself and its software environment, and to change what it does depending on what it finds [74]. In Java, reflection is used as a convenient means to handle genericity or to process Java annotations inside classes. Along with many Java features, Android has inherited the Java Reflection APIs which are packaged and included in the Android SDK for developers to use.

We have parsed Android developer blogs and reviewed some apps to understand when developers need to inspect and determine program characteristics at runtime leveraging the Java reflection feature. Through this process, we have found 4 typical ways leveraging reflective calls.

```java
//Example (1): providing genericity
Class collectionClass;
Object collectionData;
public XmlToCollectionProcessor(Str s, Class c) {
collectionClass = c;
Class c1 = Class.forName("java.util.List");
if (c1 == c) {
this.collectionData = new ArrayList();
}
Class c2 = Class.forName("java.util.Set");
if (c2 == c){
this.collectionData = new HashSet();
}}

//Example (2): maintaining backward compatibility
try {
Class.forName("android.speech.tts.TextToSpeech");
} catch (Exception ex) {
//Deal with exception
}

//Example (3): accessing hidden/internal API
//android.os.ServiceManager is a hidden class.
Class c = Class.forName("android.os.ServiceManager");
Method m = c.getMethod("getService", new Class[] {String.class});
Object o = m.invoke($obj, new String[] {"phone"});
IBinder binder = (IBinder) o;
//ITelephony is an internal class.
//The original code is called through reflection.
ITelephony.Stub.asInterface(binder);
```

Listing 2.3: Reflection Usage in Real Android Apps.

Providing Genericity. Just like in any Java-based software, Android developers can write apps by leveraging reflection to implement generic functionality. Example (1) in Listing 2.3) shows how a real-world Android app implements genericity with reflection. In this example, a fiction reader app, *sunkay.BookXueshanfeihu* (`4226F8`[iv]), uses reflection to produce the initialization of Collection List and Set.

Maintaining Backward Compatibility. In an example case, app *com.allen.cc* (`44B232`, an app for cell phone bill management) exploits reflection techniques to check at runtime the *targetSdkVersion* of a device, and, based on its value, to realize different behaviors. A similar use scenario consists in checking whether a specific class exists or not, in order to enable the use of advanced functionality whenever possible. For example, the code snippet (Example (2) in Listing 2.3), extracted from app *com.gp.monolith* (`61BF01`, a 3D game app), relies on reflection to verify whether the running Android version, includes the text-to-speech module. Such uses are widespread in the Android community as they represent the recommended way[v] of ensuring backward compatibility for different devices, and SDK versions.

[iv]In this work, we represent an app with the last six letters of its sha256 code.

[v]`http://android-developers.blogspot.com/2009/04/backward-compatibility-for-android.html`

Reinforcing App Security. In order to prevent simple reverse engineering, developers separate their app's core functionality into an independent library and load it dynamically (through reflection) when the app is launched: this is a common means to obfuscate app code. As an example, developers usually dynamically load code containing premium features that must be shipped after a separate purchase.

Accessing Hidden/Internal API. In development phase, Android developers write apps that use the `android.jar` library package containing the SDK API exposed to apps. Interestingly, in production, when apps are running on a device, the used library is actually different, i.e., richer. Indeed, some APIs (e.g., *getService()* of class `ServiceManager`) are only available in the platform SDK as they might still be unstable or were designed only for system apps. However, by using reflection, such previously hidden APIs can be exploited at runtime. Example (3), found in a wireless management app –*com.wirelessnow* (`314D51`)–, illustrates how a hidden API can be targeted by a reflective call.

2.2 Static Analysis

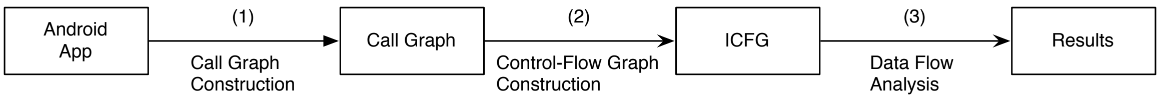


Figure 2.4: Static Analysis Overview.

Static analysis of Android apps generally involves an automated approach that takes as input the binary code (or source code in some cases) of Android apps, examines the code without executing it, and yields results by checking the code structure, the sequences of statements, and the data propagation between method calls. The main advantage of static analysis is that it can reveal bugs, errors, or vulnerabilities long before the software is released to the public. Figure 2.4 illustrates the typical process of statically analyzing Android apps: (1) Call Graph (CG) Construction; (2) Control-Flow Graph (CFG) Construction; and (3) Data Flow Analysis. We now detail them respectively.

Because Android supports the object-oriented programming scheme with the Java language, in the remainder of this section we focus on the analysis of Object-Oriented programs. Such programs are made of classes, each representing a concept (e.g. a car) and including a set of fields to represent other objects (e.g., wheels) and a set of methods containing code to manipulate objects (e.g, drive the car forward). The code in a method can call other methods to create instances of objects or manipulate existing objects.

2.2.1 Call Graph (CG) Construction

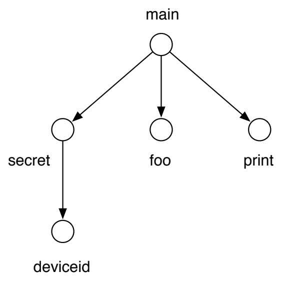
A program usually starts with a single entry point referred to in Java as the `main` method. A quick inspection of the main method's code can list the method(s) that it calls. Then, iterating this process on the code of the called methods leads to the construction of a directed graph (e.g., see Figure 2.5), commonly known as the *call graph* in program analysis. Although the concept of a call graph is standard in static analysis approaches, different concerns, such as precision requirements, may impact the algorithms used to construct a program's call graph. For Java programs, a number of popular algorithms have been proposed, including CHA [58], RTA [19], VTA [215], Andersen [5], Steensguard [208], etc., each being more or less sensitive to different properties of program executions. We detail some of the main properties in Table 2.2 to allow a clear differentiation between research works in the literature.

Table 2.2: A Summary of Different Sensitivities that CG Constructors may Take into Account.

Sensitivity	Explanation
Context	Each method has only a single model of parameter and return value.
Object	The caller objects are taken into account, i.e., the invocation of methods made on different objects are distinguished.
Flow	Methods are aware of their calling orders.
Path	The execution paths are taken into account (for conditional statements).

```
void main() {
  int x = secret();
  int y = 0;
  y = foo(x);
  print(y);
}
int secret() {
  int q = deviceid();
  return q;
}
int foo(int p) { return p; }
```

a Source Code.



b Call Graph.

Figure 2.5: Source Code of a Simple Example and its Call Graph Generated from the main Method.

2.2.2 Control-Flow Graph (CFG) Construction

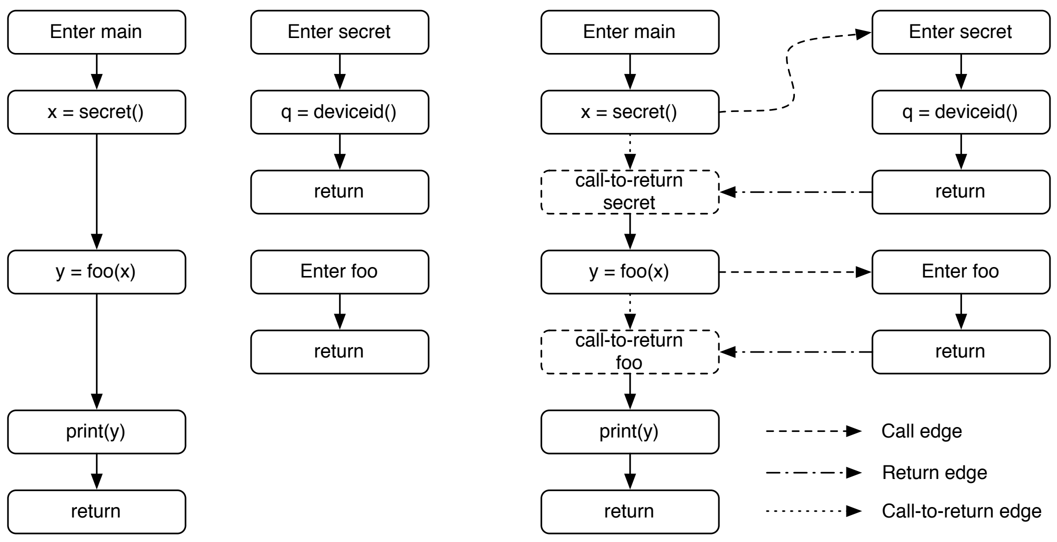
CFG differs from CG in a way that it considers not only the method calls but also the statements of each method call. CFG is frequently used in programming analysis to represent a program, where statements and their sequences are usually represented as nodes and directed edges, respectively. Let us take the program shown in Figure 2.5a as an example, based on the previous definition, we could build three CFGs with one for each of its user-defined method (cf. Figure 2.6a). Those CFGs are built through single method base, which is thus insufficient to represent Android apps that involves frequently invocations from one method to another.

To record the invocation mechanism of methods, the concept of inter-procedural control-flow graph (ICFG) is introduced. In addition to the original edges (hereinafter we call it as **Normal edges**), ICFG introduces three new types of edges to connect intra-procedural CFGs. In order to better represent those newly introduced edges, ICFG further introduce a new node, namely call-to-return node, for each method call node. As an example, node $y = foo(x)$ in method *main*'s CFG will now be represented as $y = foo(x) \rightarrow call - to - return : foo$ (cf. Figure 2.6b).

- **Call edge:** connecting caller method node to callee method node and passing information like method arguments between them.

- **Return edge:** connecting callee method node to caller method's call-to-return node and passing return value between them.

- **Call-to-return edge:** connecting caller method node to call-to-return node and passing information from the caller method to all its successor nodes. This edge typically pass information that does not concern the callee node. The purpose of introducing this edge type, along with the call-to-return node, is actually to avoid loops. Imaging there is no call-to-return node, the return edge would directly point to the caller method node, which can traverse again into the callee method node through a call edge, and then back to the caller method node through return edge, resulting in a loop.



a CFG. **b** ICFG.

Figure 2.6: CFG and ICFG of the Example Code Shown in Figure 2.5a.

2.2.3 Data-Flow Analysis

A data-flow analysis is a technique to compute at every point the possible values of program variables. Generally, data flow analyses use a so-called equation system to compute information at each statement. In equation system, a statement s can be represented as $s := < in, out, gen, kill >$, where $in, out, gen, kill$ are four sets of possible values. We now depict them:

- $in(s)$ presents the set of information that are valid before statement s.
- $out(s)$ presents the set of information valid after s.
- $gen(s)$ presents a set of newly created values.
- $kill(s)$ presents a set of existing values that are killed by s.

Let $succs(s)$ and $preds(s)$ be statement s's immediate successor and predecessor sets respectively. Based on the aforementioned definition, we can compute $in(s)$ and $out(s)$ through Formula 2.1. The function $f : in(s) \rightarrow out(s)$ that transfers $in(s)$ to $out(s)$ is called *transfer function* (or *flow function*).

$$\begin{aligned} in(s) &= \cup_{s' \in preds(s)} out(s') \\ out(s) &= gen(s) \cup (in(s) \setminus kill(s)) \end{aligned} \tag{2.1}$$

Note that Formula 2.1 works only for forward data-flow analysis. For backward data-flow analysis, we need to propagate values from $out(s)$ to $in(s)$, whose values can be computed through Formula 2.2.

$$\begin{aligned} out(s) &= \cup_{s' \in succs(s)} in(s') \\ in(s) &= kill(s) \cup (out(s) \setminus gen(s)) \end{aligned} \tag{2.2}$$

Actually, different data-flow problems may need to compute their equation system values differently (as illustrated by Formula 2.1 and Formula 2.2). As a result, different data-flow analyzers need to be implemented from scratch. To mitigate this, Reps, Horwitz and Sagiv present IFDS framework, attempting to solve the inter-procedural data-flow analysis problem in a generic manner. IFDS framework has worst-case complexity $O(ED^3)$, where E stands for the number of edges analyzed and D is the size of the analysis domain. The main idea behinds IFDS framework is to reduce any data-flow analysis problem to a simple graph-reachability problem.

Many data-flow problems such as "reaching definitions", "possibly uninitialized variables", and "available expressions" can ready be modeled and solved by IDFS framework. However, some problems that involves "environment" information cannot be encoded with IFDS, because the set of data-flow facts could be infinite. As an example, when a data fact from domain D is reached at a given statement, it may introduce other values from a secondary domain V. This relationship is the so-called environment and is denoted as $Env(D, V)$. To further support this kind of data-flow problem, the same authors present IDE framework, an extension of IFDS that effectively allows a data-flow analysis to extend the reachability problem to a value-computation problem. The IDE framework can be taken as a generalization of IFDS framework, where all IFDS problems can be represented as IDE problems.

2.3 Code Instrumentation

Code instrumentation is a widely used mechanism in different software engineering areas. It can be leveraged for reducing the analysis complexity of programs, for creating profilers and debuggers, for detecting program errors at runtime, or for securing programs through inline reference monitoring.

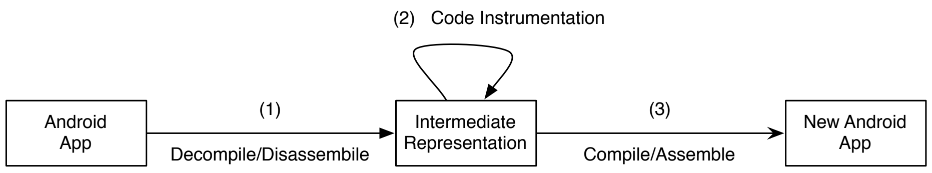


Figure 2.7: Code Instrumentation Overview.

In general, code instrumentation involves removing statements and adding new statements. Updating statements can be achieved by first removing them from the context and then adding the updated versions at the same place. Figure 2.7 illustrates the typical process of performing code instrumentation for Android apps. The intermediate representation can be different formats, as long as there are tools supporting the disassemble/assemble process. Let us take SMALI code[vi] as an example, it is frequently used as intermediate representation (IR) for reverse engineering Android apps, where apktool is leveraged to disassemble/assemble Android apps. Unfortunately, to the best of our knowledge, there is no tool that supports automatic instrumentation of SMALI code, making the instrumentation a manual activity, preventing the corresponding instrumentation from being market-scale.

Jimple code [219] is another frequently used IR for instrumenting Android apps. The disassemble/assemble process can be realized by Soot [115], with the help of Dexpler [24]. Moreover, contrary to SMALI code, Jimple-based instrumentation can be conducted automatically with the help of Soot, making it appropriate for instrumenting large set of apps. Therefore, in this dissertation, we select Jimple to perform our code instrumentation.

In this dissertation, we aim at boosting existing static analyses through code instrumentation. The reason why we choose code instrumentation is that it provides a non-invasive solution, which not only solves the analyzing problem but also makes the solution generic (be reusable) that can benefit many existing static analyzers without even modifying them. Let us take Figure 2.8 to better explain this. Assuming our goal is to detect a data-flow from method m_1 to m_4 with going through m_2 and m_3 in Android apps. Because of some challenges (e.g., ICC, reflection), there is no connection between

[vi]`https://github.com/JesusFreke/smali`

method m_2 and m_3 in the default control-flow graph, say, the default output of Soot. As a result, many existing static analyses, which simply leverage the CFG computed by Soot, would not be able to achieve our goal. At a high level, we find three different ways to tackle the aforementioned analysis problem, which are illustrated in Figure 2.8 with approach (A), (B), and (C), respectively.

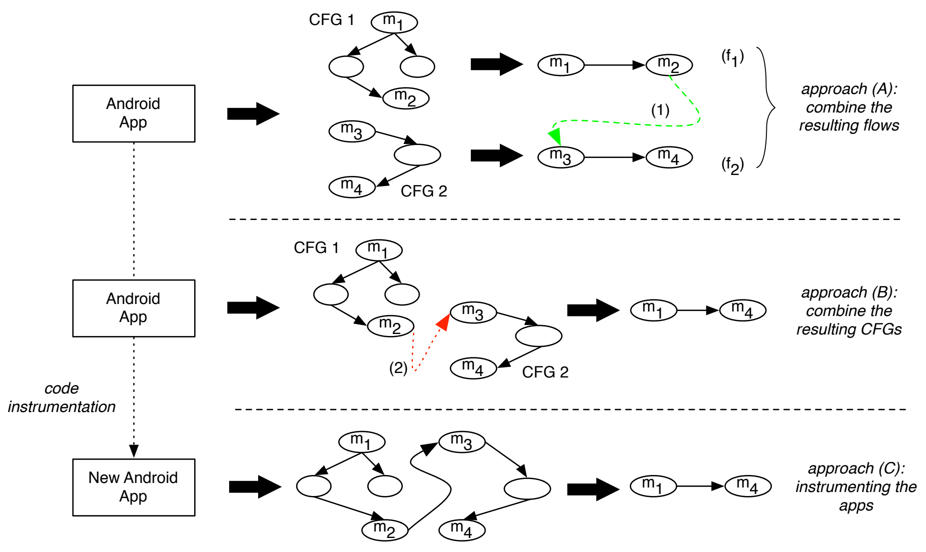

CFG 1
m1
m2
m3
m4
CFG 2
Android App
(f1)
(1)
(f2)
approach (A): combine the resulting flows
approach (B): combine the resulting CFGs
(2)
code instrumentation
New Android App
approach (C): instrumenting the apps

Figure 2.8: Benefit of Performing Code Instrumentation for Static Analysis.

In approach (A), the results of different (intra-CFG) data flows (i.e., flow f_1 and f_2) are combined to yield an inter-CFG data flow. However, because the combination is performed after the analysis, no context data (e.g., variable values such as data handled between m_2 and m_3) is available, therefore the approach requires to approximate the flows between the CFGs (e.g., here line (1)). This approach may thus lead to a significant number of false positives.

Approach (B) is an improved version of (A), it is no longer the results combined but the CFGs instead. This approach thus supports a context-sensitive inter-CFG analysis by operating on a combined CFG and on a data dependence graph between different CFGs. Nevertheless, such an approach cannot be generalized to any instance of CFGs. In practice for example, the workload for combining CFGs generated by Soot cannot even be applied in the case of CFGs generated by WALA. Hence, specific development effort must be put into each and every static analyzer to support inter-CFG analysis.

Approach (C) considers the caveats of all previous approaches by further improving approach (B) to yield a **general** approach for enabling **context-sensitive** *inter-CFG* analysis of Android apps. To this end, approach (C) solves the previous problem by instrumenting the app bytecode. More specifically, it adds auxiliary code to explicit connect m_2 to m_3, imitating the behavior through normal method calls. The generated new app (after code instrumentation) is thus immediately ready for analysis by existing static tools (which previously cannot).

To summarize, code instrumentation is capable to boost existing static analysis in a non-invasive manner to perform more extensive analysis. Furthermore, the instrumentation itself is generic (can be reused to support any static analyzer) and the output supports even context-sensitive analysis. Because of these benefits, in this dissertation, we focus on code instrumentation for static analysis.

3 An SLR on Static Analysis of Android Apps

In this SLR, we aim at providing a clear view on the state-of-the-art works that statically analyze Android apps, from which we highlight the trends of static analysis approaches, pinpoint where the focus has been put and enumerate the key aspects where future researches are still needed. Our in-depth examination on 124 identified research papers has led to several key findings: 1) Static analysis is largely performed to uncover security and privacy issues; 2) The Soot framework and the Jimple intermediate representation are the most adopted basic support tool and format, respectively; 3) Taint analysis remains the most applied technique in research approaches; 4) Most approaches support several analysis sensitivities, but very few approaches consider path-sensitivity; 5) There is no single work that has been proposed to tackle all challenges of static analysis that are related to Android programming; and 6) Only a small portion of state-of-the-art works have made their artifacts publicly available.

This chapter is based on the work published in the following technical report:

- Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static Analysis of Android Apps: A Systematic Literature Review. *Technical Report, ISBN 978-2-87971-150-8 TR-SNT-2016-3*, 2016

Contents

3.1 Introduction

As Android apps pervade all user activities, ill-designed and malicious apps have become big threats that can lead to damages of varying severity (e.g., app crashes, financial losses with malware sending premium-rate SMS, reputation issues with private data leaks). Data from anti-virus vendors and security experts regularly report on the rise of malware in the Android ecosystem. For example, G DATA has reported that the 560,671 new Android malware samples collected in the second quarter of 2015 revealed a 27% increase, compared to the malware distributed in the first quarter of the same year[i].

To deal with the aforementioned threats, the research community has investigated various aspects of Android development, and proposed a wide range of program analyses to identify syntactical errors and semantic bugs [183], to discover sensitive data leaks [10, 124], to uncover vulnerabilities [158, 181], etc. In most cases, these analyses are performed statically, i.e., without actually running the Android app code, in order not only to ensure scalability when targeting thousands of apps in stores, but also to guarantee a traversal of all possible execution paths. Unfortunately, static analysis of Android programs is not a trivial endeavour since one must account for several specific features of Android, to ensure both soundness and completeness of the analysis. Common barriers to the design and implementation of performant tools include the need to support Dalvik bytecode analysis or translation, the absence of a main entry point to start the call graph construction and the constraint to account for event handlers through which the whole Android program works. Besides these specific challenges, Android carries a number of challenges for the analysis of Java programs, such as how to resolve the targets of Java reflection statements and deal with dynamic code loading. Thus, despite much efforts in the community, state-of-the-art tools are still challenged by their lack of support for some analysis features. For example, the state-of-the-art FlowDroid [10] taint analyzer cannot track data leaks across components since it is unaware of the Android Inter-Component Communication (ICC) scheme. More recent tools which focus on ICC calls may not account for reflective calls.

Because of the various of concerns in static analysis of Android apps, it is important for the field, which has already produced substantial amount of approaches and tools, to reflect on what has already been done, and on what remains to do. Although a recent survey [213] on securing Android has mentioned some well-known static analysis approaches, a significant part of current works has been skipped from the study. Furthermore, the study only focused on general aspects of Android security research, neglecting basic characteristics about the static analyses used, and missing an in-depth analysis of the support for some Android-specific features (e.g., *XML Layout*, or *ICC*).

This review is an attempt to fulfill the need of a comprehensive study for static analysis of Android apps. To reach our goal, we performed a systematic literature review (SLR) on all the influential state-of-the-art approaches. After identifying thoroughly the set of related research publications, we perform a trend analysis and provide a detailed overview on key aspects of static analysis of Android apps such as the characteristics of static analysis, the Android-specific features, the addressed problems (e.g. security or energy leaks) and also some evaluation-based empirical results. Finally, we summarize the current limitations of static analysis of Android apps and point out potential future research directions.

The main contributions of this section are:

- We build a comprehensive and searchable repository[ii] of research works dealing with static analysis for Android apps. These works are categorized following several criteria related to their support of common analysis characteristics as well as Android-specific concerns.

[i]https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q2_2015_EN.pdf

[ii]Repository available at: http://lilicoding.github.io/SA3Repo

- We analyze in detail key aspects of static analysis to summarize the research efforts and the reached results.
- We further enumerate the limitations of current static analysis approaches (on Android) and provide insights for potential future research directions.
- Finally, we provide a trend analysis on this research field to report on the latest focus as well as the level of maturity in key analysis problems.

The remainder of this chapter is organized as follows: Section 3.2 introduces the methodology applied by this SLR, where research questions, search strategies, primary publication selections, etc. are detailed. Section 3.3 briefly outlines the dimensions this SLR investigates and Section 3.4 summarizes the corresponding findings for each dimension enumerated. Section 3.5 discusses different aspects raised by the research and practice of static Android app analysis. Finally, in Section 3.6 we introduce some closely related work and we close this chapter with conclusion in Section 3.7.

3.2 Methodology for the SLR

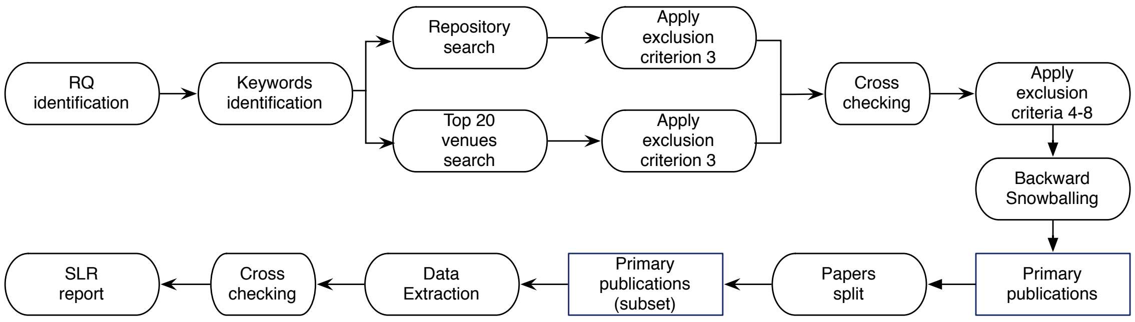


Figure 3.1: Overview of our SLR Process.

The methodology that we follow for this SLR is based on the guidelines provided by Kitchenham [111]. Figure 3.1 illustrates the protocol that we have designed to conduct the SLR:

- In the first step we define the research questions motivating this SLR, and subsequently identify the relevant information to collect from the publications in the literature (cf. Section 3.2.1).
- Then, we enumerate the different search keywords that allow us to crawl the possibly largest set of relevant publications within the scope of this SLR (cf. Section 3.2.2).
- The search process itself is conducted following two scenarios: the first one considers the well-known publication repositories, while the second one focuses on the lists of publications from top venues, including both conferences and journals (cf. Section 3.2.2).
- To limit our study to very relevant papers, we apply exclusion criteria on the search results, thus filtering out papers of likely limited interest (cf. Section 3.2.3).
- Then we merge the sets of results from both search scenarios to produce the overall list of publications to review. We further consolidate this list by applying another set of exclusion criteria based on the content of the papers' abstracts (cf. Section 3.2.3).
- Finally, we perform a lightweight backward-snowballing on the selected publications. The final list of papers is hereafter referred to as *primary publications/studies* (cf. Section 3.2.4).

Given the large number of publications relevant to the systematic literature review that we undertake to conduct, we must devise a strategy of review which guarantees that each publication is investigated thoroughly and that the extracted information is reliable. To that end, we further proceed with the following steps:

- First, we assign the primary publications to the authors of this SLR who share the heavy workload of paper examinations.
- Then, each primary publication is fully reviewed by the SLR author to whom it is attributed. Based on their reviews, each SLR author must fill a common spreadsheet with relevant information in categories that have been previously enumerated.
- To ensure that the review information for a given paper is reliable, we first cross-check these reviews among reviewers. Then, once all information is collected, we engage in a *self-checking process* where we forward our findings to the authors of the reviewed papers. These authors then confirm our investigation results or demonstrate any inaccuracies in the classifications.
- Eventually, we report on the findings to the research community.

3.2.1 Research Questions

This SLR aims to address the following research questions:

RQ1: What are the purposes of the Analyses? With this research question, we will survey the various issues targeted by static analysis, i.e., the concerns in the Android ecosystem that researchers attempt to resolve by statically analyzing app code.

RQ2: How are the analyses designed and implemented? In this second research question, we study in detail the depth of analysis that are developed by researchers. To that end, we investigate the following sub-questions:

RQ 2.1: What code representations are used in the analysis process? To facilitate analysis with existing frameworks, analyses often require that the app byte code is translated back to Java or other intermediate representations.

RQ 2.2: What fundamental techniques are used by the community of static analysis of Android apps?

RQ 2.3: What sensitivity-related metrics are applied?

RQ 2.4: What Android-specific characteristics are taken into account?

RQ3: Are the research outputs available and usable? With this research question, we are interested in whether the developed tools are readily available to practitioners and/or the reported experiments can be reproduced by other researchers. For each technical contribution, we check if the data sets used in the validation of approaches are available, and if the experimental protocol is described in detail.

RQ4: What challenges remain to be addressed? Finally, with this fourth research question we survey the issues that have not yet benefited from a significant research effort. To that end, we investigate the following questions:

RQ 4.1: To what extent are the enumerated analysis challenges covered? We survey the proportion of research approaches that account for reflective calls, native code, multi-threading, etc.

RQ 4.2: What are the trends in the analyses? We study how the focus of researchers evolved over time and whether this correlates with the needs of practitioners.

3.2.2 Search Strategy

We now detail the search keywords and the datasets that we leverage for the search of our relevant publications.

3.2.2.1 Search keywords

Thanks to the research questions outlined in Section 3.2.1, we can summarize our search terms with keywords that are (1) related to analysis activities, or (2) related to key aspects of static analysis, and (3) related to the target programs. Table 3.1 depicts the actual keywords that we used based on a manual investigation of some relevant publications.

Table 3.1: Search Keywords.

Line	Keywords
1	Analysis; Analyz*; Analys*;
2	Data-Flow; "Data Flow*"; Control-Flow; "Control Flow*"; "Information-Flow*"; "Information Flow*"; Static*; Taint;
3	Android; Mobile; Smartphone*; "Smart Phone*";

Our search string s is formed as a conjunction of the three lines of keywords, i.e., $s =: l_1$ **AND** l_2 **AND** l_3, where each line is represented as a disjunction of its keywords, e.g., $l_1 =: \{$*Analysis* **OR** *Analyz** **OR** *Analys**$\}$.

3.2.2.2 Search datasets

As shown in Fig. 3.1, our data search is based on repositories and is complemented by a check against top venues in software engineering and security. Repository search is intended for finding the relevant publications, while the top venue check is used only as an additional checking process, to ensure that the repository search has not missed any major publication. We now give more details of them separately.

Repository Search To find datasets of publications we first leverage five well-known electronic repositories, namely ACM Digital Library[iii], IEEE Xplore Digital Library[iv], SpringerLink[v], Web of Knowledge[vi] and ScienceDirect[vii]. Because in some cases the repository search engine imposes a limit to the amount of search result (e.g., meta-data) that can be downloaded, for such cases, we split the search string and iterating until we collect all relevant meta-data of publications. For example, SpringerLink only allows to collect information on the first 1,000 items from its search results. Unfortunately, by applying our predefined search string, we get more than 10,000 results on this repository. Consequently, we must split our search string to narrow down the findings and afterwards combine all the findings into a final set. In other cases, such as with ACM Digital Library, where the repository search engine does not provide a way to download a batch of search results (meta-data), we resort to python scripts to scale up the collection of relevant publications.

[iii] `http://dl.acm.org`
[iv] `http://ieeexplore.ieee.org`
[v] `http://link.springer.com`
[vi] `http://apps.webofknowledge.com`
[vii] `http://www.sciencedirect.com`

Top Venue Check A few conference and journal venues, such as the Network and Distributed System Security Symposium, have policies (e.g., open proceedings) that make their publications unavailable in the previously listed electronic repositories. Thus, to ensure that our repository search results are, to some extent, exhaustive, we consider all publications from well-known venues. For this SLR we have considered the top[viii] 20 venues: 10 venues are from the field of software engineering and programming languages while the other 10 venues are from the security and privacy field. Table 3.2 lists these venues considered at the time of review, where some venues on cryptography fields (including EUROCRYPT, CRYPTO, TCC, CHES and PKC), parallel programming (PPoPP), magazines (such as IEEE Software) and non-official proceedings (e.g., arXiv Cryptography and Security) are excluded, because those venues are mainly not the focuses of static analysis of Android apps. The *H5-index* in Table 3.2 is defined by Google Scholar as a special h-index where only those of its articles published in the last 5 complete calendar years (in our case is from 2010 to 2014) are considered. The h-index of a publication is the largest number h such that at least h articles in that publication were cited at least h times each. Intuitively, the higher *H5-index*, the better the venue.

Table 3.2: The Top 20 Venues Including Both Conference Proceedings and Journals in SE/PL and SEC fields.

Acronym	Full Name	H5-index
	Software Engineering and Programming Languages (SE/PL)	
ICSE	International Conference on Software Engineering	57
TSE	IEEE Transactions on Software Engineering	47
PLDI	SIGPLAN Conference on Programming Language Design and Implementation	46
IST	Information and Software Technology	45
POPL	ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages	45
JSS	Journal of Systems and Software	41
FSE	Foundations of Software Engineering	38
OOPSLA	Conference on Object-Oriented Programming Systems, Languages, and Applications	34
ISSTA	International Symposium on Software Testing and Analysis	31
TACAS	International Conference on Tools and Algorithms for the Construction and Analysis of Systems	31
	Security and Privacy (S&P)	
CCS	ACM Conference on Computer and Communications Security	65
S&P	IEEE Symposium on Security and Privacy	53
SEC	USENIX Security Symposium	51
TISSEC	IEEE Transactions on Information Forensics and Security	47
NDSS	Network and Distributed System Security Symposium	39
TDSC	IEEE Transactions on Dependable and Secure Computing	39
ASIACRYPT	International Conference on The Theory and Application of Cryptology and Information Security	34
COMPSEC	Computers & Security	34
ACSAC	Annual Computer Security Applications Conference	29
SOUPS	Symposium On Usable Privacy and Security	29

Our top 20 venues check is performed on DBLP[ix]. We only use such keywords that are listed in line 3 in Table 3.1 for this search. As DBLP provides papers' title only, it is not necessary for us to use the same keywords that we use in the repository search step. Ideally, all the papers that are related to smartphones (including Android, Windows, iOS and so on) are taken into account. As a result, this coarse-granularity strategy has introduced some irrelevant papers (e.g., papers that analyze iOS apps). Fortunately, because of the small number of venues, we are able to manually exclude those irrelevant papers from our interesting set, more details are given in the next section.

[viii]Following Google Scholar Metrics: `https://scholar.google.lu/citations?view_op=top_venues&hl=en`
[ix]http://dblp.uni-trier.de

3.2.3 Exclusion Criteria

The search terms provided above are purposely broad to allow the collection of a near exhaustive list of publications. However, this broadness also suggests that many of the search results may actually be irrelevant. We thus propose the following exclusion criteria to allow us focus only on the most relevant publications.

1. First to account for the common language spoken by the reviewers, and the fact that most of today's scientific works are published in English, we filter out *non-English* written publications.

2. Second, we want to focus on extensive works with detailed publications. Thus, we exclude *short papers*, i.e., heuristically, papers with less than 7 pages in LNCS single-column format or with less than 5 pages in IEEE/ACM-like double-column format. Further, it should be noted that such papers are often preliminary work that are later published in full format and are thus likely to be included in the final set.

3. Third, related to the second exclusion criteria, we attempt to identify *duplicate papers* for exclusion. Usually, those are papers published in the context of a conference venue and extended to a journal venue. We look for such papers by first comparing systematically the lists of authors, paper titles and abstract texts. Then we manually check whether suspicious pairs of identified papers share a lot of content or not. When duplication is confirmed we filter out the less extensive publication.

4. Fourth, because our search terms include "mobile" to collect most papers, the collected set includes papers about "mobile networking" or iOS/Windows platforms. We exclude such *non Android-related* papers. This exclusion criterion allows to remove over half of the collected papers, now creating the opportunity for an acceptable manual effort of assessing the relevancy of remaining papers.

5. Last, we quickly skim through the remaining papers and exclude those that target Android but *do not deal with static analysis techniques*. For example, publications about dynamic analysis/testing of Android apps are excluded.

Since Android has been a hot topic in the latest years, the set of relevant papers constituted after having applied the above exclusion criteria is still large. These are all papers that propose approaches relevant to Android, based on static analysis of the apps. We have noticed that some of the collected papers 1) do not contribute to the research on static analysis of Android apps, or 2) simply grep or match API names. For example, some approaches simply read the manifest file to list permissions requested, or simply attempt to match specific API names in function calls. Thus, we devise four more exclusion criteria to filter out such publications:

6. We exclude papers that statically analyze Android Operating System (OS) rather than Android apps. Because our main focus in this survey is to survey works related to static analysis of Android apps. As examples, we have dismissed PSCout [12] and EdgeMiner [39] in this review because they are dedicated to analyzing the Android framework.

7. We dismiss papers that do not actually parse the app program code, e.g., research papers that only perform static analysis on the meta-data (i.e., descriptions, configuration files.) of apps are excluded.

8. We filter out technical reports, such as SCanDroid [75]. Such non-peer-reviewed papers are often re-published in a conference and journal venue, and are thus likely to be included in our search set with a different title and author list. For example, the technical report paper on IccTA [128] has eventually appeared in a conference proceeding [124], which was indeed considered by this SLR.

Table 3.3: Summary of the Selection of Primary Publications. The Total Number of Searched Publications are Given only after the Merge Step.

Steps	IEEE	ACM	Springer	Elsevier	Web of Knowledge	Top-20-venues	Total
Search results	1,048	387	11,826	2,416	655	155	-
Scripts verification (with same keywords)	369	302	70	17	453	155	-
Scripts exclusion (criterion 3)	264	289	57	16	453	155	-
Merge							1123
After reviewing title/abstract (criteria 4 → 5)							302
After skimming full paper (criteria 6, 7 and 8)							142
After final discussion							118
Author recommendation							+4
Backward Snowballing							+2
Total							124

9. We also dismiss papers that simply leverage the statement sequences, or grep API names from the source/app code. As an example, we exclude Juxtapp [89], a clone detection approach, from consideration since it simply takes opcode sequences for its static analysis.

3.2.4 Backward Snowballing

As recommended in the guidelines of Kitchenham and Charters [111], we perform a lightweight[x] backward snowballing from reference lists of the articles identified via repository and top-venue search. The objective is to find additional relevant papers in the literature that may have not been matched via our search keywords.

It is tedious and time-consuming to perform this step manually. Thus, we leverage python scripts to automatically extract references from our selected primary publications. In particular, we first leverage *pdfx*[xi] to transfer a given paper from pdf to text format. Then, we retrieve all the references from the previously generated text files. To further mitigate manual efforts, we again use scripts to filter out references whose publication date fall outside of our SLR timeline and whose titles appear without keywords that are defined in the scope of this SLR. After these steps, we manually read and check the remaining references. If a given reference (paper) is in line with this work but is not yet available in our primary publication set, we then include it into our final paper set.

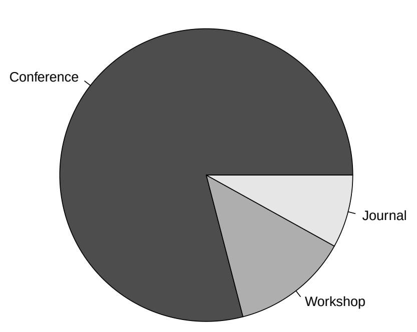
3.2.5 Primary publications selection

In this section, we give details on our final selection results of primary publications, which are summarized in Table 3.3.

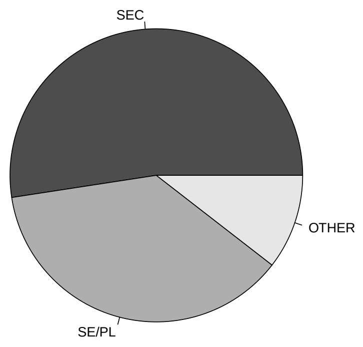
The first two lines (search results and scripts verification) provide statistics on papers found through the keywords defined previously. In the first line, we focus on the output from the repository search (with full paper search, whenever possible, because we want to collect as many relevant papers as possible in this phase). Through this repository search, we collect data such as *paper title* or *paper abstract*. The second line shows the results of an additional verification step on the collected data. More specifically, we perform automated keywords search on the data (with exactly the same keywords as the previous step). We adopt this second step because of the flaws in "advanced" search functionality of the five repositories, where the search results are often inaccurate, resulting in a set noised by some irrelevant papers. After performing the second step (line 2), the number of potential relevant papers is significantly reduced.

The third line shows the results of applying our exclusion criterion 3 (to exclude short papers) for the results of line 2. The only big difference happens in IEEE repository. We further look into the

[x] We only perform backward-snowballing once, meaning that the newly found publications are not considered for snowballing.
[xi] https://github.com/metachris/pdfx



a Type.



b Domain.

Figure 3.2: Statistics of Examined Publications.

publications of IEEE and found that those short papers are mostly 4 page conference papers with insufficient description of their proposed approaches/tools. Therefore it makes sense for us to remove those short papers from our interesting set.

Line 4 merges the results of line 3 to one set in order to avert redundant workload (otherwise, a same paper in two repositories would be reviewed twice). We have noticed that the redundancy occurs in three cases:

1. The ACM repository may contain papers that originally published in IEEE or Springer.
2. The Web of Knowledge repository may contain papers that published in Elsevier.
3. The five repositories may contain papers that appear in the top-20-venues set.

After the merge step, we split the searched papers to all the co-authors of this review. We manually review the title/abstract and examine the full content of the selected papers, with applying the exclusion criteria 4-8. With final discussion between authors, we finally select 118 papers as primary publications.

For the self-checking process, we have collected 343 distinct email addresses of 419 authors for the 88 primary publications selected in our search (up to May 2015). We then send the SLR to these authors and request their help in checking if their publications are correctly classified. Within a week, we have received 25 feedback messages which we take into account. Authors have further recommended a total of 19 papers to include in the SLR. Fifteen of them are already considered by our SLR (from Jul. 2015 to Dec. 2015). The remaining 4 of those recommended papers are found to be borderline with regards to our exclusion criteria (e.g., dynamic approaches which resort to some limited static analyses). We review them against our inclusion criteria and decide to include them (Table 3.3, line 8).

Regarding the backward-snowballing, overall, we have identified 1815 referenced articles whose publication date fall within our SLR timeline. Only 53 of these articles have titles without keywords that could be matched by search. We reviewed these papers and found that only 2 fit our criteria (i.e., deal with static analysis in the context of Android apps).

In total, our SLR examines 124 publications (the full list is shown in Table 3.7 and Table 3.8). Fig. 3.2 illustrates the distributions of these publications by types(Fig 3.2a) and publication domains (Fig. 3.2b). Over 70% of our collected papers are published in conferences. Workshops, generally co-located with top conferences, have also seen as a fair share of contributions on Android static analysis. These findings are not surprising, since the high competition in Android research forces authors to aim for targets where publication process is fast. Presentations in conferences can also

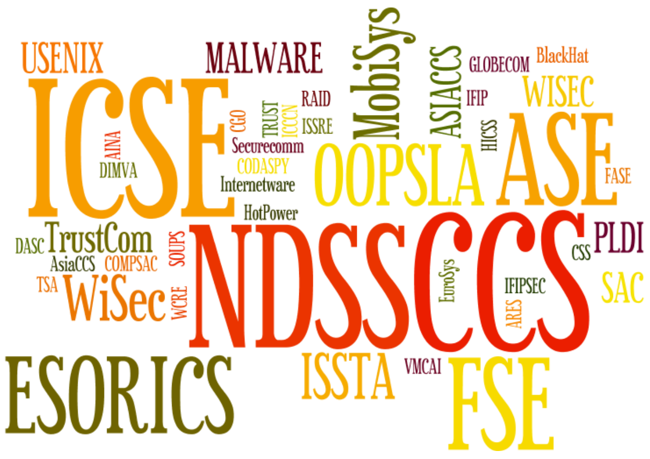


Figure 3.3: Word Cloud of All the Conference Names of Examined Publications. Journal and Workshop Publications are not Considered.

stimulate more collaborations as can be seen in most recent papers on Android analysis. We further find that half of the publications were made in Security venues while another larger proportion was published in Software Engineering venues. Very few contributions were published in other types of venues. MIGDroid [97], published in the *Network* domain and CloneCloud [50], published in the *Systems* domain, are the rare exceptions. This finding comforts our initial heuristics of considering top venues in SE/PL and SEC to verify that our repository search is exhaustive.

Fig. 3.3 shows a word cloud of the conference names where our primary publications were presented. Most of the reviewed publications are from top conference venues (e.g., ICSE, NDSS and CCS), suggesting that the state-of-the-art research has been included at a substantial scale.

3.3 Data Extraction

Once relevant papers have been collected, we build a taxonomy of the information that must be extracted from each paper in order (1) to cover the research questions enumerated above, (2) to be systematic in the assessment of each paper, and (3) to provide a baseline for classifying and comparing the different approaches. We now overview the metrics extracted from the selected publications:

Targeted Problems. Approaches are also classified on the targeted problems. Examples of problems include privacy leakage, permission management, energy optimization, etc.

Fundamental Techniques. This dimension focuses on the fundamental techniques adopted by examined primary publications. The fundamental techniques in this work include not only such techniques like *taint analysis* and *program slicing* that solve problems through different means but also the existing tools such as Soot or WALA that are leveraged.

Static Analysis Metrics. This dimension includes static analysis related metrics. Given a primary publication, we would like to check whether it is context-sensitive, flow-sensitive, path-sensitive, object-sensitive, field-sensitive, static-aware, implicit-flow-aware, and alias-aware. Besides, in this dimension, the dynamic code loading, reflection supporting, native code supporting are also inspected.

Android Characteristics. This dimension includes such metrics that are closely related to Android such as ICC, IAC (Inter-App Communication) and so on. Questions like "Do they take care of the lifecycle or callback methods?" or "Do the studied approaches support ICC or IAC?" belong to this dimension.

Evaluation Metrics. This dimension focuses on the evaluation methods of primary publications, intending to answer the question how their approaches are evaluated. To this end, this dimension will count whether their approaches are evaluated through in-the-lab apps (i.e., artificial apps with knowing the ground truth in advance) or in-the-wild apps (i.e., the real-world apps). Questions like how many in-the-wild apps are evaluated are also addressed in this dimension.

