Mining AndroZoo: A Retrospect

Li Li
SnT, University of Luxembourg, Luxembourg
li.li@uni.lu
http://lilicoding.github.io

***Abstract*—This paper presents a retrospect of an Android app collection named AndroZoo and some research works conducted on top of the collection. AndroZoo is a growing collection of Android apps from various markets including the official Google Play. At the moment, over five million Android apps have been collected. Based on AndroZoo, we have explored several directions that mine Android apps for resolving various challenges. In this work, we summarize those resolved mining challenges in three research dimensions, including code analysis, app evolution analysis, malware analysis, and present in each dimension several case studies that experimentally demonstrate the usefulness of AndroZoo.**

I. Introduction

Mobile devices such as smart phones have permeated into out daily life, where now people complete many of their tasks through mobile devices such as to carry out social networking, buy bus/train tickets, navigate car routes, etc. Nowadays people spend a significant amount of time everyday on their mobile devices, making mobile ecosystem an important target to explore.

Among several mobile operating systems, Android is the most popular one that has almost dominated the market by grabbing nearly 90% of share. The popularity of Android-based smart phones also reflects on the number of Android apps, which by now has exceeded a record of 3 million apps on Google Play[1]. Such a large number of apps provide great resources and values for researchers and practitioners to mine.

State-of-the-art works have explored the direction of mining Android apps from different aspects [1], [2], [3]. As summarized by Martin et al. [4] in their recent survey on app store analysis, literature works have tackled seven aspects related to mining Android apps, which are API, Feature, Release, Reviews, Security, Store Ecosystem, and Prediction. In our previous systematic literature review (SLR) on static analysis of Android apps [5], we additionally identified four directions that literature works also mine Android apps for, including analyzing app clones, addressing energy problems, generating test cases, and verifying code correctness. The SLR also shows that security is the most targeted aspect for Android apps.

Our work is in line with the aforementioned literature works, where we attempt to mine Android apps for achieving various purposes including to secure Android apps. Since our research community does not provide a large dataset of Android apps by the time when we started our research, we start to explore the direction of mining Android apps by first building a large enough dataset of Android apps, which are named later as AndroZoo [6]. AndroZoo is a growing collection of Android apps from various sources, including the official Google Play store. By far, it has collected over five millions of Android apps that we share to the research community for encouraging our fellow researchers to engage in reproducible experiments.

[1]https://www.appbrain.com/stats/stats-index

Based on AndroZoo, we tackle three research dimensions on mining Android apps: (1) Code analysis: We mainly perform static code analysis to address Android security problems; (2) App evolution analysis: We present some case studies around the analysis of app versions. Concretely, we focus not only on app variants that are useful for evaluating extractive Software Product Line (SPL) adoption techniques, but also on app versions including both legitimate versions that belong to the lineage of an app and repackaged versions of apps which are now proliferating in third-party markets. (3) Malware analysis: We have explored a new type of vulnerability named potential component leak and have conducted several machine learning algorithms to detect Android malware.

II. Background

To help readers better understand this work, we now briefly introduce the different artifacts shipped with Android apps that could be potentially interesting to miners to achieve their mining objectives.

An Android app is basically a zip archive file, which includes several types of files that are put together during the app releasing phase. Table I enumerates the file structure of a given Android app and a basic explanation on the functionality of those files. The most important file are *AndroidManifest.xml* and *classes.dex*. We now detail these two files separately.

TABLE I: File Structure of an Android App.

Structured Files	Description
AndroidManifest.xml	App configuration such as declaration of 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

AndroidManifest.xml provides essential information about the app to the Android system so that the system knows how to execute the app code. Indeed, it contains several important attributes that could be interesting to app miners. We now summarize them as follows:

- **package.** The *package* attribute depicts the unique name of a given Android app, which thus is frequently used to uniquely name a given app.

- **sdkVersion.** The manifest leverages the *uses-sdk* element to express an app's compatibility with one or more versions of Android platform.
- **permission.** Permission is declared to allow the app to access some sensitive actions. For example, permission *READ_PHONE_STATE* allows the app to read the phone state information like device id.
- **component declaration.** Component is the basic unit forming an Android app. Normally, all the components should be declared in the manifest, except for rare cases in which components are registered dynamically (in app code).

classes.dex contains the compiled app code, which is originally written in Java. The actual implementation of app components declared in *AndroidManifest.xml* is located in *classes.dex*. As demonstrated in Fig. 1, there are four types of components: 1) *Activity*, which represents the visible part of Android apps; 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., phone rebooting); 4) *Content Provider*, which provides a standard interface for other components/apps to access structured data.

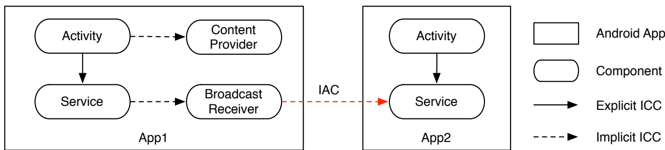

Fig. 1: Overview of Android Components.