3.4 Summary of Findings

We now report on the findings of this SLR in light of the research questions enumerated in Section 3.2.1.

3.4.1 Purposes of the Analyses

In the literature of Android, static analysis has been applied for achieving various tasks. Among others, such analysis is implemented to highlight various security issues (such as private data leaks or permission management concerns), to verify code, to compare apps for detecting clones, to automate the generation of test cases, or to assess code efficiency in terms of performance and energy consumption. We have identified eight recurring purposes of Android-targeted static analysis approaches in the literature. We detail these purposes and provide statistics of approaches that target them.

Private Data Leaks. Recently, concerns on privacy with Android apps have led researchers to focus on the private data leaks. FlowDroid [10], introduced in 2014, is probably the most advanced approach addressing this issue. It performs static taint analysis on Android apps with a flow-, context-, field-, object-sensitive and implicit flow-, lifecycle-, static-, alias-aware analysis, resulting in a highly precise approach. The associated tool has been open-sourced and many other approaches [13,112,124,125,189] have leveraged it to perform more extensive analysis.

Vulnerabilities. Security vulnerabilities are another concern for app users who must be protected against malware exploiting the data and privileges of benign apps. Many of the vulnerabilities addressed in the literature are related to the ICC mechanism and its potential misuses such as for component hijacking (i.e., gain unauthorized access to protected or private resources through exported components in vulnerable apps) or intent injection (i.e., manipulate user input data to execute code through it). For example, CHEX [158] detects potential component hijacking-based flows through reachability analysis on customized system dependence graphs. Epicc [181] and IC3 [180] are tools that propose static analysis techniques for implementing detection scenarios of inter-component vulnerabilities. Based on this, PCLeaks [125] goes one step further by performing sensitive data-flow analysis on top of component vulnerabilities, enabling it to not only know what is the issue but also to know what sensitive data will leak through that issue. Similar to PCLeaks, ContentScope [258] detects sensitive data leaks focusing on Content Provider-based vulnerabilities in Android apps.

Permission Misuse. Permission checking is a pillar in the security architecture of Android. The Android permission-based security model associates sensitive resources with a set of permissions that must be granted before access. However, as shown by Bartel et al. [23, 25], this permission model is an intrinsic risk, since apps can be granted more permissions than they actually need. Malware may indeed leverage permissions (which are unnecessary to the core app functionality) to achieve their malicious goals. PSCout [12] is currently the most extensive work that dissects the Android permission specification from Android OS source code using static analysis. However, we do not take PSCout into consideration in this SLR as our focus is static analysis of Android apps rather than Android OS.

Energy Consumption. Battery stand-by time has been a problem for mobile devices for a long time. Larger screens found in modern smartphones constitute the most energy consuming components. As shown by Li et al. [119], modern smart phones use OLED, which consumes more energy when displaying light colors than dark colors. In their investigation, the energy consumption could be reduced by 40% if more efficient web pages are built for mobile systems (e.g., in dark background). To reach this conclusion, they performed extensive program analysis on the structure of web apps, more specifically, through automatically rewriting web apps so as to generate more efficient web pages. Li et al. [117] present a tool to calculate source line level energy consumption through combining program analysis and statistical modeling.

Other Purposes. Besides the four aforementioned concerns, state-of-the-art works have also targeted less hot topics, often about highly specific issues. More representatively, works such as CryptoLint [64] and CMA [206] have leveraged static analysis to identify cryptography implementation problems in Android apps. Researchers have extended the Julia [183] static analyzer to perform code verification through formal analyses of Android programs. Other works such as AppIntent [238] and SIG-Droid [171] are proposed to automatically generate test cases for Android apps. Researchers have also leverage static analysis to perform clone detection of Android apps. As examples, DNADroid [53] and AnDarwin [54] are proposed to compute the similarity of apps based on program dependency graphs extracted from Android apps.

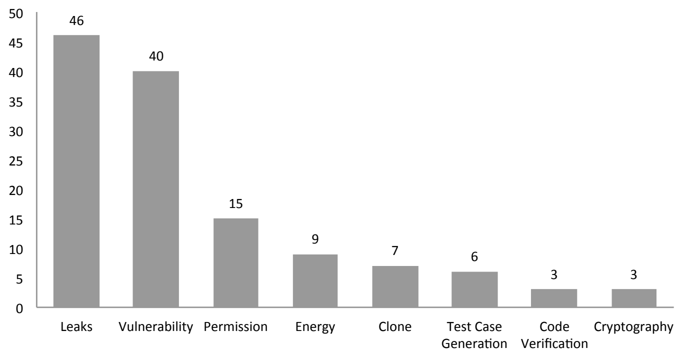


Figure 3.4: Statistics of the Analyzed Purposes Addressed by Examined Publications.

The summary statistics in Fig. 3.4 show that Security concerns are the focus of most static analysis approaches for Android. Energy efficiency is also a popular concern ahead of test case generation.

RQ 1*: Static analysis is largely performed on Android programs to uncover security and privacy issues.*

3.4.2 Form and Extent of Analysis

We now investigate how the analyses described in the literature are implemented. In particular, we study the support tools that they leverage (Section 3.4.2.1), the fundamental analysis methods that they apply (Section 3.4.2.2), the sensitivities supported by their analysis (Section 3.4.2.3) as well as the Android peculiarities that they deal with (Section 3.4.2.4).

3.4.2.1 Code Representations and Support Tools

Table 3.4 enumerates the recurrent tools that are used by approaches in the literature to support their analyses. Such tools often come as off-the-shelf components that implement common analysis processes (e.g., for the translation between bytecode forms or for the automatic construction of call-graphs). The table also provides for each tool information on the intermediate representation (IR)

Table 3.4: List of Recurrent Support Tools for Static Analysis of Android Apps.

TOOL	Brief Description	IR	Example Usages
Soot [115]	A Java/Android static analysis and optimization framework	JIMPLE, JASMIN	FlowDroid [10], IccTA [124], ApplIntent [238]
WALA[a]	A Java/Javascript static analysis framework	WALA-IR (SSA-based)	AsDroid [100], Asynchronizer [146], ORBIT [234]
Chord [210]	A Java program analysis platform	CHORD-IR (SSA-based)	CloneCloud [50]
Androguard [62, 63]	Reverse engineering, malware/-goodware analysis of Android apps	DEX_ASSEMBLER	MalloDroid [70], Relda [87]
Ded [177]	A DEX to Java bytecode translator	JAVA_CLASS	Enck et al. [68]
Dare [179]	A DEX to Java bytecode translator	JAVA_CLASS	Epicc [181], IC3 [180]
Dexpler [24]	A DEX to Jimple translator	JIMPLE	BlueSeal [205]
Smali/Baksmali[b]	A DEX to Smali translator (and verse visa)	SMALI	Woodpecker [85], SEFA [229]
Apktool[c]	A tool for reverse engineering Android apps	SMALI	PaddyFrog [228], Androlizer [28]
dex2jar[d]	A DEX to Java bytecode translator	JAVA_CLASS	DroidChecker [42], Vekris et al. [220]
dedexer[e]	A disassembler for DEX files	DEX_ASSEMBLER	Brox [161], AQUA [109]
dexdump	A disassembler for DEX files	DEX_ASSEMBLER	ScanDal [110]
dx	A Java bytecode to DEX translator	DEX_ASSEMBLER	EdgeMiner [39]
jd-gui[f]	A Java bytecode to source code translator (and also an IDE)	JAVA_CLASS	Wang et al. [225]
ASM [37, 113]	A Java manipulation and analysis framework	JAVA_CLASS	COPES [23]
BCEL[g]	A library for analyzing and instrumenting Java bytecode	JAVA_CLASS	vLens [117], Julia [183], Nyx [119]
Redexer	A reengineering tool that manipulates Android app binaries	DEX_ASSEMBLER	Brahmastra [30]

[a]http://wala.sourceforge.net
[b]http://baksmali.com
[c]http://ibotpeaches.github.io/Apktool/
[d]https://github.com/pxb1988/dex2jar
[e]http://dedexer.sourceforge.net
[f]https://github.com/java-decompiler/jd-gui
[g]https://commons.apache.org/bcel/

that it deals with. The IR is a simplified code format to represent the original Dalvik bytecode and facilitate processing since Android Dalvik itself is known to be complex and challenging to manipulate. Finally, the table highlights the usage of such tools in specific reviewed approaches.

Table 3.5 goes into more details on the adoption of the different code representations by the examined approaches. Fig. 3.5 summarizes the frequency of usages, where *Jimple*, which is used by the popular Soot tool, appears as the most used IR followed by the *Smali* intermediate representation, which is used by Apktool.

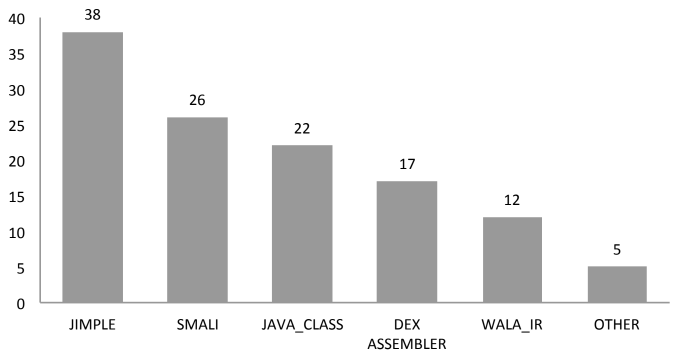


Figure 3.5: Distribution of Code Representations Used by Examined Publications.

RQ 2.1: *The Soot framework and the Jimple intermediate representation are the most adopted basic support tool and format for static analysis of Android apps.*

Table 3.5: Summary of Examined Approaches through the Code Representations They Use.

Code Representation	Publications
OTHER	SIG-Droid [171], Cortesi et al. [52], Amandroid [226], FUSE [192], Nyx [119]
SMALI	MobSafe [231], I-ARM-Droid [57], A3 [160], MIGDroid [97], SMV-Hunter [207], Androlizer [28], Wognsen et al. [227], SmartDroid [253], Woodpecker [85], CredMiner [260], SEFA [229], Uranine [191], AppCaulk [202], ContentScope [258], Apparecium [216], Zuo et al. [261], ClickRelease [170], SAAF [94], CMA [206], Anadroid [142], DroidSim [214], Chen et al. [45], SADroid [88], AdRisk [84], PaddyFrog [228], Jensen et al. [105]
JIMPLE	DescribeMe [243], DEvA [198], Capper [246], AppContext [235], DroidSafe [81], Violist [118], Bartel et al. [25], HelDroid [6], Gator [195], A5 [222], Gator2 [233], FlowDroid [10], Lotrack [143], DidFail [112], TASMAN [11], ACTEve [4], DroidJust [47], AppSealer [245], ApkCombiner [123], Bastani et al. [27], PerfChecker [152], EcoDroid [102], W2AIScanner [93], Vekris et al. [220], Apposcopy [73], COPES [23], Gator3 [233], IC3 [180], Sufatrio et al. [212], Epicc [181], Gallingani et al. [76], IccTA [124], android-app-analysis-tool [78], BlueSeal [205], AutoPPG [241], WeChecker [55], Covert [20], AppIntent [238]
DEX_ASSEMBLER	Mann et al. [165], Brox [161], Androguard2 [63], Androguard [62], MalloDroid [70], AndRadar [151], Dflow+DroidInfer [101], AQUA [109], Brahmastra [30], Redexer [106], Relda [87], CryptoLint [64], StaDynA [251], Lin et al. [144], Adagio [77], ComDroid [48], Scandal [110]
JAVA_CLASS	Chen et al. [44], AppAudit [230], PermissionFlow [201], Bartsch et al. [26], EvoDroid [163], Lu et al. [159], DroidAlarm [254], AsyncDroid [145], DroidChecker [42], DroidSIFT [244], eLens [90], CloneCloud [50], DPartner [248], Wang et al. [225], TrustDroid [250], Choi et al. [49], IFT [69], Pathak et al. [182], Pegasus [46], SIF [91], vLens [117], Julia [183]
WALA_IR	AnDarwin [54], Poeplau et al. [185], Asynchronizer [146], CHEX [158], ORBIT [234], AAPL [157], AndroidLeaks [79], A3E [14], THRESHER [31], AsDroid [100], Hopper [32], DNADroid [53]

3.4.2.2 Fundamental Analysis Methods

While all reviewed approaches build on control-flow or data-flow analyses, specific techniques are employed to enhance the results and thus to reach the target purposes. In our study, we have identified six fundamental techniques which are used, often in conjunction, in the literature.

Abstract Interpretation. Abstract interpretation is a theory of approximating the semantics of programs, where soundness of the analysis can be guaranteed and thereby to avoid yielding false negative results. A concrete implementation of abstract interpretation is through formal program analysis. As an example, Julia [183] is a tool that uses abstract interpretation to automatically static analyze Java and Android apps for the development of high-quality, formally verified products. SCanDal [110], another sound and automatic static analyzer, also leverages abstract interpretation to detect privacy leaks in Android apps.

Taint Analysis. Taint analysis is implemented by at first tainting a data if it is defined to be sensitive, and then tracking it through data-flow analysis. If a tainted data flows to a point where it should not be, specific instructions can be applied, e.g., to stop this behavior and report it to the users. As an example, AppSealer [245] leverages taint analysis to automatically generate patches for Android component hijacking attacks. When a tainted data is going to violate the predefined policies, AppSealer injects a patch before the violation to alert the app user through a pop-up dialog box. FlowDroid [10], as another example, performs static taint analysis to detect sensitive data leaks. Based on a predefined set of *source* and *sink* methods, which are automatically extracted from the Android SDK (cf. SUSI [188]), sensitive data leaks are reported if and only if the data are obtained from *source* methods (i.e., when these data are tainted) and eventually flow to *sink* methods (i.e., when violating security polices).

Symbolic Execution. Symbolic execution is an approach to generate possible program inputs, which is different from abstract interpretation, because it assumes symbolic values for inputs rather than obtaining actual inputs as abstract interpretation does. As an example, AppIntent [238] uses symbolic execution to generate a sequence of GUI manipulations that lead to data transmission. As the basic straightforward symbolic execution is too time-consuming for Android apps, AppIntent leverages the unique Android execution model to reduce the search space without sacrificing code coverage.

Program Slicing. Program slicing has been used as a common means in the field of program analysis to reduce the set of program behaviors, meanwhile to keep the interesting program behavior unchanged. Given a variable v in program p that we are interested in, a possible slice would consist of all statements in p that may affect the value of v. As an example, Hoffmann et al. [94] present a framework called SAAF to create program slices so as to perform backward data-flow analysis to track parameter values for a given Android method. CryptoLint [64] computes static program slices that terminate in calls to cryptographic API methods, and then extract the necessary information from these slices.

Code Instrumentation. Static analysis is often necessary to find sweet spots where to insert code for collecting runtime behaviour data (cf. SIF framework [91]). In recent works, code instrumentation has also been employed to address challenges for static analysis in Android apps, including artificially linking components for inter-component communication detection [124], or replacing reflection calls with standard java calls so as to reduce incomplete analyses due to broken control-flows [133]. In Android community, Arzt et al. [9] have introduced several means to instrument Android apps based on Soot. As an example, IccTA [124] instruments Android apps to reduce an inter-component taint propagation problem to an intra-component problem. Nyx [119] instruments android web app to modify the background of web pages, so as to reduce the display power consumption and thereby making web app become more energy efficient. Besides Soot, other tools/frameworks such as WALA and ASM are also able to support instrumentation of Android apps.

Type/Model Checking. Type and model checking are two prevalent approaches to program verification. The main difference between them is that type checking is usually based on syntactic and modular style whereas model checking is usually defined in a semantic and whole-program style. Actually, this difference makes these two approaches complementary to one another: type checking is good at explaining why a program was accepted while model checking is good at explaining why a program was rejected [175]. As an example, COVERT [20] first extracts relevant security specifications from a given app and then applies a formal model checking engine to verify whether the analyzed app is safe or not. For type checking, Cassandra [156] is presented to enable users of mobile devices to check whether Android apps comply with their personal privacy requirements even before installing these apps. Ernst et al. [69] also present a type checking system for Android apps, which checks the information flow type qualifiers and ensures that only such flows defined beforehand can occur at run time.

Table 3.6 provides information on the works that use different techniques. The summary statistics show that taint analysis, which is used for tracking data, is the most applied technique (30.6% of primary publications), while 18.5% primary publications involve code instrumentation and 12.1% primary publications have applied program slicing technique in their approaches. Type/Model checking, abstract interpretation, and symbolic execution are applied in 6.5% primary publications respectively.

RQ 2.2: *Taint analysis remains the most applied technique in static analysis of Android apps. This is in line with the finding in RQ1 which shows that the most recurrent purpose of state-of-the-art approaches is on security and privacy.*

3.4.2.3 Static Analysis Sensitivities

We now investigate the depth of the analyses presented in the primary publications. To that end we assess the sensitivities (cf. Figure 3.6) that different approaches take into account. *Field-sensitivity* appears to be the most considered with 48 primary publications taking it into account. This finding is

Table 3.6: Summary through the Adoption of Different Fundamental Techniques (Only Relevant Works, e.g., with at Least One Check are Presented).

Techniques	Publications	Percentage[xii]
Abstract Interpretation	Mann et al. [165], Anadroid [142], Cortesi et al. [52], Rocha et al. [194], Lu et al. [159], Hopper [32], Scandal [110], Julia [183]	6.5%
Taint Analysis	MobSafe [231], CHEX [158], PermissionFlow [201], DescribeMe [243], Cortesi et al. [52], Capper [246], AppContext [235], DroidSafe [81], DroidChecker [42], HelDroid [6], FlowDroid [10], Lotrack [143], Amandroid [226], DidFail [112], FUSE [192], CredMiner [260], SEFA [229], TASMAN [11], Uranine [191], Mann et al. [165], AppCaulk [202], Brox [161], AppAudit [230], AndroidLeaks [79], Apparecium [216], DroidJust [47], AppSealer [245], Bastani et al. [27], Dflow+DroidInfer [101], W2AIScanner [93], Apposcopy [73], Sufatrio et al. [212], Anadroid [142], TrustDroid [250], AAPL [157], Gallingani et al. [76], IccTA [124], WeChecker [55]	30.6%
Symbolic Execution	W2AIScanner [93], SIG-Droid [171], ACTEve [4], Gallingani et al. [76], ClickRelease [170], TASMAN [11], AppIntent [238], Jensen et al. [105]	6.5%
Program Slicing	MobSafe [231], AppCaulk [202], Brox [161], Poeplau et al. [185], Capper [246], AndroidLeaks [79], Apparecium [216], AppSealer [245], Hopper [32], eLens [90], SAAF [94], AQUA [109], Rocha et al. [194], CryptoLint [64], CredMiner [260]	12.1%
Code Instrumentation	Cassandra [156], Uranine [191], AppCaulk [202], I-ARM-Droid [57], Androguard2 [63], Capper [246], DroidSafe [81], ACTEve [4], AsyncDroid [145], AppSealer [245], Bastani et al. [27], CMA [206], SmartDroid [253], Brahmastra [30], Sufatrio et al. [212], ORBIT [234], Rocha et al. [194], IccTA [124], android-app-analysis-tool [78], DidFail [112], SIF [91], vLens [117], Nyx [119]	18.5%
Type/Model Checking	Mann et al. [165], Choi et al. [49], Lu et al. [159], DroidAlarm [254], IFT [69], Dflow+DroidInfer [101], SADroid [88], Covert [20]	6.5%

reasonable since Android apps are generally written in Java, an Object-Oriented Programming (OOP) language where object fields are pervasively used to hold data. *Context-sensitivity* and *Flow-sensitivity* are also largely taken into account (with 42 and 40 publications respectively). The least considered sensitivity is *Path-sensitivity* (only 6 publications), probability due to the scalability issues that it raises.

In theory, the more sensitivities considered, the more precise the analysis is. Indeed, as shown by Arzt et al. [10], the authors of FlowDroid, who claimed that the design of being context-, flow-, field-, and object-sensitive maximizes precision and recall, i.e., aims at minimizing the number of missed leaks and false warnings. Livshits et al. [154, 155] also stated that both context-sensitivity and path-sensitivity are necessary to achieve a low false positive rate (i.e., higher precision). It is thus reasonable to state that only two approaches, namely TRESHER [31] and Hopper [32], achieves high precision by taking into account all sensitivities. However, given the relatively high precision of existing state-of-the-art works, it seems unnecessary to support all sensitivities to be useful in practice.

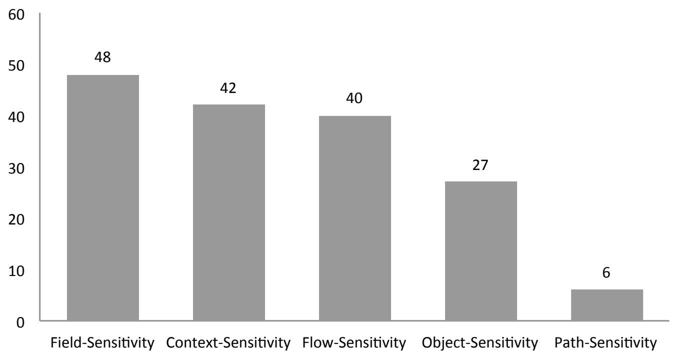


Figure 3.6: Statistics of Sensitivities Considered by the Publications.

RQ 2.3*: Most approaches support up to 3 of the 5 sensitivities in Call-Graph construction for static analysis. Path-sensitivity is the least taken into account by the Android research community.*

3.4.2.4 Android Specificities

Although Android apps are written in Java, they present specific characteristics in their functioning. Typically, they extensively make use of a set of lifecycle event-based methods that the system requires to interact with apps, and rely on the inter-component communication mechanism to make application parts interact. These characteristics however may constitute challenges for a static analysis approach.

Component Lifecycle. Because lifecycle callback methods (i.e., *onStop()*, *onStart()*, *onRestart()*, *onPause()* and *onResume()*) have neither connection among them nor directly with app code, it is challenging for static analysis approaches to build control-flow graphs (e.g., to continuously keep track of sensitive data flows). We found that 57 of the reviewed publications propose static analysis approaches that take into account component lifecyle.

UI Callbacks. Besides lifecycle methods, a number of callbacks are used in Android to handle various events. In particular, UI events are detected by the system and notified to developer apps through callback methods (e.g., to react when a user clicks on a button). There are several such callbacks defined in various Android classes. Similar to lifecycle methods, taking into account such callback methods leads to a more complete control-flow graph. Our review reveals that 64 of examined publications are considering specific analysis processes that take into account callback methods.

EntryPoint. Most static approaches for Android apps must build an entry point, in the form of a dummy main, to allow the construction of call-graph by state-of-the-art tools such as Soot and WALA. Seventy four publications from our set have explicitly discussed their handling of the single entry-point issue.

ICC. The inter-component communication (ICC) is well-known to challenge static analysis of Android programs. Recently, several works have focused on its inner-working to highlight vulnerabilities and malicious activities in Android apps. Among the set of collected primary publications, 30 research papers explicitly deal with ICC. As examples, Epicc [181] and IC3 [180] attempt to extract the necessary information of ICC in Android apps which can support other approaches, including IccTA [124], and DidFail [112] , in performing ICC-aware analyses across components. AmanDroid [226] also resolves ICC information for supporting inter-component data-flow analysis, for the purpose of vetting the security of Android apps.

IAC. The inter-app communication (IAC) mechanism extends the ICC mechanism for components across different apps. Because, most approaches focus on analyzing a single app, IAC-supported analyses are scarce in the literature. We found only 6 publications that deal with such scenarios of interactions. A main challenge of tackling IAC-aware analyses is the support of scalability for market-scale analyses.

XML-Layout. The structure of user interfaces of Android apps are defined by layouts, which can be declared in either XML configurations or Java code. The XML layout mechanism provides a well-defined vocabulary corresponding to the View classes, sub-classes and also their possible event handlers. We found that 30 publications, from our set, take into account XML layouts to support more complete analysis scenarios. Mostly, those analyses of tracking callback methods indicated in XML files but which are not explicitly registered in the app code. For example, the code snippets in Listing 3.1 provide two distinct, but equivalent, implementations for registering a listener to a layout Button. If the XML implementation is retained, function `myFancyMethod` would be considered as dead code for any analyzer not taking into account XML information.

```
//Registration of onClickListener in app code
Button btn = (Button) findViewById(R.id.mybutton);
btn.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
myFancyMethod(v);
}
});

//Registration of onClickListener through XML
<Button android:id="@+id/mybutton"
android:text="Click Me!"
android:onClick="myFancyMethod" />
```

Listing 3.1: Registering a Callback Function for a Button

Overall, Figure 3.7 summarizes the support for addressing the enumerated challenges by approaches from the literature. We illustrate in this figure only those publications from our collected set that are explicitly addressing at least one of the challenges.

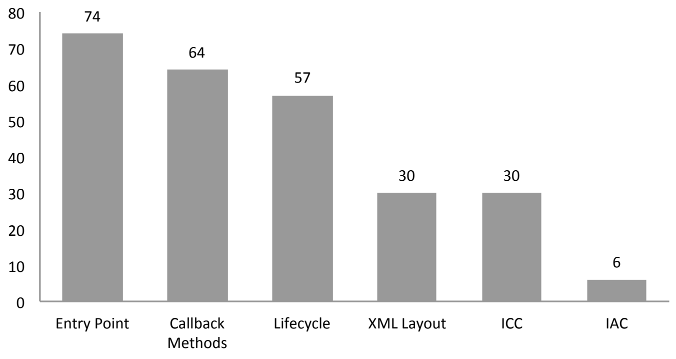


Figure 3.7: Statistics of Android Specificities Addressed by the Publications.

> ***RQ 2.4:*** *Only few works in the literature has proposed to tackle at once all challenges due to Android specificities. Instead, most approaches select to deal partially with those challenges, which are further delivered directly within their implementation (as a whole), leaving little opportunities for reuse by other approaches.*

3.4.3 Usability of Research Output

We now investigate whether the works behind our primary publications have produced usable tools and whether their evaluations are extensive enough to make their conclusions reliable or meaningful.

Among the 124 reviewed papers, 41 (i.e., only 33%) have provided a publicly available tool. This finding suggests that, currently, despite the increasing size of the community working on static analysis of Android apps, sharing of research efforts is still limited. Among those 41 publicly available approaches, 34 of them are further open-sourced.

We now consider how researchers in the field of static analysis of Android apps evaluate their approaches. We differentiate *in-the-lab experiments*, which are mainly performed with a few hand-crafted test cases to highlight efficacy and/or correctness, from *in-the-wild experiments*, which consider a large number of real-world apps to demonstrate efficiency and/or scalability. In-the-lab experiments help to quantify an approach through standard metrics (e.g., precision and recall), which is very difficult to obtain through in-the-wild experiments, because of missing of ground truth. In-the-wild

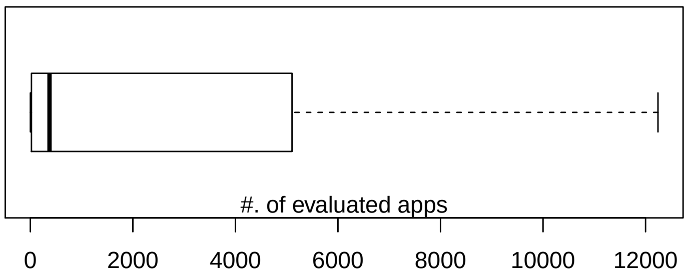
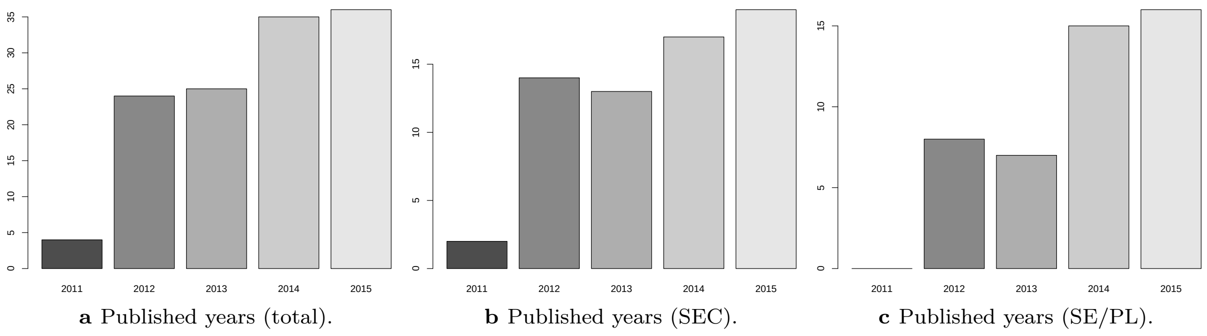


Figure 3.8: Distribution of the Number of Evaluated Apps (for All the Publications Reviewed in This Work).



a Published years (total). **b** Published years (SEC). **c** Published years (SE/PL).

Figure 3.9: Distribution of Examined Publications through Published Year.

experiments are however also essential for static approaches. They are dedicated for finding and possibly solving real problems of real-word apps, which may have already been used by thousands of users. Among all the reviewed approaches, only 7 of them have taken into account in-the-lab and in-the-wild experiments at the same time.

Figure 3.8 further plots the distribution of the number of apps evaluated through in-the-wild experiments by the approaches reviewed in this work. The median number of apps that those in-the-wild experiments consider are 374. The maximum number of evaluated apps is 318,515, which is considered by AndRadar [151].

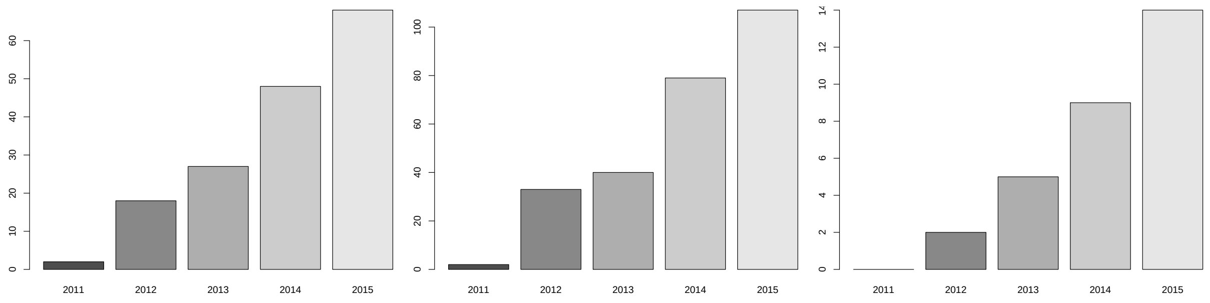
***RQ3**: Only a small portion of state-of-the-art works that perform static analysis on Android apps have made their contributions available in public (e.g., tool support). Among those approaches, only a few have evaluated their approaches in both in-the-lab and in-the-wild experiments.*

3.4.4 Trends and Overlooked Challenges

Although Android is a recent ecosystem, research on analyzing its programs has flourished. We investigate the general trends in the research and make an overview of the challenges that are (or are not) dealt with.

3.4.4.1 Trend Analysis

Fig. 3.9 shows the distribution of publications from our set according to their year of publication. Research papers meeting our criteria appear to have started in 2011, about two and a half years after the first commercial release of Android in September 2008. Then, a rush of papers have ensued for both Security and Software engineering communities.



a Trend of sensitivity. **b** Trend of awareness. **c** Trend analysis.

Figure 3.10: Trend of ICC.

```java
SmsManager sms = SmsManager.getDefault();
//sms.sendTextMessage("+49 1234", null, "123", null, null);
for (int i = 0; i < 123; i++)
sms.sendTextMessage("+49 1234", null, "count", null, null);
```

Listing 3.2: Example of an Implicit Flow.

Fig. 3.10 shows that, as time goes by, research works are considering more sensitivities and addressing more challenges to produce precise analyzers which are aware of more and more analysis specificities. We further look into the ICC challenge for static analyzers to show the rapid increase of related publications.

> ***RQ 4.1:*** *Research on static analysis for Android is maturing, yielding more analysis approaches which consider more analysis sensitivities and are aware of more specificities of Android.*

3.4.4.2 Dealing with Analysis Challenges

We now discuss our findings on the different challenges addressed in analyses to make them static-, implicit-flow, alias-, dynamic-code-loading-, reflection-, native-code-, and multi-threading-aware.

We consider an approach to be static-aware when it takes into account *static* object values in Java program to improve analysis precision. Thirty two approaches explicitly take this into account. Thirty primary publications consider aliases. Both challenges are the most considered in approaches from the literature as they are essential for performing precise static analysis.

We found 23 primary publications which take into account multi-threading. We further investigate these supports since multi-threading is well-known to be challenging even in Java ecosystem. We note that those approaches partially solve multi-threading issues in a very simple manner, based on a predefined whitelist of multi-threading mechanisms. For example, when *Thread.start()* is launched, they simply bridge the gap between method *start()* and *run()* through an explicit call graph edge. However, other complex multi-threading processes (e.g., those unknown in advance) or the synchronization among different threads are not yet addressed by the community of static analysis researchers for Android apps.

Another challenge is implicit flows, i.e., flow information inferred from control-flow dependencies [240]. Let us take Listing 3.2 as an example, if an Android app does not send out message 123 directly, but sends 123 times the word "count", the attacker can actually gain the same information as if the app had directly sent the 123 value.

The remaining challenges include reflection, native code and dynamic code loading (DCL) which are taken into account by 15, 3 and 3 publications respectively.

Fig 3.11 provides information on which challenges are addressed by the studied papers.

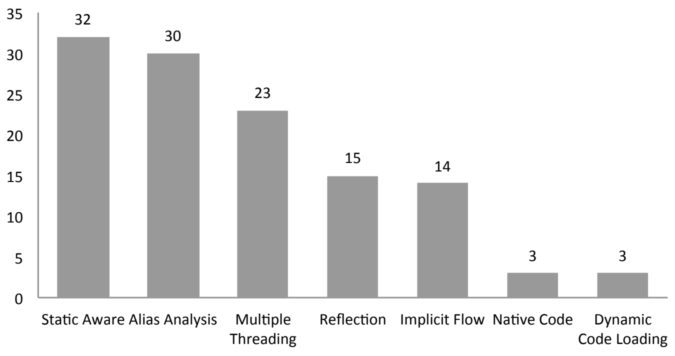


Figure 3.11: Statistics of Different Aspects of Static Analysis.

RQ 4.2: *There are a number of analysis challenges that remain to be addressed further by the community to build approaches that are aware of implicit-flows, dynamic code loading features, reflective calls, native code and multi-threading, so as to implement sound and highly precise static analyzers. As it is hard for a single tool to efficiently address all the challenges we discussed in this section, we suggest that those challenges should be solved in a reusable way so that the forthcoming approaches can directly benefit from them.*

3.5 Discussion on Findings

The findings yielded while investigating the research questions of this SLR constitute as many discussion points around the research and practice of Android.

3.5.1 Security will remain a strong focus of Android research

Static analysis of Android apps, as shown in the investigation of RQ1, is largely focused on uncovering security and privacy issues. This suggests that security and privacy are a big concern for both users and researchers nowadays. Several studies have already shown how the permission system can be misused [25] and more recently how app uninstallation can leave residual data which can be exploited in attacks [247].

On the one hand, the open source nature of Android development code base is central in the interest that it generates in the research community. While it is easy for malware writers to find exploitable security holes, researchers can also more readily collect data, perform experiments and test solutions on this platform.

On the other hand, time-to-market pressure is substantially higher in the mobile ecosystem than in traditional desktop computing, making testing a neglected concern by developers and users.

3.5.2 Considering external code is essential for a sound analysis

There are two types of external code available in Android apps outside the main *classes.dex*: Dalvik bytecode (often hidden via the extension of another file format such as xml) and binary code. Dalvik bytecode can be accessed through reflection and dynamic code loading (DCL) while binary code can be leveraged via the Java native interface (JNI) APIs. Unfortunately, both DCL and native code are rarely considered by the state-of-the-art static analyzers. As a result, current analyses, which do not consider DCL and native code in their implementation, miss the opportunity to discover problems hidden inside external code, leading to incomplete results.

3.5.3 Improving basic support tools such as Soot is a priority

The majority of research works reviewed in this SLR build their analyses based on support tools such as Soot and WALA. Unfortunately, these tools present various limitations. For example, the transformation performed by Soot to translate Dalvik bytecode into Jimple and back to bytecode is still without guarantees that the rebuilt application is runnable (i.e., will not crash during execution). Several approaches and tools that instrument apps, such as AppSealer, are impacted by this limitation. It is thus essential that researchers focus on contributing in improving the static analysis and transformations supported by these support tools. It is noteworthy that a small improvement in the performance of these tools (e.g., providing more precise call graph) will benefit many more research approaches (e.g., all the approaches relying on call graph construction).

3.5.4 Sensitivities are key in the trade-off between precision and performance

The more sensitivities a static analysis takes into account, the more precise its results will be. However, as showcased in FlowDroid [10], this precision will come at the cost of performance: execution then becomes time consuming, and may even fail on more corner case apps. Limiting the sensitivities may yield some false positives and false negatives, but will produce an approach that could be successfully applied at the scale of markets. For instance, an imprecise but fast approach could be used to quickly filter a huge set of applications. The remaining application set, which should be much smaller than the initial set, is then analyzed using a more precise, but much slower, analysis.

3.5.5 Android specificities can help improve code coverage

Android specificities such as lifecycle awareness are a must to not miss any code of the application related to individual Android components. For an ICC-aware static analysis, handling the inter-component communication precisely would increase the connectivity of the call-graph and would thus reduce the overall analysis time. Indeed, some connection between components would now be recognized as over-approximations and will no longer be taken into account by the analysis.

The analysis for Android specificities are often intertwined with the main static analysis itself. The precision of one analysis has a direct impact on the other. For instance, the lifecycle awareness requires to cover the whole call graph. Having a precise call graph, by handling many sensitivities, may increase the precision in considering component's lifecycle.

3.5.6 Researchers must strive to provide clean and reusable artifacts

Our SLR showed that most approaches select a subset of challenges that they address directly within their implementation, while providing little opportunity for reuse in other approaches. Furthermore, only a few have made their tools publicly available. This leads to a situation where redundant contributions are made in the community without any comparison among state-of-the-art to further advance the research. Researchers should thus be strongly encouraged to make available at least the datasets used in their assessment experiments.

3.5.7 The booming Android ecosystem is appealing for holistic analysis

The number of apps considered by the state-of-the-art works is much less than the total number of existing Android apps (e.g., over 2.4 million apps available on Google Play)[xiii]. There is thus a need to develop static analyzers that are capable of market-scale analysis. Furthermore, since attackers can orchestrate several apps to perform advanced attacks in order to bypass analyzers that only target single application [66], there is also a need to perform compositional analysis among multiple Android apps. Unfortunately, inter-app analysis is not yet well investigated by the community. Actually, less than 5% of approaches examined in this SLR have taken into account inter-app analysis. More urgently, researchers should at least consider all the apps installed in a device as a whole to perform holistic analysis and contribute to minimize end-user exposure to security threats.

3.5.8 Combining multiple approaches could be the key towards highly precise analysis

Static analysis leads to over-approximations for multiple reasons, one being that it analyzes all code, including dead code. As a result, static analyses may generate false positives. On the contrary, dynamic analyses under-approximates, as it is challenging to cover all code, and thus tend to produce false negatives. Both approaches are thus complementary and can be combined to perform practical analyses. Typically, dynamic analysis can focus on checking if the reported results of static analysis are false positives, thus reducing the number of false alarms.

Besides combining static and dynamic analysis approaches, the community should consider also combining several static approaches to conduct highly precise analysis. Recently, TASMAN [11] has proposed to perform targeted symbolic execution to remove false data leaks reported by FlowDroid, where FlowDroid leverages static taint analysis to detect data leaks. TASMAN's promising results are encouraging for the research direction on using multiple static approaches to ensure a minimum false positives in analysis approaches.

3.6 Related Work

To the best of our knowledge, there is no systematic literature review in the research area of static analysis of Android apps. There is also no survey that specifically focuses on this research area. However, several Android security related surveys have been proposed in the literature. Unlike our approach, these approaches are actually not done systematically (not SLRs). As a result, there are always some well-known publications missing. Indeed, our review in this report has shown better coverage than those surveys in terms of publications in the area of static analysis of Android apps.

Sufatrio et al. [213] present a survey on general Android security, including both static and dynamic approaches. This survey first introduces a taxonomy with five main categories based on the existing security solutions on Android. Then, it classifies existing works into those five categories and thereby comparatively examines them. In the end, this survey has highlighted the limitation of existing works and also discussed potential future research directions. The survey shows in particular, that most research solutions addressing security issues in Android are leveraging static analysis. This relates to the finding in our SLR that most static analysis works for Android target security concerns. In their discussion of static analysis for Android, they also enumerate the different challenges of Android programming (e.g., multiple entry points, GUI, call-backs in event-based system) that analyzers must account for. In our SLR, we detail those challenges and further clarify which works deal with which challenge.

[xiii]https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

Faruki et al. [71] present another survey mainly focusing on a set of known Android malware detection approaches/tools, e.g., the growth of malware, the existence of anti-analysis techniques (i.e., being able to detect and thus evade analyzing systems [221]) . This survey highlights the need to take into account the existence of anti-analysis techniques (i.e., techniques that allow a malicious sample to detect and evade analyzing systems [221]). For example, they discuss how traditional signature-based and static analysis-based approaches can be vulnerable to stealthy techniques such as encryption. Eventually, it offers an overall overview for the directions that must be taken to tackle the remaining issues in statically analyzing Android apps.

Rashidi et al. [187] present another survey on existing Android security threats and security enforcement solutions. This survey classifies Android security mechanisms into four dimensions: Information Leaks, Vulnerabilities (Privilege Escalation and Colluding), Denial of Service (DoS) attacks, and App Clones. Our SLR finds that security papers using static analysis indeed target issues mainly in three of the dimensions they consider. We have not found any work that statically detects or hints on DoS attacks, which is reasonable. In contrast with their approach, the systematic nature of the SLR, allowed us to find papers in another security-related dimension, namely Cryptography misuses.

Haris et al. [92] present a survey focusing on privacy leaks and their associated risks in mobile computing. This survey has studied privacy in the area of mobile connectivity (e.g., cellular and surveillance technology) and in the area of mobile sensing (e.g., users prospects on sensor data). Besides, the authors have studied not only Android-specific leakages but also other mobile platforms including iOS and Windows. Similarly, [114] and [211] present state-of-the-art reviews with considering multiple mobile platforms.

More recently, Martin et al. [167] produce a technical report that reviews the state-of-the-art works on app store analysis in the software engineering field. In their survey, they have reported both non-technical and technical information to learn behaviors and trends of software repositories. More specifically, they have reviewed the literature works in 7 dimensions: API Analysis, Feature Analysis, Review Analysis, Security, Store Ecosystem, Size and Effort Prediction, and Others, including 127 non-technical and 60 technical papers. In their findings, they report that security is a pervasive concern in reviewed papers. This finding is inline with one of our SLR finding stating that static analysis is mainly used for the purpose of assessing app security. The whole focus of their work is however different from ours.

Sadeghi et al. [197] also present a technical report that studies the taxonomy and qualitative comparison of program analysis techniques, with a special focus on Android security. They have examined 100 research papers, including both static analysis and dynamic analysis approaches. Comparing to their review, in this SLR, we focus on static analysis of Android apps only. Among the findings, Sadeghi et al. show that the most used intermediate representation (IR) for their examined approaches are Jimple, accounting for 29%, which is in line with our findings. Besides, they also show that it is difficult to perform replication study on security-based researches, because most research tools and their evaluated artifacts are not publicly available.

All in all, our SLR differs from all the aforementioned surveys in a way that we exclusively focus on static analysis of Android apps. We believe those surveys can compliment ours and thus to provide a better view on the landscape of Android-based researches.

3.7 Conclusions

Research on static analysis of Android apps is quickly maturing, producing more and more advanced approaches for statically uncovering security issues in app code. To summarize the state-of-the-art and enumerate the challenges to be addressed by the research community we have conducted a systematic literature review of publications on approaches involving the use of static analysis on Android apps.

In the process of this review, we have collected 124 research papers published in Software engineering, programming languages and security conference and journal venues.

Our review has consisted of investigating the categories of issues targeted by static analysis, the fundamental techniques leveraged in the approaches, the implementation of the analysis itself (i.e., which analysis sensitivities are considered, and what Android characteristics are taken into account?), how the evaluation has been performed, and whether the research output is available for use by the community.

We have found that, (1) most analyses are performed to uncover security flaws in Android apps; (2) many approaches are built on top of a single analysis framework, namely Soot; (3) taint analysis is the most applied fundamental analysis technique in the publications; (4) although most approaches support multiple sensitivities, path sensitivity appears overlooked; (5) all approaches have missed to consider at least 1 characteristic of Android programming in their analysis; (6) finally, research contributions artifacts, such as tools and datasets, are often unpublished.

Table 3.7: The Full List of Examined Publications (Part I).