Android components can communicate with each other, where the communication is referred by the literature as inter-component communication (ICC), e.g., from *Activity* to *Service* in Fig. 1. Actually, as shown in Fig. 1, there are two types of ICC: (1) explicit ICC, where the target component is explicitly set and (2) implicit ICC, where the target component is not explicitly set but is specified by some meta information such as Action, Category. For implicit ICC, the final communication is determined by the Android system based on the specified meta information. The Android system searches in the installed apps for all the possible target components, either within the same app or from other apps. If the target component is from a different app, the communication is then referred by the literature as inter-app communication (IAC), e.g., from *Broadcast Receiver* in App1 to *Service* in App2 in Fig. 1.

III. AndroZoo

AndroZoo is a growing collection of Android apps from several sources, including the official Google Play market, various alternative markets such as Anzhi and Slideme, and some open-source repositories like F-Droid. At the moment, AndroZoo contains more than five million apps. Fig. 2 plots the distribution of DEX size (i.e., size of *classes.dex*) of our collected AndroZoo apps. The DEX size ranges from several hundred kilobytes to over 10 megabytes, demonstrating that our dataset is highly diverse and thus can be leveraged to conduct various mining challenges. We have released our dataset to the research community, with Restful APIs to facilitate the access of Android apps. We hope our dataset can be well leveraged by the community to promote the field of mining Android apps, and also to encourage our fellow researchers to engage in reproducible experiments.

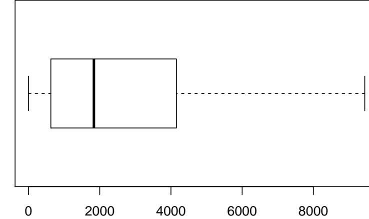

Fig. 2: Distribution of DEX Size of AndroZoo Apps (in KB).

In addition to Android apps, within AndroZoo, we also make available some pre-computed artifacts such as the signing certificate of Android apps, the malicious status of Android apps, etc, trying our best to boost the mining process for potential miners.

IV. Research Dimensions

Fig. 3 presents an overview of the research dimensions that we have explored so far based on the AndroZoo apps and their meta-data that we pre-compute for facilitating high level analyses. In a nutshell, our research on mining Android apps are mainly located in three dimensions: (1) Code analysis; (2) App evolution analysis; and (3) Malware analysis; We now detail those research dimensions respectively.

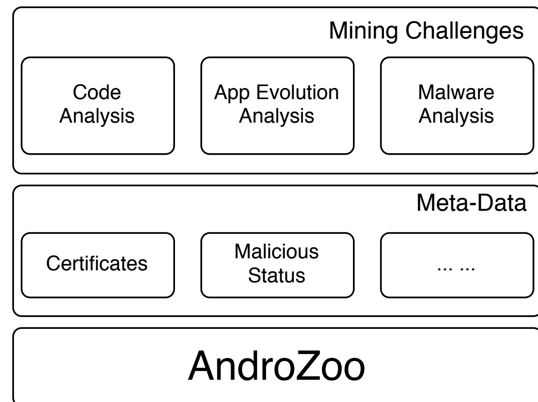

Fig. 3: Overview of Our Research Dimensions.

A. Code Analysis

ICC-Aware Analysis. Since Android apps can leak sensitive information carelessly or maliciously and malicious apps manipulate significantly more ICCs than benign apps, we propose a static analysis approach named IccTA to detect privacy leaks crossing Android components, and crossing apps

with the help of ApkCombiner [7]. Unlike state-of-the-art approaches, which mainly detect privacy leaks within single component, IccTA propogates context information among components to support inter-component communication (ICC) analysis [8], [9].

IccTA applies a code instrumentation based approach, i.e., change the code before analyzing, in order to make the ICC analysis reusable to existing intra-component analyzers. Listing 1 illustrates a code snippet showing an ICC-based privacy leak. The device id, considered as sensitive information, is obtained and stored into an Intent object (lines 8-10) in Activity1. Then, an ICC method *startActivity* is called, which switches the current execution from Activity1 to Activity2. Finally, in Activity2, the device id is retrieved and is eventually sent out of the device through *sendTextMessage* (lines 15-17).

```
 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 1: Example of an ICC Leak.

This privacy leak cannot be detected by intra-component analyzers such as FlowDroid [10], because the switching between Activity1 and Activity2 is unfortunately decided only by the system and it is non-trivial to obtain it directly at the code level [11], [12]. Therefore, in this work we present IccTA, a code instrumentation based approach, which modifies the code to be analyzed in a way that inter-component feature is mitigated. As an example, Listing 2 demonstates the modifications made by IccTA for the ICC leak example shown in Listing 1. The ICC method *startActivity* is replaced by a helper method that simulates the ICC through Java code, resulting in a simplified code snippet where ICC is no longer appearing. As a consequence, existing intra-component analyzers such as FlowDroid can now detect the privacy leak shown in Listing 1 without any modification.

Reflection-Aware Analysis. Like ICC we introduced previously, reflection is another challenge that usually causes static analyzers to yield false negatives. Android developers heavily use reflection in their apps for legitimate reasons such as providing genericity, maintaining backward compatibility, accessing inaccessible APIs [13], but also significantly for hiding malicious actions. Unfortunately, based on our SLR [5], most state-of-the-art works do not take into account the presence of reflective calls. Hence, we present DroidRA [14], [15],