Year	Title
2015	StaDynA : Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications [251]
2015	Using targeted symbolic execution for reducing false-positives in dataflow analysis [11]
2015	Interactively verifying absence of explicit information flows in Android apps [27]
2015	Uranine: Real-time Privacy Leakage Monitoring without System Modification for Android [191]
2015	Information-Flow Analysis of Android Applications in DroidSafe [81]
2015	Scalable and precise taint analysis for Android [101]
2015	Static Control-Flow Analysis of User-Driven Callbacks in Android Applications [233]
2015	Composite Constant Propagation: Application to Android Inter-Component Communication Analysis [180]
2015	ApkCombiner : Combining Multiple Android Apps to Support Inter-App Analysis [123]
2015	Harvesting developer credentials in Android apps [260]
2015	Static Detection and Automatic Exploitation of Intent Message Vulnerabilities in Android Applications [76]
2015	Web-to-Application Injection Attacks on Android: Characterization and Detection [93]
2015	Datacentric Semantics for Verification of Privacy Policy Compliance by Mobile Applications [52]
2015	Automatically Detecting SSL Error-Handling Vulnerabilities in Hybrid Mobile Web Apps [261]
2015	Effective Real-Time Android Application Auditing [230]
2015	COVERT: Compositional Analysis of Android Inter-App Permission Leakage [20]
2015	Apparecium: Revealing Data Flows in Android Applications [216]
2015	SIG-Droid: Automated system input generation for Android applications [171]
2015	PaddyFrog: systematically detecting confused deputy vulnerability in Android applications [228]
2015	Towards Automatic Generation of Security-Centric Descriptions for Android Apps [243]
2015	Study and Refactoring of Android Asynchronous Programming [145]
2015	IccTA : Detecting Inter-Component Privacy Leaks in Android Apps [124]
2015	String Analysis for Java and Android Applications [118]
2015	HelDroid: Dissecting and Detecting Mobile Ransomware [6]
2015	Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution [170]
2015	DroidJust: automated functionality-aware privacy leakage analysis for Android applications [47]
2015	Static Window Transition Graphs for Android [233]
2015	Accurate Specification for Robust Detection of Malicious Behavior in Mobile Environments [212]
2015	Detecting Event Anomalies in Event-Based Systems [198]
2015	WeChecker: efficient and precise detection of privilege escalation vulnerabilities in Android apps [55]
2015	AutoPPG: Towards Automatic Generation of Privacy Policy for Android Applications [241]
2015	Selective Control-Flow Abstraction via Jumping [32]
2015	EcoDroid: an approach for energy-based ranking of Android apps [102]
2015	A Permission verification approach for android mobile applications [78]
2015	AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context [235]
2015	Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting [157]
2014	Cassandra: Towards a Certifying App Store for Android [156]
2014	Systematic Analysis and Detection of Misconfiguration Vulnerabilities in Android Smartphones [88]
2014	Brahmastra: Driving Apps to Test the Security of Third-Party Components [30]
2014	Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications [206]
2014	Execute this! analyzing unsafe and malicious dynamic code loading in android applications [185]
2014	AndRadar: Fast Discovery of Android Applications in Alternative Markets [151]
2014	Achieving accuracy and scalability simultaneously in detecting application clones on Android markets [45]
2014	Android taint flow analysis for app sets [112]
2014	Static Reference Analysis for GUI Objects in Android Software [195]
2014	Protection against Code Obfuscation Attacks based on control dependencies in Android Systems [82]
2014	Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android [25]
2014	Tracking Load-time Configuration Options [143]
2014	Modeling Users' Mobile App Privacy Preferences : Restoring Usability in a Sea of Permission Settings [144]
2014	Characterizing and detecting performance bugs for smartphone applications [152]
2014	Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs Categories and Subject Descriptors [244]
2014	AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking Into Android Apps [202]
2014	Retrofitting concurrency for Android applications through refactoring [146]
2014	Making web applications more energy efficient for OLED smartphones [119]
2014	Formalisation and analysis of Dalvik bytecode [227]
2014	MIGDroid: Detecting APP-Repackaging Android malware via method invocation graph [97]
2014	AsDroid : Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction [100]
2014	Apposcopy : Semantics-Based Detection of Android Malware Through Static Analysis [73]
2014	FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps [10]
2014	A5 : Automated Analysis of Adversarial Android Applications [222]
2014	Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph [214]
2014	A type and effect system for activation flow of components in Android programs [49]
2014	AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications [245]

Table 3.8: The Full List of Examined Publications (Part II).

Year	Title
2014	Collaborative Verification of Information Flow for a High-Assurance App Store [69]
2014	Information flows as a permission mechanism [205]
2014	SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps [207]
2014	Amandroid : A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps Categories and Subject Descriptors [226]
2014	EvoDroid: segmented evolutionary testing of Android apps [163]
2014	Efficient, Context-Aware Privacy Leakage Confinement for Android Applications without Firmware Modding [246]
2014	Multi-App Security Analysis with FUSE : Statically Detecting Android App Collusion [192]
2014	Detecting Mobile Application Malicious Behaviors Based on Data Flow of Source Code [44]
2013	Effective Inter-Component Communication Mapping in Android with Epicc : An Essential Step Towards Holistic Security Analysis [181]
2013	Contextual Policy Enforcement in Android Applications with Permission Event Graphs [46]
2013	The impact of vendor customizations on android security [229]
2013	A3: Automatic Analysis of Android Malware [160]
2013	Detecting passive content leaks and pollution in android applications [258]
2013	AppIntent : Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection [238]
2013	The Transitivity-of-Trust Problem in Android Application Interaction [26]
2013	MobSafe: cloud computing based forensic analysis for massive mobile applications using data mining [231]
2013	Targeted and depth-first exploration for systematic testing of android apps [14]
2013	Hybrid static-runtime information flow and declassification enforcement [194]
2013	A grey-box approach for automated GUI-model generation of mobile applications [234]
2013	SIF: A Selective Instrumentation Framework for Mobile Applications [91]
2013	Thresher: precise refutations for heap reachability [31]
2013	automated testing with targeted event sequence generation [105]
2013	Calculating source line level energy information for Android applications [117]
2013	Automatic Detection of Inter-application Permission Leaks in Android Applications [201]
2013	Characterizing and detecting resource leaks in Android applications [87]
2013	Scalable semantics-based detection of similar Android applications [54]
2013	Slicing droids: program slicing for smali code [94]
2013	Sound and precise malware analysis for android via pushdown reachability and entry-point saturation [142]
2013	DroidAlarm: an all-sided static analysis tool for android privilege-escalation malware [254]
2013	An empirical study of cryptographic misuse in android applications [64]
2013	Estimating mobile application energy consumption using program analysis [90]
2013	Detecting GPS information leakage in Android applications [161]
2013	Structural Detection of Android Malware using Embedded Call Graphs [77]
2012	DroidChecker : Analyzing Android Applications for Capability Leak [42]
2012	Towards verifying android apps for the absence of no-sleep energy bugs [220]
2012	Automatically securing permission-based software by reducing the attack surface: An application to android [23]
2012	CHEX : Statically Vetting Android Apps for Component Hijacking Vulnerabilities Categories and Subject Descriptors [158]
2012	Unsafe exposure analysis of mobile in-app advertisements [84]
2012	A framework for static detection of privacy leaks in android applications [165]
2012	Refactoring android Java code for on-demand computation offloading [248]
2012	Detect and optimize the energy consumption of mobile app through static analysis: an initial research [225]
2012	Dr . Android and Mr . Hide : Fine-grained Permissions in Android Applications Categories and Subject Descriptors [106]
2012	Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications [253]
2012	I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications [57]
2012	Systematic Detection of Capability Leaks in Stock Android Smartphones. [85]
2012	Model-based static source code analysis of java programs with applications to android security [159]
2012	Automated Concolic Testing of Smartphone Apps [4]
2012	what is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps [182]
2012	Android: Static analysis using similarity distance [62]
2012	TrustDroid: Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking [250]
2012	Attack of the clones: detecting cloned applications on Android markets [53]
2012	Detecting control flow in smarphones: Combining static and dynamic analyses [83]
2012	AQUA: Android QUery Analyzer [109]
2012	Scandal: Static Analyzer for Detecting Privacy Leaks in Android Applications [110]
2012	Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security [70]
2012	AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale [79]
2012	Static analysis of Android programs [183]
2011	Analyzing Inter-Application Communication in Android [48]
2011	Android : From Reversing to Decompilation [63]
2011	Clonecloud: elastic execution between mobile device and cloud [50]
2011	Using Static Analysis for Automatic Assessment and Mitigation of Unwanted and Malicious Activities Within Android Applications [28]

4 Bridging Inter-Component Communication

Inter-Component Communication (ICC) mechanism is actually conducted by the Android system at runtime, making static analyzers that simply manipulate app-level code ICC-unaware. Therefore, in this chapter, we propose an instrumentation-based approach which represents explicitly ICC with traditional Java calls. As a result, existing static analyzers, without any modification, are able to perform ICC-aware analysis. Moreover, because Inter-App Communication (IAC) shares the same mechanism of ICC, we propose to combine multiple Android apps to further support IAC-aware analysis.

This chapter is based on the work published in the following papers:

- Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015
- Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *Proceedings of the 30th International Information Security and Privacy Conference (IFIP SEC)*, 2015

Contents

4.1 Introduction

The success of the Android OS in its user base as well as in its developer base can partly be attributed to its communication model, named Inter-Component Communication (ICC), which promotes the development of loosely-coupled applications. By dividing applications into components that can exchange data within a single application or even across several applications, Android encourages software reuse, and thus reduces developer burden.

Unfortunately, the ICC model, which provides a message passing mechanism for data exchange among components, can be misused by malicious apps to threaten user privacy. Indeed, researchers have shown that Android apps frequently send user's private data outside the device without their prior consent [257]. Those applications are said to leak private data. Recently, researchers have investigated ICC methods as features for vulnerability detection [164], in lieu of permissions and API calls. However, there is still a lack of a comprehensive study on the characteristics of the usage of ICCs by Android malware. Typically, what is the extent of the presence of privacy leaks in Android malware?

To answer such a question, an Android analysis tool has to be developed for tracking privacy leaks. Although, most of the privacy leaks are simple, i.e., easily identifiable as they operate within a single component, there have recently been reports of cross-components privacy leaks [135]. Thus, analyzing components separately is not enough to detect leaks: it is necessary to perform an inter-component analysis of applications. Android app analysts could leverage such a tool to identify malicious apps that leak private data. For the tool to be useful, it has to be highly precise and minimize the false positive rate when reporting applications leaking private data.

In this chapter, we use a static taint analysis technique to find privacy leaks, e.g., paths from sensitive data, called sources, to statements sending the data outside the application or device, called sinks. A path may be within a single component or cross multiple components (or apps). State-of-the-art approaches using static analysis to detect privacy leaks on Android apps mainly focus on detecting intra-component sensitive data leaks. CHEX [158], for example, uses static analysis to detect component hijacking vulnerabilities by tracking taints between sensitive sources and sinks. FlowDroid [10] performs taint analysis within single components of Android applications but with a better precision. More recently, Amandroid [226] has been proposed to detect ICC-based privacy leaks in Android apps. However, it does not currently tackle `Content Provider`, one of the four Android components. It is also not sensitive to some complicated ICC methods such as `bindService` and `startActivityForResult`.

Thus, we propose IccTA, an Inter-component communication Taint Analysis tool, for a sound and precise detection of ICC links and leaks. Although our approach is generic and can be used for any data-flow analysis, we focus in this chapter on using IccTA to detect ICC-based privacy leaks. To verify our approach, we developed 22 apps containing ICC-based privacy leaks. We have added these applications to DroidBench[i], an open test suite for evaluating the effectiveness and accuracy of taint analysis tools specific for Android apps. The 22 apps cover the top 8 used ICC methods illustrated in Table 4.1 (we give more details later). Besides, we test IccTA on 15,000 real-world apps randomly selected from Google Play market in which we detect 2,395 ICC leaks in 337 apps. We also launch IccTA on the MalGenome set containing 1260 malware, where IccTA reports 108 apps with 534 ICC leaks. By comparing the detecting rate $r = \frac{\#\ of\ detected\ apps}{\#\ of\ tested\ apps}$ of the two data sets, we found that $r_{MalGenome} = 8.6\%$ is much higher than $r_{GooglePlay} = 2.2\%$. Thus, we can conclude that *ICC are significantly used by malware to leak private data*, making ICC a potential feature for malware detection.

The privacy leaks in Android are further exacerbated by the fact that several applications can interact and "collaborate" to leak data using the inter-app communication (IAC) mechanism. IAC and ICC

[i]https://github.com/secure-software-engineering/DroidBench

are similar in Android, and thus present the same vulnerabilities. Unfortunately, state-of-the-art analysis tools are focused on ICC by analyzing a single app at a time. Consequently, inter-app privacy leaks cannot be identified and managed by existing tools and approaches from the literature.

In this chapter, we propose to empower existing static analysis tools for Android to work beyond the boundaries of a single app, so as to highlight security flaws in the interactions between two or more apps. To that end we have designed and developed a tool called ApkCombiner which takes as input several apps that may cohabit in the same device, and yields a single app package (i.e., apk) combining the different components from the different apps. The resulting package is ensured to be ready for analysis by existing tools. Thus, since the IAC mechanism is the same as the ICC mechanism, by combining apps, ApkCombiner reduces an IAC problem to an ICC problem, allowing existing tools to indirectly perform inter-app analysis without any modification.

During the combination of multiple Android apps, some classes may conflict with one another. In this work, we take into account two types of conflict: 1) the conflicted classes are exactly same (same name and same content), we solve this type of conflicts by simply dropping the duplicated classes and 2) the conflicted classes are different (same name but different content), we solve this type of conflicts by first renaming the conflicted classes, and then ensuring that all dependencies and calls related to those classes are respected throughout the app code.

The contributions of this chapter are as follows:

- We present the findings of an empirical study on the use of ICC in Android malware and benign apps.
- We propose a novel methodology to resolve the ICC problem by directly connecting the discontinuities of Android apps at the code level. More specifically, we provide a tool-based approach called IccTA, which leverages FlowDroid to perform inter-component static taint analysis.
- We discuss the need for tools to support inter-app analysis, and present a non-intrusive approach that can be leveraged by existing tools which are focused on intra-app analysis. In particular, we provide a prototype implementation called ApkCombiner that uses an effective algorithm to solve different conflicts which may arise during the combination of multiple Android apps.
- Finally, we present an assessment of IccTA using i) the DroidBench and ICC-Bench test suites, ii) 15,000 real-world Android applications, iii) 1,260 malware apps from MalGenome and an evaluation of ApkCombiner using both benchmark apps and real-world Android apps.

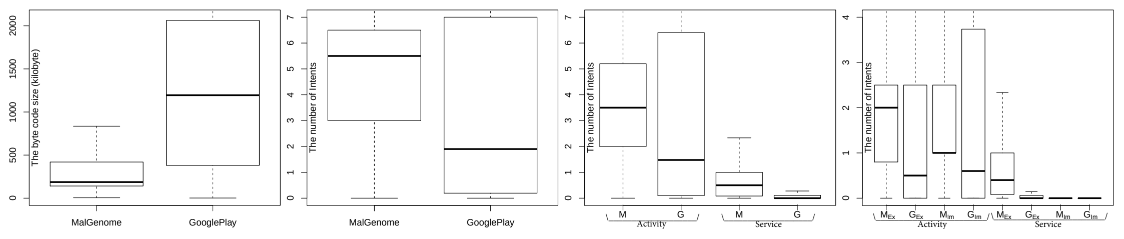
We make available online our full implementation as an open source project, along with the extended DroidBench apps and the scripts to reproduce our experimental results on

`https://sites.google.com/site/icctawebpage/`

The remainder of this chapter is organized as follows. We first motivate our approach in Section 4.2. Then, we detail in Section 4.3 how our approach instruments Android apps for bridging ICC for static analysis. Next, we point our how our approach combines Android apps and how our approach solves class conflicts in Section 4.4. Section 4.5 evaluates our approach and Section 4.6 discusses some closely related work. Finally in Section 4.7 we close this work with conclusion.

4.2 Motivation

To motivate our work, we present an empirical study on how ICCs are used in Android apps, to expose the difference of usage between malware and benign apps (cf. Section 4.2.1). We then give concrete examples to introduce ICC and IAC leaks in Section 4.2.2 and Section 4.2.3, respectively.



a The Tytecode Size **b** The Total Intents **c** Activity & Service **d** Explicit & Implicit

Figure 4.1: The Comparison between MalGenome and GooglePlay Apps. (“M” Means MalGenome, “G” Means GooglePlay, “Ex” Means Explicit and “Im” Means Implicit).

4.2.1 ICC Usage in Android Apps

To the best of our knowledge there have been no empirical investigation of the usage of ICC in Android apps. Yet, given the importance of ICC in the Android development model, as well as its potential correlation with malware functioning as introduced above, a thorough study on real-world apps can provide answers to the following important questions:

How often are ICCs used in Android apps? This question will lead to the investigation of the extent to which each of the different ICC methods are used in apps, and what types of components they are targeting.

What kind of Intents are used for ICC in apps? This question is important to estimate the differences in the instantiations of *implicit Intents* and *explicit Intents*.

Is the usage of ICC different between malware and benign apps? Investigating this question may open directions for malware detection research, which seeks reliable app features to use in machine-learning processes.

Datasets: We attempt to provide answers to the questions above, using two distinct datasets of over one thousand applications each.

The first dataset, named *MalGenome*, includes $1,023$ Android malware samples collected by Zhou et al. [257]. Although the originally published dataset contains $1,260$ apps, some of these apps share the same package names, therefore we consider them as duplicates.

The second dataset, hereafter referred to as *GooglePlay*, is a set of $1,023$ Android apps[ii] randomly selected from the official Google market.

Results: We now present the findings of our investigation.

First, we compute the usage rate of ICC methods. To that end, we parse the app byte code to count the instances of specific method calls based on a catalog of ICC method names. This analysis was performed on all the $1,023 \times 2$ apps. Table 4.1 shows the usage rate of the ICC method names, separating the top 8 most used, from all other ICC methods.

The *# of Calls* represents the absolute number of ICC method calls from the entire two sets. The *# of Apps* represents the number of apps using at least once the corresponding ICC method. 96.4% of the apps in our dataset use the `startActivity` ICC method, which accounts for 56.01% of the total ICC methods calls. `startActivity` is used to launch a new Activity component, e.g., to switch from one user interface window to another. The second most used ICC method is `startActivityForResult` which also launches a new Activity component. In this case however, the flow goes back to the calling component. Then follows `query`, an ICC method used to access content providers. `startService`, which appears in 78.1% of the apps, is used to launch a new Service.

[ii]We choose $1,023$ apps to avoid a class imbalance issue.

Table 4.1: The Top 8 Used ICC Methods. When These Methods are Overloaded, We Consider the One with most Number of Calls.

ICC Method	# of Calls	# of Apps
startActivity	54,334 (56.01%)	1,972 (96.4%)
startActivityForResult	11,118 (11.46%)	1,409 (68.7%)
query	8,191 (8.44%)	1,374 (67.2%)
startService	6,660 (6.86%)	1,597 (78.1%)
sendBroadcast	5,119 (5.28%)	1,035 (50.1%)
insert	2,164 (2.23%)	780 (38.1%)
bindService	1,638 (1.69%)	512 (25.0%)
delete	1,633 (1.69%)	472 (23.1%)
Other ICC Methods	6,155 (6.34%)	-
Total	97,012 (100%)	-

We now investigate the number of Intents[iii] leveraged by the dataset apps. We compare the sizes of apps across the MalGenome and GooglePlay sets. Fig. 4.1a represents the boxplot of the size of the apps for both sets. The median value for the MalGenome set is 187 KB whereas the median value is 1195 KB for the GooglePlay set. We ensure that this difference of median sizes between the datasets is significantly different by performing a Mann-Whitney-Wilcoxon (MWW). The resulting p-value confirms that the difference is significant at a significance level[iv] at 0.001.

To account for any potential bias that the difference of app sizes between datasets may introduce, we proceed to normalize all results according to a unit of dex code size. A normalized result nR is obtained by applying the formula $nR = \frac{iR}{\lceil bS \div 100K \rceil}$, where iR is the initial result and bS is the byte code size.

Absolute number of Intents: Fig. 4.1b represents the boxplot of the normalized numbers of Intents per apps. The median number of Intents is 5.5 per 100KB per app for the MalGenome dataset, and 1.9 for the GooglePlay dataset. The MWW test again shows that this difference is statistically significant.

Number of Intents vs Component types: We further investigate the difference of number of Intents per app in the two datasets by comparing the usage of Intents for ICC exchange with specific types of components. Fig. 4.1c shows the boxplot of number of Intents used to launch an *Activity* (left two) and a *Service* (right two). The median values are respectively 3.50 and 0.50 for the MalGenome dataset, and 1.48 and 0.00 for the GooglePlay dataset.

Explicit Intents vs Implicit Intents: Table 4.2 provides comparison data on the proportion of *implicit* and *explicit* Intents among the overall numbers of *Intents*. In the MalGenome set, 27,278 Intents in total are found, where 14,034 (51.4%) are *explicit Intents*. In the Google Play set, however, only 40.1% (22,955 out of 56,213) of the Intents are *explicit Intents*.

Fig. 4.1d presents boxplots detailing the normalized number of implicit and explicit Intents for both Activity and Service. The median values between the MalGenome dataset and the GooglePlay dataset are close in the case of implicit Intents. On the other hand, there is a larger difference for explicit Intents. We further confirm that this difference is statistically significant via the MWW test, using a significance level of $\alpha = 0.0001$.

This empirical investigation of ICC in malware and benign apps highlights the importance of ICC in the context of Android app security management. In particular, the focus of this study has demonstrated that some Android properties, e.g., possibility to explicitly target a component, thus bypassing user's choice, are exploited by malicious apps. Such apps can indeed *leak* private data

[iii] Note that we do not distinguish the difference between Intents and ICC methods since basically an Intent is corresponding to an ICC method.

[iv] Given a significance level $\alpha = 0.001$, if p-value $< \alpha$, there is one chance in a thousand that the difference between the datasets is due to a coincidence.

Table 4.2: Comparison of the Use of Intents in the Data Sets.

Dataset	Activity (expl./impl.)	Service (expl./impl.)	Receiver (expl./impl.)	Total (expl./impl.)
MalGenome	8803/9569	5204/422	27/3253	14034/13244
Google Play	20018/30461	2715/828	222/1969	22955/ 33258

```
 1 //TelephonyManager telMnger; (default)
 2 //SmsManager sms; (default)
 3 class Activity1 extends Activity {
 4 void onCreate(Bundle state) {
 5 Button to2 = (Button) findViewById(to2a);
 6 to2.setOnClickListener(new OnClickListener(){
 7 void onClick(View v) {
 8 String id = telMnger.getDeviceId();
 9 Intent i = new Intent(Activity1.this,Activity2.class);
10 i.putExtra("sensitive", id);
11 Activity1.this.startActivity(i);
12 }});}}
13 class Activity2 extends Activity {
14 void onStart() {
15 Intent i = getIntent();
16 String s = i.getStringExtra("sensitive");
17 sms.sendTextMessage(number,null,s,null,null);
18 }}
```

Listing 4.1: A Running Example.

across components. After presenting our approach for detecting leaks that are related to ICC, we will perform a final experiment to investigate whether there is a correlation between the number of Intents and the number of ICC leaks. A positive correlation will thus provide confirmation that ICC can be explored as a feature for malware detection.

4.2.2 ICC Leak Example

We define a privacy leak as a path from sensitive data, called *source*, to statements sending this data outside the application or device, called *sink*. A path may be within a single component or across multiple components. In this chapter, the *sources* and *sinks* we use are provided by SUSI [188].

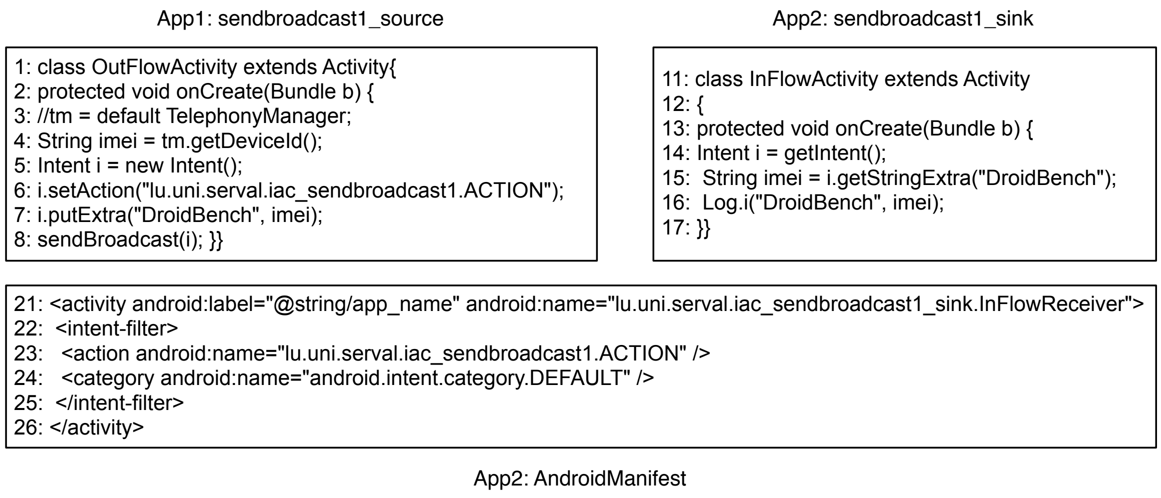
Listing 4.1 illustrates the concept of ICC leak through a concrete example. The code snippets present two Activities: $\texttt{Activity}_1$ and $\texttt{Activity}_2$. $\texttt{Activity}_1$ registers an anonymous button listener for the *to2* button (lines 5-11). An ICC method `startActivity` is used by this anonymous listener. When button *to2* is clicked, the `onClick` method is executed and the user interface will change to $\texttt{Activity}_2$. An `Intent` containing the device ID (lines 15), considered as sensitive data, is then exchanged between the two components by first attaching the data to the Intent with the `putExtra` method (lines 10) and then by invoking the ICC method `startActivity` (lines 11). Note that the Intent is created by explicitly specifying the target class ($\texttt{Activity}_2$).

In this example, `sendTextMessage` is systematically executed when $\texttt{Activity}_2$ is loaded since `onStart` is in the execution lifecycle of an `Activity`. The data retrieved from the `Intent` is thus sent as a SMS message to the specified phone number: *there is an ICC leak triggered by button to2.* When *to2* is clicked, the device ID is transferred from $\texttt{Activity}_1$ to $\texttt{Activity}_2$ and then outside the application.

In this chapter, we aim to perform static taint analysis for Android apps to detect such inter-component communication (ICC) based privacy leaks. In static taint analysis, a leak k corresponds to a sequence of statements, which starts from a *source* s and ends with a *sink* d. *Sources* are identified as they return private data from the user's point of view into the application code, while *Sinks* are identified as they send data out of the application. An ICC leak is a special leak which contains, in its statement sequence, at least one ICC method. Normally, $C(s) \neq C(d)$, where $C(s)$ means the component of method s. But in some cases, $C(s)$ can equal to $C(d)$. Take ICC method `startActivityForResult` as an example, component C_1 can use this method to start a component C_2 (in method m_1). Once C_2 finishes running, C_1 runs again (in method m_2) with some result data returned from C_2. An ICC leak may occur as $m_1 \rightarrow C_2 \rightarrow m_2$ but in this situation $C(m_1) == C(m_2)$.

4.2.3 IAC Leak Example

Fig. 4.2 presents a running example that shows an IAC vulnerability. The example is extracted from a test case of DroidBench[v] referred among its list as *InterAppCommunication_sendBroadcast1*. The example consists of two Android applications, referred to as *sendBroadcast1_source* and *sendBroadcast1_sink*.


App1: sendbroadcast1_source
App2: sendbroadcast1_sink
App2: AndroidManifest

```
1: class OutFlowActivity extends Activity{
2: protected void onCreate(Bundle b) {
3: //tm = default TelephonyManager;
4: String imei = tm.getDeviceId();
5: Intent i = new Intent();
6: i.setAction("lu.uni.serval.iac_sendbroadcast1.ACTION");
7: i.putExtra("DroidBench", imei);
8: sendBroadcast(i); }}
```

```
11: class InFlowActivity extends Activity
12: {
13: protected void onCreate(Bundle b) {
14: Intent i = getIntent();
15:  String imei = i.getStringExtra("DroidBench");
16:  Log.i("DroidBench", imei);
17: }}
```

```
21: <activity android:label="@string/app_name" android:name="lu.uni.serval.iac_sendbroadcast1_sink.InFlowReceiver">
22:  <intent-filter>
23:   <action android:name="lu.uni.serval.iac_sendbroadcast1.ACTION" />
24:   <category android:name="android.intent.category.DEFAULT" />
25:  </intent-filter>
26: </activity>
```

Figure 4.2: A Running Example that Shows an Inter-app Vulnerability.

App *sendBroadcast1_source* contains a simple *Activity* component, named `OutFlowActivity`, which first obtains the device ID (line 4) and then stores it into an Intent (line 7) which is then forwarded to other components (potentially in other applications since the Intent is implicit (line 6)). App *sendBroadcast1_sink* contains a component called `InFlowActivity`, which first extracts data from the received Intent and then logs it into disk.

In this example, we consider device ID, which is protected by a permission check of the Android system, to be sensitive data. We also consider the *log()* method to be dangerous behavior since it writes data into disk, therefore leaving it accessible to any applications, including anyone which does not have permission to access the device ID through the Android OS. Thanks to the declarations in the *Manifest* file in the package of *sendBroadcast1_sink*, `OutFlowActivity` is able to communicate with `InFlowActivity` using the *sendBroadcast()* ICC method. Thus, through the interaction between these two apps, a sensitive data can be leaked.

Unfortunately, the current state-of-the-art static analysis tools, including IccTA for which we will detail in this chapter, by default, cannot tackle this kind of IAC problem. Since IccTA is designed to be efficient in statically identifying bugs and leaks across components inside a single app, we aim at enabling them to do the same across applications. We further put a constrain on remaining non intrusive, i.e., to avoid applying any modification on them, so as to avoid introducing limitations or new bugs in these tools. We thus propose ApkCombiner, which, by combining multiple apps into

[v]DroidBench is a set of hand-crafted Android apps used as a ground truth dataset to evaluate how well static and dynamic security tools find data leaks. `https://github.com/secure-software-engineering/DroidBench`

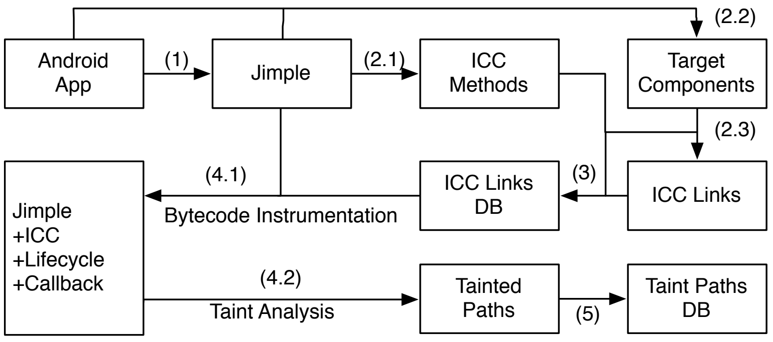


Figure 4.3: Overview of IccTA.

one, reduces the IAC problem to an ICC problem that state-of-the-art tools can solve in an intra-app analysis.

4.3 IccTA

In this section, we present an overview of our tool called IccTA, which is designed to detect ICC leaks in Section 4.3.1. IccTA uses a two-step approach: 1) ICC links extraction; 2) Taint flow analysis for ICC. Sections 4.3.2 and 4.3.3 detail these two steps respectively.

4.3.1 Overview

Fig. 4.3 shows the overview of IccTA, our open source tool to detect ICC leaks. Even if Android apps are implemented in Java, an app is compiled into Dalvik bytecode instead of the traditional Java bytecode. In a first step, IccTA uses Dexpler [24] to transform this Dalvik bytecode into *Jimple*, a Soot's internal representation [115]. Soot is a popular framework to analyze Java based apps. In the second step (arrows 2.$*$), IccTA extracts the ICC links, and in step 3, stores them as well as all the collected data (e.g., ICC call parameters or Intent Filter values) into a database. Based on the ICC links, in step 4.1, IccTA modifies the Jimple representation to directly connect the components to enable data-flow analysis between components. In step 4.2, by using a modified version of FlowDroid [10], a high precise intra-component taint analysis tool for Android apps, IccTA builds a complete control-flow graph of the whole Android application. This allows propagating the context (e.g., the value of Intents) between Android components and yielding a highly precise data-flow analysis. To the best of our knowledge, this is the first approach that precisely connects components for data-flow analysis. At last (step 5), IccTA stores the reported tainted paths (leaks) into database.

In both steps (3) and (5), we store all the results including the ICC methods with their attribute values such as URI and Intent, the target components with their Intent Filter values, the built ICC links and the reported ICC leaks into a database. This allows to only analyze an app once, and then reuse the results from the database. Besides, we can easily based on the database to perform future statistically study. The database also enables us to easily build inter-app links. In the next two sections, we detail the main technical contributions of IccTA, which lie in steps (2) and (4).

4.3.2 ICC Links Extraction

In this section, we detail our approach to extract the ICC links of the analyzed apps. An ICC link $l : m \to C$ is used to link two components in which the source component contains an ICC method m

(A)
```
// modifications of Activity1
-  Activity1.this.startActivity(i);
+  IpcSC.redirect0(i);
```

(B)
```
// creation of a helper class
+class IpcSC {
+ static void redirect0(Intent i) {
+   Activity2 a2 = new Activity2(i);
+   a2.dummyMain();
+ }
+}
```

(C)
```
// modifications in Activity2
+public Activity2(Intent i) {
+  this.intent_for_ipc = i;
+}
public Intent getIntent() {
+ return this.intent_for_ipc;
}
+public void dummyMain() {
+ // lifecycle and callbacks
+ // are called here
+}
```

Figure 4.4: Handling `startActivity` ICC Method.

that holds information (e.g., the class name for an explicit Intent or the *action, category, mimetype, ...* information for an implicit Intent) to access the target component C.

As shown in Fig. 4.3, IccTA uses three steps to extract the ICC links from an app. In step (2.1), IccTA leverages Epicc [181] to obtain the ICC methods and their parameters (e.g., *action* of Intents). Epicc is a tool, based on Soot and Heros [33], to identify ICC methods as well as their parameter values (e.g., *action*, *category*). In IccTA, we use IC3 [180], an advanced tool that implements the idea of Epicc, to also parse the `URIs` (e.g., *scheme*, *host*) to support `Content Provider` related ICC methods (e.g., `query`) and to fully support the data field of Intents [178]. In step (2.2), IccTA identifies all the possible target components by parsing the configuration file named `AndroidManifest` of an app to retrieve the values of the *Intent Filters*. In some situations, analyzing the bytecode is also necessary since `Broadcast Receiver` can be registered at runtime. In step (2.3), we match ICC methods with their target components, i.e., the *Intents* with *Intent Filters*, through the rules introduced by the Android documentation[vi].

4.3.3 Taint Flow Analysis for ICC

In this section, we detail our instrumentation approach to perform taint flow analysis for ICC. Actually, there are three types of discontinuities in Android: (1) ICC methods, (2) lifecycle methods and (3) callback methods. We first describe how IccTA tackles ICC methods in Section 4.3.3.1. Then, we detail how IccTA resolves lifecycle and callback methods in Section 4.3.3.2.

4.3.3.1 ICC Methods

In step (4.1) of Fig. 4.3, the *Jimple* code is instrumented by IccTA to connect components. This code modification is required for all ICC methods (listed in Table 4.1). The main idea of the transformation is to replace an ICC method call with an instantiation of the target component with the appropriate *Intent*. We detail these modifications for the two most used ICC methods: `startActivity` and `startActivityForResult`. We handle ICC methods for *Service*s and *Broadcast Receiver*s in a similar way.

StartActivity. Fig. 4.4 shows the code transformation done by IccTA for the ICC link between $\texttt{Activity}_1$ and $\texttt{Activity}_2$ of our running example. IccTA first creates a helper class named `IpcSC` (B in Fig. 4.4) which acts as a bridge connecting the source and destination components. Then, the `startActivity` ICC method is removed and replaced by a statement calling the generated helper method (`redirect0`) (A).

In (C), IccTA generates a constructor method taking an `Intent` as parameter, a `dummyMain` method to call all related methods of the component (i.e., lifecycle and callback methods) and overrides the `getIntent` method. An Intent is transferred by the Android system from the caller component to the

[vi]https://developer.android.com/reference/android/content/IntentFilter.html

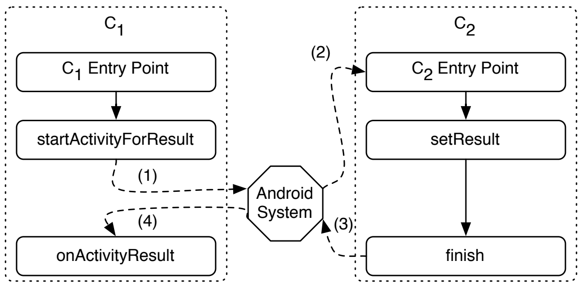


Figure 4.5: The Control-flow of `startActivityForResult`.

callee component. We model the behavior of the Android system by explicitly transferring the Intent to the destination component using a customized constructor method, `Activity`$_2$`(Intent i)`, which takes an `Intent` as its parameter and stores the Intent to a newly generated field `intent_for_ipc`. The original `getIntent` method asks the Android system for the incoming Intent object. The new `getIntent` method models the Android system behavior by returning the Intent object given as parameter to the new constructor method.

The helper method `redirect0` constructs an object of type `Activity`$_2$ (the target component) and initializes the new object with the `Intent` given as parameter to the helper method. Then, it calls the `dummyMain` method of `Activity`$_2$.

To resolve the target component, i.e., to automatically infer what is the type that has to be used in the method `redirect0` (in our example, to infer `Activity`$_2$), IccTA uses the ICC links stored in step (3) in which not only the explicit Intents but also the implicit Intents are resolved. Therefore, there is no difference for IccTA to handle explicit or implicit Intents based ICCs.

StartActivityForResult. A component C_1 can use this method to start a component C_2. Once C_2 finishes running, C_1 runs again with some result data returned from C_2. Fig. 4.5 shows the control-flow mechanism of `startActivityForResult` ICC method. There are two discontinuities: one from (1) to (2), similar to the discontinuity of the `startActivity` method, and the other from (3) to (4).

The `startActivityForResult` ICC method has a more complex semantic compared to common ICC methods that only trigger one-way communication between components (e.g., `startActivity`). Fig. 4.6 shows how the code is instrumented to handle the `startActivityForResult` method for Fig. 4.5. To stay consistent with common ICC methods, we do not instrument the `finish` method of `C`$_2$ to call `onActivityResult` method. Instead, we generate a field `intent_for_ar` to store the *Intent* which will be transferred back to `C`$_1$. The *Intent* that will be transferred back is set by the `setResult` method. We override the `setResult` method to store the value of *Intent* to `intent_for_ar`. The helper method `IpcSC.redirect0` does two modifications to link these two components directly. First, it calls the `dummyMain` method of the destination component. Then, it calls the `onActivityResult` method of the source component.

4.3.3.2 Lifecycle and Callback Methods

One challenge when analyzing Android applications is to tackle callback and lifecycle methods of components. An introduction about lifecycle and callback methods can be found in [126]. There is no direct call among those methods in the code of applications since the Android system handles lifecycles and callbacks. For callback methods, we need to take care of not only the methods triggered by the User Interface (UI) events (e.g., `onClick`) but also callbacks triggered by Java or the Android system (e.g., the `onCreate` method). In Android, every component has its own lifecycle methods. To solve this problem, IccTA generates a `dummyMain` method for each component in which we model

(A)
```
- act.startActivityForResult(i);
+ IpcSC.redirect0(act, i);
```

(B)
```
+class IpcSC {
+ static void redirect0(C1 c1,
+                         Intent i){
+   C2 c2 = new C2(i);
+   c2.dummyMain();
+   Intent retI = c2.getIntentFAR();
+   c1.onActivityResult(retI);
+ }
+}
```

(C)
```
void setResult(Intent i) {
+ this.intent_for_ar = i;
}
+public Intent getIntentFAR() {
+ return this.intent_for_ar;
+}
```

Figure 4.6: Handing the `startActivityForResult` ICC Method. (A) and (C) Represents the Modified Code of $\texttt{C}_1$ and $\texttt{C}_2$ Respectively. (B) is the Glue Code Connecting $\texttt{C}_1$ and $\texttt{C}_2$. Some Method Parameters are not Represented to Simplify the Code.

all the methods mentioned above so that our CFG based approach is aware of them. Note that FlowDroid also generates a `dummyMain` method, but it is generated for the whole app instead of for each component like we do.

4.4 ApkCombiner

We now discuss the design and implementation of ApkCombiner. First we present an overview of the approach in Section 4.4.1 before providing details on how we address the case of conflicting code, typically same-name classes, when combining apps (cf. Section 4.4.2). Although, for the sake of simplicity, we describe the case of merging two apps, the approach, and the prototype tool, can perform on merging any number of apps.

4.4.1 Overview

The main objective of our work is to enable Android-targeted state-of-the-art static analysis tools, which have proven to be effective in *intra*-app analyses, to perform as well in *inter*-app analyses. ApkCombiner takes a set of Android apps as input and yields a new Android app in output. The newly generated app contains all the features of the input apps except for their IAC features: there is no more IAC but only ICC in the new generated app.

The different steps of how ApkCombiner works are shown in Fig. 4.7. Each app is first disassembled into *smali* files and a Manifest file using a tool for reverse engineering Android apk files, namely android-apktool [vii]. Second, all files from the apps are checked together for conflicts and integrated (with conflicts solved) into a directory. The Manifest files, one from each app, are merged into a single Manifest file. Finally, ApkCombiner assembles the smali files and the Manifest file along with all other resources, such as image files, into a single apk. Although potential conflicts on such extra-resources may be met, ApkCombiner does not take them into account since the objective is not to produce a runnable apk, but an apk that can be analyzed statically.

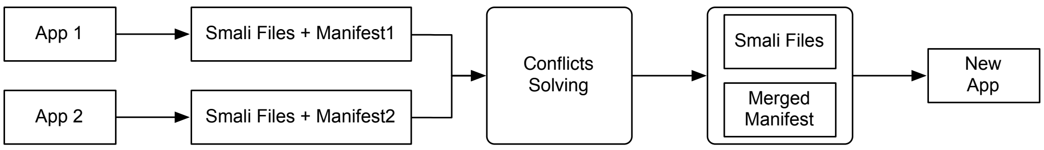


Figure 4.7: Working Steps of ApkCombiner.

[vii] https://code.google.com/p/android-apktool/

4.4.2 Resolution of Conflicts

Our prototype of ApkCombiner is focused on solving conflicts that may arise in the merging of code from two different apps. Such conflicts occur when two classes have the same name (up to the package level, i.e., the absolutely full qualified name). Thus, given class c_1 in app a_1 and class c_2 in app a_2, if $name(c_1) = name(c_2)$, we consider that there is a conflict between a_1 and a_2.

Fig. 4.8 illustrates the process of conflict checks we use. ApkCombiner considers that there is no conflict when two classes are named differently. If the name of two classes are the same, ApkCombiner distinguishes two cases according to the content of the classes. In a first type of conflict, the classes share the same name and their content is also the same (after verification of their footprint with the cryptographic hash), In this case, one copy of the class files is simply dropped. In the second type of conflict, i.e., when the content of the conflicting files are different, a thorough refactoring is necessary. This type of conflict occurs when, for example, two classes are actually from two different versions of the same library used in the two apps.

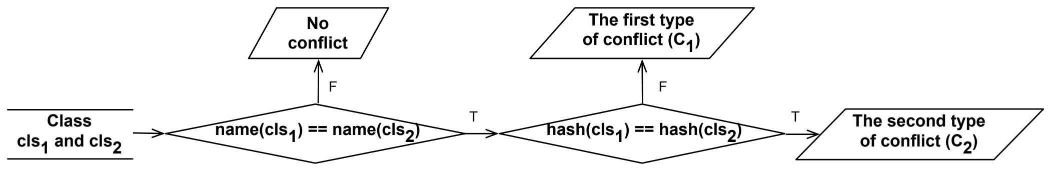


Figure 4.8: The Conflict Checking Process of ApkCombiner. Class cls_1 and cls_2 are from Different Apps.

Algorithm 1 details the described strategy for solving conflicts during merging as implemented by the procedure `CheckAndSolveConflicts()`. Given two sets ($set1$ and $set2$) of class files corresponding to the code of two apps (a_1 and a_2), the algorithm must identify and manage all conflicts.

Algorithm 1 Checking and Solving Conflicts

```
1: procedure CheckAndSolveConflicts(set1, set2)
2:     $confliSameMap \leftarrow new\ Map()$
3:     $confliDiffMap \leftarrow new\ Map()$
4:     for all $cls1 \in set1$ do
5:         if $set2.contain(cls1)$ then
6:             $cls2 \leftarrow set2.get(cls1)$
7:             if $hash(class(cls1))\ ==\ hash(class(cls2))$ then
8:                 $confliSameMap.put(cls1, cls2)$
9:             else
10:                $confliDiffMap.put(cls1, cls2)$
11:            end if
12:        end if
13:    end for
14:    if $empty(confliSameMap, confliDiffMap)$ then
15:        return
16:    end if
17:    for all $cls1, cls2 \in confliSameMap$ do
18:        $remove\ class(cls2)$
19:        if $isComponent(cls2)$ then
20:            $remove\ cls2\ from\ Manifest2$
21:        end if
22:    end for
23:    for all $cls1, cls2 \in confliDiffMap$ do
24:        $rename\ cls2$
25:        $solvingDependence(cls2, set2)$
26:        if $isComponent(cls2)$ then
27:            $rename\ cls2\ in\ Manifest2$
28:        end if
29:    end for
30: end procedure
```

First, two maps, referred to as $confliSameMap$ and $confliDiffMap$ are created to keep track of the classes that belong to the two types of conflict (lines 2-3). After identifying the kind of conflict that exists for each pair of classes across the two sets, the algorithm can attempt to solve the eventual conflicts. This resolution is performed in a two-step process. In step 1 (lines 17-22), the algorithm addresses the cases of type 1 conflicts. In step 2, type 2 conflicts are solved by refactoring the code.

Refactoring the code to solve conflicts is not as straightforward as renaming the conflicting classes. Indeed, there is a lot of dependencies to consider within the code of other classes. Procedure `solvingDependence()`, in line 25, is used to handle these dependencies, where we take into account three types of dependencies: 1) for a given class c we need to rename, another class c_i may use it as one of its attribute, 2) method m_i of class c_i may hold a parameter of c and 3) statement s_i of method m_i may use c as a variable. For the third type of dependency, we deal with statements that instantiate the variable as well as access the variable's attributes and methods because only such statements hold information related to class c.

To combine multiple Android apps to one, we need not only to integrate the different apps' bytecode, but also to merge their Manifest files. In particular the merge of Manifest files must take into account the fact that some classes where dropped while others were renamed. If those classes represent Android components, and not helper code, these changes should be reflected in the final Manifest of the new app (line 20 and 27).

4.5 Evaluation

Our evaluation addresses the following research questions:

RQ1 How does IccTA compare with existing tools?

RQ2 Can IccTA find ICC leaks in real-world apps?

RQ3 Can ApkCombiner support IccTA for IAC leaks?

RQ4 What is the runtime performance of IccTA and ApkCombiner?

All the experiments discussed in this section are performed on a Core i7 CPU running a Java VM with 8GB of heap size.

4.5.1 RQ1: Comparing IccTA With Existing Tools

In this research question, we compare IccTA with four existing tools: FlowDroid [10], IBM AppScan Source 9.0[viii], DidFail [112] and Amandroid [226]. FlowDroid is a state-of-the-art open-source tool for intra-component static taint analysis, AppScan Source is a commercial tool released by IBM, while DidFail and Amandroid are two recent state-of-the-art tools for detecting Android ICC leaks. All the tools are able to directly analyze Android bytecode except AppScan Source, which is only able to analyze the source code of the apps. Unfortunately, we were unable to compare IccTA with other static taint analysis tools as either they fail to report any leaks (e.g., SCanDroid [75]) or their authors did not make them available (e.g., SEFA [229]).

[viii]http://www-03.ibm.com/software/products/en/appscan-source

4.5.1.1 Experimental Setup

We assess the efficacy of all aforementioned tools by running them against about 30 test cases, for ICC leaks, from two benchmarks: DroidBench and ICC-Bench.

DroidBench. DroidBench is a set of hand crafted Android applications for which all leaks are known in advance. These leaks are used as *ground truth* to evaluate how well static and dynamic security tools find data leaks. As all the samples in DroidBench are intra-component privacy leaks, we developed 22 test cases to extend DroidBench with ICC leaks. The new set of test cases covers each of the top 8 ICC methods in Table 4.1. Among the 22 new test case applications, we included four (`startActivity{4,5,6,7}`) that do not contain any privacy leaks and thus will help detect false alarm rates of analysis tools. Finally, for each test case application we add an unreachable component containing a sink. These unreachable components are used to flag tools that do not properly construct links between components.

ICC-Bench. ICC-Bench is another set of apps introduced by Amandroid [226]. It contains 9 test case applications, where one of them uses explicit Intents, 6 of them use implicit Intents and the remaining two use dynamic techniques to register the target component. However, each of the test case app contains only one ICC leak and do not contain any unreachable component as DroidBench does. Because the source code of apps in the ICC-Bench were not available, we could not evaluate AppScan on this benchmark.

4.5.1.2 ICC Data Leak Test

Table 4.3 presents the results for comparing how related tools perform in the detection of ICC leaks. All 31 (22 added to DroidBench + 9 from ICC-Bench) test cases, and the corresponding detection outcome for the tools are listed in this table.

FlowDroid. Because FlowDroid has already been evaluated on the first version of DroidBench [10], we present in Table 4.3 its test results for the newly added 22 test cases which are dedicated to ICC leaks. Although, as mentioned earlier, FlowDroid was initially proposed to detect leaks in single Android components, we can use FlowDroid in a way that it computes paths for all individual components and then combines all these paths together (whether there is a real link or not). Thus, we expect FlowDroid to detect most of the leaks, although with false positives. Results of Table 4.3 confirm this, since FlowDroid shows a high recall (70.0%) and a low precision (27.4%). Furthermore, FlowDroid misses three more leaks than IccTA in `bindService{2,3,4}`. After investigation, we discovered that this is due to the fact that FlowDroid does not consider some callback methods for service components.

AppScan Source 9.0. AppScan requires a lot of manual initialization work since it has no default sources/sinks configuration file and is unable to analyze Android applications without specifying the entry points of every component. We define the `getDeviceId` and `log` methods, which we always use in DroidBench for ICC leaks, as source and sink, respectively. We also add all components' entry point methods (such as `onCreate` for Activities) as callback methods so AppScan knows where to start the analysis. AppScan is natively unable to detect inter-component data-flows and only detects intra-component flows. AppScan has the same drawbacks as FlowDroid and should have a high recall and low precision on DroidBench. We use an additional script to combine the flows between components. As expected, AppScan's recall is high (56.5%) and its precision is low (21.0%). Compared to FlowDroid, AppScan does worse. Indeed, AppScan does not correctly handle `startActivityForResult` and thus misses leaks going through methods receiving results from the called Activities in `startActivityForResult{2,3,4}` test cases.

DidFail. Since DidFail does not handle explicit ICC, it fails to report leaks for test cases that use explicit Intents between components. For implicit ICC, it is able to report all the leaks for test cases `Implicit{1,2,3,4,5,6}` even when those implicit ICCs use advanced features like `mimetype` or `data`. However, DidFail fails on case `startActivity{4,5}` test cases indicate that DidFail is not sensitive on `mimetype` and `data`. Our assumption is that DidFail uses an over-approximation approach to build implicit ICC links. As long as `action` and `category` are matched, an ICC link is constructed. Indeed, `startActivity{4,5}` use `mimetype` or `data`, but do not contain any real ICC link. Because DidFail currently only focuses on `Activity`, it fails to report any leak for the `Service`, `Broadcast Receiver` (dynamically registered or not) and `Content Provider` test cases.

Amandroid. Amandroid is the most recent state-of-the-art tool that is able to detect ICC leaks. Overall, for the 31 test cases, Amandroid reaches a precision of 78.9% (15 true positives, 4 false positives) and a recall of 51.7% (14 missed leaks). Three of the missed leaks and two of the false alarmed leaks are caused by `startActivityForResult`, where Amandroid is not able to combine `setResult` method to `onActivityResult` method. The `startService2` test case uses `IntentService` instead of `Service` which is used by test case `startService1` to implement the service. Amandroid is able to report a leak on `startService1` but fails to report a leak on `startService2`. This indicates that it does not completely model `Service`'s lifecycles. When the callback method changes from `onStartCommand` to `onHandleIntent`, Amandroid is not able to deal with it anymore. Eight other missed leaks indicate that Amandroid currently does not handle the `bindService` method and `Content Provider` components. Amandroid reports two false positives for `startActivity{6,7}`, which indicates that it is not able to distinguish the extra keys of an Intent. Indeed, `startActivity{6,7}` do not contain any leaks because they use different extra keys for the transferred Intent. Finally, Amandroid misses a leak on test case `DynRegister2` because `DynRegister2` uses string operations (e.g., `StringBuilder` Objects to contact multiple strings) which Amandroid cannot parse.

IccTA. Our tool, IccTA, also misses a leak on case `DynRegister2` like Amandroid, because, currently, it cannot parse complicated string operations as well. The same reason causes IccTA to yield a false positive on case `startActivity7`, where one extra key is built through complicated string operations. The current version of IccTA performs a simple string analysis to distinguish the extra keys of an Intent between one another.

4.5.2 RQ2: Detecting ICC Leaks

To evaluate our approach, we launch IccTA on two Android app sets: 1) *MalGenome* which contains 1260 Malware apps and 2) from *GooglePlay*, with 15,000 randomly selected apps.

For *MalGenome*, IccTA reports 108 apps ($r_{MalGenome} = 8.7\%$) containing at least one ICC leak, with a total of 534 leaks. And for *GooglePlay*, IccTA detects 337 apps ($r_{GooglePlay} = 2.2\%$) with 2,395 ICC leaks. Since $r_{MalGenome}$ is significantly higher than $r_{GooglePlay}$, we can conclude that malware indeed use ICC to leak private data. We further studied the correlations between the number of Intents and the number of detected ICC leaks for the two data sets. In this study, only apps that contain ICC leaks are considered. Interestingly, our results show that there is no correlation for *GooglePlay* apps. However, there exists a positive correlation for *MalGenome*. The Spearman's rho for *MalGenome* yielded the value 0.42 (p-value < 0.001), suggesting that the malware do use ICC to leak private data.

In total, IccTA detects 445 (108 + 337) apps from the *MalGenome* and *GooglePlay* sets. We summarize the most frequently used *source* methods and *sink* categories (Java classes) from those apps in Table 4.4. The most used *source* method is `getLongitude`: it is used 427 times. The most used *sink* category is `SharedPreferences`: it is used 1188 times. The reason why we study *sink* category instead of *sink* methods is that there are a lot of *sink* methods belonging to a same *sink* category (e.g., Log *sink* category includes eight *sink* methods which save private data to disk).

We further studied the $< sourceMethod, sinkCategory >$ pairs of the detected leaks. We found that the most frequently used pair is $< getLongitude, SharedPreferences >$, which happened

Table 4.3: Test Results on DroidBench and ICC-Bench, where Multiple Circles in One Row Means Multiple Leaks Expected and an All Empty Row Means no Leaks Expected as well as no Leaks Reported.

† Indicates the Tool Crashed on that Test Case. Because FlowDroid and AppScan are not Able to Directly Report ICC Leaks, We Try Our Best to Manually Match Their Results to Report ICC Leaks. For the Rest Tools, We only Consider Their Reported ICC Leaks.

⊛ = True Positive (Correct Warning), ⋆ = False Positive (False Warning), ○ = False Negative (Missed Leak)

Test Case	# of Comp.	Unreachable Comp.	Explicit ICC	FlowDroid	AppScan	DidFail	Amandroid	IccTA
				DroidBench				
startActivity1	3	T	T	⊛ ⋆	⊛ ⋆	○	⊛	⊛
startActivity2	4	T	T	⊛ (4 ⋆)	⊛ (4 ⋆)	○	⊛	⊛
startActivity3	6	T	T	⊛ (32 ⋆)	⊛ (32 ⋆)	○	⊛	⊛
startActivity4	3	T	F	⋆ ⋆	⋆ ⋆	⋆		
startActivity5	3	T	F	⋆ ⋆	⋆ ⋆	⋆		
startActivity6	3	T	T	⋆ ⋆	⋆ ⋆		⋆	
startActivity7	3	T	T	⋆ ⋆	⋆ ⋆		⋆	⋆
startActivityForResult1	3	T	T	⊛	⊛	○	⊛	⊛
startActivityForResult2	3	T	T	⊛	○	○	○	⊛
startActivityForResult3	3	T	T	⊛ ⋆	○	○	○ ⋆	⊛
startActivityForResult4	3	T	T	⊛ ⊛ ⋆	⊛ ○	○ ○	⊛ ○ ⋆	⊛ ⊛
startService1	3	T	T	⊛ ⋆	⊛ ⋆	○	⊛	⊛
startService2	3	T	T	⊛ ⋆	⊛ ⋆	○	○	⊛
bindService1	3	T	T	⊛ ⋆	⊛ ⋆	○	○	⊛
bindService2	3	T	T	○	○	○ †	○	⊛
bindService3	3	T	T	○	○	○ †	○	⊛
bindService4	3	T	T	⊛ ⋆ ○	⊛ ⋆ ○	○ ○	○ ○	⊛ ⊛
sendBroadcast1	3	T	F	⊛ ⋆	⊛ ⋆	○	⊛	⊛
insert1	3	T	F	○	○	○	○	⊛
delete1	3	T	F	○	○	○	○	⊛
update1	3	T	F	○	○	○	○	⊛
query1	3	T	F	○	○	○	○	⊛
				ICC-Bench				
Explicit1	2	F	T	⊛	-	○	⊛	⊛
Implicit1	2	F	F	⊛	-	⊛	⊛	⊛
Implicit2	2	F	F	⊛	-	⊛	⊛	⊛
Implicit3	2	F	F	⊛	-	⊛	⊛	⊛
Implicit4	2	F	F	⊛	-	⊛	⊛	⊛
Implicit5	3	F	F	⊛ ⋆	-	⊛	⊛	⊛
Implicit6	2	F	F	⊛	-	⊛	⊛	⊛
DynRegister1	2	F	F	○	-	○ †	⊛	⊛
DynRegister2	2	F	F	○	-	○ †	○	○
				Sum, Precision, Recall and F_1				
⊛ , higher is better	-	-	-	20	10	6	15	28
⋆ , lower is better	-	-	-	53	46	2	4	1
○ , lower is better	-	-	-	9	10	23	14	1
Precision p = ⊛/(⊛ + ⋆)	-	-	-	27.4%	17.9%	75%	78.9%	96.6%
Recall r = ⊛/(⊛ + ○)	-	-	-	70.0%	50.0%	20.7%	51.7%	96.6%
F_1-measure $2pr/(p+r)$	-	-	-	0.39	0.26	0.32	0.63	0.97

208 times. For example, in *MalGenome*, app *com.evilsunflower.farmer* obtains its longitude in class `SetPreferences` and transfers it into component `PushService`, in which the longitude is leaked. It also frequently happened in *GooglePlay* such as in app *infire.beautyleg.sexy.girls* and *ro.an.moneymanagerfree*.

Now, we give one case study to describe the detail of a leak. *com.wanpu.shuijinddp (version 11)* is an app in which an ICC leak has been reported by IccTA. It takes the device id (we consider the device id as sensitive data) as an unique user id to communicate with a remote server[ix] via HTTP. It first reads the device id and stores the id to a private field in class `com.waps.AppConnect`. Then, method `showOffers` of class `AppConnect` transfers the device id to component `OffersWebView` in which the device id has been sent to a remote server through a HTTP parameter. In this case, the device id has been leaked to a specified remote server through an ICC. Besides, the device id may be captured by hackers since it only uses HTTP instead of HTTPS to communicate with the remote server.

Finally, we investigated the total reported leaks and we found that 1812 out of 2929 (61.9%) leaks are leaked through `Service` components. These findings are interesting since using ICC makes leak detection difficult for analysis tools, while using `Services` hides those leaks to the user. Indeed Service components are running in the background with no interaction with the user (contrary to Activity components).

[ix] http://app.dwap.com:8000/action/

Table 4.4: The Top 5 Used Source Methods and Sink Categories

Method/Type	**Counts(#)**	**Detail**
	Source Methods	
getLongitude	427	get longitude
getLatitude	302	get latitude
getDeviceId	289	get IMEI or ESN
getLastKnownLocation	141	get location
getLine1Number	71	get phone no. of line 1
	Sink Categories	
SharedPreferences	1188	putInt, putString
HTTP	665	execute
Log	301	error or warn
File	38	write(string)
Message	15	sendTextMessage

4.5.3 RQ3: Detecting IAC Leaks

We now leverage IccTA to validate our approach for investigating the effectiveness of ApkCombiner in supporting existing tools for performing inter-app analyses.

With ApkCombiner we build app packages by combining pairs of apps. We then feed IccTA with these newly generated apps and assess its analysis results. We evaluate the use of IccTA in combination with ApkCombiner in two steps. In the first step, we evaluate the impact of ApkCombiner on DroidBench, which includes three test cases related to inter-app communication leaks. We found that IccTA is able to report inter-app privacy leaks for the analyzed apps by analyzing the combined package provided by ApkCombiner. To the best of our knowledge, DidFail is currently the only tool which claims to be able to perform static inter-app analysis for privacy leaks. We therefore compare DidFail with our approach associated to an existing state-of-the-art tool for intra-app analysis. The results in Table 4.5 based the DroidBench benchmark show that IccTA, while it cannot handle inter-app analysis alone, outperforms DidFail when it is supported by ApkCombiner. The reason why DidFail fails on two test cases is that at the moment DidFail only focuses on Activity-based privacy leaks.

Table 4.5: Comparison between IccTA, DidFail and ApkCombiner+IccTA.

Test Case (from DroidBench)	IccTA	DidFail	ApkCombiner+IccTA
InterAppCommunication_startactivity1	✗	✓	✓
InterAppCommunication_startservice1	✗	✗	✓
InterAppCommunication_sendbroadcast1	✗	✗	✓

In the second step, we evaluate ApkCombiner on 3,000 real Android apps. We first build an IAC graph through the results of our extended Epicc [124, 181], where an app stands for a node and an inter-app communication is modeled as an edge. For each of such edges, we launched ApkCombiner on the associated pair of apps and then used IccTA on the generated app.

We were thus able to discover an IAC leak between app *Ibadah Evaluation*[x] and app *ClipStore*[xi]. In the Ibadah Evaluation apk code, the source method *findViewById* is called in component `com.bi.mutabaah.id.activity.Statistic`, where the data of a *TextView* is obtained. Then this data is stored into an Intent along with two extras, a subject named `android.intent.extra.SUBJECT` and the text referred to as `android.intent.extra.TEXT`. Subsequently, the triggering method *startActivity* is used to transfer the Intent data to the ClipStrore app which extracts the data from the Intent with the same extra names and writes all the data into a file named `clip.txt`. Note that we consider saving sensitive data into disk as a leak.

[x]https://worldapks.com/ibadah-evaluation/, `com.bi.mutabaah.id` in our dataset.
[xi]https://worldapks.com/clipstore/, `jp.benishouga.clipstore` in our dataset.

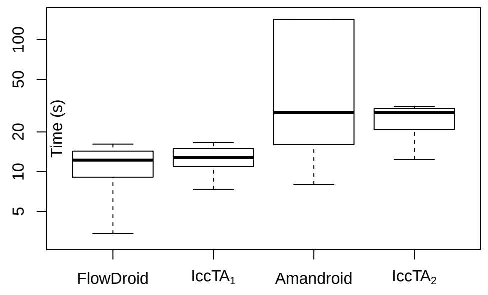


Figure 4.9: The Runtime Performance Comparison among Amandroid, FlowDroid and IccTA. IccTA$_1$ does not Count the ICC Links Extraction Time while IccTA$_2$ does. All the Experiments are Performed with the Default Options.

4.5.4 RQ4: Time Performance

We now investigate the time complexity of IccTA and ApkCombiner.

Performance of IccTA. We present the runtime performance analysis of FlowDroid, Amandroid and IccTA in Fig. 4.9. We randomly selected 50 apps from our *Googleplay* set for our study. Among those, only 18 apps have been successfully analyzed by all three tools altogether.

First, we compare the performance between FlowDroid and IccTA$_1$ to check whether our bytecode instrumentation step influences the final performance or not. As shown in Fig. 4.9, the performance of IccTA$_1$ is almost as good as FlowDroid. Indeed, an ICC link introduces 50 lines of *Jimple* code on average, which is negligible comparing to the total code lines (e.g., 412,090 lines for 1 megabyte bytecode on average).

Second, we compare the performance between IccTA$_2$ and Amandroid. In this case, we take into account the ICC links extraction time for a fair comparison since Amandroid also builds the ICC links. Fig. 4.9 shows that the median values of IccTA and Amandroid are similar. However, the runtime performance of IccTA$_2$ presents significantly less variation than Amandroid's, suggesting that Amandroid is highly sensitive to different properties (e.g., size) of the app.

Performance of ApkCombiner. The evaluation of time performance investigates the scalability of our approach. Indeed, a user may have on its device dozens apps that cohabit together. Thus, the inter-app analysis may require a fast combination of all those apps. Fig. 4.10 plots the running times[xii] of ApkCombiner for each combined app. The running time is plotted against the sum size of each pair of apps (we use the bytecode size, as resource files that are not considered in the merging may introduce a bias).

Let C_1 and C_2 represent the successful combinations where conflicts of, respectively, first type and second type were solved. Consequently, $C_1 \cup C_2 \cup \overline{C_1 \cup C_2}$ represents all the successful combinations.

Fig. 7.4a plots the time performance for all combinations. The linear regression between the plots shows that there is a correlation between the execution time and the bytecode size. Comparing with Fig. 7.4b, we note that the slope of the regression is lower when we do not consider combinations that lead to conflicts. The limited difference in slope values (0.246 against 0.157) indicates that the conflict solving module is not a runtime bottleneck.

The differences between Fig. 4.10c, Fig. 4.10d and Fig. 4.10e further confirm how the resolution of second type conflicts requires more execution time than the resolution of first type conflicts.

[xii]Note that in this chapter we consider the wall clock time (from start to finish of the execution). That means not only the actual CPU time but also the waiting time (e.g., waiting for I/O) are taken into account.

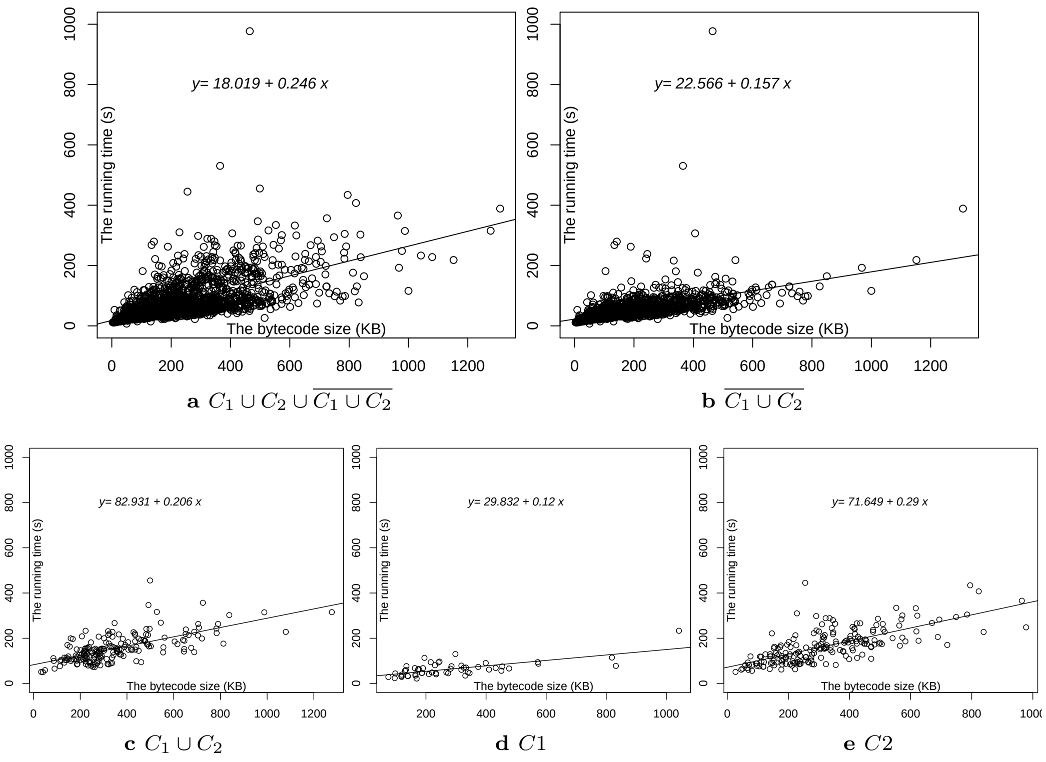


Figure 4.10: Time Performance against the Byte Code Size. C_1 Represent the Set of Combinations where First Type Conflicts were Solved, while C_2 Represents the Set of Combinations with Second Type Conflicts.

4.6 Related Work

As far as we know, IccTA and ApkCombiner are the first approach to seamlessly connect Android components (or apps) through code instrumentation in order to perform ICC based static taint analysis [127–129]. By using a code instrumentation technique [9], the state of the context and data (e.g., an *Intent*) is transferred between components.

Amandroid [226] performs an ICC analysis to detect ICC leaks, and has been developed concurrently with IccTA. Amandroid needs to build an Inter-component Data Flow Graph (ICDFG) and an Data Dependence Graph (DDG) to perform ICC analysis. Since IccTA uses an instrumentation approach, it does not need to additionally build such assistant graphs. Amandroid provides a general framework to enable analysts to build a customized analysis on Android apps. IccTA provides a *source/sink* configuration to achieve the same function. Amandroid is not able to analyze `Content Provider` as well as some ICC methods such as `bindService` and `startActivityForResult`. Finally, our instrumentation approach is more flexible, and enables generating an app with all components linked at the code level. This app can then be analyzed by any static analysis tool (e.g., Soot or Wala[xiii]).

DidFail [112] also leverages FlowDroid and Epicc to detect ICC leaks. Currently, it focuses on ICC leaks between `Activities` through implicit Intents. Thus, it will miss leaks involving explicit Intents and components other than `Activities`. Also, it does not handle some parameters for implicit Intents (such as `mimetype` and `data`) and thus generates false links between components. The consequence of that is a higher false positive rate.

Bati [17] is another approach proposed in this field. It features a context-, flow-, object-, and path-sensitive analysis for Android apps. Besides, it also models the lifecycle methods of components including even the asynchronous communicated threads.

[xiii] `http://wala.sourceforge.net`

SCanDroid [75] and SEFA [229] are another two tools that perform ICC analysis. However, neither of them keeps the context between components and thus are less precise than IccTA by design. ComDroid [48] and Epicc [181] are two tools that tackle the ICC problem, but mainly focus on ICC vulnerabilities and do not taint data. CHEX [158] is a tool to detect component hijacking vulnerabilities in Android applications by tracking taints between sensitive sources and externally accessible interfaces. However, it is limited to at most 1-object-sensitivity which leads to imprecision in practice. PCLeaks performs data-flow analysis to detect potential component leaks, which not only includes component hijacking vulnerabilities, but also component launch (or injection) vulnerabilities [121, 122, 125]. ContentScope [258] is another tool that tackles potential component leaks, but it only analyzes `Content Provider` components.

Multiple works use static analysis to detect intra-component privacy leaks in Android apps [10, 16, 79, 110, 165, 237]. AndroidLeaks [79] states the ability to handle the Android lifecycle including callback methods, but it is not context-sensitive which precludes the precise analysis of many practical scenarios. AsDroid [100] and AppIntent [238] are two other tools using static analysis to detect privacy leaks in Android apps. Both of them try to analyze if a data leak is a feature of the application or not. This kind of analysis is out of the scope of this chapter.

Multiple prior works investigated privacy leaks on systems other than Android. PiOS [65] uses program slicing and reachability analysis to detect the possible privacy leaks in iOS apps. TAJ [218] and Andromeda [217] uses the same taint analysis technique to identify privacy leaks in web applications.

Except privacy leaks detection, there has been a rich body of work on other Android security issues [23, 25, 72, 92, 177, 259] such as energy bugs [51, 148] and SSL vulnerabilities [70, 207]. Our work can complement their research by providing a highly precise control-flow graph to enable them to perform inter-component data-flow analysis and consequently to get better results.

Other approaches dynamically track the sensitive data to report security issues. TaintDroid [67] is one of the most sophisticated dynamic taint tracking system. TaintDroid uses a modified Dalvik virtual machine to track flows of private data. CopperDroid [193] is another dynamic testing tool which observes interactions between Android components and the Linux system to reconstruct high-level behavior and uses some special stimulation techniques to exercise the app to find malicious activities. Several other systems, including AppFence [95], Aurasium [232], AppGuard [18] and BetterPermission [22] try to mitigate the privacy leak problem by dynamically monitoring the tested apps.

However, those dynamic approaches can be fooled by specifically designed methods to circumvent security tracking [200]. Thus, dynamic tracking approaches may miss some data leaks and yield an under-approximation. On the other hand, static analysis approaches may yield an over-approximation because all the application's code is analyzed even code that will never be executed at runtime. These two approaches are complementary when analyzing Android applications for data leaks.

4.7 Conclusion

This chapter addresses the major challenge of performing data-flow analysis across multiple components for Android apps. We have presented IccTA, an open source tool, to perform ICC based taint analysis. In particular, we demonstrate that IccTA can detect ICC based privacy leaks by providing a highly precise control-flow graph through instrumentation of the code of applications. Unlike previous approaches, IccTA enables a data-flow analysis between two components and adequately models the lifecycle and callback methods to detect ICC based privacy leaks. When running IccTA on DroidBench and ICC-Bench, it reaches a precision of 96.6% and a recall of 96.6%. When running IccTA on a set of 1,260 apps of the MalGenome project, it reports 534 ICC leaks in 108 apps (8.6%). When running IccTA on a set of 15,000 real-world apps randomly selected from Google Play market, it detects 2,395

ICC leaks in 337 apps (2.2%). Other existing privacy detecting tools (e.g., AndroidLeaks) could benefit by implementing our approach to perform ICC based privacy leaks detection.

Furthermore, we discussed ApkCombiner, a tool-based approach for reducing an Inter-App Communication problem into an intra-app Inter-Component Communication problem by combining multiple Android apps into one. After the combination, existing intra-app analysis approaches can be applied on the generated Android app to indirectly report inter-app results. Since we combine apps at code level, our approach is context-aware and general. We evaluate ApkCombiner to demonstrate that, despite a conflict resolution algorithm that requires a time-consuming refactoring process, the approach is scalable. We further showed that it can improve the capabilities of existing state-of-the-art tools. For example, we showed that using ApkCombiner can enable tools such as IccTA to discover IAC privacy leaks in real-world apps.

5 Taming Reflection

Android developers heavily use reflection in their apps for legitimate reasons, but also significantly for hiding malicious actions. Unfortunately, current state-of-the-art static analysis tools for Android are challenged by the presence of reflective calls which they usually ignore. Thus, the results of their security analysis, e.g., for private data leaks, are inconsistent given the measures taken by malware writers to elude static detection. We propose the DroidRA instrumentation-based approach, which generally shares the same idea as we applied in the previous section for bridging ICC, to address this issue in a non-invasive way. With DroidRA, we reduce the resolution of reflective calls to a composite constant propagation problem. We leverage the COAL solver to infer the values of reflection targets, and we eventually instrument this app to represent the corresponding traditional Java call for each reflective call. Our approach allows to boost an app so that it can be immediately analyzable, including by such static analyzers that were not reflection-aware. In this chapter, we evaluate DroidRA on benchmark apps as well as on real-world apps, and demonstrate that it can allow state-of-the-art tools to provide more sound and complete analysis results.

This chapter is based on the work published in the following paper:

- Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. DroidRA: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016
- Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Tool Demonstration Track*, 2016

Contents

5.1 Introduction

Reflection is a property that, in some modern programming languages, enables a running program to examine itself and its software environment, and to change what it does depending on what it finds [74]. In Java, reflection is used as a convenient means to handle genericity or to process Java annotations inside classes. Along with many Java features, Android has inherited the Java Reflection APIs which are packaged and included in the Android SDK for developers to use. Because of the fragmentation of the Android ecosystem, where many different versions of Android are concurrently active on various devices, reflection is essential as it allows developers, with the same application package, to target devices running different versions of Android. Indeed, developers may use reflection techniques to determine, at runtime, if a specific class or method is available before proceeding to use it. This allows the developer to leverage, in the same application, new APIs where available while still maintaining backward compatibility for older devices. Reflection is also used by developers to exploit Android hidden and private APIs, as these APIs are not exposed in the developer SDK and consequently cannot be invoked through traditional Java method calls.

Unfortunately, recent studies on Android malware have shown that malware writers are using reflection as a powerful technique to hide malicious operation [108]. In particular, reflection can be used to hide the real purpose, e.g., by invoking a method at runtime to escape static scanning, or simply to deliver malicious code [7]. We have conducted a quick review of recent contributions on static analysis-based approaches for Android, and have found that over 90% of publications [135] from top conferences (including ICSE and ISSTA) do not tackle reflection. Indeed, most state-of-the-art approaches and tools for static analysis of Android simply ignore the use of reflection [73, 100] or may treat it partially [81, 192]. By doing so, the literature has produced tools that provide analysis results which are *incomplete*, since some parts of the program may not be included in the app call graph, and *unsound*, since the analysis does not take into account some hidden method invocations or potential writes to object fields. In this regard, a recent study by Rastogi et al. [190] has shown that reflection has made most of the current static analysis tools perform poorly on malware detection.

Tackling reflection is however challenging for static analysis tools. There exist ad-hoc implementations (e.g., in [192]) for dealing with specific cases of reflection patterns. Such approaches cannot unfortunately be re-exploited in other static analysis tools. However, there is a nascent commitment in the Android research community to propose solutions for improving the analysis of reflection. For example, in a recent work [21], Barros *et al.* propose an approach for resolving reflective calls in their Checker static analysis framework [69]. Their approach however 1) requires application source code (which is not available for most Android apps), 2) targets specific check analyses based on developer annotations (which thus needs additional developer efforts, e.g., learn the target framework, find the right place to annotate.) and 3) does not provide a way to directly benefit existing static analyzers, i.e., to support them in performing reflection-aware analyses.

Our aim in this chapter is to deal with reflection in a non-invasive way so that state-of-the-art analysis tools can benefit from this work to better analyze application packages from app markets. To that end, we present DroidRA, an instrumentation-based approach, for automatically taming reflection in Android apps. In DroidRA, the targets of reflective calls are determined after running a constraint solver to output a regular expression satisfying the constraints generated by an inter-procedural, context-sensitive and flow-sensitive static analysis. A Booster module is then implemented to augment reflective calls with their corresponding explicit standard Java calls. Because some code can be loaded dynamically, before our reflection analysis, we also use heuristics to extract any would-be dynamically-loaded code (e.g., a jar file with *classes.dex* inside) into our working space. Indeed, our reflection taming approach hinges on the assumption that all the code that exist in the app package may be loaded at runtime and thus should be considered for analysis.

This chapter makes the following contributions:

- We provide insights on the use of reflection in Android apps, based on an analysis of 500 apps randomly selected from a repository of apps collected from Google Play, third-party markets, as well as malware samples [3]. Our findings show that 1) a large portion of Android apps relies on reflective calls and that 2) reflective calls are usually used with some common patterns. We further show that reflective calls can be discriminated between malicious and benign apps.
- We designed and implemented DroidRA – an approach that aims at boosting existing state-of-the-art static analysis for Android by taming reflection in Android apps. DroidRA models the use of reflection with COAL [180] and is able to resolve the targets of reflective calls through a constraint solving mechanism. By instrumenting Android apps to augment reflective calls with their corresponding explicit standard Java calls, DroidRA complements existing analysis approaches.
- We evaluated DroidRA on a set of real applications and report on the coverage of reflection methods that DroidRA identifies and inspects. We further rely on well-known benchmarks to investigate the impact that DroidRA has on improving the performance of state-of-the-art static analyzers. In particular, we show how DroidRA is useful in uncovering dangerous code (e.g., sensitive API calls, sensitive data leaks [10, 124]) which was not visible to existing analyzers.
- We release DroidRA and the associated benchmarks as open source[i], not only to foster research in this direction, but also to support practitioners in their analysis needs.

The remainder of this paper is organized as follows. Section 5.2 investigates the use of reflection in Android apps and motivates this work through a concrete example. Section 5.3 presents our approach DroidRA, including its design and implementing details. Section 5.4 reports on the evaluation of DroidRA while Section 5.5 discusses the related work. Finally, Section 5.6 concludes this chapter.

5.2 Motivation

To better motivate our work, in this section, we first look into the adoption of reflection in Android apps in Section 5.2.1. After that, we leverage a concrete example to show why taming reflection is important for static analysis of Android apps (cf. Section 5.2.2).

5.2.1 Reflection Adoption in Android Apps

To investigate the use of reflection in real Android apps, we consider a large research repository of over 2 millions apps crawled from Google Play, third-party markets and known malware samples [3]. We randomly select 500 apps from this repository and parse the bytecode of each app, searching for reflective calls. The strategy used consists in considering any call to a method implemented by the four reflection-related classes[ii] as a reflective call, except such methods that are overridden from `java.lang.Object`.

5.2.1.1 Overall usage of reflection

Our analysis shows that reflection usage is widespread in Android apps, with 87.6% (438/500) of apps making reflective calls. On average, each of the flagged apps uses 138 reflective calls. Table 5.1 summarizes the top 10 methods used in reflective calls.

[i]`https://github.com/serval-snt-uni-lu/DroidRA.git`

[ii] `java.lang.reflect.Field`, `java.lang.reflect.Method`, `java.lang.Class`, and `java.lang.reflect.Constructor`.

Table 5.1: Top 10 Used Reflection Methods and Their Argument Type: Either (C): `Class`, (M): `Method` or (F): `Field`.

Method (belonging class)	# of Calls	# of Apps
getName (C)	12,588	283 (56.6%)
getSimpleName (C)	5,956	87 (17.4%)
isAssignableFrom (C)	4,886	164 (32.8%)
invoke (M)	3,026	223 (44.6%)
getClassLoader (C)	2,218	163 (32.6%)
forName (C)	2,141	227 (45.4%)
getMethod (C)	1,715	135 (27.0%)
desiredAssertionStatus (C)	1,218	202 (40.4%)
get (F)	1,139	177 (35.4%)
getCanonicalName (C)	1,115	388 (77.6%)
Others	24,708	4 (8%)
Total	60,710	438 (87.6%)

Table 5.2: Top 5 Patterns of Reflective Calls Sequences.

Sequence pattern	Occurrences
Class.forName() → getMethod() → invoke()	133
getName() → getName() → getName()	120
getDeclaredMethod() → setAccessible() → invoke()	110
getName() → isAssignableFrom() → getName()	92
getFields() → getAnnotation() → set() → ...	88

We perform another study to check whether most reflective calls are only contributed by common advertisement libraries. We thus exclude reflective calls that are invoked by common ad libraries[iii]. Our results show that there are still 382 (76.4%) apps whose non-ad code include reflective calls, suggesting the use of reflection in primary app code.

5.2.1.2 Patterns of reflective calls

In order to have a clear picture of how one can spot and deal with reflection, we further investigate the sequences of reflective calls and summarize the patterns used by developers to implement Android program behaviour with reflection. We consider all method calls within the 500 apps and focus on the reflection-related sequences that are extracted following a simple, thus fast, approach considering the natural order in which the bytecode statements are yielded by Soot. We find 34,957 such sequences (including 1 or more reflective call). An isolated reflective call is relatively straightforward to resolve as its parameter is usually a String value (e.g., name of class to instantiate). However, e.g., when a method in the instantiated class must be invoked, other reflective calls may be necessary (e.g., to get the message name in object of class), which may complicate the reflection target resolution. We found 45 distinct patterns of sequences containing at least three reflective calls. Table 5.2 details the top five sequences: in most cases, reflection is used to access methods and fields of a given class which may be identified or loaded at runtime. This confirms the fundamental functionality of reflective calls which is to access methods/fields.

We further investigate the 45 distinct patterns to focus on reflective calls that are potentially dangerous as they may change program state. Thus we mainly focus on sequences that include a method invocation (sequences 1 and 3 in Table 5.2) or access a field value in the code (sequence 5). Taking into account all the relevant patterns, including 976 sequences, we infer the common pattern which is represented in Figure 5.1. This pattern illustrates how the reflection mechanism allows to

[iii]We take into account 12 common libraries, which are published by [13] and are also used by [121]. We believe that a bigger library set like the one provided by Li et al. [130] could further improve our results.

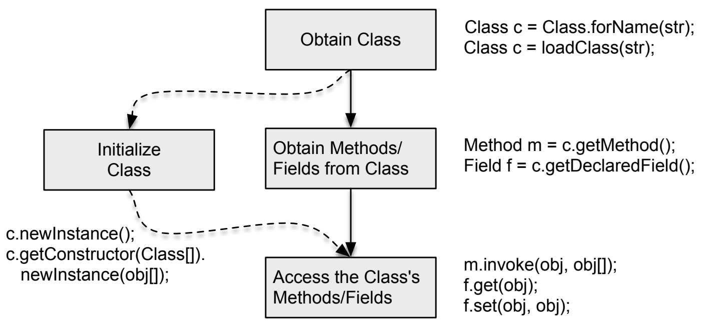

Obtain Class
Class c = Class.forName(str);
Class c = loadClass(str);
Initialize Class
Obtain Methods/ Fields from Class
Method m = c.getMethod();
Field f = c.getDeclaredField();
c.newInstance();
c.getConstructor(Class[]).
newInstance(obj[]);
Access the Class's Methods/Fields
m.invoke(obj, obj[]);
f.get(obj);
f.set(obj, obj);

Figure 5.1: Abstract Pattern of Reflection Usage and Some Possible Examples.

obtain methods/fields dynamically. These methods and fields can be used directly when they are statically declared (solid arrows in figure 5.1); they may otherwise require initializing an object of the class, e.g., also through a reflective call to the corresponding constructor (dotted arrows). With this common pattern, we can model most typical usages of reflection which can hinder state-of-the-art static analysis approaches.

The model yielded allows to consider different cases in reflective call resolution: In some simple cases, a string analysis is sufficient to extract the value of the call parameter; In other cases however, where class objects are manipulated to point to methods indicated in field values, simple string analysis cannot be used to help mapping the flow of a malicious operation. Finally, in some cases, there is a need to track back to the initialization of an object by another reflective call to resolve the target.

5.2.1.3 Reflection in malicious vs benign apps

Given the potential of using reflection to hide malicious operations, which could not be statically detected, we investigate whether reflection usage can constitute in discriminating between malicious and benign apps. We build a basic machine learning classification scheme with reflective calls as the unique feature set. For the experiments, we use RandomForest ensemble learning algorithm and build our ground truth by leveraging VirusTotal scanning results[iv] on the randomly selected 500 apps. 10-Fold cross validation experiments[v] yield a performance of 97.5% for F-Measure, the harmonic mean of precision and recall. Although such a high performance can be favored by our experimental settings, it nonetheless suggests that reflective calls may constitute a discriminate feature between malware and goodware.

5.2.2 Reflection Example

Millions of Android apps are spread in different markets with more and more security and privacy alerts from anti-virus vendors. A recent report from Symantec shows that in 2014 they classified about 1 million of Android apps as malware[vi] on a total of 6.3 millions of apps analyzed. These facts urge for practical and scalable approaches and tools targeting the security analysis of large sets of Android apps. As example of such approaches, static taint analyzers aim at tracking data across control-flow paths to detect potential privacy leaks.

Let us consider the FlowDroid [10] state-of-the-art approach as a concrete example. FlowDroid is used to detect private data leaks from sensitive sources, such as contact information or device identification numbers, to sensitive sinks, such as sending HTTP posts or short messages. FlowDroid

[iv]`https://www.virustotal.com`: we consider an app *a* to be malicious if any of the anti-virus products from VirusTotal has flagged it as malicious. Otherwise *a* is a benign app.

[v]To address the imbalance dataset problem (only 20 out of the 500 apps are malicious), we perform oversampling using SMOTE [43].

[vi]`http://know.symantec.com/LP=1123`

```
 1 TelephonyManager telephonyManager = //default;
 2 String imei = telephonyManager.getDeviceId();
 3 Class c = Class.forName("de.ecspride.ReflectiveClass");
 4 Object o = c.newInstance();
 5 Method m = c.getMethod("setIme" + "i", String.class);
 6 m.invoke(o, imei);
 7 Method m2 = c.getMethod("getImei");
 8 String s = (String) m2.invoke(o);
 9 SmsManager sms = SmsManager.getDefault();
10 sms.sendTextMessage("+49 1234", null, s, null, null);
```

Listing 5.1: Code Excerpt of *de.ecspride.MainActivity* from DroidBench's `Reflection3.apk`.

has demonstrated promising results, however, they suffer from limitations inherent to the challenges of static analysis in Android for taking into account reflection, class loading or native code support. In this chapter, we focus on taming reflection in Android apps to allow state-of-the-art tools such as FlowDroid to significantly improve their results. Reflection breaks the traditional call graph construction mechanism in static analysis, resulting in an incomplete control-flow graph (CFG) and consequently leading to insufficient results. Dealing with reflection in static analysis tools is however challenging. Even the Soot Java optimization framework, on top of which most state-of-the-art approaches are built, does not address the case of reflective calls in its analyses. Thus, overall, taming reflection at the app level will enable better analysis by state-of-the-art analysis tools to detect security issues for app users.

We consider the case of an app included in the DroidBench benchmark [10]. The *Reflection3* benchmark app is known to be improperly analyzed by many tools, including FlowDroid, because it makes use of reflective calls. In this example app (Listing 5.1), class `ReflectiveClass` is first retrieved (line 3) and initialized (line 4). Then, two methods (*setImei()* and *getImei()*) from this class are reflectively shipped and invoked (lines 5-8). *setImei()*, which is matched by concatenating two strings, will store the device ID, which was obtained at line 2, into field *imei* of class `ReflectiveClass` (line 6). *getImei()*, similarly, gets back the device ID into the current context so that it can be sent outside the device via SMS to a hard-coded (i.e., not provided by user) phone number (line 10).

The operation implemented in this code sample is malicious since the device ID is taken as sensitive private information. The purpose of the reflective calls, which appear between the obtaining of the device ID and its leakage outside the device, is to elude any taint tracking by confusing the traditional flow. Thus, statically detecting such leaks becomes non trivial. For example, analyzing string patterns can be challenged intentionally by developers, as it was done in line 5. Furthermore, simple string analysis is not enough to resolve reflective calls, because not only the method name (e.g., *getImei* for method $m2$) but also the method's declaring class name (e.g., *ReflectiveClass* for $m2$) are needed. These values must therefore be matched and tracked together: this is known as a composite constant propagation problem.

5.3 DroidRA

In this section, we first present an overview of the architecture of DroidRA in Section 5.3.1. More specifically, DroidRA adopts a three-module approach: JPM, RAM, and BOM, for which we will detail them in Section 5.3.2, Section 5.3.3, and Section 5.3.4, respectively.

5.3.1 Overview

Our work is directed towards a twofold aim: (1) to resolve reflective call targets in order to expose all program behaviours, especially for analyses that must track private data; (2) to *unbreak* app control-flow in the presence of reflective calls in order to allow static analyzers to produce more precise results.

Figure 5.2 presents an overview of the architecture of the DroidRA approach involving three modules. (1) The first module named *JPM* prepares the Android app to be properly inspected. (2) The second module named *RAM* spots reflective calls and retrieves the values of their associated parameters (i.e., class/method/field names). All resolved reflection target values are made available to the analysts for use in their own tools and approaches. (3) Leveraging the information yielded by the RAM module, the *BOM* module instruments the app and transforms it into a new app where reflective calls are augmented with standard java calls. The objective of BOM is to produce an equivalent app whose analysis by state-of-the-art tools will yield more precise results [120, 136].

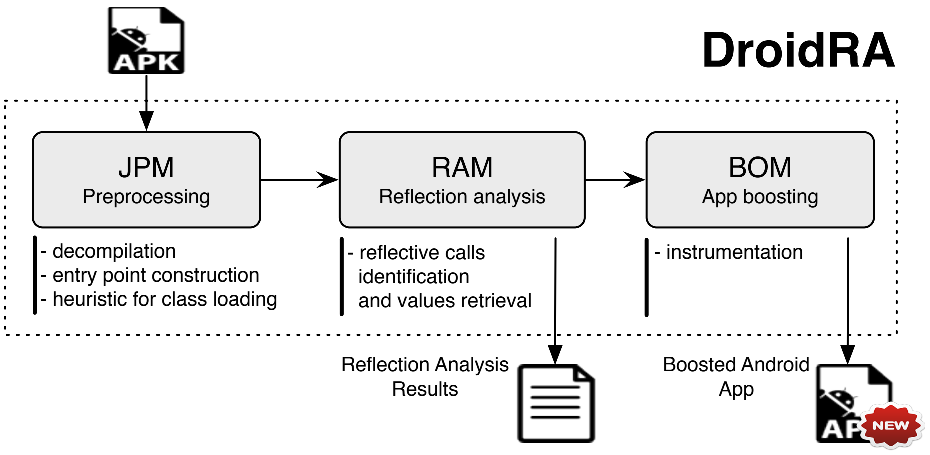

APK
DroidRA
JPM
Preprocessing
RAM
Reflection analysis
BOM
App boosting
- decompilation
- entry point construction
- heuristic for class loading
- reflective calls identification and values retrieval
- instrumentation
Reflection Analysis Results
Boosted Android App
NEW

Figure 5.2: Overview of DroidRA.

5.3.2 JPM – Jimple Preprocessing Module

Android programming presents specific characteristics that require app code to be preprocessed before Java standard analysis tools can be used on it. First, an Android app is distributed as an *apk* file in which the code is presented in the form of Dalvik bytecode, a specific format for Android apps. Our analysis and code instrumentation will however manipulate code in Jimple, the intermediate representation required by Soot [115], a Java optimization framework. As a result, in a first step JPM leverages the Dexpler [24] translator to decompile the apk and output Jimple code.

Second, similar to any other static approaches for Android, DroidRA needs to start analysis from a single entry-point. Unfortunately, Android apps do not have a well-defined entry-point, e.g., *main()* in Java applications. But instead, they have multiple entry-points since each component that declares `Intent Filters` (which defines the capabilities of a component) is a possible entry-point. To address this challenge, we use the same approach as in the FlowDroid [10] state-of-the-art work on Android analysis, that is, to artificially assemble a dummy main method, taking into account all possible components including their lifecycle methods (e.g., *onCreate()* and *onStop()*) and all possible callback methods (e.g., *onClick()*). This enables the static analyzer to build an inter-procedural control-flow graph and consequently to traverse all the app code.

Third, we aim to analyze the entire available app code, including such code that is dynamically loaded (e.g., at runtime). Dynamic Code Loading (DCL), however, is yet another challenge for static analysis, as some would-be loaded classes, which would be added at runtime (e.g., downloaded from a remote server), may not exist at all at static analysis time. In this work, we focus on dynamically loaded code that is included in the apk file (although in a separated archive file) and which can then be accessed statically. We assume that this way of storing locally the code to be dynamically is the most widespread. In any case, Google Play policy explicitly states that an app downloaded from

Google Play may not modify, replace or update its own APK binary code using any method other than Google Play's update mechanism[vii].

In practice, our DCL analysis is performed through heuristics: Given an app a, we first unzip[viii] it and then traverse all its embedded files, noted as set F. For each file $f \in F$, if it is a Java archive format (the file extension could vary from *dat*, *bin* to *db*), then we recursively look into it, to check whether it contains a *dex* file through its magic number (*035*). All retrieved *dex* (usually come with *classes.dex*) files are then taken into account for any further analysis of the app.

We tested this heuristics-based process for finding DCL code by analyzing 1,000 malicious apps randomly selected from our data set. We found that 348 (34.8%) apps contain additional code, which could be dynamically loaded at runtime. Among the 348 apps, we collect 1,014 archives that contain an extra *classes.dex* file, giving an average of 2.9 "archives with code" per app. We also found that the 1,014 archives are redundant in many apps: there are actually only 74 distinct archive names. For example, library *bootablemodule.jar* (which contains a *classes.dex* file) has been used by 115 apps. This library package was recently studied in a dynamic approach [251].

5.3.3 RAM – Reflection Analysis Module

The Reflection Analysis Module identifies reflective calls in a given app and maps their target string/object values. For instance, considering the motivating example from the DroidBench app presented in Listing 5.1, the aim with RAM is to extract not only the method name in the `m2.invoke(o)` reflective call (line 8 in Listing 5.1), but also the class name that $m2$ belongs to. In other words, we have to associate $m2$ with *getImei*, but also o with *de.ecspride.ReflectiveClass*. To that end, based on the motivation example and our study of reflective call patterns, we observe that the reflection problem can be modeled as a constant propagation problem within an Android Inter-procedural Control-Flow Graph. Indeed, mapping a reflective call eventually consists in resolving the value of its parameters (i.e., name and type) through a context-sensitive and flow-sensitive inter-procedural data-flow analysis. The purpose is to obtain highly precise results, which are very important since the app will be automatically instrumented without any manual check of the results. Let us consider the resolution of the value of $m2$ in line 8 ('*String s = (String) m2.invoke(o)*' in Listing 5.1) as an example: if we cannot precisely extract the class name that $m2$ belongs to, say, our RAM tells that $m2$ belongs to class *TelephonyManager*, rather than the right class *ReflectiveClass*, then, during instrumentation, we will write code calling $m2$ as a member of *TelephonyManager*, which would yield an exception at runtime (e.g., no such method error), and consequently fail the static analysis.

To build a mapping from reflective calls to their target string/object values, our static analysis adopts an inter-procedural, context-sensitive, flow-sensitive analysis approach leveraging the composite COnstant propAgation Language (COAL) [180] for specifying the reflection problem. In order to use COAL, the first step is to model the reflection analysis problem independently from any app using the abstract pattern of reflective call inferred in Section 5.2.1.2. This generic model is specified by composite objects, e.g., a reflective method is being specified as an object (in COAL) with two fields: the method name and its declaring class name. Once reflection analysis has been modeled, we build on top of the COAL solver to implement a specific analyzer for reflection. This analyzer then performs composite constant propagation to solve the previously defined composite objects and thereby to infer the reflective call target values.

COAL-based Reflection Analysis. We now illustrate a simple example shown in Listing 5.2 to better explain the constant propagation of reflection-related values for class `Method`. Specifications for all other reflection-related classes are defined similarly. All specifications will be open-sourced eventually. Based on the specification shown in Listing 5.2, the COAL solver generates the semilattice that represents the analysis domain. In this case, the `Method` has two string fields, where Class

[vii] `https://play.google.com/about/developer-content-policy.html`
[viii] The format of an *apk* is actually a compressed ZIP archive.

```
 1 //Java/Android code
 2 Class c; Method m;
 3 if (b) {
 4 c = first.Type.class;
 5 m = c.getMethod("method1");
 6 } else {
 7 c = second.Type.class;
 8 m = c.getMethod("method2");
 9 }
10 m.invoke(someArguments);
11 //Simplified COAL specification (partial)
12 class Method {
13 Class declaringClass_method;
14 String name_method;
15 mod gen <Class: Method getMethod(String,Class[])>{
16 -1: replace declaringClass_method;
17 0: replace name_method; }
18 query <Method: Object invoke(Object,Object[])>{
19 -1: type java.lang.reflect.Method; }
20 }
```

Listing 5.2: Example of COAL-based Reflection Analysis for Class *Method*. Similar Specifications Apply to All Other Reflection Classes.

types (strings of characters) are modeled as fully qualified class names. In the COAL abstraction, each value on an execution path is represented by a tuple, in which each tuple element is a field value. More formally, let $\mathcal{S}$ be the set of all strings in the program and let $B = (S \cup \{\omega\}) \times (S \cup \{\omega\})$, where ω represents an unknown value. Then the analysis domain is the semilattice $L = (2^B, \subseteq)$, where for any set X, 2^X is the power set of X, and elements in 2^B represent the set of values of `Method` variables across all execution paths. Semilattice L has a bottom element $\perp = \varnothing$ and its top element is the set of all elements in B. For example, the following equation models the value of object `m` at line 10 of Listing 5.2:

$$\{(\texttt{first.Type}, \texttt{method1}), (\texttt{second.Type}, \texttt{method2})\} \tag{5.1}$$

The first tuple in Equation (5.1) represents the value of Method object `m` contributed by the first branch of the `if` statement. The second tuple, on the other hand, models the value on the fall-through branch.

In order to generate transfer functions for the calls to `getMethod`, the COAL solver relies on the specification presented in lines 15-17 of Listing 5.2. The modifier `mod` statement specifies the signature of the `getMethod` method and it describes how the method modifies the state of the program. The `gen` keyword specifies that the method generates a new object of type Method (i.e., it is a factory function). Statement `-1: replace declaringClass_method` indicates that the name of the Class object on which the method is called (e.g., `first.Type` at line 4) is used as the field `declaringClass_method` of the generated object. Note that in this statement the special `-1` index indicates a reference to the instance on which the method call is made, for example object `c` at line 5. Finally, statement `0: replace name_method` indicates that the first argument (as indicated by index 0) of the method is used as the `name_method` field of the generated object.

At the start of the propagation performed by the COAL solver, all values are associated with $\perp$. Then the COAL solver generates transfer functions that model the influence of program statements on the values associated with reflection. Following the formalism from [180], for any $v \in L$, we define function $init_v$ such that $init_v(\perp) = v$. By using the specification at lines 15-17, the COAL solver generates function $init_{\{(\texttt{first.Type},\texttt{method1})\}}$ for the statement at line 5. The function that summarizes the statement at line 8 is defined in a similar manner as $init_{\{(\texttt{second.Type},\texttt{method2})\}}$. Thus, when taking the join of $init_{\{(\texttt{first.Type},\texttt{method1})\}}(\perp)$ with $init_{\{(\texttt{second.Type},\texttt{method2})\}}(\perp)$, we obtain the value given by Equation (5.1).

```
1 Object[] objs = new Object[2];
2 objs[0] = "ISSTA";
3 objs[1] = 2016;
4 m.invoke(null, objs);
5 //m(String,int)
```

Listing 5.3: Example of Use of a Varargs Parameter.

The COAL specification in Listing 5.2 includes a `query` statement at lines 18-19. This causes the COAL solver to compute the values of objects of interest at specific program points. In our example, the `query` statement includes the signature for the `invoke` method. The `-1: type Method` statement specifies that objects on which the `invoke` method is called have type `Method`. Thus using this specification the COAL solver will compute the possible values of object `m` at line 10 of Listing 5.2.

Improvements to the COAL Solver. In the process of this work, we have contributed to several improvements of the COAL solver that now enables it to perform efficiently for resolving targets of reflective calls. At first, we extended both the COAL language and the solver to be able to query the values of objects on which instance calls are made. For example, this allowed us to query the value of object *m* in statement *m.invoke(obj, args)*. Second, we added limited support for arrays of objects such that the values of object arrays can be propagated to array elements. More specifically, if an array a is associated with values v_1, v_2, ..., v_n, for any i array element $a[i]$, we mark it as potentially containing all the values (from v_1 to v_n). While this may not be precise in theory, in the case of reflection analysis, the arrays of constructors, returned by method *getConstructors()*, that we consider typically only have a few elements. Thus, this improvement, which ensures that the propagation of constructors is done, is precise enough in practice. Finally, we performed various optimizations to improve overall performance of the COAL solver[ix].

We now detail an example of difficulty that we have encountered to retrieve the string/object values. The difficulty is due to the fact that some reflection calls such as *m.invoke(Object, Object[])* take as parameter a varargs[x]. The problem here is that the object array is not the real parameter of the method m. Indeed, the parameters are instead the elements of the array. This keeps us from extracting the appropriate method for instrumentation.

Let us consider the example code snippet in Listing 5.3. By only looking in line 4, we would infer that the parameter of the method m is $objs$. Whereas actually m has two parameters: a *String* and an *int* (as showed in line 5). To solve this problem and infer the correct list of parameters, we perform a backward analysis for each object array. For example, from *objs* in line 4, we go back to consider both line 2 and line 3, and infer that 1) the first parameter of m is a *String* whose value is $ISSTA$, 2) the second parameter is an *int* whose value is 2016.

5.3.4 BOM – Booster Module

The Booster Module considers as input an Android app represented by its Jimple instructions and the reflection analysis results yielded by the RAM module. The output of BOM is a new *reflection-aware analysis-friendly* app where instrumentation has conservatively augmented reflective calls with the appropriate standard Java calls: reflective calls will remain in the app code to conserve its initial behaviour for runtime execution, while standard calls are included in the call graph to allow only static exploration of once-hidden paths. For example, in the case of Listing 5.1, the aim is to augment "$m.invoke(o, imei)$" with "$o.setImei(imei)$" where o is a concrete instance of class *de.ecspride.ReflectiveClass* (i.e. explicitly instantiated with the *new* operator). Boosting

[ix]https://github.com/siis/coal.git
[x]http://docs.oracle.com/javase/7/docs/technotes/guides/language/varargs.html

```
 1 Class c = Class.forName("de.ecspride.ReflectiveClass");
 2 Object o = c.newInstance();
 3 + if (1 == BoM.check())
 4 +   o = new ReflectiveClass();
 5 m.invoke(o, imei);
 6 + if (1 == BoM.check())
 7 +   o.setImei(imei);
 8 String s = (String) m2.invoke(o);
 9 + if (1 == BoM.check())
10 +   s = (String) o.getImei();
```

Listing 5.4: The Boosting Results of Our Motivating Example.

approaches have been successful in the past in state-of-the-art frameworks for improving analysis of specific software by reducing the cause of analysis failures. TamiFlex [34] deals with reflection in standard Java software in this way, while IccTA [124] explicitly connects components, to improve Inter-Component Communication analysis.

Let us consider again our motivation example presented in Listing 5.1 to better illustrate the instrumentation proposed by BOM. Listing 5.4 presents the boosting results of Listing 5.1. Our instrumentation tactic is straightforward: for instance if a reflection call initializes a class, we explicitly represent the statement with the Java standard *new* operator (line 4 in Listing 5.4). If a method is reflectively invoked (lines 5 and 8), we explicitly call it as well (lines 7 and 10). This instrumentation is possible thanks to the mapping of reflective call targets yielded by the RAM module. The target resolution in RAM indeed exposes that (1) object c is actually an instance of class `ReflectiveClass`; (2) object m represents method *setImei* of class `ReflectiveClass` with a String parameter *imei*; (3) object $m2$ represents method *getImei* of class `ReflectiveClass`.

This example illustrates why reflection target resolution is not a simple string analysis problem. In this case, the support of composite object-analysis in RAM is warranted: In line 1 of Listing 5.4, c is actually an object, yet the boosting logic requires information that this represents class name "ReflectiveClass".

Observant readers may have noticed that the new injected code is always guarded by a conditional to add a path for the traditional calls. The *check()* method is declared in an interface whose implementation is not included for static analysis (otherwise a precise analyzer could have computed its constant return value). However for runtime execution, *check()* always returns `false`, preventing paths added by BOM from ever being executed. Thus, The opaque predicate keeps the new injected code from changing the app behavior, while all sound static analysis can safely assume that the path can be executed.

Additional Instrumentations. BOM performs additional instrumentations that are not directly related to the Reflection problem. Nevertheless, these instrumentations are useful to improve the completeness of other static analyses. We remind that the goal of our approach is to enable existing analyzers such as FlowDroid to perform reflection-aware static analysis in a way that improves their security results. For instance FlowDroid aims at detecting data leaks with taint-flow static analysis. In the presence of dynamic class loading, FlowDroid stops its analysis when a class has to be loaded. We already explained how DroidRA tackles this problem with the JPM module (cf. Section 5.3.2). However, not all the classes which have to be loaded are accessible. One of the reasons is that some files are encrypted, which prevents the analysis from statically accessing them. For example, app *com.ivan.oneuninstall* contains an archive file called *Grid_Red_Attract.apk*, which contains another archive file called *tu.zip* that has been encrypted. Because it is unrealistic to implement a brute-force technique to the password, we simply exclude such apps from our analysis. However, to allow tools such as FlowDroid to continue their analyses, we propose an instrumentation that conservatively solves this problem: we explicitly mock all the classes, methods and fields that are reported by the

RAM module[xi] but are not existing in the current class path (i.e. they are neither present in the initial code of the apk, nor in the code "extracted" by the JPM module).

Let us take an example to illustrate our instrumentation. Consider the instruction "$result = o.inc(a_1, a_2)$" where the method *inc* is not accessible and where a_1 is tainted. Without any modification of this code, a standard analyzer would stop its analysis. Our instrumentation consists in creating the method *inc* (and the associated class if required) in a way that the *taints* of a_1 and a_2 can be propagated. Concretely, the instrumented method *inc* will contain the following instruction: $return\ (Object)\ (a_1.toString() + a_2.toString())$, assuming that the type of *result* is *Object*.

5.4 Evaluation

Overall, our goal was to enable existing state-of-the-art Android analyzers to perform reflection-aware static analysis, thus improving the soundness and completeness of their approaches. The evaluation of DroidRA thus investigates whether this objective is fulfilled. To that end, we consider answering the following research questions:

RQ1 What is the coverage of reflection calls that DroidRA identifies and inspects?

RQ2 How does DroidRA compare with state-of-the-art approaches for resolving reflective call targets in Android?

RQ3 Does DroidRA support existing static analyzers to build sounder call graphs of Android apps?

RQ4 Does DroidRA support existing static analyzers to yield reflection-aware results?

All the experiments investigated in this section are conducted on a server with 24 processors and 100GB memory. For such tools that are running on top of Java VM, we execute them with 24GB heap size.

5.4.1 RQ1: Coverage of Reflective Calls

The goal of our reflection analysis is to provide necessary information for analysts (or other approaches) to better understand how reflections are used by Android apps. Thus, instead of considering all reflection-related methods, in this experiment, we select such methods that are most interesting for analysts. These include: 1) methods that acquire *Method*, *Constructor* and *Field* objects. Those method call sequences are falling in our common pattern (cf. Figure 5.1) and are critical as they can be used, e.g., to exchange data between normal explicit code and reflectively hidden code parts. For these calls, we perform a composite analysis and inspect the related class names and method/field names if applicable; and 2) methods that contain at least one string parameter. For these methods, we explore their string parameter's possible values.

We use the corpus of 500 apps selected in Section 5.2, to investigate the coverage of reflection calls. From each app with reflective calls we extract two information:

1. **Reached:** The total number of reflective calls that are identified by our RAM reflection analysis module.
2. **Resolved:** The number of reflective calls that are successfully resolved (i.e., the values of relevant class, method and field names can be extracted) by our reflection analysis.

[xi]This means that we only take into account reflective calls.

Our experimental results are shown in Figure 5.3, which illustrates with boxplots the performance of DroidRA in reaching reflective calls from the dummy main, and in resolving their targets. Compared to the total number of reached reflective calls, in average, DroidRA is able to correctly resolve 81.2% of the targets.

These off-targets are mainly explained by 1) the limitations of static analysis, where runtime values (e.g., user configuration or user inputs) cannot be solved statically at compile time; 2) the limitations of our COAL solver, e.g., currently it is not able to fully propagate arrays of objects, although we have provided a limited improvement on this.

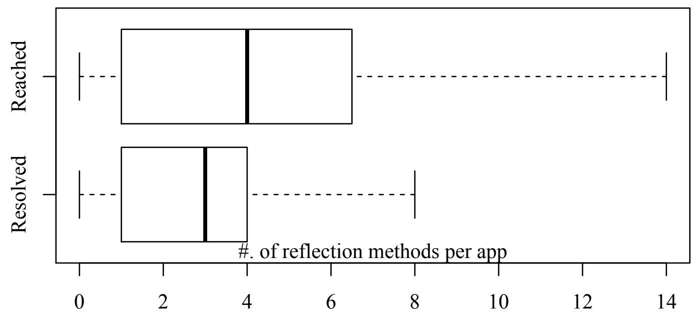


Figure 5.3: Results of the Coverage of Reflection Methods.

5.4.2 RQ2: Comparison with Checker

The approach proposed by Barros et al. [21] is the closest work to ours. Their recent publication presents an approach, hereon referred to as *Checker*, to address reflection in the Information Checker Framework (IFC) [69]. We thus compare both approaches using their evaluation dataset which consists of 10 apps from the F-Droid open-source apps repository[xii]. Table 5.3 lists the 10 apps and provides comparative results between Checker and DroidRA. Checker has been evaluated by providing statistics on methods and constructors related to reflective invocations. We thus consider the same settings for the comparison. Moreover, note that we apply DroidRA directly on the bytecode of the apps while Checker is applied on source code. Additionally, our approach does not need extra developer efforts while Checker needs manual annotations, e.g., one has to pinpoint good places to put appropriate annotations.

Overall, as shown in Table 5.3, DroidRA resolves 9 more method/constructors than Checker. Now we give more details on these results. For app *RemoteKeyboard*, DroidRA missed one method and Checker reports that it is not able to resolve it as well. Our further investigation shows that it is impossible for static approaches to resolve the reflective call in this case as the reflection target is a runtime user input (a class name for a shell implementation). Other advanced approaches like SUPOR [99] could be leveraged to supplement this shortfall. For app *VimTouch*, DroidRA refuses to report a reflective call, namely method *Service.stopForeground*, because its caller method *ServiceForegroundCompat.stopForeground* is not invoked at all by other methods, letting it becomes unreachable from our entry method.

For app *ComicsReader*, DroidRA has resolved one more reflective method than Checker. We manually verify in the source code that the additional reflective call is a True Positives of DroidRA. However, with *ComicsReader*, DroidRA missed one method[xiii], although it resolved two additional reflective calls that Checker missed. This missed method is actually located in a UI-gadget class which is not an Android component (e.g., Activity). Since our dummy main only considers Android components

[xii] `https://f-droid.org`
[xiii] *View.setSystemUiVisibility().*

Table 5.3: The Comparison Results between DroidRA and Checker, where cons Means the Number of Resolved Constructors.

App	Checker		DroidRA	
	methods	cons	methods	cons
AbstractArt	1	0	1	0
arXiv	14	0	14	0
Bluez IME	4	2	4	2
ComicsReader	6	0	7	0
MultiPicture	1	0	1	0
PrimitiveFTP	2	0	2	7
RemoteKeyboard	1^{α}	0	0	3
SuperGenPass	1	0	1	0
VimTouch	3^{β}	0	2	0
VLCRemote	1	0	1	0

$^{\alpha}$ Reached but not resolved.
$^{\beta}$ One from dead code.

of an app as potential entry-points, DroidRA further failed to reach this method from its dummy main.

Last but not the least, we have found 10 more constructors located in libraries embedded in the studied apps. Because Checker only checks the source code of apps, it could not reach and resolve them.

5.4.3 RQ3: Call graph construction

An essential step of performing precise and sound static analysis is to build at least a complete program's method call graph (CG), which will be used by static analyzers to visit all the reachable code, and thus perform a sound analysis. Indeed, methods that are not included in the CG would never be analyzed since these methods are unreachable from the analyzer's point of view. We investigate whether DroidRA is able to enrich an app's CG. To that end we build the CG of each of the apps before and after they are instrumented by BOM. Our CG construction experiments are performed with the popular Soot framework [115]: we consider the CHA [58] algorithm, which is the default algorithm for CG construction in Soot, and the more recent Spark [116] algorithm which was demonstrated to improve over CHA. Spark was demonstrated to be more precise than CHA, and thus producing fewer edges in its constructed CG.

On average, in our study dataset of 500 apps, for each app, DroidRA improves by 3.8% and 0.6% the number of edges in the CG constructed with Spark and CHA respectively. Since CHA is less precise than Spark, CHA yields far more CG edges, and thus the proportion of edges added thanks to DroidRA is smaller than for Spark.

We highlight the case of three real-world apps from our study dataset in Table 5.4. The CG edges added (i.e., Diff column) vary between apps. We have further analyzed the added edges to check whether they reach sensitive API methods[xiv] (e.g., `ActivityManager.getRunningTasks(int)`) which are protected by a system permission (e.g., `GET_TASKS`). The recorded number of such newly reachable APIs (see Perm column in Table 5.4) further demonstrates how taming reflection can allow static analysis to check the suspicious call sequences that are hidden by reflective calls. We confirmed that this app is flagged as malicious by 24 anti-virus products from VirusTotal.

Case Study: **`org.bl.cadone`.** We consider the example of app *org.bl.cadone* to further highlight the improvement in CG construction. We have computed the call graph (CG) of this app with CHA and, for the benefit of presentation clarity, we have simplified it into a class dependency graph where

[xiv]The list of sensitive API methods are collected from PScout [12].

Table 5.4: The Call Graph Results of Three Apps We Highlight for Our Evaluation. Perm Column Means the Number of Call Graph Edges that are Actually Empowered by DroidRA and are Accessing Permission Protected APIs.

Package	Algo	Original	DroidRA	Diff	Perm
com.boyaa.bildf	Spark	714	22,867	22,153	3
	CHA	172,476	190,436	17,960	51
org.bl.cadone	Spark	694	951	257	0
	CHA	172,415	187,079	14,664	16
com.audi.light	Spark	6,028	6,246	218	0
	CHA	174,007	174,060	53	0

all CG edges between methods of two classes are transformed into a single graph edge where all nodes representing methods from a single class are merged into a single node representing this class.

Figure 5.4 presents the CFG with 14,664 new edges added after applying DroidRA. Black edges represent nodes and edges that were available in the original version of the app. The new edges (and nodes) have been represented in green. Some of them however reach sensitive APIs, and are highlighted in red. We found that, among the 8 permissions that protect the 16 sensitive APIs (included in 8 classes) that are now reachable, 6 (i.e., 75%) are of the *dangerous* level[xv], which further suggests that the corresponding reflective calls were meant to hide dangerous actions.

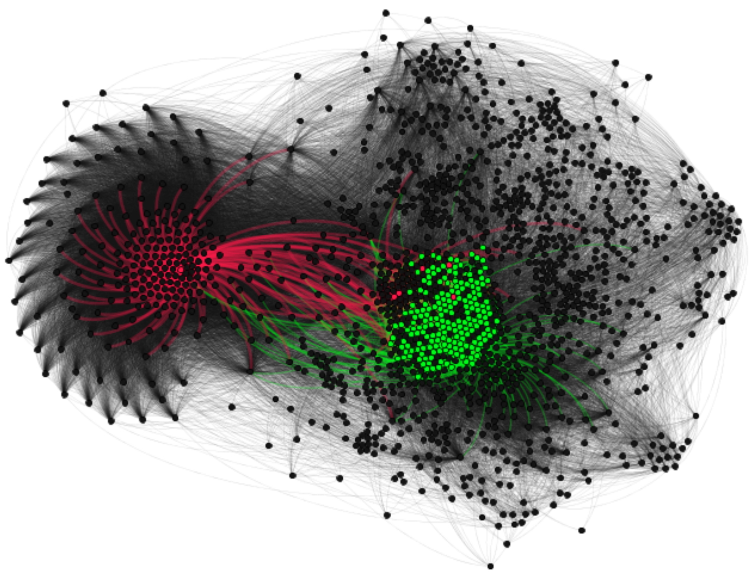


Figure 5.4: The Class Dependency Graph (Simplified Call Graph) of App *org.bl.cadone* (based on CHA Algorithm). **Black** Color Shows the Originally Edges/Nodes, Green Color Shows Edges/Nodes that are Introduced by DroidRA while Red Color Shows New Edges that are Further Protected by Permissions.

5.4.4 RQ4: Improvement of Static Analysis

We consider the state-of-the-art tool FlowDroid and its ICC-based extension called IccTA for assessing to what extent DroidRA can support static analyzers in yielding reflection-aware results. Our experiments are based on benchmark apps, for which the ground truth of reflective calls is known, and on real-world apps, for which we check whether the runtime performance of DroidRA will not prevent its use in complement to other static analyzers.

[xv]`http://developer.android.com/guide/topics/manifest/permission-element.html`

Table 5.5: The 13 Test Cases We Use in Our In-the-lab Experiments. These 13 Cases Follow the Common Pattern of Figure 5.1 and Each Case Contains Exactly One Sensitive Data Leak.

Case	Source	Reflection Usage	IccTA	DroidRA+IccTA
1	DroidBench	forName() → newInstance()	✓	✓
2	DroidBench	forName() → newInstance()	✗	✓
3	DroidBench	forName() → newInstance() → m.invoke() → m.invoke()	✗	✓
4	DroidBench	forName() → newInstance()	✗	✓
5	New	forName() → getConstructor() → newInstance()	✗	✓
6	New	forName() → getConstructors() → newInstance()	✗	✓
7	New	forName() → getConstructor() → newInstance() → m.invoke() → m.invoke()	✗	✓
8	New	loadClass() → newInstance()	✗	✓
9	New	loadClass() → newInstance() → f.set() → m.invoke()	✗	✓
10	New	forName() → getConstructor() → newInstance() → f.get()	✗	✓
11	New	startActivity() → forName() → newInstance() → m.invoke() → m.invoke()	✗	✓
12	New	forName() → getConstructor() → newInstance() → f.set() → f.get()	✗	✓
13	New	forName() → getConstructor() → newInstance() → getFields() → f.set() → f.get()	✗	✗

DroidRA on benchmark apps. We assess the efficacy of DroidRA on 13 test case apps for reflection-based sensitive data leaks. 4 of these apps are from the Droidbench benchmark where they allowed to show the limitations of FlowDroid and IccTA. We further consider 9 other test cases to include other reflective call patterns (e.g., the top used sequences , cf. Table 5.2). Since the test cases are handcrafted, the data leak (e.g., leak of *device id* via *SMS*), are known in advance. In 12 of the apps, the leak is intra-component, while in the 11^{th} it is inter-component.

Table 5.5 provides details on the reflective calls and whether the associated data-leak is identified by the static analysis of IccTA and/or DroidRA-supported IccTA. Expectedly, IccTA alone only succeeds on the first test case, $\mathit{Reflection}_1$ in DroidBench, where the reflective calls are not in the data-leak path, thus not requiring a reflective call resolution for the taint analysis to detect the leak. However, IccTA fails on all other 12 test cases. This was expected since IccTA is not a reflection-aware approach. When reflection is tamed in the test cases by DroidRA, IccTA gains the ability to detect a leak on 11 out of 12 test cases. In test case 13, the reflective call is not resolved because the reflection method *getFields()* returns an array of fields that the current implementation of constant propagation cannot manage to resolve. Indeed, we have enhanced COAL with limited support to propagate array elements, complex field arrays are not addressed. Nevertheless, constructor arrays can now be resolved, allowing DroidRA to tame reflection in test case 6.

DroidRA on real-world apps. To investigate the impact of DroidRA on the static analysis results of real-word apps, we consider a random set of 100 real-world apps that contain reflective calls and that contain at least one sensitive data leak (discovered by IccTA). Comparing to using IccTA on original apps, the instrumentation by DroidRA impacts the final results by allowing IccTA to report on median 1 more leak in a reflection-aware setting.

Runtime performance of DroidRA. We investigate the time performance of DroidRA to check whether time overhead of DroidRA app will not be an obstacle to practical usage in complement with state-of-the-art static analyzers. On the previous set of 91 apps, we measure the time performance of the three modules of DroidRA. The median value for JPM, RAM, BOM on the apps are 24 seconds, 21 seconds and 8 seconds respectively leading to a total median value of 53 seconds for DroidRA. This value is reasonable in comparison with the execution of tools such as IccTA or FlowDroid which can run for several minutes and even hours on a given app.

5.5 Related Work

Research on static analysis of Android apps presents strong limitations related to reflection handling [28, 79, 125, 238]. Authors of recent approaches explicitly acknowledge such limitations, indicating that they ignore reflection in their approaches [124, 180, 235] or failing to state whether reflective calls are handled [233] in their approach.

The closest work to ours was concurrently proposed by Barros et al. [21] within their Checker framework. Their work differs from ours in several ways: first, the design of their approach focus on helping developers checking the information-flow in their own apps, using annotations in the source code; this limits the potential use of their approach by security analysts in large markets of Android apps such as GooglePlay or AppChina. Second, they build on intra-procedural type inference system to resolve reflective calls, while we build on an inter-procedural precise and context-sensitive analysis. Third, our approach is non-invasive for existing analysis tools who can now be boosted when presented with apps where reflection is tamed.

Reflection, by itself, has been investigated in several works for Java applications. Most notably, Bodden et al. [34] have presented TamiFlex for aiding static analysis in the presence of reflections in Java programs. Similar to our approach, TamiFlex is implemented on top of Soot and includes a Booster module, which enriches Java programs by "materializing" reflection methods into traditional Java calls. However, DroidRA manipulates Jimple code directly while TamiFlex works on Java bytecode. Furthermore, DroidRA is a pure static approach while TamiFlex needs to execute programs after creating logging points for reflection methods to extract reflection values. Finally, although Android apps are written in Java, TamiFlex cannot even be applied to Android apps as it uses a special Java API that is not available in Android [251]. Another work that tackles reflection for Java has been done by Livshits et al. [153], in which points-to analysis is leveraged to approximate the targets of reflection calls. Unlike our approach, their approach needs users to provide an per-app specification in order to resolve reflections, which is difficult to apply for a large scale analysis. Similarly, Braux et al. [36] propose a static approach to optimize reflection calls at compile time, for the purpose of increasing time performance.

Regarding Dynamic Code Loading in Android, Poeplau et al. [185] have proposed a systematic review on how and why Android apps load additional code dynamically. In their work, they adopt an approach that attempts to build a super CFG by replacing any *invoke()* call with the target method's entry point. This approach however fails to take into account the *newInstance()* reflective method call, which initializes objects, resulting in a context-insensitive approach, potentially leading to more false positives. StaDynA [251] was proposed to address the problem of dynamic code loading in Android apps at runtime. This approach requires a modified version of the Android framework to log all triggering actions of reflective calls. StaDynA is thus not market-scalable, and present a coverage issue in dynamic execution. Our approach, DroidRA, provides a better solution for reflective method calls, can be leveraged to compliment these approaches, so as to enhance them to conduct better analysis.

Instrumenting Android apps to strengthen static analysis is not new [9]. For example, as we detailed in Chapter 4, IccTA [124] is proposed to instrument Android apps to bridge ICC gaps and eventually enable inter-component static analysis. AppSealer [245] instruments Android apps for generating vulnerability-specific patches, which prevent component hijacking attacks at runtime. Other approaches [202, 246] apply the same idea, which injects shadow code into Android apps, to perform privacy leaks prevention.

5.6 Conclusion

This chapter addresses a long time challenge that is to perform reflection-aware static analysis on Android apps. We have presented DroidRA, an open source tool, to perform reflection analysis, which models the identification of reflective calls as a composite constant propagation problem through the COAL declarative language, and leverages the COAL solver to automatically infer reflection-based values. We remind the reader that these reflective-based values can be directly used as basis for many whole-program analyses of Android apps. We further illustrate this point by providing a booster module, which is based on the previously inferred results to augment apps with traditional Java calls, leading to a non-invasive way of supporting existing static analyzers in performing reflection-aware analysis, without any modification or configuration. Through various evaluations we have demonstrated the benefits and performance of DroidRA.

6 Detaching Common Library

The packaging model of Android apps requires the entire code necessary for the execution of an app to be shipped into one single apk file. Thus, an analysis of Android apps often visits code which is not part of the functionality delivered by the app. Such code is often contributed by the common libraries which are used pervasively by all apps. Unfortunately, Android analyses, e.g., piggybacked app detection or malware detection, can produce inaccurate results if they do not take into account the impact of common libraries, which constitute noise in app features. Thus, there is a need to detach common library (through code instrumentation) for other analyses.

Despite some efforts on investigating Android libraries, the momentum of Android research has not yet produced a complete set of common libraries to further support in-depth analysis of Android apps. In this paper, we leverage a dataset of about 1.5 million apps from Google Play to harvest potential common libraries, including advertisement libraries. With several steps of refinements, we finally collect by far the largest set of 1,113 libraries supporting common functionality and 240 libraries for advertisement. We use the dataset to investigates several aspects of Android libraries, including their popularity and their proportion in Android app code. Based on these datasets, we have further performed several empirical investigations to confirm the motivations behind our work.

This chapter is based on the work published in the following paper:

- Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016

Contents

6.1 Introduction

The research community has produced a large body of work for mitigating the emerged threats in the Android ecosystem, essentially to guard the security and privacy of users. For scalability and practicability reasons, a substantial number of the proposed approaches [67, 124, 226] rely on static analysis to parse the entire code shipped in the app package to find security problems in code instructions, to extract features for further processing or simply to compare apps in large repositories. Unfortunately, because Android development paradigm allows to easily include third-party code, in the form of libraries, a significant portion of an app is eventually irrelevant to certain analyses (e.g., piggybacking analysis). Common libraries embedded in app code thus constitute a significant barrier for the static exploration of applications code. For example, as we will detail later, common library would introduce both false positives and false negatives for piggybacked detection approaches. Besides, static code analysis is also often challenged by computing power and memory requirements. In the case of FlowDroid [10], the state-of-the-art static taint analysis tool for Android apps, it was reported that the analysis time can be too high [13]. As demonstrated in Wang et al.'s findings [224], where 60% of app's code are contributed by common libraries, which would thus indicate roughly that over half of the CPU and memory consumption could be wasted on irrelevant library code, threatening the performance of the analyzer. Therefore, there is a need to detach common library from Android apps, in attempt to increase the precision of results and improve the performance of analysis process. We thus leverage code instrumentation[i] in this work to detach common library from Android apps.

The aforementioned case constitutes strong motivations for automatically identifying once a large set of **common libraries** from market-scale apps, which could then be used by other approaches to immediately take such libraries into account. A straightforward solution for achieving such a task is to build a comprehensive *whitelist* of common libraries. Wang et al. [224] claim to have collected more than 600 different common libraries to improve their repackaged app detection process. However, this collection is not available to the community, and may not be representative in other datasets. Other approaches [35, 80, 84, 184] build on top of limited whitelists collected using simplistic heuristics and containing between only 9 (AdDroid [184]) and 103 (Book et al. [35]) libraries.

In this work we investigate the use of common libraries in Android based on a dataset of around 1.5 million apps collected from the official Google Play market. In particular, we build and maintain a comprehensive whitelist of 1,113 Android common libraries that we share with the communities. Our approach identifies common libraries based on the assumption that they are used by many apps as such, i.e., without developer modification. We further label those libraries to distinguish between **advertisement libraries** (or ad libraries, a specific type of common libraries) and others, using heuristics defined from our manual investigations.

Overall, in this chapter, we make the following contributions:

- An approach to automatically harvest common libraries from market-scale Android apps. In this work, we collect 1,113 common libraries from a dataset of around 1.5 million Android apps.
- A discriminative study of advertisement libraries, for which 240 common libraries are recognized as ad libraries.
- An empirical investigation and evaluation of the use of common libraries in Android apps. We show that there are indeed significant differences in the use of common libraries between benign and malicious apps. Besides, we also show that our harvested common libraries are indeed useful for other approaches, e.g., to reduce both false positive and false negative rates for piggybacked apps detection.

[i]Since we have already demonstrated in the previous two chapters that code instrumentation is an effective approach for boosting static security analysis, and it is actually quite simple to perform code instrumentation for detaching common library, we will no longer detail the working process of code instrumentation in this chapter.

- Two comprehensive *whitelist*s of Android libraries (one for common libraries and the other for ad libraries), that we make available online to the Android research community at: `https://github.com/serval-snt-uni-lu/CommonLibraries.git`.

The rest of this chapter is organized as follows: We first introduce two concrete examples to better explain the problems we attempt to address in Section 6.2. Then, we present our approach and its implementation details in Section 6.3. In Section 6.4, we present our investigated data set and the overall results. We then empirically evaluate our findings in Section 6.5, followed by related works in Section 6.6. Finally, in Section 6.7, we conclude this chapter.

6.2 Motivating

We now motivate our work by discussing the impact of filtering out libraries from apps when performing piggybacking detection. Piggybacking is an operation that consists in taking an existing app, unpacking it, then modifying it by adding a (generally malicious) new payload and re-signing it, before distributing it as a new app. Like repackaged apps (where a payload is not necessarily added), piggybacked apps are now pervasive in the Android ecosystem where they further constitute an easy way to build and distribute malware [255, 256]. A typical approach for detecting piggybacked apps consists in performing pairwise comparisons to identify the original app that was actually piggybacked. In the process of computing similarity however, libraries, which may account for a large portion of apps, can influence towards inaccurate results. We present two real-world examples of pairs of apps where the presence of libraries can lead to a mislabelling of a legitimate app as piggybacked or a failure to flag a piggybacked app as such.

6.2.1 Mislabeling Legitimate apps as Piggybacked

We consider in Fig. 6.1a the case of two apps (*air.starq.game.ZRcards9ers* and *com.taiweishiye.tom.pkjjtss*) collected from an Android market. The packages in their code structure are very similar when considering the common libraries that they integrate: one app has 86% of its code[ii] that is also contained in the other app. However, considering the results of a prior investigation of a set of 1,169 known legitimate/piggybacked app pairs where we found that most of the similarity degree ranges between 81% and 100%, we could set a threshold of 80% for

[ii]The percentage is computed based on method level, where more details will be given in Section 6.3.2.

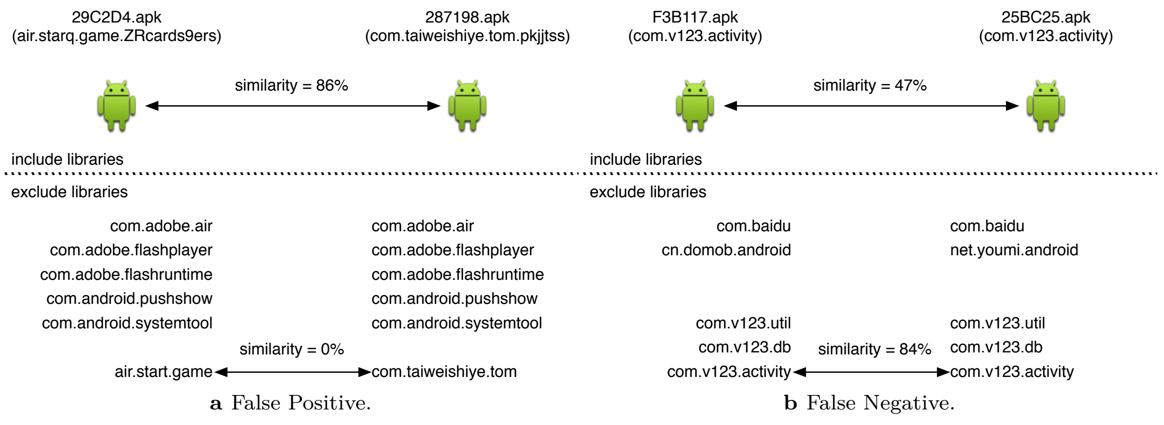


Figure 6.1: Two Motivating Examples Show the Importance of Excluding Common Libraries (through Instrumentation) in Order to Perform Precise Piggybacked Apps Detection. Note that *F3B117.apk* and *25BC25.apk* are Actually Signed by Different Certificates although They Share a Same Package Name.

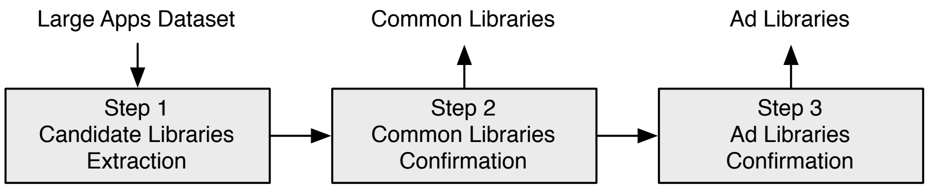


Figure 6.2: Overview of Our Detaching Common Library Approach.

identifying piggybacking cases. This unfortunately would lead to a mislabeling in the above case. Indeed, a detailed analysis of both apps shows that they are actually using several common libraries (e.g., `com.android` and `com.adobe`). Excluding such libraries from the similarity computation, the similarity degree falls down to 0%, leaving no room for a false positive prediction.

6.2.2 Missing True Piggybacked Apps

We now consider in Fig. 6.1b two apps which are known to be a legitimate/piggybacked app pair. These apps share the main package called *com.v123.activity*. However, library *cn.domob.android* was replaced in the piggybacked app with library *net.youmi.android* to redirect the revenues of the legitimate app to another developer. Nevertheless, although these two apps are piggybacked from one to another, their similarity degree is only at 47%, which would constitute a false negative in our detection scheme with a threshold at 80%. However, if the detection system identified first the common libraries and dismissed them during pairwise comparison, the similarity degree would reach 84%, leading to a successful prediction.

Overall, the validity of pairwise comparison for piggybacking detection could be threatened when substantial parts of app code are common library code. Thus, to limit both false positives and false negatives, detaching common library is now more and more considered in state-of-the-art repackaging and piggybacking detection approaches [45, 89, 224]. However, the whitelists that they leveraged is built based on manual investigations or automatically with limited datasets. Furthermore, these whitelists are seldom available to other researchers in the community.

One objective *of our work is to provide to the community a comprehensive list of common libraries, which can be used as a whitelist for detaching and thus supporting static code-based analyses of Android apps.*

6.3 Identification of Common Libraries

In this section we provide details on the approach that we have devised to collect common libraries for the study.

Process Overview: Fig. 6.2 illustrates the general process of our approach, which is dedicated to harvest common libraries in Android apps and identifying advertisement libraries among them.

First, for our approach to make sense, we need a large and representative dataset of Android apps. Then, as a first step, we visit all the apps in the dataset and rank all packages in terms of the frequency of their appearance in apps. For the sake of simplicity, we assume that a package with the same name in several apps is a candidate library. Thus, *Step 1* outputs a ranked list of candidate libraries, where the highest ranked candidate library has the most recurring package name in the dataset. In the second step we perform a more fine-grained pairwise comparison of candidate library code within apps. The objective of *Step 2* is to confirm as common library packages those recurring packages that have the same name and are very similar in their code. Finally, in *Step 3*, we further investigate the harvested libraries to label those that are advertisement libraries and thus may be

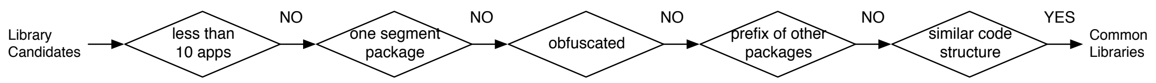


Figure 6.3: Refinement Process for Common Libraries Identification.

treated differently in some Android analysis approaches. We now provide details on how each step works in the following three subsections.

6.3.1 Step 1: Candidate Libraries Extraction

We assume that common libraries are such software packages that are:

- used in a large number of apps – recurring packages have very high probability of being common libraries.
- used by developers without modifications – their code must be similar across apps. Hu et al. [96] have found that over 80% of libraries are indeed used without modification in their dataset of 100,000 Google Play apps.

Building on those assumptions, and leveraging a large dataset, we extract all package names from Android apps and cluster them based on their frequency of occurrence in the dataset. Theoretically, packages that appear in at least two apps could be taken as candidate libraries[iii]. To reduce the number of distinct packages considered as candidate libraries, and which must be further processed we consider two constraints:

- We only consider the first three segments[iv] of package name or the entire name if there are less than 3 segments. With this constraint we manage to limit the number of redundant subpackages while still guaranteeing a large diversity in package names.
- We also exclude packages with names starting with *android.support*. Indeed, there are many sub-packages within this package and they are used pervasively in Android apps. Furthermore, since these are part of the Android framework, we do not consider them in our study.

6.3.2 Step 2: Common Libraries Confirmation

Because package naming is done in Java programming with limited constraints, any two packages may share the same name while being completely different in terms of code functionality. Also, the frequency of a package name may actually be contributed by piggybacking operations, obfuscation activities (e.g., *a.a.a* or *com.a* are recurrent in many obfuscated apps) or simplistic naming (e.g., *debug* or *mobile* package names). Thus, we must refine the list collected in the previous step with code similarity measurements to find actual code packages used as common libraries.

Beforehand, given the expensive property of pairwise comparison, we use heuristics to exclude from the candidate libraries outputted by *Step 1*, those packages which would be irrelevant. Our refinement process is shown in Fig. 6.3.

1) At first, we focus on those packages whose names appear in more than 10 apps to reduce the number of candidate libraries to the most relevant ones.

2) Then, we remove such packages whose names contain only one segment. Although such short names are indeed likely to be redundant in several apps, they are not likely to be those of packages that will

[iii]Actually this may not be true if the apps are from a same developer. However, since we are performing experiments on a large set of apps, this small deviation will not impact our final results.

[iv]In this work, we use the term segment to describe each domain of different levels, e.g., for package *org.example*, we say it contains two segments, which are *org* and *example*.

be distributed as common libraries. Indeed, to prevent package name collisions, one convention in Java package naming[v] recommends organizations/development teams to use their reversed Internet domain names (e.g., com.facebook) to begin their package names, which justifies our assumption that common libraties, intended for wide distribution, have package names with several segments.

3) Next, we undertake to exclude packages with obfuscated code. However, because there is currently no advanced approach for checking whether a package is obfuscated or not, we build on a naive approach based on observations that we inspect from several obfuscated apps: every package that contains a single letter segment (e.g., *d* of *com.idreamsky.d*) is considered as obfuscated.

4) To further reduce the number of candidates, we exclude such packages that are prefixes of other packages (e.g., we remove package *com.sansec* if package *com.sansec.AESlib* exists). The idea behind this decision is that on the one hand long packages would indicate more fine-grained examination while, on the other hand, short packages would increase the chance of being duplicated (by accident).

5) Finally in the last step, we perform package similarity analysis to discriminate common library packages from normal app code package. Given p, a package name, and A, a set of apps which include a package named p, our similarity analysis works in three steps:

5.1) Pairwise combinations of apps. We consider all the pairwise combinations of apps with package name p. Recall that every considered package name p was selected as candidate because it appears in at least 10 apps. Thus, for any given p, there are at least $\binom{10}{2} = 45$ pairs to compare. Google's ad package *com.google.ads* is the one for which A is the largest (247,394 apps), leading to a over 30 billions pairs that require comparisons. For scalability reasons, we randomly select for each case of a package name p, 10 pairs of apps, allowing us to assess whether this package name indeed represents a common package code across the apps.

5.2) Method Comparisons. Analysis of a pair of apps is performed by computing the similarity between their methods. This similarity takes into account not only the signatures of apps but also their respective contents. Two methods, from two different apps, with the same signature are said to be *identical* only when their contents are the same. Otherwise, they are simply said to be *similar*. Such methods may exist between two packages of the same library in several cases: a method in one library package may been modified to insert malicious payload during piggybacking operations; different obfuscation algorithms applied on different apps that include the same library may produce methods with the same signature but different contents. To limit the impact of obfuscation, we proceed to abstract the contents of methods by comparing the types of statements (e.g., "invoke") in the Jimple code, leaving out all names of variables/fields/methods. However, since obfuscation is not expected to modify SDK API methods, we also take into account the names of such methods. Eventually, the similarity of methods is computed as a simple text differencing.

5.3) Similarity Analysis. In the last step, we finally perform pairwise similarity analysis for packages with the same name p. There are two thresholds, namely t_p and t_a, which are involved in the similarity analysis. First, we consider that two packages p_1 and p_2 correspond to the same common library p if p_1 and p_2 are identical or are at least similar up to a threshold t_p. Second, because of the known common phenomenon of repackaging/piggybacking in Android, which may nullify the package similarity (because they are probably from the same original app), we must dismiss cases where the similarity score of the pair of apps (app_1 and app_2) is higher than a threshold t_a. Note that the similarity between apps is computed at the method level (i.e., what percentage of methods are identical or similar between the apps?).

To summarize, for similarity analysis, given a pair of apps (app_1, app_2), we compute four metrics: *identical* (i.e., the number of methods that are exactly the same, both in terms of signatures and implementation), *similar* (i.e., the number of methods having the same signature but with different contents), *deleted* (i.e., the number of methods that exist in app_1 but not in app_2), and *new* (i.e., the number of methods existing only in app_2). These metrics are indeed good indicators for comparison

[v] `https://newcircle.com/bookshelf/java_fundamentals_tutorial/packaging`

and have been leveraged in state-of-the-art Android similarity tools, such as Androguard [62]. Given these metrics, we can compute the similarity between the pair (app_1, app_2) using Formula 6.1.

$$similarity = max\{\frac{identical}{total - new}, \frac{identical}{total - deleted}\} \tag{6.1}$$

where

$$total = identical + similar + deleted + new \tag{6.2}$$

Note that we use the same formula to perform the similarity analysis of a given pair of packages ($p_1, p2$), except that the metrics are computed by counting methods in packages rather than in apps (e.g. *identical* is the number of methods that are exactly the same in p_1 and $p2$, *deleted* is the number of methods that exist in p_1 but not in p_2.)

6.3.3 Step 3: Identification of Ad Libraries

A specific example of type of widespread common libraries in Android is advertisement libraries. Such libraries are indeed used pervasively as they constitute one of the main ways for app developers to be rewarded for their development effort. Ad libraries are also often inserted during piggybacking to redirect revenues. Their presence in an app often lead antivirus products to flag them as adware. Recent approaches for Android security analysis are now processing ad library code in a specific way to reduce false positives. For example, MUDFLOW [13] simply does not report any potential sensitive data leaks through ad libraries, as they might be legitimate. To that end, they have leveraged a limited whitelist of 12 libraries. In this context, we propose to further mine our collected set of common libraries to identify a large set of ad libraries which could be leveraged to improve the results of Android analyses. To that end, we consider a basic method of detection based on the library name and a more semantic approach based on the characteristics of ad libraries.

6.3.3.1 Keywords matching

We note that ad library package names generally contain keywords that include the term *"ad"*. Widespread examples of such packages are *com.google.ads* and *com.adsdk.sdk*. Unfortunately, simply matching "ad" in the package name would lead to substantial portion of false positives as several library package names have "ad" in their segments which are common words (e.g., shadow, gadget, load, adapter, adobe). Thus, to work around this limitation, we collect all English words containing "ad" from SCOWL[vi] (accounting for a total of 13,385 words), and dismiss packages containing such words as potential ad libraries.

6.3.3.2 Ad features investigations

We consider samples from a list of ad packages summarized by Grace et al. [84] and manually investigate how ad libraries differentiate from other common libraries, and infer a set of features whose presence in a package would justify the tag of ad library.

[vi] Spell Checker Oriented Word Lists: http://wordlist.aspell.net

Internet usage All investigated libraries unsurprisingly require access to Internet to remotely upload to a server some viewing statistics and update ad contents. Thus, apps integrating add libraries also require the `android.permission.INTERNET` permission. Given this fact, we can already exclude a number of common libraries, which appear in apps without Internet access, as ad libraries. However, given that an app may requests the *INTERNET* permission for its own needs, we cannot immediately state that a common library in such an app is an add library. Instead we must investigate whether the code of such an app indeed declares uses Internet-related APIs. To that end, we leveraged the whitelist of such APIs, originally shared by PSCout [12], to produce candidate ad libraries among the common libraries.

Components declaration Our manual investigations have also revealed that ad libraries often contain components, mainly Activities, for facilitating users' ad-related interactions (e.g., switching to a new full-screen ad page when users click on an advertisement banner). As a concrete example, MoPub[vii] is an advertisement library targeting both Android and iOS. To integrate this library in their apps, developers must declare four components in their apps' *manifest* file. One component in particular, *MraidVideoPlayerActivity* is necessary for video ads to work properly. Thus, when a library package is associated to a declared component, we flag it as a potential ad library.

Views declaration In Android, advertisements are generally set to be visualized, which in Android programming imply the use of view gadgets (i.e., classes extended from `android.view.View`). Thus, we check whether there are *View*-based classes under a common library to flag it as candidate ad library.

6.3.4 Implementation Details

We implement our approach through several languages such as Java and shell/Python scripts. In *step 1*, we leverage *Apktool*[viii] to disassemble Android apps. Given an Android app, we extract the prefixes of paths of *smali* files (a format used by *Apktool* to represent Android apps' code) to represent its packages. Then, we cluster all the packages of investigated apps together and rank them through their repeated times. The packages whose size are greater than a given threshold are selected as library candidates.

The code similarity analysis in *step 2* and the ad library conformation in *step 3* are implemented in Java. More specifically, both of them leverage Soot [115] to achieve their functionality and work in the *Jimple* code level, where Soot is a framework for analyzing and transforming Java/Android apps while Jimple is an intermediate representation of Soot. The transformation from Android Dalvik bytecode into Jimple code is powered by Dexpler [24], which currently is available as a plugin in Soot.

6.4 Dataset and Results

In this section, we first disclose our evaluated data set in Section 6.4.1 and then we present our overall findings including both common libraries and also ad libraries in Section 6.4.2. Finally, we present further statistics on the libraries in Section 6.4.3 and Section 6.4.4.

[vii] https://github.com/mopub/mopub-android-sdk
[viii] https://ibotpeaches.github.io/Apktool/

6.4.1 Dataset

Our data set is made up of 1,455,516 (around 1.5 million) apps that are collected from the official Google market (*Google Play*) over several months. This data set has already been applied for large-scale experiments on Android researches such as malware detection [121] and piggybacked apps detection [139]. We have sent all the apps into VirusTotal to check whether they are malicious or not. Among the 1,455,516 apps, 311,490 (nearly 21%) of them are flagged by at least one anti-virus product hosted on VirusTotal while 65,079 (nearly 4%) apps are flagged by at least five anti-virus products.

6.4.2 Overall Results

Table 6.1: Summary of Our Investigation Results.

Type	Number
#. of packages (total)	7,710,505
#. of packages (distinct)	676,674
#. of packages ($N_{shared\_apps} > 10$)	19,725
#. of packages (one segment)	613
#. of packages (obfuscated)	1,461
#. of packages (prefix of others)	919
Size of final set of candidate common libraries	**16,732**

Table 6.1 illustrates the overall results of our investigation on a data set of around 1.5 million apps. In total, we collect 676,674 distinct package names, where we filter out 656,949 package names that are used by at most 10 apps, leading to a set of 19,715 package names. We further dismiss 2,993 from consideration thanks to our library refinement process. Those 2,993 package names are composed of 613 one segment packages, 1,461 obfuscated packages and 919 packages that are prefix of other packages. Finally, we perform pairwise similarity analysis for 16,732 packages. For each package, we randomly select 10 pairs of apps to do the comparison. As long as there are positive results, we consider it as a common library, and verse visa.

6.4.2.1 Results of Common Libraries

Table 6.2: Results of Common Libraries with Different Thresholds: t_p for Package-level and t_a for App-level. Common Libraries are Select if and only if Their Package-level Similarities are Bigger than t_p while Their App-level Similarities are Smaller than t_a.

$t_p \backslash t_a$	0.1	0.2	0.3	0.4
0.9	1,113	2,148	3,173	4,072
0.8	1,363	2,564	3,715	4,685
0.7	1,573	2,898	4,117	5,144
0.6	1,735	3,179	4,452	5,509

Our common libraries selection is actually depending on the two thresholds introduced in Section 6.3: t_a for app-level similarity and t_p for package-level similarity. The precision of our results is positively correlated to t_p while negatively correlated to t_a. Indeed, the bigger t_p is, the higher the probability that a given candidate library is an actual common library, giving the assumption that libraries are not modified when they are used among apps. On the other hand, the smaller t_a is, the lower the probability that the compared two apps are repackaged/piggybacked from one to another. Recall that if two apps are repackaged/piggybacked from one to another, the similarity of packages would become meaningless, as in this case, most packages would be the same, without being necessarily common libraries.

Table 6.2 illustrates the results of common libraries with different thresholds. The final number of common libraries range from 1,113 to 5,509. To better refer to our results in the remainder of the paper, we name $CL_{p,a}$ the set of Common Libraries that are selected with the thresholds t_p and t_a. For example, $CL_{9,1}$ stands for the precise set of 1,113 common libraries we harvest with $t_p = 0.9$ and $t_a = 0.1$, while $CL_{6,4}$ stands for the more "loose" set of common libraries, which although is the biggest set, contains potential more false positives (less precise than $CL_{9,1}$).

6.4.2.2 Results of Ad Libraries

Table 6.3: Results of Ad Libraries.

Description	#. of Libraries
Ad-related keyword matching	275
Ad characteristic-based investigating	822
Merge (conservative ad libraries)	1050
Manual confirmation (keyword matching)	222
Manual confirmation (characteristic investigating)	137
Merge (precise ad libraries)	240

We then distill ad libraries from the previously harvested common libraries. We start from the $CL_{6,4}$ library set and performs two types of refinement: 1) ad-related keywords matching and 2) ad characteristic-based investigation. The refinement results are presented in Table 6.3.

Ad-related keywords matching. By following the process described in Section 6.3.3, we were able to automatically harvest 275 ad libraries.

Ad characteristic-based investigating. We have observed three characteristics that ad libraries may have in Section 6.3.3. Fig. 6.4 shows the results of our investigation. Among the 5,509 libraries in $CL_{6,4}$, 1,248 of them request the *INTERNET* permission, 1,560 have declared *View* gadgets and 1,388 have declared components. The intersection results are also illustrated in Fig. 6.4. In this work, we take the intersection of all the three characteristics as potential ad libraries, leading to a set of 822 ad libraries.

In the next step, we merge the aforementioned two ad libraries sets, leading to a set of 1,050 ad libraries. In the remainder of the paper, we name this set AD_{1050}.

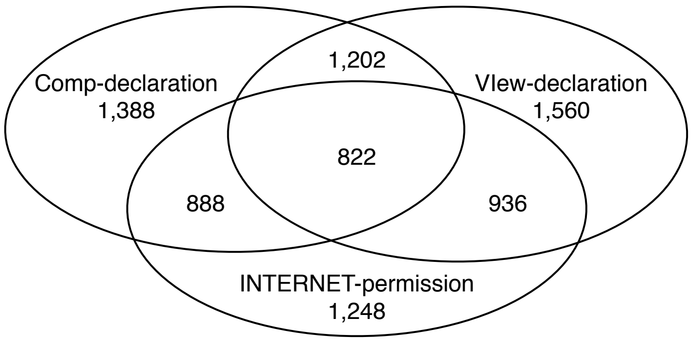


Figure 6.4: Investigation Results of Different Characteristics for Ad Libraries.

Manual confirmation. As far as we know, AD_{1050} is currently the largest set of ad libraries existing in the community. However, because we start from $CL_{6,4}$, mainly to start with the biggest set (minimizing the miss of libraries), AD_{1050} may contain false positives. To this end, we perform a fast but aggressive manual refinement, where only clear ad libraries[ix] are taken into account. As a

[ix]Their corresponding web pages have explicitly claimed that they function advertisements.

result, 240 libraries are confirmed as ad libraries[x], hereinafter we refer to this set as AD_{240}. This 240 ad libraries are highly precise. We argue that a highly precise ad library set is important, which plays as a basement that makes it possible for other approaches to also yield precise results.

6.4.3 Popularity of Common Libraries

Fig. 6.5 lists the top 20 common libraries and indicates, for each, the number of apps in which they are used. The top used library is *com.google.ads*, which is used by 247,394 apps (nearly 17%) of our data set. Moreover, the results suggest that developers often use libraries which are proposed by popular (well-known) companies such as Google or Facebook.

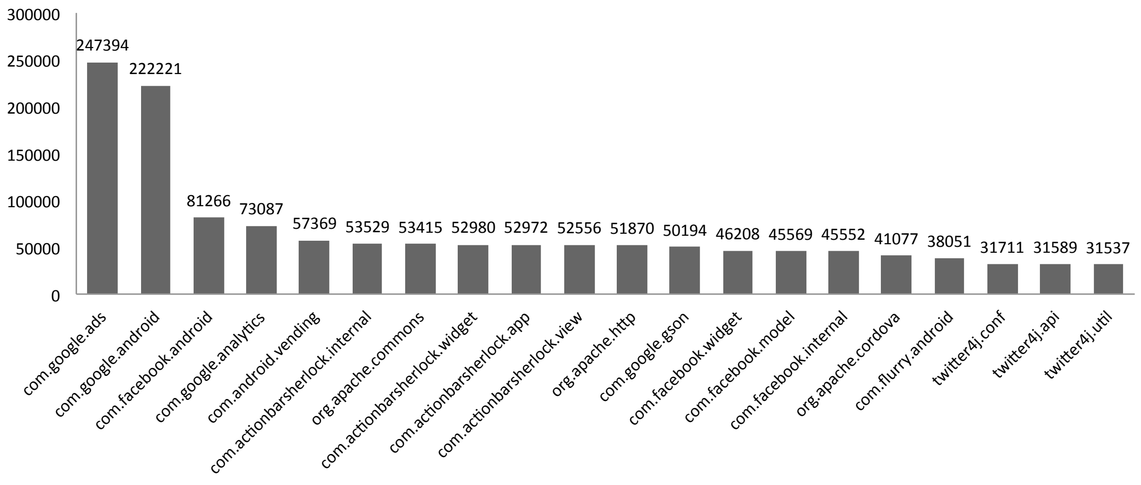


Figure 6.5: Popularity of the Top 20 Common Libraries in Our Investigation, and the Number of Apps in which They are Used.

6.4.4 Proportion of Library code in App code

We then look into the percentage of Android apps code which come from libraries. To this end, we consider the libraries present in $CL_{9,1}$ and a set of 10,000 apps randomly selected from our initial set of apps. For each app, we compute the size of the $CL_{9,1}$ libraries ($size_{lib}$, in bytes) presented in the app and the size of the whole app ($size_{app}$). We finally compute the portion p of the use of common libraries through $p = size_{lib}/size_{app}$. The experimental results vary from 0 to 0.99, giving a median value 0.41. Among the 10,000 apps, 4,293 (42.9%) of them have used more code in libraries than in their real logic ($p >= 0.5$). This results show that **Android apps are indeed using common libraries pervasively, i.e., 41% of an Android app code on average is contributed by common libraries**.

6.5 Evaluation

Our investigation into libraries have also revealed interesting findings on the use of libraries: 1) Well-used libraries, such as *unit3d*, are often used as the compromising point for malicious apps. 2) Malware writers often name their malicious components after famous and pervasively used libraries from reputed firms: e.g., the *DroidKungFu* malware family spreads malicious payload within a package called *com.google.update*. Our similarity analysis allowed to detect such fraud by further investigating outliers.

[x]This does not mean the remaining 810 libraries are not ad libraries.

Furthermore, we have also found that, except for improving the time performance of static analysis, our collected common libraries can be also leveraged to enhance other analyses. In this chapter, we discuss two cases where our harvested common libraries show significant improvements to the performance of Android analysis approaches.

6.5.1 Machine Learning for Malware Detection

We investigate the case of machine-learning based approaches for Android, and study the impact of ignoring or taking into account common libraries on the accuracy of prediction. We consider a case study based on MUDFLOW [13] and its dataset. This dataset contains sensitive data leaks information for 15,096 malicious apps and 2,800 benign apps. MUDFLOW is a relevant example as the authors have originally foreseen the problem with libraries and thus attempted to exclude a small set of ad libraries in their experiments. With our large harvested dataset of common libraries, we investigate the performance gap that can be achieved by excluding more known libraries.

MUDFLOW performs machine learning to mine apps that use sensitive data abnormally. More specifically, MUDFLOW takes each distinct type of sensitive data leak (from pre-defined *source* to *sink*) and performs an one-class classification to detect abnormal apps. One-class classification is realistic in their experimental settings with their imbalanced data set (they have much more malicious apps than benign apps).

Since our goal is not to replicate MUDFLOW (along with its sophisticated library-unrelated parameters), but to evaluate the impact of excluding common libraries for machine learning, we propose to implement a slightly simplified approach for our experiments. Unlike MUDFLOW, which constructs a training set based on benign apps and then applies it to predict unknown apps, we simply perform 10-fold cross validation in our evaluation. As we are working on the same imbalanced data, we also choose one-class classification.

We have performed four types of experiments, which are detailed below:

- **E5:** We evaluate on all the 15,096 malicious apps. The feature set is made up of distinct sensitive data leaks. Instead of taking into account *source* and *sink* methods, each data leak is represented through the *source* and *sink* categories (e.g., methods like *Log.i()*, *Log.e()* are represented as category *LOG*).
- **E6:** This experiment has similar settings as in **E5**, except that such sensitive data leaks that are contributed by the 12 ad libraries considered by MUDFLOW are excluded.
- **E7:** This experiment has similar settings as in **E6**. In this case however, the excluded set of libraries is the most constrained set of 1,113 libraries harvested in our work.
- **E8:** This experiment has similar settings as in **E7**. In this case however, the excluded set of libraries is constituted by libraries selected based on a more loose definition of libraries. The excluding set contains 5,509 libraries, which may include a number of false positives.

The results of these four experiments are shown in Table 6.4. Comparing **E7** to **E5**, the accuracy is indeed increased, which suggests that the presence of common libraries code could confuse machine learning-based classifier. However, the accuracy remained the same between **E6** and **E5**, suggesting that the MUDFLOW *whitelist*, which contains 12 libraries, is too small to impact the final results. Interestingly, with our largest set of libraries, the accuracy of **E8** decreases slightly comparing to that of **E7**. This suggests that the precision in common library identification is important: excluding non-library code will eventually decrease the overall performance for machine learning.

Table 6.4: Results of Our Machine Learning based Experiments Performed on the Data Set Provided by MUDFLOW [13].

Seq.	#. of Features	Excluding Libs	Accuracy
1	109	0	81.85%
2	109	12 (MUDFLOW [13])	81.85%
3	109	1,113 ($t_p = 0.9$,$t_a = 0.1$)	83.10%
4	108	5,509 ($t_p = 0.6$,$t_a = 0.4$)	83.01%

6.5.2 Piggybacking Detection

Recall that in Section 6.2, we have shown that piggybacking detection approaches are likely to yield false positives and false negatives if code contributed by common libraries are not taken into account. We provide more empirical evidence of such threats.

For our experiments we rely on a set of pairs of apps that we have collected in the similarity analysis of Step 2 and regrouped into two categories: the first category, *FNData*, contains 761 pairs of apps where the similarity score for each pair is below 50% while the second category, *FPData*, includes 1,100 pairs of apps where the similarity score of each pair of apps is over 80%. Given the previously justified threshold of 80% for deciding on piggybacking (cf. Section 6.2), we assume that all pairs in *FPData* are piggybacked pairs while those in *FNData* are not. We now explore again the similarity scores of the pairs when excluding from each app the common libraries (in CL_{91}) they may include.

False positives elimination. Among the 1,100 pairs of apps in *FPData*, 1,029 (93.5%) remained similar above the 80% threshold. 71 (6.5%) pairs of apps now have similarity scores below 80%, and can no longer be supposed to be piggybacked pairs. We manually verified 10 of pairs and found that they are indeed not piggybacked.

False negatives re-classification. Among the 761 pairs of apps in *FNData*, 110 (14%) have higher similarity scores, among which 2 pairs are now beyond the threshold of 80% which would allows to re-classify them as piggybacked pairs. We have manually verified and confirmed that these two pairs of apps are piggybacked pairs: one pair was previously used in our motivating example section (Fig. 6.1b).

6.6 Related Work

At first, we discuss a batch of works that investigate the issues related to libraries. Then, we show that even if libraries are not harmful by themselves, they threaten the validity of other approaches. Finally, we summarize the works that are dedicated to the identification of Android libraries.

Problems of Libraries. As reported by Hu et al. [96], Android libraries are currently suffering three threats: 1) the library modification threat, where normal libraries can be modified to be malicious. Our previous work has also confirmed this findings [139]. 2) the masquerading threat, e.g., a well-known malware family called *DroidKungFu* uses names such as *com.google.update* to pretend the services are provided by Google [257]. 3) the aggressive library threat, where some legitimate libraries have aggressive behaviors such as collecting users' email address.

Other works [35, 124, 125] done by us and by others, have also shown that some libraries frequently and aggressively collect (leak) users' private information. For instance, the most common leaked information is the device id, which is used by ad libraries to uniquely identify a user. This findings are in line with the investigation of Stevens et al. [209], in which the authors show that, through libraries, users can be tracked by a network sniffer across ad providers and by an ad provider across apps. Besides, they also argue that ad libraries usually require permissions beyond their real needs and some bad programmed libraries use Android's Javascript extension mechanism insecurely. AdRisk [84] focuses on detecting privacy and security risks posed by ad libraries. Most notably, it shows that some libraries even execute untrusted code from internet sources. Moreover, those untrusted code are fetched through an unsafe mechanism, which by itself has caused serious security risks.

Gui et al. [86] have shown that free ad libraries actually come with hidden cost for developers such as the rating of apps. As reported by Mojica et al. [172], ad libraries are indeed impacting the ratings of Android apps. Moreover, As illustrated by Demetriou et al. [59], ad libraries can also learn from the list of installed apps on a mobile device to infer a user's sensitive attributes like gender, age, etc.

Although our work in this paper is not dedicated to identify problems of libraries, our findings, the list of common (ad) libraries, can definitely benefit other approaches (e.g., API studies [29,131,147–149,196]) by giving them a good starting point for thorough analysis. For instance, Mojica et al. [196] leverages Software Bertillonage [56] to investigate reuse among Android apps. They have found that most of the classes reused are actually from third-party libraries. With the help of our big common library set, the precision of their results could be much improved.

Research Works threatened by Libraries. Researchers have noticed Android libraries will definitely influence the results of app clone detection [45, 53, 54, 186, 224], where most of them use a list of libraries as a whitelist. As an example, Chen et al. [45] leverage a whitelist containing 73 libraries in their approach, which is far from being a complete whitelist of existing libraries, as shown in [224], over 600 distinct libraries have been identified. However, this list is not publicly available. Besides, comparing to our findings in this paper, this list is also considerably incomplete.

For clone detection, it is important to detect and filter out third-party libraries, as the results may be doomed if the studied apps are dominated by common libraries. As an example, the very first approach presented by Linares et al. [150] shows that the results are statistically significant differences between including and excluding third-party libraries. Not only for clone detection, but also for machine learning-based malware detection, the results are threatened by common libraries. MUDFLOW [13], as an example, uses a list of 12 well-known ad libraries as a whitelist, to exclude such features that fill in them. A later work done by Li et al. [121] also leverage that list in their machine learning-based malware detection.

Our work, in this paper, provides a comprehensive list of common libraries, that can be leveraged by other approaches and thus to significantly refine their results.

Identification of Libraries. Backes et al. [15] present a light-weight and effective approach called LibScout to detect third-party libraries in Android apps. They devise a profile matching algorithm to compute a similarity score for identifying library versions. Their approach has eventually identified 164 distinct third-party libraries with 2,065 different versions. Wang et al. [224] uses an automated clustering technique to detect common libraries, in which they have found over 600 distinct libraries. Our approach is in line with their assumptions on common libraries, however, we come with a different approach and we also discriminate ad libraries from common libraries, for which they have not. The recent LibRadar [162] approach extends WuKong's clustering approach to generate unique profiles for each detected cluster. Profiles are built from the frequency of API calls within distinct packages in a cluster, which is also capable of being used for fast library identification. Another approach called AdDetect [176], identifies Android ad libraries through their semantics (e.g., the usage of Android components, or specific APIs) and then performs a ML-based classification to detect ad libraries. However, this approach does not report any findings that can benefit the Android research community.

6.7 Conclusion

We have presented our process for collecting a set of 1,113 common libraries and 240 ad libraries from a dataset of about 1.5 million Android apps. To the best of our knowledge, these two sets are the largest ones that are publicly accessible in the community of Android research. We empirically illustrate how these two library sets can be used as *whitelist*s by Android analysis approaches to improve their time performances. More specifically, we have shown that two approaches, namely machine learning-based malware detection, and piggybacking detection, can also benefit from our harvested libraries.

7 Understanding Malicious Behaviors

The previous three chapters have demonstrated that existing static security analyzers could be amplified by code instrumentation techniques to pinpoint more malicious apps. However, our endeavor is still unable to justify how malicious payloads are introduced. As a perspective for our future research, we thus conduct a thorough dissection on malicious apps, in an attempt to understand how malicious apps are actually built. In this chapter, our investigation will specifically focus on piggybacked malware, a special type of malicious apps, whose malicious payloads are easily identifiable.

Although recent research has produced approaches and tools to identify piggybacked apps, the literature lacks a comprehensive investigation into such phenomenon. We fill this gap by 1) systematically building a large set of piggybacked and benign apps pairs, which we release to the community, 2) empirically studying the characteristics of malicious piggybacked apps in comparison with their benign counterparts, and 3) providing insights on piggybacking processes. Among several findings providing insights analysis techniques should build upon to improve the overall detection and classification accuracy of piggybacked apps, we show that piggybacking operations not only concern app code, but also extensively manipulates app resource files, largely contradicting common beliefs.

This chapter is based on the work published in the following technical reports:

- Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics & Security (TIFS)*, 2017
- Li Li, Daoyuan Li, Tegawendé F Bissyandé, David Lo, Jacques Klein, and Yves Le Traon. Ungrafting Malicious Code from Piggybacked Android Apps. *Technique Report*, 2016

Contents

7.1 Introduction

Unfortunately, unlike for traditional software executables, Android app package elements can be easily modified by third parties [22]. Malware writers can thus build on top of popular apps to ensure a wide diffusion of their malicious code within the Android ecosystem. Indeed, it may be effective to simply unpack a benign, preferably popular, app and then *graft* some malicious code on it before repackaging it and distributing it for free. The resulting app, which thus piggybacks a malicious payload, is referred to as a *piggybacked* app.

Previous studies, have exposed statistics suggesting that malware is written at an industrial scale and that a given malicious component can be extensively reused in a bulk of malware [257]. These findings support the assumption that most malware might be simply piggybacked versions of official applications. Evidence of the widespread use of piggybacking by malware writers is provided in MalGenome [257], a reference dataset in the Android security community, where 80% of the malicious samples are known to be built via repackaging other apps. A more recent analysis highlights the fact that even Google Play security checkers are challenged in detecting fake apps[i], providing further evidence that the problem is widespread.

The problem of detecting piggybacked apps is eventually related to the problem of detecting app clones and repackaged apps. State-of-the-art techniques, such as DroidMoss [256] and DNADroid [53], have focused on performing pairwise similarity comparisons of app code. To alleviate the scalability problem that exists in such approaches, PiggyApp [255] builds, for every app, a vector with normalized values of semantic features extracted from components implementing the app's primary functionality. Thus, instead of computing the similarity between apps based on their code, PiggyApp computes, as a proxy, the distance between their corresponding vectors. This approach however also remains impractical, since one would require the dataset to contain exhaustively the piggybacked apps as well as their corresponding original apps. In practice however, many piggybacked apps are likely to be uploaded into different markets than where the original app can be found.

Overall, the aforementioned limitations could be overcome with approaches that leverage more knowledge on how piggybacked apps are built. Instead of a brute-force comparison, one could use semantic features that hint on probable piggybacking scenarios.

The goal of our work is to extensively investigate piggybacked apps for understanding how piggybacking is performed. We make the following contributions:

- We contribute to the research community efforts by building and sharing the first publicly available dataset[ii] on piggybacked apps. This dataset, which was collected in a systematic way and includes for each piggybacked app its associated original app, can be used as a reference ground truth for assessing approaches.
- We conduct a comprehensive study of piggybacked apps and report our findings on how piggybacked apps differentiate from benign apps, what actions are performed by piggybackers, what payloads are inserted, etc. Among several insights, we find that piggybacking is done with little sophistication, in many cases automatically, and often via library code.

We now provide a taxonomy to which we will refer to in the remainder of this work. Figure 7.1 illustrates the constituting parts of a piggybacked app. Such malware are built by taking a given original app, referred to in the literature [255] as the **carrier**, and grafting to it a malicious payload, referred to as the **rider**. The malicious behaviour will be triggered thanks to the **hook** code that is inserted by the malware writer to connect his rider code to the carrier app code. The hook thus defines the point where carrier context is switched into the rider context in the execution flow.

[i]`http://goo.gl/kAFjkQ` – Posted on 24 February 2016
[ii]`https://github.com/serval-snt-uni-lu/Piggybacking`

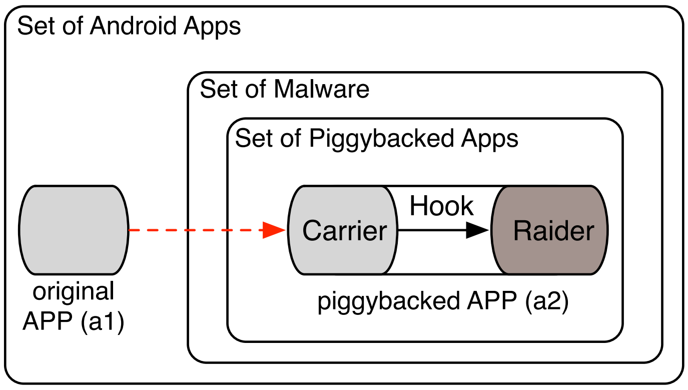


Figure 7.1: Piggybacking Terminology.

The remainder of this chapter is organized as follows. Section 7.2 details the dataset of piggybacked apps we collect. Section 7.3 presents the detailed investigation we conduct on our collected dataset, including both dimensions and findings. Section 7.4 discusses the potential outcomes of this dissection. Section 7.5 enumerates related work and Section 7.6 concludes this chapter.

7.2 Dataset Collection

Research on Android security is challenged by the scarcity of datasets and benchmarks. Despite the abundance of studies and approaches on detection of piggybacked apps, access to the associated datasets is limited. Furthermore, while other studies focus on various kinds of repackaged apps, ours targets exclusively piggybacked malware. Finally, related work apply their approach on random datasets and then manually verify the findings to compute performance. Conversely, we take a different approach and automate the collection of a ground truth that we share with the community[iii].

7.2.1 Process

Our collection is based on AndroZoo [3], a large repository of millions of apps crawled over several months from several markets (including Google Play, appchina, anzhi), open source repositories (including F-Droid) and researcher datasets (such as the MalGenome dataset). We follow the process in Figure 7.2 to collect the ground truth dataset of piggybacked apps.

First, we send all apps to VirusTotal to collect their associated anti-virus scanning reports. Based on VirusTotal's results, we classify the set of apps to two subsets: one contains only benign apps while the other contains only malicious apps.

Second, we filter out irrelevant results, in an attempt to only focus on building piggybacking pairs. This step has further been divided into three sub-steps. In step (2.1), we filter the apps based on Application Package name recorded in the Manifest file of each app, which should identify uniquely the app[iv]. Considering the identicality of package names, we were able to focus on a subset of about 540 thousand pairs $< app_g, app_m >$ of benign (app_g) and malicious[v] (app_m) apps sharing the same package name. This step yields our first set of candidate pairs. Before proceeding to the next step, we ensure that for each pair, the creation date of the benign app precede the one of the malicious app. In step (2.2), we do not consider cases where a developer may piggyback his own app to include new

[iii] `https://github.com/serval-snt-uni-lu/Piggybacking`

[iv] Two apps with the same Application Package name cannot be installed on the same device. New versions of an app keep the package name, allowing updates instead of multiple installs.

[v] In this study, we consider an app to be malware if at least one of the anti-virus products from VirusTotal has labeled it as such.

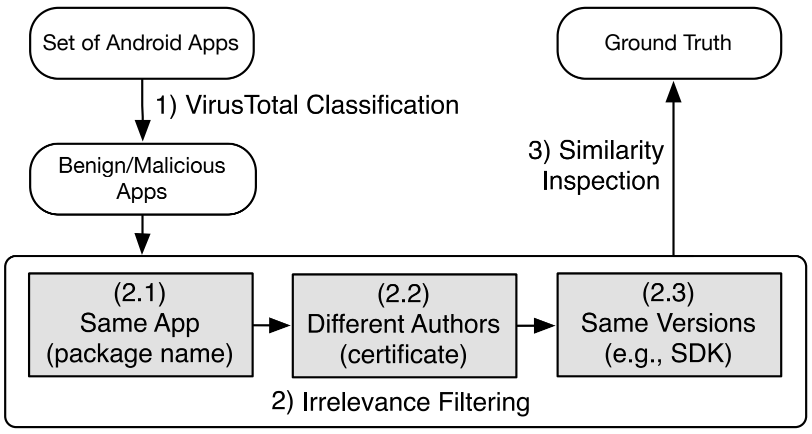

Set of Android Apps
1) VirusTotal Classification
Benign/Malicious Apps
Ground Truth
3) Similarity Inspection
(2.1) Same App (package name)
(2.2) Different Authors (certificate)
(2.3) Same Versions (e.g., SDK)
2) Irrelevance Filtering

Figure 7.2: The Ground Truth Build Process.

payload. Indeed, we consider piggybacking to be essentially a parasite activity, where one developer exploits the work of another to carry his malicious payload. Furthermore, developers may later “piggyback” their own app, e.g., to insert an advertising component to collect revenue, transforming these apps to adware (often classified as malware). We choose to clean the dataset from such apps. Thus, we discard all pairs where both apps are signed with the same developer certificate. This step brings the subset to about 70 thousands pairs. In step (2.3), we focus on cases where piggybackers do not modify version numbers and do not entirely re-implement functionality which would require new SDK versions. By considering pairs where both apps share the same version number and SDK requirements, we are able to compile a promising dataset of 1,497 app pairs where one app potentially piggybacks the other to include a malicious payload.

Finally, in order to build a final ground truth, we must validate the relevance of each pair as a piggybacking pair. To that end, we perform a *similarity analysis* where we expect that, given a pair of apps $< app_g, app_m >$, app_g's code is part of app_m and app_m includes new code to constitute the malicious payload.

7.2.2 Similarity Analysis

Similarity analysis is essential to understand the difference (or similarity) of two apps such as a pair of piggybacked apps. Given a pair of apps (app$_1$, app$_2$), we compute the following four metrics that have already been adopted by state-of-the-art tools such as Androguard [62]:

- *identical* – a given method code is exactly the same in both apps.
- *similar* – a given method has slightly changed (at the instruction level) between the two apps.
- *new* – a method has been added in the piggybacked app.
- *deleted* – a method has been deleted from the carrier code when including it in the piggybacked app.

Based on these metrics, we can now calculate the similarity score of pair (app$_1$, app$_2$) using Formula 7.1, for which we have already leveraged in Chapter 6.

$$similarity = max\{\frac{identical}{total - new}, \frac{identical}{total - deleted}\} \tag{7.1}$$

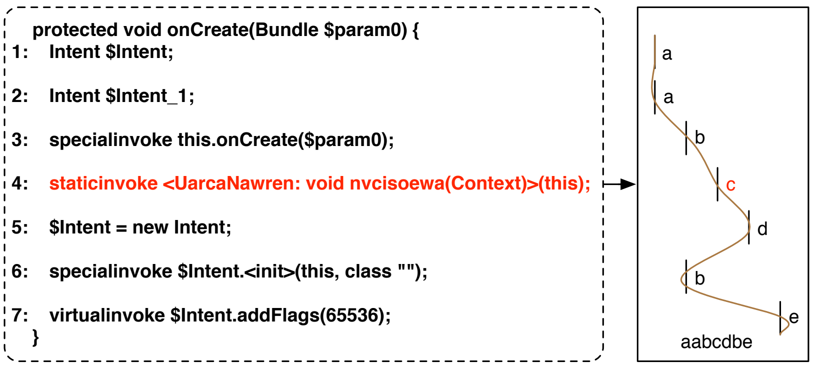
where

$$total = identical + similar + deleted + new \tag{7.2}$$

7.2.2.1 Code to text Representation

In order to enable a fast comparison, our similarity analysis is based on an abstract text that represents the logic of each app method. Given two Android apps, our similarity analysis compares their included methods based on the sequence of statements that they implement. We first pre-define a mapping between statement types[vi] and printable characters. For example, a static invocation statement could be mapped to the character *c*. Based on this mapping, every method can be represented at a high level as a small string. In the example of Figure 7.3, the content of method *onCreate()* is represented by string *aabcdbe*.

In addition to the fast comparison scheme, applying code to text representation for similarity analysis brings in another advantage: it is resilient to simple obfuscation scenarios that change the names of classes, methods and variables. As cons, the similarity measure is only an approximation.



```
protected void onCreate(Bundle $param0) {
   Intent $Intent;

   Intent $Intent_1;

   specialinvoke this.onCreate($param0);

   staticinvoke <UarcaNawren: void nvcisoewa(Context)>(this);

   $Intent = new Intent;

   specialinvoke $Intent.<init>(this, class "");

   virtualinvoke $Intent.addFlags(65536);
}
```

Figure 7.3: Illustration of Code to Text Transformation. Characters in the Right Box are Representations of the Type of Statements.

7.2.2.2 Ground truth inference

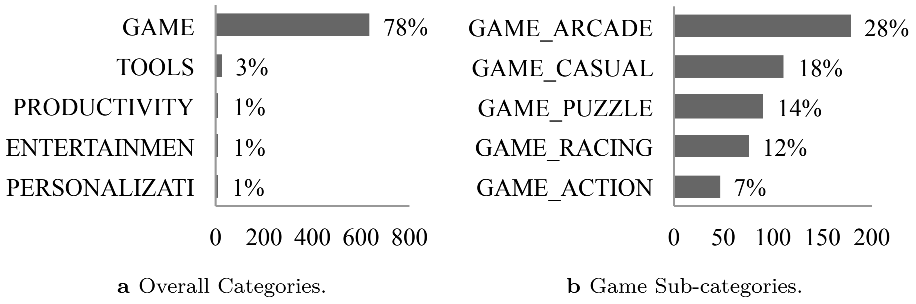
Our similarity analysis allowed to consolidate our dataset by validating that apps in a piggybacking pair were indeed very similar in terms of code content; pairs with code similarity of less than a pre-defined 80% threshold are filtered out.

We further drop cases of app pairs where no new code has been added (e.g., only resource files have been modified/added). The final set of ground truth is now composed of 950 pairs of apps.

7.2.2.3 Hook and rider identification

Methods that are found to be similar between a pair of app represent a sweet spot for characterizing piggybacking. Given a method m present in two artifacts a_1 and a_2 but differing in their implementation, our similarity analysis takes one step further by performing a fine-grained analysis to localize the added/changed/deleted statements of m in a_2 w.r.t a_1. Such statements are considered as the hooks via which the malware writer connects his malicious code to the original app (a_1). Rider code is then inferred as the set of added methods and classes in the control-flow graph that are reachable from the hook statements.

[vi]https://ssebuild.cased.de/nightly/soot/javadoc/index.html



a Overall Categories. **b** Game Sub-categories.

Figure 7.4: Top 5 Categories (Left) and Game Sub-categories (Right) of Android Apps Used to Carry Piggybacking Payloads.

7.2.3 Dataset Characterization

For each piggybacking pair in our dataset, we collect the metadata information (i.e., creation date, description, categorization, download count) of the original app by looking in their Manifest file and by crawling description pages from specialized websites[vii].

Variety of app categories. Statistics of app categories, illustrated in Figure 7.4, show that Game apps are the most representative category in our piggybacked datasets. A random sampling on our dataset of 2 million apps shows that Games also constitute the largest category of apps. Our dataset of piggybacking pairs further includes piggybacked apps from a variety of 22 categories such as Productivity, Tools, Entertainment, Personalization, Social Networking, Shopping, Sports, Weather, Transportation or News.

Distribution channels. Piggybacked apps from our dataset were found in several markets. While many were distributed on alternative markets such as anzhi (71%) and appChina (14%), some are actually spread via the official Google Play market (2.2%).

Popularity of apps. The distribution of download counts of original apps in piggybacking pairs and of a random sample of benign apps from Google Play, shows that mainly popular apps are leveraged by malware writers to carry their malicious payloads (cf. Figure 7.5).

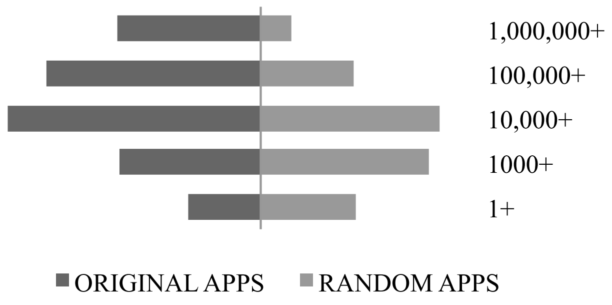


Figure 7.5: Download Counts of Apps Leveraged for Piggybacking vs. Apps Randomly Sampled in GooglePlay. (Sets are of Same Size)

Variety of malicious behaviour. We have collected AV reports from around 50 AV engines provided by VirusTotal for the piggybacked apps. As a proxy to the type/family of malware they are categorized in, we rely on AV labels. Our piggybacked apps were qualified with over 1000 distinct labels by the AVs. Even considering that each AV has its own naming system, this high number suggests a variety of malicious behaviour implemented in the piggybacked payloads.

[vii]Notably `bestappsmarket.com`

Piggybacking delay. For each piggybacking pair, we compute the time difference to explore the time delay before an app is leveraged for piggybacking. Distribution in Figure 7.6 shows that piggybacking operation can be done on apps of any age, with a median delay of 33 days. On average however, malware writers wait half a year before using a “known” app to spread malicious payload.

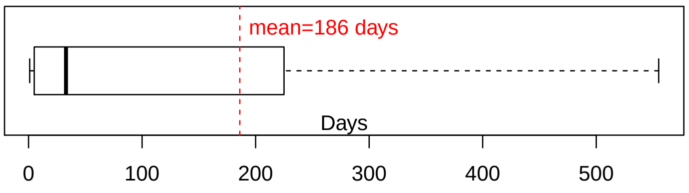


Figure 7.6: App Age when it is Piggybacked (Median=33 Days)

Piggybacking pair similarity. Figure 7.7 plots the similarity analysis results of our collected dataset. Generally, a small amount of methods in carrier are modified in the piggybacked code and the remaining methods are kept identical. In most cases, piggybacked apps also introduce new methods while piggybacked apps remove methods from the original carrier code in only a few cases.

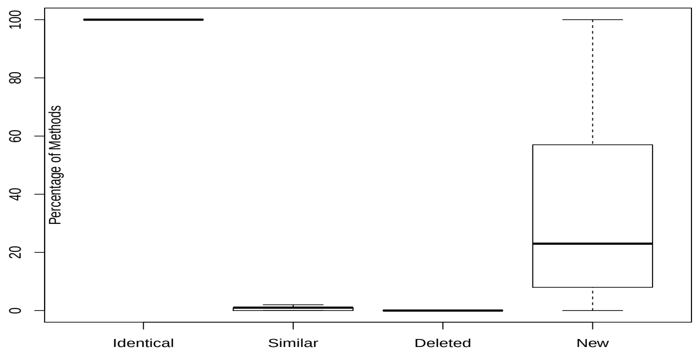


Figure 7.7: Overview of the Similarity Analysis Findings for the Pairs of Apps in Our Collected Dataset.

7.3 Understanding Piggybacking

Our investigation into Android app piggybacking is carried out through a dissection of Android piggybacked apps. We overview in Section 7.3.1 the various dimensions of dissection that we used to answer several relevant research questions for understanding the piggybacking process. Subsequently, we detail our study and the findings in Section 7.3.2.

7.3.1 Dimensions of Dissection

Our study explores several aspects of Android app piggybacking. Overall we look into:

- ***Which*** *app elements are manipulated by piggybackers?*

An Android application package includes a *classes.dex* file that contains the main code implementing app functionality. Besides this code, the apk file includes icons and background image files, layout description XML files, as well as other resource files such as library Jar files and archives containing extra code.

Image and layout resource files are important for malware writers who must ensure, beyond advertised functionality, that the malicious app has the proper “look and feel”. Package names, launcher activity and component lists are also essential elements described in the Manifest file that could be manipulated during piggybacking to ensure that the app is transparently installed on user devices, even when the original app is already installed, and that the malicious code is triggered seamlessly. We investigate the extent of these modifications to answer the following research questions:

RQ-01 Are resource files as much concerned by piggybacking operations as app bytecode?

RQ-02 What types of resources are included in the malicious payload?

RQ-03 Are rider code samples redundant across piggybacked apps?

RQ-04 What changes are applied to the Manifest file during piggybacking?

RQ-05 What developer certificates can be found in piggybacked apps?

- ***How*** *app functionality and behaviour are impacted?*

Android apps are made up of four types of components: 1) Activity, which is used to represent the visible part of Android apps; 2) Service, which is dedicated to execute tasks in the background; 3) Broadcast Receiver, which waits for receiving user-specific or system events; and 4) Content Provider, which plays as a standard means for structural data access. These components may not be equally important for spreading malicious payloads.

App components use *Intents* as the primary means for exchanging information. When the Android OS resolves an intent which is not explicitly targeted to a specific component, it will look for all registered components that have *Intent filters* with actions matching the intent action. Indeed, Intent filters are used by apps to declare their capabilities. Piggybacked apps may declare new capabilities to trick users into triggering themselves the malicious behaviour.

Finally, every Android app must be granted the necessary permissions to access every sensitive resource or API required for its functioning. If the malicious code interacts with more sensitive resources than the original app, new permissions must be requested in the Manifest file.

In the following research questions, we investigate in details the scenarios mentioned above.

RQ-06 Does the malicious payload require extra-permissions to execute?

RQ-07 Which types of components are more manipulated in piggybacking?

RQ-08 Is piggybacking always performed manually for every target popular app?

RQ-09 What kind of actions, encoded in *Intents*, are created by piggybacked rider code?

RQ-10 Are there new sensitive data leaks performed for the piggybacking needs?

- ***Where*** *malicious code is hooked into benign apps?*

To elude detection, malware writers must identify the sweet spots to graft malicious rider code to the benign carrier code. Programmatically, in Android, there are mainly two ways to ensure that rider code will be triggered during app execution. We refer to the two ways as **type**$_1$ and **type**$_2$ hooks. Type$_1$ hooks modify carrier code to insert specific method calls that connect to rider code, which does not need to modify the Manifest file. Conversely, Type$_2$ hooks come in as rider components (need to register in Manifest), which can be launched independently (e.g., via user actions such as clicking, or system events). It is thus important to investigate to what extent piggybackers place hooks that ensure that the rider code will be executed (e.g., with only type$_2$ hooks or both?).

Piggybackers may also attempt to elude static detection of malicious behaviour by dynamically loading parts of the rider code. Finally, rider code may deliver malicious behaviour from various families. We investigate these aspects through the following research questions:

RQ-11 Is the injected rider code complex?

RQ-12 Is the piggybacking rider code limited to the statically accessible code?

RQ-13 How tightly is the rider code integrated with the carrier code?

RQ-14 Are hook code samples specific to each piggybacked app?

RQ-15 What families of malware are spread via piggybacking?

7.3.2 Findings

In this section we report on the findings yielded by our investigation of the research questions outlined above. For each of the findings, named from **F1** to **F20**, we provide the take-home message before providing details on the analysis.

When a finding involves a comparison of a characteristic of piggybacked app w.r.t original apps, we have ensured that **the difference is statistically significant** by performing the Mann-Whitney-Wilcoxon (MWW) test. MWW is a non-parametric statistical hypothesis test for assessing the statistical significance of the difference between the distributions in two datasets [166]. We adopt this test as it does not assume any specific distribution, a suitable property for our experimental setting. In this work, we consider a significance level at $\alpha = 0.001$[viii].

For readers' quick reference, we enumerate in Table 7.1, the findings that were yielded in the investigation of each Research question.

Table 7.1: Contributions of Findings to Research Question.

Research Question	Contributing Findings	Research Question	Contributing Findings
RQ-01	F1, F2	RQ-09	F15
RQ-02	F1, F2	RQ-10	F20
RQ-03	F13, F14, F15	RQ-11	F17, F18
RQ-04	F4, F5, F8, F9 F10, F11	RQ-12	F2, F17
RQ-05	F3, F19	RQ-13	F12, F13
RQ-06	F4, F5, F6, F7	RQ-14	F12, F13
RQ-07	F8	RQ-15	F16
RQ-08	F3, F6		

F1▶ *The realisation of malicious behaviour is often accompanied by a manipulation (i.e., adding/removing/replacing) of app resource files.*

In our dataset collection, we only considered piggybacking cases where code has been modified to deviate from original behaviour implementation. Thus, we focus on investigating how other app elements are treated during piggybacking. As illustrated in Fig. 7.8, most (91%) piggybacked apps have added new resource files to the original apps, while only a few (6%) have left the resources files of the original apps untouched.

We first investigate the cases where the piggybacked app has removed resource files from the app carrying its malicious payload. Fig. 7.9 highlights that many image files are removed as well as some Java serialized objects. At a lesser extent, Plain text files, which may contain configuration information, and XML files, which may describe layouts, are also often removed/replaced.

We further investigate the removal of resource files during piggybacking. A common case that we have found is that resource file removal actually corresponds to files renaming. For example, for the piggybacking pair (*A801CF*[ix] to *59F8A1*) , six PNG files under the *drawable* directory have been

[viii] When the null hypothesis is rejected, there is one chance in a thousand that this is due to a coincidence.

[ix] In this paper, we refer to apps using the last six letters of their SHA256 hash, to enable quick retrieval in Androzoo.

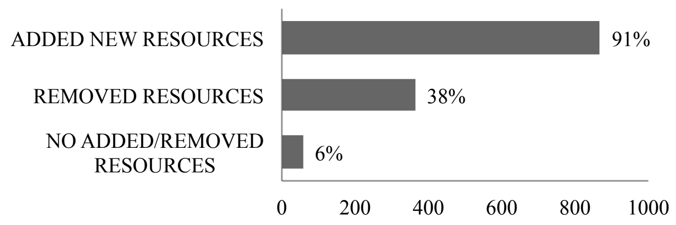


Figure 7.8: Distribution of piggybacked apps that add/remove resource files to/from their original counterparts.

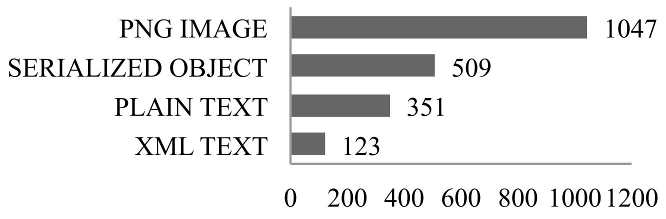


Figure 7.9: Top types of resource files removed during piggybacking.

slightly renamed. Besides file names, file extensions (e.g., from PNG to GIF) and parent directories (e.g., from *drawable-480dpi* to *drawable-xxhdpi*) can be changed during piggybacking. This finding suggests that piggybackers attempt to mitigate any possibility of being identified in basic approaches, e.g., through resource-based similarity analysis. We have computed for example the similarity scores in the piggybacking pair (*A801CF* to *59F8A1*), and found that the resource-based similarity score is 72.3% while the code-based similarity score reaches 99.9%.

F2▶ *Piggybacking modifies app behaviour mostly by tampering with existing original app code.*

Although *classes.dex* remains the main site where app functionality is implemented, Android apps can carry extra code that may be invoked at runtime. We investigate the resource files added by piggybacked apps to check that they do not actually constitute the more subtle means used by malware writers to insert malicious payload. Fig. 7.10 shows that most of the added resources are media data (i.e., audio, video and image files). Extra DEX code have been added in only 8% of piggybacked apps. In 10% of cases, native compiled code in ELF format has been added.

These results suggest that sophisticated piggybacking which manipulates extra code is still limited. In most cases, malware writers simply modify the original app code.

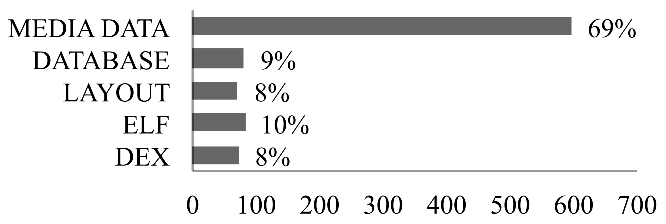


Figure 7.10: Distribution of piggybacked apps according to the types of added resource files.

F3▶ *Piggybacked apps are potentially built in batches.*

In our effort towards understanding app piggybacking, we investigate the developers who sign the certificates of those apps. We found that some certificates have been used for several apps. For example, RANGFEI.RSA certificate key appears in 71 piggybacked apps of our dataset, suggesting that the owner of this signature is intensively producing batches of piggybacked apps.

We further consider the case of the 71 piggybacked apps signed with RANGFEI.RSA to investigate whether the developer injected similar malicious payloads in the apps. To that end, we first cluster the 71 apps through the Expectation-Maximization algorithm [173] using rider class names as features, yielding 8 clusters. We then investigate how different apps in one cluster are from apps in the others. To that end, we consider the set of labels and compute the Jaccard distance between sets of labels across clusters. We found that the clusters were highly distant (average distance > 0.7), suggesting different payloads in the associated piggybacked apps.

F4▶ *Piggybacking often requires new permissions to allow the realisation of malicious behaviour.*

For every piggybacking pair, we check in the Manifest files how requested permissions differ. We found that 812 (85%) piggybacked apps have requested new permissions that were not initially requested by their original counterparts. Fig. 7.11 enumerates the top 10 newly added permissions in our dataset of piggybacked apps. 6 out these 10 permissions are for allowing access to sensitive user/phone information. We note that almost half of the piggybacked apps require the WAKE_LOCK permission which enables it to keep device processor from sleeping or screen from dimming. This permission is mostly used by apps that must keep executing tasks even when the user is not actually using his device.

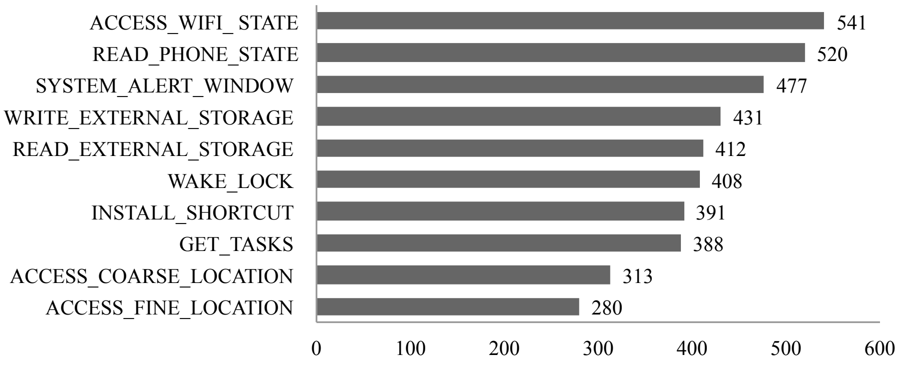


Figure 7.11: Top 10 permissions that are newly added.

F5▶ *Some permissions appear to be more requested by piggybacked apps than non-piggybacked apps.*

From the newly added permissions of Fig. 7.11, we note that permission SYSTEM_ALERT_WINDOW, which is requested by 458 (i.e., ≃50%) piggybacked apps, is only requested by 26 (i.e., 3%) original benign apps (as illustrated in Fig. 7.12). When considering the top 10 newly added permissions only, we note that they are requested, on average, by 65% piggybacked apps and only 26% benign original apps.

Fig. 7.13 further shows that, excluding outliers, a given piggybacked app requests a minimum of 5, and median number of 7 (out of the 10) from the top permissions list, while original benign apps only request a maximum of 4.

F6▶ *Piggybacking is probably largely automated.*

Since permissions are a key element in Android app programming, a malware writer must systematically ensure that the necessary permissions are granted to its app to enable the working of inserted malicious payloads. We investigate how permissions are added, and note that there seems to be a naive automated process for adding permissions. Indeed, we found that permissions requests are often duplicated within the same Manifest file. For example, permission READ_PHONE_STATE, which

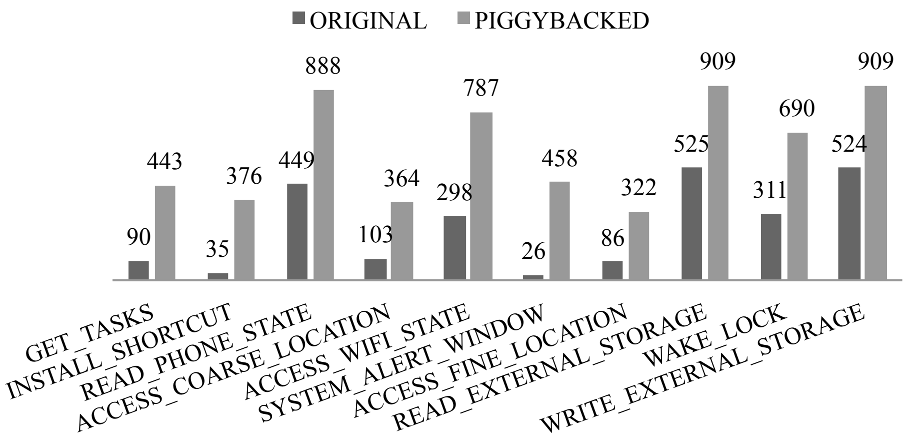


Figure 7.12: Distribution of requests of the top 10 permissions across original and piggybacked apps.

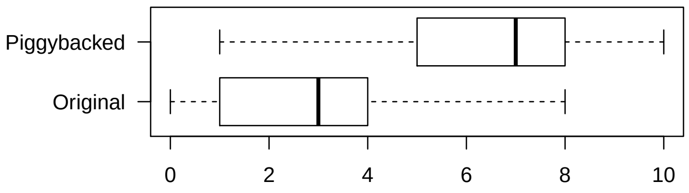


Figure 7.13: Distribution of the number of top-10 permissions requested by original apps and piggybacked apps.

is among the top newly added permissions by piggybacked apps, has been systematically duplicated in 58 apps. Often both permission requests appear consecutively in the Manifest file as illustrated in Listing 7.1 which is extracted from app *713414*, suggesting that the second permission was not manually added by a developer.

```
uses-perm:"android.permission.ACCESS_WIFI_STATE"
uses-perm:"android.permission.READ_PHONE_STATE"
uses-perm:"android.permission.READ_PHONE_STATE"
uses-perm:"android.permission.GET_ACCOUNTS"
```

Listing 7.1: Example of duplicated permission declaration.

In our dataset, we have found that there is at least one duplicate permission for 590 piggybacked apps, where 576 of them have a permission existing already in the original apps.

F7▶ *Piggybacked apps overly request permissions, while leveraging permissions requested by their original apps.*

As illustrated in Fig. 7.13, and discussed previously, piggybacked apps appear to "systematically" include a larger set of permissions than their original counterparts. We investigate the control-flow of piggybacked apps to check whether they reached the sensitive APIs that are protected by their requested permissions (based on PScout's results [12]). We found that 759 (or 80%) piggybacked apps have declared more permissions than necessary. This finding is inline with previous studies [25] on Android apps, including legitimate apps, which showed that app authors are usually unaware of what exact permissions are needed for the APIs leveraged in their apps. Similarly, our results suggest that piggybacked app writers are not aware of the permissions needed for the APIs accessed by their injected payloads.

Fig. 7.14 shows that half of the piggybacked apps have requested at least two more permissions that they do not leverage. At the same time, rider code actually use APIs protected by permissions originally requested by original apps.

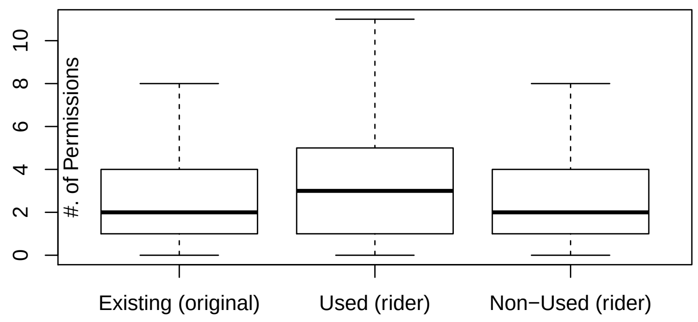


Figure 7.14: Distribution of the number of permissions relating to the rider code.

F8▶ *Most piggybacked apps now include new user interfaces, implement new receivers and services, but do not add new database structures.*

Fig. 7.15 highlights the statistics of components added by piggybacking. 834 (or 88%) of apps have added Activity components during piggybacking: these represent new user interfaces that did not exist in the original apps. We manually check a random sample of those components and find that they are mostly for displaying advertizement pages. We checked all such apps and their associated original apps and found that they added new ad libraries to redirect the ad revenues to the accounts of piggybackers. Fig. 7.16 shows that half of the piggybacked apps even add several Activity components. Those components can represent different advertisement libraries. As an example, the piggybacking process that led to app *90D8A5* has brought in two ad libraries: *com.kuguo.ad* and *net.crazymedia.iad*. Although this phenomenon is not very common, it does suggest that piggybackers may take steps to maximize the opportunities of gaining benefits.

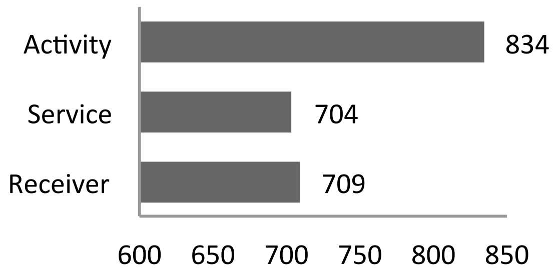


Figure 7.15: # of piggybacked apps adding at least one new component.

Service and Broadcast Receiver components are also largely injected by piggybacking payload. While Broadcast Receivers register to device events (e.g., WIFI connection is active) so that the malicious code can be triggered, Service components run background tasks which can deplete device resources (cf. recent clicker trojans[x] found in GooglePlay). We found that, in most cases, when a *service* is injected, a *receiver* will be correspondingly injected. In this case, the receiver plays as a proxy for ensuring that the service will be launched. Indeed, it is much easier for trigger malicious behaviour via a receiver than to make a user start a service. This finding has also been reported at a smaller scale in our previous work [121].

Interestingly, we have found that some newly injected components are shipped with empty implementations. In Listing 7.2, component *com.idpack.IRE* is a newly injected class in app *39DE41*.

[x]`http://goo.gl/kAFjkQ`

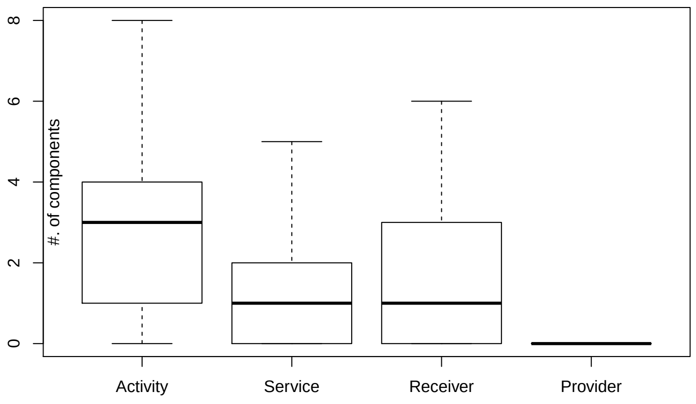


Figure 7.16: Distribution of the number of components newly added.

In the manifest file, *com.idpack.IRE* has been declared with a lot of capabilities (i.e., six actions), making it likely to be triggered. For example, it will be launched when a new app is installed (cf. line 7) or uninstalled (cf. line 8). However, as shown in line 2, the implementation of component *com.idpack.IRE* is surprisingly empty. When executing the component, it will reuse the implementation of its parent class named *com.is.p.Re*. Unfortunately, this indirection is often ignored in trivial static analysis approaches, leaving the real implementation of injected components unanalyzed. It would be interesting to investigate to what extent such tricks can impact the results of existing static analysis approaches. This, however, is out of the scope of this paper and therefore we consider it for future work.

```
 1 //The implementation of component com.idpack.IRE
 2 public class com.idpack.IRE extends com.is.p.Re { }
 3
 4 //The manifest declaration of component com.idpack.IRE
 5 <receiver android:name="com.idpack.IRE">
 6 <intent-filter>
 7 action:"android.intent.action.PACKAGE_ADDED"
 8 action:"android.intent.action.PACKAGE_REMOVED"
 9 <data android:scheme="package" />
10 </intent-filter>
11 <intent-filter>
12 action:"android.net.conn.CONNECTIVITY_CHANGE"
13 action:"android.intent.action.USER_PRESENT"
14 action:"com.lseiei.downloadManager"
15 action:"com.cdib.b"
16 </intent-filter>
17 </receiver>
```

Listing 7.2: An example of empty component implementation (app *39DE41*).

Finally, we note that no piggybacked apps add new Content Provider components. This finding is inline with the intuition that malicious apps are not targeted at structuring data for sharing with other apps.

F9▶ *Piggybacking often consists in inserting a component that offers the same capabilities as an existing component in the original app.*

We noted that piggybacking may add a component with a given capability which was already declared for another component in the carrier app. This is likely typical of piggybacking since there is no need in a benign app to implement several components with the same capabilities (e.g., two PDF reader components in the same app). For example, in our ground truth dataset, we have found that in each of 551 (i.e., 58%) piggybacked apps, several components have the same declared capability. In contrast, 6% of the benign original apps included components with the same capability.

```
receiver:com.sumase.nuatie.wulley.RsollyActivity
action:"android.intent.action.PACKAGE_ADDED"
action:"android.net.conn.CONNECTIVITY_CHANGE"
action:"android.intent.action.USER_PRESENT"
receiver:com.fuonw.suoetl.cuoll.TsenaActivity
action:"android.intent.action.PACKAGE_ADDED"
action:"android.net.conn.CONNECTIVITY_CHANGE"
action:"android.intent.action.BOOT_COMPLETED"
receiver:com.hunstun.tallsyen.blawe.RsekleeActivity
action:"android.intent.action.PACKAGE_ADDED"
action:"android.net.conn.CONNECTIVITY_CHANGE"
action:"android.intent.action.USER_PRESENT"
receiver:com.luotuy.mustn.VenrowoActivity
action:"android.intent.action.PACKAGE_ADDED"
action:"android.net.conn.CONNECTIVITY_CHANGE"
action:"android.intent.action.USER_PRESENT"
```

Listing 7.3: An example of duplicated component capabilities (app *4C35CC*).

We then perform a manual investigation of the duplicated component capabilities and make two observations: 1) In most cases, duplicated component capabilities are associated with *Broadcast Receivers*. This finding is intuitively normal since receivers can be easily triggered (e.g., by system events) and consequently can readily lead to the execution of other (targeted) components. 2) All the duplicated component capabilities are likely to be newly injected. Because piggybackers intend to maximize their benefits, they will usually inject at the same time different modules. Those modules are independent from each other and each of them will attempt to maximize its possibility of being executed. As a result, each module declares a receiver to listen to popular system events, further resulting in duplicated component capabilities. Listing 7.3 shows an example of duplicated component capabilities. All the four receivers (possibly from four modules as indicated by the name of receivers) are newly injected and all of them have declared the same capabilities.

Fig. 7.17 enumerates the top 10 duplicated capabilities that are leveraged by piggybacked apps. The three capabilities presented in Listing 7.3 are found to be the top duplicated capabilities by piggybackers. Actually, the corresponding system events, including 1) new app installed (*PACKAGE_ADDED*), 2) internet connection changed (*CONNECTIVITY_CHANGE*), and 3) phone unlocked (*USER_PRESENT*), are commonly-fired events in Android devices.

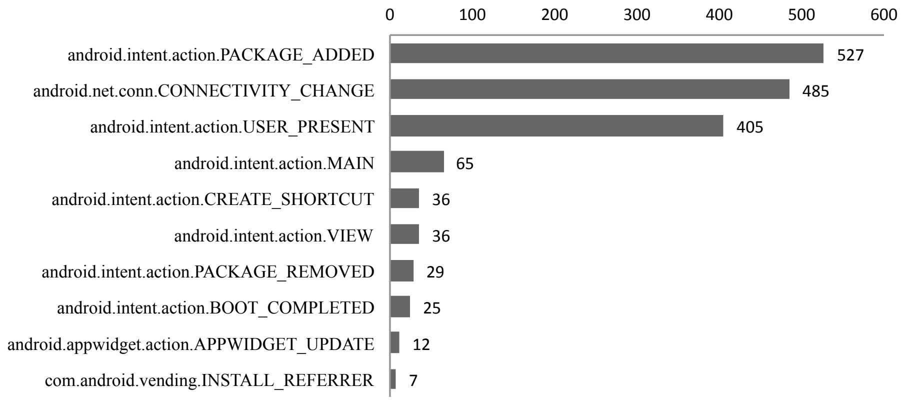


Figure 7.17: Top 10 duplicated component capabilities (actions).

F10▶ *Piggybacked apps can simply trick users by changing the launcher component in the app, in order to trigger the execution of rider code.*

As previously explained, Android apps include in their Manifest file an Application package name that uniquely identifies the app. They also list in the Manifest file all important components, such as the `LAUNCHER` component with its class name. Generally Application package name and Launcher component name are identical or related identically. However, when a malware writer is subverting app users, she/he can replace the original Launcher with a component from his malicious payload. The app example from Listing 7.4 illustrates such a case where the app's package (`se.illusionlabs.labyrinth.full`, line 1) and launcher (`com.loading.MainFirstActivity`, line 10) differ.

```
 1 <manifest package="se.illusionlabs.labyrinth.full">
 2 activity:"se.illusionlabs.labyrinth.full.
 3 StartUpActivity"
 4 action:"android.intent.action.MAIN"
 5 -    category:"android.intent.category.LAUNCHER"
 6 +   activity:"com.loading.MainFirstActivity"
 7 +    action:"android.intent.action.MAIN"
 8 +    category:"android.intent.category.LAUNCHER"
 9 +   receiver:"com.daoyoudao.ad.CAdR"
10 +    action:"android.intent.action.PACKAGE_ADDED"
11 +    <data android:scheme="package" />
12 +   service:"com.daoyoudao.dankeAd.DankeService"
13 +    action:"com.daoyoudao.dankeAd.DankeService"
14 </manifest>
```

Listing 7.4: Simplified view of the *manifest* file of *se.illusionlabs.labyrinth.full* (app's sha256 ends with `7EB789`).

We investigate our piggybacking pairs to identify cases where the piggybacked apps has changed their `LAUNCHER` component, comparing to the original benign app. Table 7.2 illustrates some examples on the original and updated `LAUNCHER`. These changes in the Manifest file are essential for piggybackers to ensure that their code is run. We found 73 cases in our dataset where the piggybacked app switched the original launcher component to one component that was added as part of its rider code.

Table 7.2: Illustrative examples of launcher changes.

Original Launcher	Updated Launcher
com.unity3d.player.UnityPlayerActivity	com.sorted.android.probe.ProbeMain
com.ubermind.ilightr.iLightrActivity	com.geinimi.custom.Ad3034_30340001
com.virgil.basketball.BasketBallActivity	cn.cmgame.billing.ui.GameOpenActivity
game.main.CatchThatKid_UIActivity	cn.dena.mobage.android.MobageActivity
jp.seec.escape.stalking.EscapeStalking	com.pujiahh.Main

F11▶ *Piggybacking is often characterized by a naming mismatch between existing and inserted components.*

Since Android apps are mostly developed in Java, different modules in the application package come in the form of Java packages. In this programming model, developer code is structured in a way that its own packages have related names with the Application package name (generally sharing the same hierarchical root, e.g., `com.example.*`). When an app is piggybacked, the inserted code comes as separated modules constituting the rider code with different package names. Thus, the proportions in components that share the application package names can also be indicative of piggybacking. Such diversity of package names can be seen in the example of Listing 7.4. The presented excerpt already contains three different packages. Since Android apps make extensive use of Google framework libraries, we systematically exclude those in our analysis to improve the chance of capturing the true diversity brought by piggybacking.

Fig. 7.18 illustrates that piggybacked apps present a higher diversity of packages names in comparison with their original benign counterparts.

F12▶ *Malicious piggybacked payload is generally connected to the benign carrier code via a single method call statement.*

Table 7.3: Top five $type_1$ hooks used by piggybacked apps.

Method	# piggybacked apps
$< com.basyatw.bcpawsen.DaywtanActivity : void\ Tawo(android.content.Context) >$	42
$< com.gamegod.touydig : void\ init(android.content.Context) >$	39
$< com.geseng.Dienghla : void\ init(android.content.Context) >$	34
$< com.gamegod.touydig : void\ destroy(android.content.Context) >$	24
$< com.tpzfw.yopwsn.Aervlrey : void\ Tdasen(android.content.Context) >$	15

Table 7.4: Top five methods in carrier code where hook statements are inserted.

Method	# piggybacked apps
$< com.unity3d.player.UnityPlayerProxyActivity : void\ onCreate(android.os.Bundle) >$	95
$< com.ansca.corona.CoronaActivity : void\ onCreate(android.os.Bundle) >$	29
$< org.andengine.ui.activity.BaseGameActivity : void\ onCreate(android.os.Bundle) >$	11
$< com.prime31.UnityPlayerProxyActivity : void\ onCreate(android.os.Bundle) >$	10
$< com.g5e.KDLauncherActivity : void\ onCreate(android.os.Bundle) >$	8

Table 7.5: Statistics on the pairwise similarity between rider code of piggybacked apps.

Jaccard Distance	Method-Level		Class-Level		Android-API-Level	
	# of Apps	# of Relevant Pairwise Combinations	# of Apps	# of Relevant Pairwise Combinations	# of Apps	# of Relevant Pairwise Combinations
$= 0$	599	8,890	605	9,078	791	11,482
≤ 0.05	806	10,178	794	10,814	914	18,662
≤ 0.1	821	10,384	813	11,222	933	25,788
≤ 0.2	845	10,718	833	11,836	967	42,660

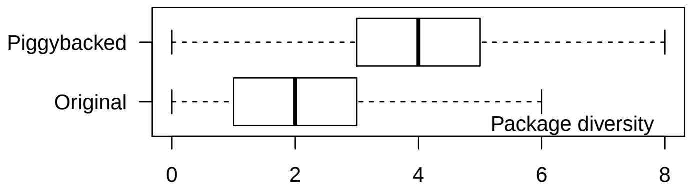


Figure 7.18: Distribution of the number of different packages.

Table 7.6: Top five Android APIs that are interested by rider code.

Method	Count
$< android.net.Uri : android.net.Uri\ parse(java.lang.String) >$	955
$< android.content.Context : java.lang.Object\ getSystemService(java.lang.String) >$	953
$< android.content.BroadcastReceiver : void\ < init > () >$	949
$< android.content.Context : android.content.pm.PackageManager\ getPackageManager() >$	941
$< android.app.Activity : void\ < init > () >$	939

Listing 7.5 illustrates a snippet showing an example of $type_1$ hook (cf. definition in Section 7.3.1), where the hook (line 4) is placed immediately at the beginning of the *onCreate* method to trigger the malicious payload when component *UnityPlayerProxyActivity* is activated. The fact that a *static* method is leveraged somewhat suggests that piggybackers attempt to pay least effort to connect benign carrier code to rider code. The simplified implementation of *touydig.init()* is shown in lines 11-20. Generally, this method is used to initialize three modules which are simultaneously injected by piggybackers. All the three modules, including *umeng*, *kuguo*, and *crazymedia*, are injected to provide advertizement and consequently to redirect the revenues from the original developers to piggybackers.

In our dataset, 699 (74%) piggybacked apps use $type_1$ hooks to connect rider code to carrier code. Among those, 438 (i.e., 63% of $type_1$) hooks only include a single statement, creating a weak link between the call-graphs of rider and carrier code. Table 7.3 enumerates the most redundant hook statements used in piggybacked apps.

```
 1 //In class UnityPlayerProxyActivity
 2 protected void onCreate(Bundle b) {
 3 this.onCreate(b);
 4 + com.gamegod.touydig.init(this);
 5 $r2 = new String[2];
 6 $r2[0] = "com...UnityPlayerActivity";
 7 $r2[1] = "com...UnityPlayerNativeActivity";
 8 UnityPlayerProxyActivity.copyPlayerPrefs(this, $r2);
 9 }
10 
11 class com.gamegod.touydig {
12 public static void init(Context $param0) {
13 //initialize umeng module
14 $String = $bundle.getString("UMENG_CHANNEL");
15 //initialize kuguo module
16 $KuguoAdsManager = KuguoAdsManager.getInstance();
17 $KuguoAdsManager.setCooId($param0, "ae63b5208de5422f9313d577b3d0aa41");
18 //initialize crazymedia module
19 net.crazymedia.iad.AdPushManager.init($param0, "8787", "wx1x6nkz277ruch5", false);
20 }}
```

Listing 7.5: Simplified view of the snippet of app *com.SkillpodMedia.farmpopfrenzy* (app's sha256 ends with `BF3978`). The '+' sign at Line 4 indicates the statement that was added to the original code.

F13▶ *Piggybacking hooks are generally placed within library code rather than in core app code.*

We look into the carrier method implementations where hook statements are placed. Table 7.4 presents the top 5 methods where hook statements can be recurrently traced to. Interestingly, all five methods are actually part of well-known libraries that are used for benign functionality. We further found that these libraries actually propose utility functions for Game apps.

F14▶ *Injected payload is often reused across several piggybacked apps.*

We compute the pairwise similarity of rider code using the Jaccard distance at the class level, method level and API level. When the set of classes, methods or used APIs are the same between two apps, the distance is zero. When there is no overlap this distance amounts to 1. Table 7.5 provides statistics which reveal that more than half piggybacked apps include a rider code which can be found in another piggybacked app. The Jaccard distance indeed amounts to 0 for 599, 605 and 791 piggybacked apps when comparing at the method, class and API levels respectively. Theoretically, There are in total $\binom{2}{\#ofapps}$ relevant combinations.

The similarity of comparison results at the class, method and API levels, suggest that when piggybackers reuse an existing rider code sample, they rarely make modifications. Finally, there are more pairs of rider code which are similar at the API level, suggesting that many piggybacking payloads often include code that perform similar tasks (e.g., for advertizements).

F15▶ *Piggybacking adds code which performs sensitive actions, often without referring to device users.*

Intents are special objects used by Android apps to exchange data across components. It allows app components to reuse functionalities implemented in other components (within or outside the app). Intent objects include a field for indicating the action that must be performed on the data exchanged (e.g., VIEW a PDF file, or DIAL a number). Fig. 7.19 enumerates the top actions that can be seen in all intents used by rider code across the piggybacked apps. In most cases, intents transfer data to be viewed, a necessary action for displaying advertizements. In many cases also, rider code installs shortcuts, performs dialing, shares the data outside the app, etc. We also note that two of the top actions (the *6th* and *8th*) are not system-defined. Instead they are defined in user code, and their presence in many piggybacked apps further confirms the reuse of payloads across apps.

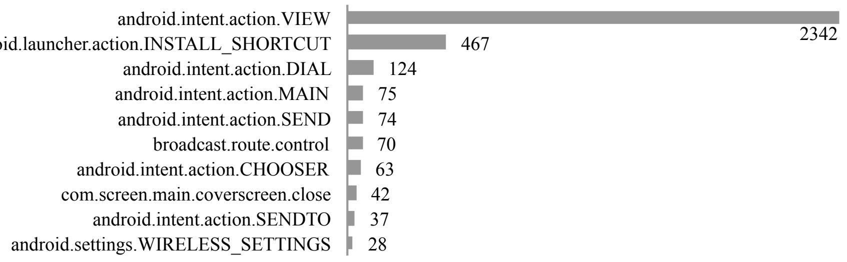


Figure 7.19: Top 10 actions from Intents brought in rider code.

Intents are either implicit (the component which will realize the action will be selected by the user) or explicit (a component is directly designated, bypassing user's choice). Differences in distributions presented in Fig. 7.20 suggest that piggybacking rider code contain more explicit intents than implicit intents. In contrast, as shown in [124], benign apps actually attempt to use more implicit Intents (for showing Activities).

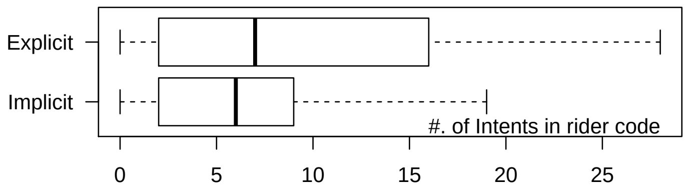


Figure 7.20: Use of explicit vs. implicit Intents in rider code.

F16▶ *Piggybacking operations spread well-known malicious behaviour types.*

We investigate the labels provided by AV engines and extract the top appearing names to classify malware types. Table 7.7 provides statistics on the three outstanding types. Although most are flagged as Adware, we note that there are overlapping with Trojans and Spyware. This is inline with the recent analysis of a family of trojans by ESET, showing that they act as adware.

Table 7.7: Outstanding malware types reported by AV engines on the piggybacked apps.

App type	# of piggybacked apps	# of distinct labels
Adware	888	230
Trojan	495	451
Spyware	192	94

We further match the AV labels against 6 well-known malware family names reported in [257]. We found that our dataset contains samples belonging to each family as described in Table 7.8. For example, clustering the rider code of piggybacked apps in the GingerMaster[xi] family based on class names and with EM algorithm yields 5 clusters, one of which contains over 80 samples, suggesting a high reuse of a malicious payload code.

Table 7.8: Number of piggybacked apps in known malware families.

ADRD	BaseBridge	Geinimi	GingerMaster	GoldDream	Pjapps
11	5	21	149	11	9

F17▶ *Piggybacked apps increasingly hide malicious actions via the use of reflection and dynamic class loading.*

[xi]These malware collect and send user info, including device ID and phone number, to a remote server.

The Android system provides DexClassLoader, a class loader that loads classes from .jar and .apk files. It can be used by malware writers to execute code not installed as part of their apps. We found 185 (19%) piggybacked apps whose riders dynamically load code. Such schemes are often used by malware writers to break the control-flow of app, and thus challenge static detection of suspicious behaviour. Fig. 7.21 provides the ratio of piggybacked apps using reflection and dynamic code loading. The ratio has substantially improved in recent years.

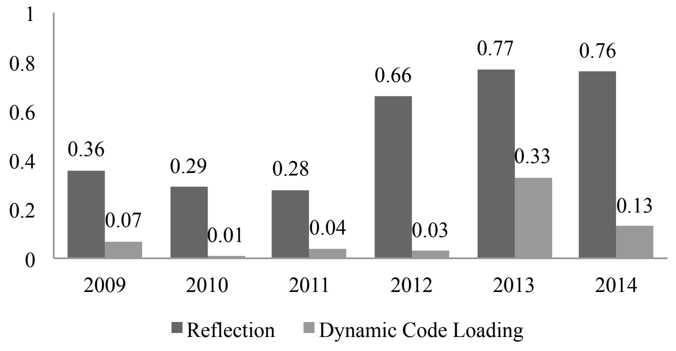


Figure 7.21: Trend in the use of reflection and dynamic class loading by piggybacked apps.

F18▶ *Piggybacking code densifies the overall app's call graph, while rider code can even largely exceed in size the carrier code.*

We compute the proportion of piggybacked app code which is actually brought by rider code. Fig. 7.22 (left) highlights that, for most apps (71%), the rider code is smaller than the carrier code. However, for 29% of the piggybacked apps in our dataset, the rider code is bigger in terms of LOCs than the carrier code.

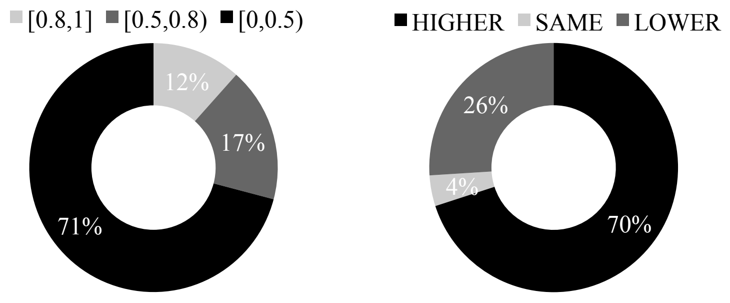


Figure 7.22: Impact of Piggybacking on rider code proportion (left) and CG density (right).

We further build the call graphs of piggybacked apps and compare them to the call graphs of their corresponding original apps. Then we compute the density metric[xii] using the number of real edges in the graph divided by the total number of possible edges. Distribution in Fig. 7.22 (right) highlights that in the large majority of cases, piggybacking code increases[xiii] the call graph density. This increase can be achieved by injecting complex rider code with higher density than the carrier code. Indeed, complex rider code is essential for piggybackers to make their code more difficult to understand by analysts. In other words, this finding is expected because the injected rider code will likely explore malicious scenarios that piggybackers wish to leave hidden to manual review and static analysis. On the other hand, in 1/4 cases the call graph density has lowered. We have sampled some piggybacked

[xii] We leverage the Toolkit of GraphStream (http://graphstream-project.org) to compute the density metric. The computation is implemented in a complexity of O(1).

[xiii] As for all comparisons, we remind the reader that we have checked that the difference was statistically significant.

apps in such cases, and found that most use $type_2$ hooks (i.e., via events). Only a few piggybacked apps kept the same density metric values.

F19▶ *Piggybacked app writers are seldom authors of benign apps.*

We investigate the certificates used to sign the apps in our dataset. We collected 703 distinct certificates in original apps against only 194 distinct certificates in piggybacked apps. We only found 14 certificates where each have been used to sign both a piggybacked and another app in the set of originals. Looking into those certificates, we found that it is inline with the well-known copy/paste problem. As an example, we have found that a certificate issued to Android Debug, which should only be used to develop and test an app, is applied to released apps. Several of the overlapping certificates were actually due to this issue.

F20▶ *Piggybacking code brings more execution paths where sensitive data can be leaked.*

Private data leaks in Android are currently a main focus in the research and practice community. We investigate the static behaviour of piggybacked apps w.r.t the data flows that they include using IccTA [124]. Fig. 7.23 show that piggybacked apps include, on median, 15 more flows from a sensitive source to a sensitive sink than original apps. We also found that these flows mostly represent data exchange across components. As an example, we have found 12 such leaks where Context information is collected, then transferred to another component which eventually forwards it outside the device via SMS.

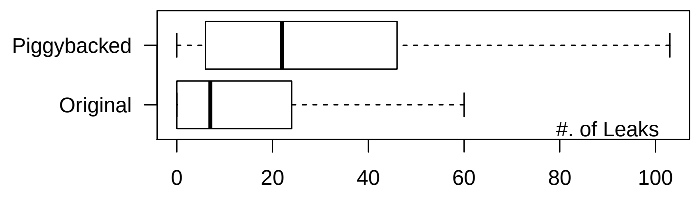


Figure 7.23: Distributions of the number of leaks.

7.4 Discussion

We now explore three potential outcomes of our findings (cf. Section 7.4.1). Then, we discuss two problematic assumptions that have been commonly made in the Android community (Section 7.4.2) and the potential threats to validity of this study (cf. Section 7.4.3).

7.4.1 Potential Outcomes

Understanding malicious Android apps (through piggybacked apps) can help in pushing further a number of research directions:

1. Practical detection of piggybacked apps: With this work, we hope to follow on the steps of the MalGenome project and drive new research on the automated detection of piggybacked apps. Indeed, by enumerating high-level characteristics of Android malware samples, MalGenome has opened several directions in the research on malware detection, most of which have either focused on detecting specific malware types (e.g., malware leaking private data [124]), or are exploiting app features, such as permissions requested, in Machine Learning classification [8]. Similarly, we expect the devise of machine learning-based detection approaches which can **effectively model piggybacking** and thus avoid the need for impractical and unscalable pairwise comparisons across app markets. Indeed, if we consider piggybacked detection as a classification problem, we can address the main limitation in state-of-the-art works which all require the original app in order to

search for potential piggybacked apps that use it as a carrier. In a similar research, researchers have already shown with Clonewise that it was possible to treat clone detection as a classification problem [41]. In preliminary experiments, using a feature set limited to basic features whose extraction was inexpensive (Permissions, Intents, rider API uses) we built classifiers that achieved acceptable performance in precision and recall (both over 70%) with 10-Fold cross validation on our dataset. More sophisticated and discriminative features can be explored from our findings. Indeed, as shown in our technique report [139], with a set of features including *new declared capabilities*, *duplicated permissions*, *diversity of packages*, etc., we are able to achieve a high performance of 97% for both precision and recall, suggesting the features built on our findings are suitable for discriminating piggybacked apps from non-piggybacked apps.

2. Explainable malware detection: Current Machine Learning classifiers [38,60,199,239] for Android malware detection are too generic to be relevant in the wild: features currently used in the literature, such as n-grams, permissions or system calls, allow to flag apps without providing any hint on which malicious actions are actually expected. A collection of piggybacked apps provides new opportunities for **retrieving a variety of malicious payloads** that can be investigated to infer fine-grained semantic features for malware detection.

3. Malicious code localization: In their fight against malware, practitioners are regularly challenged by the need to localize malicious code within an app, e.g., in order to remove/block it, or to characterize its impact. Our work can be leveraged towards **understanding how malicious code are hooked in benign app code**, as well as which code features may potentially help statically discriminate malicious parts from benign parts. In our preliminary experiments [137,139], based on the identified behaviors of piggybacked apps, we are able to automatically localize the hook code with an accuracy@5 (the hook code is within the top 5 packages we ranked based on their probabilities of being hook code) of 83%, without knowing the original counterparts of the dissected piggybacked apps.

7.4.2 Validity of Common Assumptions

Based on our study, we are able to put in perspective an assumption used in the literature for reducing the search space of piggybacking pairs. With FSquaDRA [252], Zhauniarovich et al. have proposed to rely on similarity of apps' resource files to detect piggybacking. Finding F1 of our work however supports that resource files can be extensively manipulated during piggybacking.

We further plot in Figure 7.24 the distribution of pairwise similarity for the piggybacking pairs of our dataset to highlight the significant difference between relying on resource-based similarity and code-based similarity.

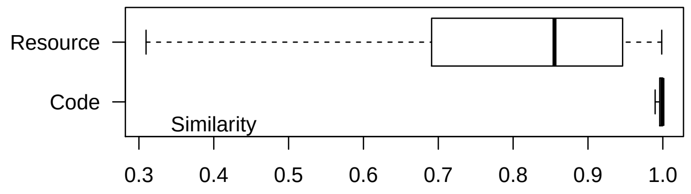


Figure 7.24: Similarity Distribution.

Another common assumption in the literature is that library code is noisy for static analysis approaches who must parse a large portion of code which is not part of app core functionality, leading to false positives. Thus, recent research works [13,130] first heuristically prune known common libraries before applying their approaches on apps. We have shown however with our investigation that libraries, in particular popular libraries, are essential to malware writers who can use them to seamlessly insert hooks for executing malicious rider code. Thus, there is a special need for analysts who want to filter out common libraries from their analyses to ensure that the detached libraries will not introduce side-effects.

7.4.3 Threats to Validity

The main threat to validity of our study lies in the exhaustivity of our dataset. However, we have made our best effort to leverage the AndroZoo largest available research dataset of over 2 million Android apps [3] to search for piggybacked apps. Furthermore our apps span a long timeline of 6 years. Another threat to validity is that we do not include apps from the last year or so, since the collection of AV reports in AndroZoo is delayed. We will continuously update the dataset of piggybacked apps within the AndroZoo project [3].

With regards to the restrictions used in the identification of piggybacking pairs, we would like to note for the readers that our focus in this work is not to exhaustively detect all piggybacking pairs. Instead, we aimed for collecting a sufficient number of obvious, thus accurate, piggybacking pairs in order to be able to dissect and understand piggybacking processes.

The original apps in our ground truth may not be the final original apps, i.e., they may also be repackaged versions of a previous app. To the best of our knowledge, there is no straightforward way to pinpoint whether a given original app from a piggybacking pair is the true original app. Therefore, it remains an interesting future work to the community.

AV labels from VirusTotal engines may not be perfect because of AV disagreements [107]. However, in this work, we only use them to get quick insights. Finally, by construction, our dataset of piggybacked apps has been built with a constraint on the maliciousness (in the sense of AV detection) of the piggybacked app by a different developer. Other "piggybacking" operations may be performed by the same developer with "benign" code.

Recently, there are many Android app packers (e.g., Bangcle, Ijiami, etc.) introduced for comprehensive protection of Android apps, attempting to prevent Android apps from being reverse engineered and repackaged/piggybacked. Intuitively, it becomes much harder to piggyback packed apps. Unfortunately, state-of-the-art approaches including AppSpear [236] and DexHunter [249] have already demonstrated promising results for extracting DEX code from packed apps, making it still possible to piggyback even packed apps.

7.5 Related Work

Dissection studies: In recent years there have been a number of studies dissecting Android malware [68, 72, 257]. Genome [257] provides an interesting overview on the landscape of Android malware based on samples collected from forums and via experiments. Our approach for piggybacked apps is however more systematic. Besides in Android community, dissection studies have also been widely performed in other communities. For example, Jakobsson et al. [103] have presented a comprehensive analysis on crimeware, attempting to understand current and emerging security threats including bot networks, click fraud, etc.

Repackaged/Cloned app detection: Although the scope of repackaged app detection is beyond simple code, researchers have proposed to rely on traditional code clone detection techniques to identify similar apps [53, 54, 61, 256]. With DNADroid [53], Crussell et al. presented a method based on program dependency graphs comparison. The authors later built on their DNADroid approach to build AnDarwin [54], an approach that uses multi-clustering to avoid the scalability issues induced by pairwise comparisons. DroidMOSS [256] leverages fuzzy hashing on applications' OpCodes to build a database of application fingerprints that can then be searched through pairwise similarity comparisons. Shahriar and Clincy [203] use frequencies of occurrence of various OpCodes to cluster Android malware into families. Finally, in [45], the authors use a geometry characteristic of dependency graphs to measure the similarity between methods in two apps, to then compute similarity score of two apps.

Instead of relying on code, other approaches build upon the similarity of app "metadata". For instance, Zhauniarovich et al. proposed FSquaDRA [252] to compute a measure of similarity on the apps' resource files. Similarly, ResDroid [204] uses application resources such as GUI description files to extract features that can then be clustered to detect similar apps. In their large-scale study, Viennot, Garcia, and Nieh [223] also used assets and resources to demonstrate the presence of large quantities of either rebranded or cloned applications in the official Google Play market.

Piggybacked app search and Malware variants detection: Cesare and Xiang [40] have proposed to use similarity on Control Flow Graphs to detect variants of known malware. Hu, Chiueh, and Shin [98] described SMIT, a scalable approach relying on pruning function Call Graphs of x86 malware to reduce the cost of computing graph distances. SMIT leverages a Vantage Point Tree but for large scale malware indexing and queries. Similarly, BitShred [104] focuses on large-scale malware triage analysis by using feature hashing techniques to dramatically reduce the dimensions in the constructed malware feature space. After reduction, pair-wise comparison is still necessary to infer similar malware families.

Recently, Meng et al. [169] present an automatic analysis framework, namely SMART, to comprehend, detect, and classify malicious Android apps. At first, SMART learns deterministic symbolic automaton (DSA) from known malware families. Then, it extracts semantic features from the summarized DSA to classify new malware. The authors have also presented a framework called Mystique to automatically generate malware variants [168], in a hope to prepare labeled malware for boosting the evaluation of existing malware detection approaches.

PiggyApp [255] is the work that improves over their previous work, namely DroidMoss, to deal with repackaged app detection. PiggyApp is based on the assumption that a piece of code added to an already existing app will be loosely coupled with rest of the application's code. Consequently, given an app, they build its program dependency graph, and assigns weights to the edges in accordance to the degree of relationship between the packages. Then using an agglomerative algorithm to cluster the packages and select primary modules. To find piggybacked apps, they perform comparison between primary modules of apps. To escape the scalability problem with pair-wise comparisons, they rely on the Vantage Point Tree data structure to partition the metric space.

7.6 Conclusion

We have investigated Android piggybacked apps to provide the research community with a comprehensive characterization of piggybacking. We then build on our findings to put in perspective some common assumptions in the literature. We expect this study to initiate various research directions for practical and scalable piggybacked app detection, explainable malware detection, malicious code localization, and so on.

8 Conclusions and Future Work

In this chapter, we revisit the main contributions of this dissertation and present potential future research directions.

Contents

8.1 Conclusions

In this work, we presented techniques for boosting static security analysis of Android apps based on code instrumentation. We select code instrumentation because it is able to 1) provide generic solutions where the instrumented code can benefit any existing up-front static analyzers and 2) support context-sensitive analysis since the disconnected parts can be bridged via glue code. More specifically, we scheduled this dissertation in three parts: 1) Systematic literature review on static analysis of Android apps; 2) Code instrumentation for static security analysis; and 3) Thorough understanding on Android malicious behaviors. We now detail them.

Regarding the first part, our objective is to provide a clear view on the state-of-the-art works that secure Android apps through static analysis and thereby to highlight the trends, pinpoint the main focus of static analysis approaches, and enumerate the key aspects of future directions. We thus performed an SLR that surveyed five dimensions of static analyses: problems targeted, fundamental techniques leveraged, static analysis sensitivities considered, Android characteristics taken into account, and the scalability of evaluation performed. The results of review show that the research community is still facing a number of challenges for building approaches that are aware altogether of implicit flows, dynamic code loading features, reflective calls, native code and multi-threading features, in order to implement sound and highly precise static analysis approaches.

For the second part, we presented three different approaches to support static security analysis, where all of them have leveraged code instrumentation to mitigate static analysis challenges. In particular, we first proposed an approach called IccTA, which performs ICC-aware static taint analysis for detecting inter-component privacy leaks. The code instrumentation technique is applied to reduce the challenge of inter-component communication, where the ICC mechanism is simulated with injected glue code. Similarly, following the idea of IccTA, we proposed another approach called DroidRA, which represents reflective calls with traditional Java calls. As a result, the ICC calls and reflective calls no longer exist for static analysis. Thus, any existing static analyzer, even if it does not support ICC and reflection, can yield ICC-aware and reflection-aware results. Finally, based on our experiments, we found that common libraries are very noisy for static security analysis approaches, as what we shown in this dissertation, both piggybacking detection and ML-based malware detection could be doomed by the existence of common libraries. We come to an instrumentation-based approach which detaches common libraries from a given app before sending them to the final analysis. Our experimental results demonstrated that detaching common libraries can indeed improve the accuracy of other approaches.

As a final part, we conducted an in-depth dissecting on piggybacked apps, where we are able to identify and un-graft their malicious payloads, to have a better understanding on the malicious behaviors of Android apps. Our dissection is an attempt to go one step further by not only detecting malicious apps but also being able to explain why a given app is malicious.

8.2 Future Work

We now summarize potential future directions that are in line with this dissertation.

- **Supporting Code Instrumentation with Framework.** The code instrumentation process in this dissertation are actually conducted separately, making a lot of efforts that could be reused wasted. Therefore, as future work, we would like to implement a generic framework to support code instrumentation. This framework can further be integrated into Apkpler, a plugin-based system for reusing instrumentation-based solutions (e.g., making IccTA available for other approaches).

- **Considering more Challenging Artifacts for Instrumentation.** As it is hard for a single tool to efficiently address all of the various challenges of Android programming that make analysis difficult, we propose in this dissertation instrumentation-based approaches to manipulate the app code for reducing the analysis complexity, e.g., transforming a hard problem to a easy-resolvable one. In this dissertation, we have already taken into account the ICC and reflection challenges. The experimental results have demonstrated that code instrumentation is promising for supporting static analysis. However, there are many other artifacts such as native code, multiple threads in Android apps. In our future work, we would like to apply code instrumentation to reducing these challenging artifacts as well, in order to facilitate static security analysis.
- **Predicting Piggybacked Apps through Semantic Features.** Our thorough investigation into the Android piggybacking behaviors has led to many promising findings. Our previous work, which applies only simple features built from a small number of findings, has presented promising results for predicting piggybacked apps. By considering the whole set of findings, the prediction results could be further improved. We thus plan to put more efforts into this direction in an attempt to build semantic features for ML-based piggybacked apps detection.

List of papers, tools & services

Papers included in this dissertation:

- Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static Analysis of Android Apps: A Systematic Literature Review. *Technical Report, ISBN 978-2-87971-150-8 TR-SNT-2016-3*, 2016
- Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015
- Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. *arXiv preprint arXiv:1404.7431*, 2014
- Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *Proceedings of the 30th International Information Security and Privacy Conference (IFIP SEC)*, 2015
- Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. DroidRA: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016
- Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Tool Demonstration Track*, 2016
- Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016
- Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics & Security (TIFS)*, 2017
- Li Li, Daoyuan Li, Tegawendé F Bissyandé, David Lo, Jacques Klein, and Yves Le Traon. Ungrafting Malicious Code from Piggybacked Android Apps. *Technique Report*, 2016

Papers not included in this dissertation:

- Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME)*, 2016
- Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Parameter Values of Android APIs: A Preliminary Study on 100,000 Apps. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016

- Li Li. Boosting static analysis of android apps through code instrumentation. In *Proceedings of the 38th International Conference on Software Engineering, Doctoral Sysposium (ICSE-DS)*, 2016
- Damien Octeau, Somesh Jha, Matthew Dering, Patrick Mcdaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43th Symposium on Principles of Programming Languages (POPL)*, 2016
- Li Li, Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Mining families of android applications for extractive spl adoption. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC)*, 2016
- Li Li, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Towards a generic framework for automating extensive analysis of android applications. In *Proceedings of the 31st ACM/SIGAPP Symposium on Applied Computing (SAC)*, 2016
- Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. Automatically Locating Malicious Packages in Piggybacked Android Apps. *Technique Report*, 2016
- Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Potential Component Leaks in Android Apps: An Investigation into a new Feature Set for Malware Detection. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2015
- Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. A study of potential component leaks in android apps. *Technical Report*, 2015
- Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Using an instrumentation based approach to detect inter-component leaks in android apps. In *Grande Region Security and Reliability Day*, 2015
- Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Using a path matching algorithm to detect inter-component leaks in android apps. In *Grande Region Security and Reliability Day*, 2014
- Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Detecting privacy leaks in android apps. *International Symposium on Engineering Secure Software and Systems - Doctoral Symposium (ESSoS-DS2014)*, 2014
- Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2014

Tools and Datasets:

- **IccTA** - `https://github.com/lilicoding/soot-infoflow-android-iccta`
- **DroidRA** - `https://github.com/lilicoding/DroidRA`
- **Common Libraries** - `https://github.com/lilicoding/CommonLibraries`
- **Piggybacked Benchmark Apps** - `https://github.com/lilicoding/Piggybacking`
- **Static Analysis of Android Apps Repository** - `https://github.com/lilicoding/SA3Repo`

Services (Reviewer):

- IEEE Transactions on Information Forensics and Security
- Journal of Computer Virology and Hacking Techniques
- Pervasive and Mobile Computing
- Frontiers of Information Technology & Electronic Engineering

Services (External Reviewer):

- SANER'17, CODASPY'17, ICSE'17, NDSS'17
- ICECCS'16, QRS'16, MobileSoft'16, ESSoS'16, CODASPY'16
- COMPSAC'15, ICSE-NIER'15, ICSE'14
- IEEE Transactions on Software Engineering
- Information and Software Technology
- ACM Computing Surveys

Bibliography

[1] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. Sok: Lessons learned from android security research for appified software platforms. In *IEEE S&P*, 2016.

[2] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 83–93. ACM, 2015.

[3] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *The 13th International Conference on Mining Software Repositories, Data Showcase track*, 2016.

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[5] Lars Ole Andersen. Program analysis and specialization for the c programming language. *PhD Thesis at the University of Cophenhagen*, 1994.

[6] Nicoló Andronio, Stefano Zanero, and Federico Maggi. Heldroid: Dissecting and detecting mobile ransomware. In *Research in Attacks, Intrusions, and Defenses.* 2015.

[7] Axelle Apvrille and Ruchna Nigam. Obfuscation in android malware, and how to fight back. *Virus Bulletin*, 2014.

[8] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[9] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Instrumenting android and java applications as easy as abc. In *Runtime Verification*, 2013.

[10] Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014)*, 2014.

[11] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. Using targeted symbolic execution for reducing false-positives in dataflow analysis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 1–6. ACM, 2015.

[12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[13] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2015.

[14] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *ACM SIGPLAN Notices*, 2013.

[15] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM, 2016.

[16] Michael Backes, Sven Bugiel, Erik Derr, Sebastian Gerling, and Christian Hammer. R-droid: Leveraging android app analysis with static slice optimization. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 129–140. ACM, 2016.

[17] Michael Backes, Sven Bugiel, Erik Derr, and Christian Hammer. Taking android app vetting to the next level with path-sensitive value analysis. 2014.

[18] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard: enforcing user requirements on android apps. In *Proceedings of the 19th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2013.

[19] David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. *ACM Sigplan Notices*, 1996.

[20] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. Covert: Compositional analysis of android inter-app permission leakage. *Technical Report*, 2015.

[21] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Armorim, and Michael D. Ernst. Static analysis of implicit control flow:resolving java reflection and android intents. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2015.

[22] Alexandre Bartel, Jacques Klein, Martin Monperrus, Kevin Allix, and Yves Le Traon. Improving privacy on android smartphones through in-vivo bytecode instrumentation. *Technical Report*, 2012.

[23] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference On Automated Software Engineering*, 2012.

[24] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *SOAP*, 2012.

[25] Alexandre Bartel, John Klein, Martin Monperrus, and Yves Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *Software Engineering, IEEE Transactions on*, 2014.

[26] Steffen Bartsch, Bernhard Berger, Michaela Bunke, and Karsten Sohr. The transitivity-of-trust problem in android application interaction. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, 2013.

[27] Osbert Bastani, Saswat Anand, and Alex Aiken. Interactively verifying absence of explicit information flows in android apps. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[28] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, 2011.

[29] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of api change-and fault-proneness on the user ratings of android apps. *Software Engineering, IEEE Transactions on*, 2015.

[30] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[31] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: Precise refutations for heap reachability. In *ACM SIGPLAN Notices*, 2013.

[32] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[33] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, 2012.

[34] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[35] Theodore Book and Dan S Wallach. A case of collusion: A study of the interface between ad libraries and their apps. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, 2013.

[36] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in java. *ACM SIGPLAN Notices*, 1999.

[37] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 2002.

[38] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. A classifier of malicious android applications. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, 2013.

[39] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[40] Silvio Cesare and Yang Xiang. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing - Volume 107*, 2010.

[41] Silvio Cesare, Yang Xiang, and Jun Zhang. Clonewise: Detecting package-level clones using machine learning. In *Security and Privacy in Communication Networks*, 2013.

[42] Patrick P. F. Chan, Lucas C. K. Hui, and S. M. Yiu. DroidChecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[43] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 2002.

[44] Chia-Mei Chen, Je-Ming Lin, and Gu-Hsin Lai. Detecting mobile application malicious behaviors based on data flow of source code. In *Trustworthy Systems and their Applications (TSA), 2014 International Conference on*, 2014.

[45] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[46] Kevin Zhijie Chen, Noah M Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *NDSS*, 2013.

[47] Xin Chen and Sencun Zhu. Droidjust: automated functionality-aware privacy leakage analysis for android applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2015.

[48] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011.

[49] Kwanghoon Choi and Byeong-Mo Chang. A type and effect system for activation flow of components in android programs. *Information Processing Letters*, 2014.

[50] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, 2011.

[51] Luis Corral, Anton B Georgiev, Alberto Sillitti, Giancarlo Succi, and Tihomir Vachkov. Analysis of offloading as an approach for energy-aware applications on android os: A case study on image processing. In *International Conference on Mobile Web and Information Systems*, 2014.

[52] Agostino Cortesi, Pietro Ferrara, Marco Pistoia, and Omer Tripp. Datacentric semantics for verification of privacy policy compliance by mobile applications. In *Verification, Model Checking, and Abstract Interpretation*, 2015.

[53] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security–ESORICS 2012*, 2012.

[54] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of semantically similar android applications. In *Computer Security–ESORICS 2013*, 2013.

[55] Xingmin Cui, Jingxuan Wang, Lucas CK Hui, Zhongwei Xie, Tian Zeng, and SM Yiu. Wechecker: efficient and precise detection of privilege escalation vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2015.

[56] Julius Davies, Daniel M German, Michael W Godfrey, and Abram Hindle. Software bertillonage: Determining the provenance of software development artifacts. *Empirical Software Engineering*, 2013.

[57] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *Mobile Security Technologies*, 2012.

[58] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, 1995.

[59] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A Gunter. Free for all! assessing user data exposure to advertising libraries on android. 2016.

[60] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[61] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, 2014.

[62] Anthony Desnos. Android: Static analysis using similarity distance. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, 2012.

[63] Anthony Desnos and Geoffroy Gueguen. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, 2011.

[64] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[65] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. In *The Network and Distributed System Security Symposium (NDSS 2011)*, 2011.

[66] Karim O Elish, Danfeng Yao, and Barbara G Ryder. On the need of precise inter-app icc classification for detecting android malware collusions. In *Most@S&P*, 2015.

[67] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[68] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, 2011.

[69] Michael D Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, et al. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[70] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[71] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Gaur, Marco Conti, and Raj Muttukrishnan. Android security: A survey of issues, malware penetration and defenses. *IEEE Communications Surveys & Tutorials*, 2015.

[72] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011.

[73] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[74] Ira R Forman, Nate Forman, and John Vlissides Ibm. Java reflection in action. *Technical Report*, 2004.

[75] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.

[76] Daniele Gallingani, Rigel Gjomemo, VN Venkatakrishnan, and Stefano Zanero. Static detection and automatic exploitation of intent message vulnerabilities in android applications. *Technical Report*, 2015.

[77] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, 2013.

[78] Dimitris Geneiatakis, Igor Nai Fovino, Ioannis Kounelis, and Paquale Stirparo. A permission verification approach for android mobile applications. *Computers & Security*, 2015.

[79] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *TRUST*, 2012.

[80] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, 2013.

[81] Michael I Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of android applications in droidsafe. In *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

[82] Mariem Graa, Nora Cuppens Boulahia, Frédéric Cuppens, and Ana Cavalliy. Protection against code obfuscation attacks based on control dependencies in android systems. In *Software Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on*, 2014.

[83] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, and Ana Cavalli. Detecting control flow in smarphones: Combining static and dynamic analyses. In *Cyberspace Safety and Security*, 2012.

[84] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[85] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[86] Jiaping Gui, Stuart Mcilroy, Meiyappan Nagappan, and William GJ Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2015.

[87] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. Characterizing and detecting resource leaks in android applications. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, 2013.

[88] Zhihui Han, Liang Cheng, Yang Zhang, Shuke Zeng, Yi Deng, and Xiaoshan Sun. Systematic analysis and detection of misconfiguration vulnerabilities in android smartphones. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, 2014.

[89] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2012.

[90] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Software Engineering (ICSE), 2013 35th International Conference on*, 2013.

[91] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Sif: a selective instrumentation framework for mobile applications. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services (MobiSys)*, 2013.

[92] Muhammad Haris, Hamed Haddadi, and Pan Hui. Privacy leakage in mobile computing: Tools, methods, and characteristics. *arXiv preprint arXiv:1410.4978*, 2014.

[93] Behnaz Hassanshahi, Yaoqi Jia, Roland HC Yap, Prateek Saxena, and Zhenkai Liang. Web-to-application injection attacks on android: Characterization and detection. In *Computer Security–ESORICS 2015*, 2015.

[94] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing droids: program slicing for smali code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013.

[95] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[96] Wenhui Hu, Damien Octeau, Patrick McDaniel, and Peng Liu. Duet: Library integrity verification for android applications. In *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2014.

[97] Wenjun Hu, Jing Tao, Xiaobo Ma, Wenyu Zhou, Shuang Zhao, and Ting Han. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, 2014.

[98] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[99] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 977–992, 2015.

[100] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[101] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015.

[102] Reyhaneh Jabbarvand, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul Ammann. Ecodroid: an approach for energy-based ranking of android apps. In *Proceedings of the Fourth International Workshop on Green and Sustainable Software*, 2015.

[103] Markus Jakobsson and Zulfikar Ramzan. *Crimeware: understanding new attacks and defenses.* Addison-Wesley Professional, 2008.

[104] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[105] Casper S Jensen, Mukul R Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013.

[106] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

[107] Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Brad Miller, Vaishaal Shankar, Rekha Bachwani, Anthony D. Joseph, and J. D. Tygar. Better malware ground truth: Techniques for weighting anti-virus vendor labels. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, 2015.

[108] Mikhail Kazdagli, Ling Huang, Vijay Reddi, and Mohit Tiwari. Morpheus: Benchmarking computational diversity in mobile malware. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, 2014.

[109] Chon Ju Kim and Phyllis Frankl. Aqua: Android query analyzer. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, 2012.

[110] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 2012.

[111] Barbara Kitchenham. Procedures for performing systematic reviews. *Technical Report*, 2004.

[112] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, 2014.

[113] Eugene Kuleshov. Using the asm framework to implement common java bytecode transformation patterns. *Aspect-Oriented Software Development*, 2007.

[114] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. A survey on security for mobile devices. *Communications Surveys & Tutorials, IEEE*, 2013.

[115] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[116] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *Compiler Construction*, 2003.

[117] Ding Li, Shuai Hao, William GJ Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013.

[118] Ding Li, Yingjun Lyu, Mian Wan, and William GJ Halfond. String analysis for java and android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[119] Ding Li, Angelica Huyen Tran, and William GJ Halfond. Making web applications more energy efficient for oled smartphones. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[120] Li Li. Boosting static analysis of android apps through code instrumentation. In *Proceedings of the 38th International Conference on Software Engineering, Doctoral Sysposium (ICSE-DS)*, 2016.

[121] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Potential Component Leaks in Android Apps: An Investigation into a new Feature Set for Malware Detection. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2015.

[122] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. A study of potential component leaks in android apps. *Technical Report*, 2015.

[123] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *Proceedings of the 30th International Information Security and Privacy Conference (IFIP SEC)*, 2015.

[124] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015.

[125] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2014.

[126] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Detecting privacy leaks in android apps. *International Symposium on Engineering Secure Software and Systems - Doctoral Symposium (ESSoS-DS2014)*, 2014.

[127] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Using a path matching algorithm to detect inter-component leaks in android apps. In *Grande Region Security and Reliability Day*, 2014.

[128] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. *arXiv preprint arXiv:1404.7431*, 2014.

[129] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Using an instrumentation based approach to detect inter-component leaks in android apps. In *Grande Region Security and Reliability Day*, 2015.

[130] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[131] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Parameter Values of Android APIs: A Preliminary Study on 100,000 Apps. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[132] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

[133] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. DroidRA: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016.

[134] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Tool Demonstration Track*, 2016.

[135] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static Analysis of Android Apps: A Systematic Literature Review. *Technical Report, ISBN 978-2-87971-150-8 TR-SNT-2016-3*, 2016.

[136] Li Li, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Towards a generic framework for automating extensive analysis of android applications. In *Proceedings of the 31st ACM/SIGAPP Symposium on Applied Computing (SAC)*, 2016.

[137] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. Automatically Locating Malicious Packages in Piggybacked Android Apps. *Technique Report*, 2016.

[138] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics & Security (TIFS)*, 2017.

[139] Li Li, Daoyuan Li, Tegawendé F Bissyandé, David Lo, Jacques Klein, and Yves Le Traon. Ungrafting Malicious Code from Piggybacked Android Apps. *Technique Report*, 2016.

[140] Li Li, Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Mining families of android applications for extractive spl adoption. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC)*, 2016.

[141] Chieh-Jan Mike Liang, Nicholas D Lane, Niels Brouwers, Li Zhang, Börje F Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, et al. Caiipa: automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 519–530. ACM, 2014.

[142] Shuying Liang, Andrew W Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, 2013.

[143] Max Lillack, Christian Kästner, and Eric Bodden. Tracking load-time configuration options. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014.

[144] Jialiu Lin, Bin Lin, Norman Sadeh, and Jason Hong. Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings. In *Symposium on Usable Privacy and Security (SOUPS)*, 2014.

[145] Yu Lin, Semih Okur, and Danny Dig. Study and refactoring of android asynchronous programming. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, 2015.

[146] Yu Lin, Cosmin Radoi, and Danny Dig. Retrofitting concurrency for android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[147] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *FSE*, 2013.

[148] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014.

[149] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *ICPC*, 2014.

[150] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014.

[151] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. Andradar: fast discovery of android applications in alternative markets. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2014.

[152] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[153] Benjamin Livshits, John Whaley, and Monica S Lam. Reflection analysis for java. In *Programming Languages and Systems*, 2005.

[154] V Benjamin Livshits and Monica S Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. *ACM SIGSOFT Software Engineering Notes*, 28(5):317–326, 2003.

[155] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *Usenix Security*, 2005.

[156] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a certifying app store for android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, 2014.

[157] Kangjie Lu, Zhichun Li, Vasileios P Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *NDSS*, 2015.

[158] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[159] Zheng Lu and Supratik Mukhopadhyay. Model-based static source code analysis of java programs with applications to android security. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, 2012.

[160] Zhang Luoshi, Niu Yan, Wu Xiao, Wang Zhaoguo, and Xue Yibo. A3: Automatic analysis of android malware. In *1st International Workshop on Cloud Computing and Information Security*, 2013.

[161] Siyuan Ma, Zhushou Tang, Qiuyu Xiao, Jiafa Liu, Tran Triet Duong, Xiaodong Lin, and Haojin Zhu. Detecting gps information leakage in android applications. In *Global Communications Conference (GLOBECOM), 2013 IEEE*, 2013.

[162] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 653–656. ACM, 2016.

[163] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[164] Sam Malek, Hamid Bagheri, and Alireza Sadeghi. Automated detection and mitigation of inter-application security vulnerabilities in android (invited talk). In *International Workshop on Software Development Lifecycle for Mobile (DeMobile)*, 2014.

[165] Christopher Mann and Artem Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 2012.

[166] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 1947.

[167] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *RN*, 2016.

[168] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security*, pages 365–376. ACM, 2016.

[169] Guozhu Meng, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan. Semantic modelling of android malware for effective malware comprehension, detection, and classification. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 306–317. ACM, 2016.

[170] Kristopher Micinski, Jonathan Fetter-Degges, Jinseong Jeon, Jeffrey S Foster, and Michael R Clarkson. Checking interaction-based declassification policies for android using symbolic execution. In *Computer Security–ESORICS 2015*, 2015.

[171] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. Sig-droid: Automated system input generation for android applications. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, 2015.

[172] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, Theodore Berger, Steffen Dienst, and Ahmed E Hassan. Impact of ad libraries on ratings of android mobile apps. *Software, IEEE*, 2014.

[173] Tood K Moon. The expectation-maximization algorithm. *Signal processing magazine, IEEE*, 13(6):47–60, 1996.

[174] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. *ICST'16*, 2016.

[175] Mayur Naik and Jens Palsberg. A type system equivalent to a model checker. In *Programming Languages and Systems*, 2005.

[176] Arun Narayanan, Lihui Chen, and Chee Keong Chan. Addetect: Automated detection of android ad libraries using semantic analysis. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*, 2014.

[177] Damien Octeau, William Enck, and Patrick McDaniel. The ded decompiler. *Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Tech. Rep. NAS-TR-0140-2010*, 2010.

[178] Damien Octeau, Somesh Jha, Matthew Dering, Patrick Mcdaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43th Symposium on Principles of Programming Languages (POPL)*, 2016.

[179] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *FSE*, 2012.

[180] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[181] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[182] Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel P Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 2012.

[183] Étienne Payet and Fausto Spoto. Static analysis of android programs. *Information and Software Technology*, 2012.

[184] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.

[185] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[186] Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang. Plagiarizing smartphone applications: attack strategies and defense techniques. In *International Symposium on Engineering Secure Software and Systems*, pages 106–120. Springer, 2012.

[187] Bahman Rashidi and Carol Fung. A survey of android security threats and defenses. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 2015.

[188] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *The 2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[189] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. Droidforce: enforcing complex, data-centric, system-wide policies in android. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, 2014.

[190] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013.

[191] Vaibhav Rastogi, Zhengyang Qu, Jedidiah McClurg, Yinzhi Cao, and Yan Chen. Uranine: Real-time privacy leakage monitoring without system modification for android. In *Security and Privacy in Communication Networks*, 2015.

[192] Tristan Ravitch, E Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, 2014.

[193] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[194] Bruno PS Rocha, Marco Conti, Sandro Etalle, and Bruno Crispo. Hybrid static-runtime information flow and declassification enforcement. *Information Forensics and Security, IEEE Transactions on*, 2013.

[195] Atanas Rountev and Dacong Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[196] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. Understanding reuse in the android market. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, 2012.

[197] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android apps. *Technical Report*, 2016.

[198] Gholamreza Safi, Arman Shahbazian, WG Halfond, and Nenad Medvidovic. Detecting event anomalies in event-based systems. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015.

[199] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *Intelligence and Security Informatics Conference (EISIC), 2012 European*, 2012.

[200] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *10th International Conference on Security and Cryptography (SECRYPT)*, 2013.

[201] Dragos Sbîrlea, Michael G Burke, Salvatore Guarnieri, Marco Pistoia, and Vivek Sarkar. Automatic detection of inter-application permission leaks in android applications. *IBM Journal of Research and Development*, 2013.

[202] Julian Schutte, Dennis Titze, and JM De Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, 2014.

[203] H. Shahriar and V. Clincy. Detection of repackaged android malware. In *Internet Technology and Secured Transactions (ICITST), 2014 9th International Conference for*, 2014.

[204] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[205] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y Ko, and Lukasz Ziarek. Information flows as a permission mechanism. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014.

[206] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*, 2014.

[207] David Sounthiraraj, Justin Sahs, Garrett Greenwood, Zhiqiang Lin, and Latifur Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2014.

[208] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1996.

[209] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*, 2012.

[210] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 2001.

[211] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. Evolution, detection and analysis of malware for smart devices. *Communications Surveys & Tutorials, IEEE*, 2014.

[212] Sufatrio, Tong-Wei Chua, Darell JJ Tan, and Vrizlynn LL Thing. Accurate specification for robust detection of malicious behavior in mobile environments. In *Computer Security–ESORICS 2015*, 2015.

[213] Darell JJ Sufatrio, Tan, Tong-Wei Chua, and Vrizlynn LL Thing. Securing android: A survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR)*, 2015.

[214] Xin Sun, Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Detecting code reuse in android applications using component-based control flow graph. In *ICT Systems Security and Privacy Protection*, 2014.

[215] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, 2000.

[216] Dennis Titze and Julian Schütte. Apparecium: Revealing data flows in android applications. In *Proceedings of the 29th International Conference on Advanced Information Networking and Applications (AINA)*, 2015.

[217] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, Salvatore Guarnieri, et al. Andromeda: Accurate and scalable security analysis of web applications. In *FASE-International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software-2013*, 2013.

[218] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. In *ACM Sigplan Notices*, 2009.

[219] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. *Technical Report*, 1998.

[220] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. Towards verifying android apps for the absence of no-sleep energy bugs. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, 2012.

[221] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014.

[222] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, 2014.

[223] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. *SIGMETRICS Perform. Eval. Rev.*, 2014.

[224] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: a scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015.

[225] Jingtian Wang, Xiaoquan Wu, Jun Wei, et al. Detect and optimize the energy consumption of mobile app through static analysis: an initial research. In *Proceedings of the Fourth Asia-Pacific Symposium on Internetware*, 2012.

[226] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *CCS*, 2014.

[227] Erik Ramsgaard Wognsen, Henrik Søndberg Karlsen, Mads Chr Olesen, and René Rydhof Hansen. Formalisation and analysis of dalvik bytecode. *Science of Computer Programming*, 2014.

[228] Jianliang Wu, Tingting Cui, Tao Ban, Shanqing Guo, and Lizhen Cui. Paddyfrog: systematically detecting confused deputy vulnerability in android applications. *Security and Communication Networks*, 2015.

[229] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[230] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, 2015.

[231] Jianlin Xu, Yifan Yu, Zhen Chen, Bin Cao, Wenyu Dong, Yu Guo, and Junwei Cao. Mobsafe: cloud computing based forensic analysis for massive mobile applications using data mining. *Tsinghua Science and Technology*, 2013.

[232] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium*, 2012.

[233] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *International Conference on Software Engineering (ICSE)*, 2015.

[234] Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Fundamental Approaches to Software Engineering*, 2013.

[235] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proc. of the International Conference on Software Engineering (ICSE)*, 2015.

[236] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 359–381. Springer, 2015.

[237] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *WCSE*, 2012.

[238] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[239] S.Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new android malware detection approach using bayesian classification. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, 2013.

[240] Wei You, Bin Liang, Jingzhe Li, Wenchang Shi, and Xiangyu Zhang. Android implicit information flow demystified. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 585–590. ACM, 2015.

[241] Le Yu, Tao Zhang, Xiapu Luo, and Lei Xue. Autoppg: Towards automatic generation of privacy policy for android applications. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015.

[242] Razieh Nokhbeh Zaeem, Mukul R Prasad, and Sarfraz Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 183–192. IEEE, 2014.

[243] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. Towards automatic generation of security-centric descriptions for android apps. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[244] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[245] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS*, 2014.

[246] Mu Zhang and Heng Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014.

[247] Xiao Zhang, Kailiang Ying, Yousra Aafer, Zhenshen Qiu, and Wenliang Du. Life after app uninstallation: Are the data still alive? data residue attacks on android. NDSS '16.

[248] Ying Zhang, Gang Huang, Xuanzhe Liu, Wei Zhang, Hong Mei, and Shunxiang Yang. Refactoring android java code for on-demand computation offloading. In *ACM SIGPLAN Notices*, 2012.

[249] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: toward extracting hidden code from packed android applications. In *European Symposium on Research in Computer Security*, pages 293–311. Springer, 2015.

[250] Zhibo Zhao and Fernando C Colon Osono. "trustdroid": Preventing the use of smartphones for information leaking in corporate networks through the used of static analysis taint tracking. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, 2012.

[251] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015.

[252] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. Fsquadra: Fast detection of repackaged applications. In *Data and Applications Security and Privacy XXVIII*, 2014.

[253] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

[254] Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Droidalarm: an all-sided static analysis tool for android privilege-escalation malware. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013.

[255] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, 2013.

[256] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, 2012.

[257] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[258] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[259] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

[260] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2015.

[261] Chaoshun Zuo, Jianliang Wu, and Shanqing Guo. Automatically detecting ssl error-handling vulnerabilities in hybrid mobile web apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015.