```
 1   // modifications of Activity1
 2 - Activity1.this.startActivity(i);
 3 + IpcSC.redirect0(i);
 4   // creation of a helper class
 5 + class IpcSC {
 6 +  static void redirect0(Intent i) {
 7 +   Activity2 a2 = new Activity2(i);
 8 +   a2.dummyMain();
 9 +  }
10 + }
11   // modifications in Activity2
12 + public void dummyMain() {
13 +  // lifecycle and callbacks
14 +  // are called here
15 + }
16 + public Activity2(Intent i) {
17 +   this.intent_for_ipc = i; }
18 + public Intent getIntent() {
19 +  return this.intent_for_ipc; }
```

Listing 2: Code Instrumentation for *startActivity*.

a code instrumentation based approach, to tackle this issue in a non-invasive way. We first model the reflection analysis problem to a constant string propagation problem and then leverage the COAL solver [11] to infer the values of reflection-related targets. Finally, we instrument the app code to replace reflective calls by traditional Java calls where the separated parts due to reflection are now connected. Experimental results demonstrate that DroidRA is capable of supporting state-of-the-art static approaches to provide more sound and complete analyses.

Common Library Analysis. In a preliminary study, Wang et al. [16] have found that over 60% of Android apps's code is from common libraries, which may not be relevant to certain analyses such as repackaging analysis. Indeed, there are a number of researches that exclude libraries before performing their analyses [17], [18], [19]. Despite some efforts on investigating common libraries, the momentum of Android reserach has not yet produced a complete set of libraries that can be taken as a whitelist to support further analysis. Therefore, in this work, we leverage AndroZoo apps to perform a heuristic-based approach with several steps of refinements to harvest potential common libraries, for which we eventually collect 1,113 general libraries and 240 advertisement libraries. Furthermore, based on the collected libraries, we have performed several empirical studies that confirm our motivation: certain analyses such as repackaging analysis and malware detection should not take into account the library code, which constitute noise in app features, in order to produce accurate results.

B. App Evolution Analysis

App Variant Analysis. App variants (or family, e.g., the different products of a same company) usually embrace valuable information for evaluating extractive Software Product Line (SPL) adoption techniques, e.g., the reuse practices among app variants [20].

Fig. 4 presents a tree model to select app variants from a set of to-be mined Android apps, where each non-leaf node is represented by a package segment (e.g., baidu) while leaf node is represented through the remaining package segments

(e.g., BaiduMap). A branch from the root node (i.e., com) to a leaf node (e.g., BaiduMap) represents an unique package name (i.e., the Baidu Map). Because each Android app can have different versions, e.g., each update will result in a version, each leaf node has further been affiliated with a list of meta-data of app versions. The time line of the vertical axis shows that the affiliated list is ranked through times and the apps it contains are actually different versions of the app indicated by the leaf node. Given a time point, the tree model also gives a way to identify family variants. For example, as shown in the red dashed rectangle, given the latest time point, we are able to collect a set of variants for company com.baidu.

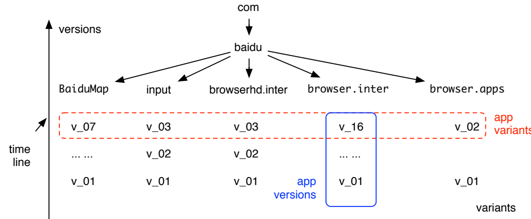

Fig. 4: A Simplified Example Showing how App Variants and Versions are Selected.

In this research direction, based on a clustering-based approach, we have collected in total 75,963 families of apps: The median number of variants in a family is three and 760 collected families have over 100 variants. Through a preliminary study on the collected app families, we have identified several reuse cases adopted by Android app variants, including library reuse, automated app generation, content-driven variants and device-driven variants [21]. With an advanced study, we believe that more reuse cases, including semantic reuses, can be identified on top of our collect app family variants.

Pairwise Similarity Analysis. We present a research framework called SimiDroid [22] that supports multi-level pairwise similarity comparison of Android apps, aiming at supporting the understanding of similarities or changes among app versions and among repackaged apps. SimiDroid is designed as a plugin-based framework that has already integrated various comparison methods such as code-based or resource-based comparisons. In addition to detect similar Android apps, we also perform a number of case studies on AndroZoo apps to demonstrate the suitability of SimiDroid in providing explanation hints for different usage scenarios. For example, we have leveraged SimiDroid to check and validate the hypothesis of multi-generation repackaging, where an original app identified in a repackaging pair is actually a repackaged app from a prior repackaging generation [23].

Piggybacking Behavior Understanding. Fig. 5 presents some basic terms related to Android app piggybacking. Basically, the working process of piggybacking is like this: Given an original Android app (referred to as carrier), attackers first unpack it and then modify its code by injecting some additional code (referred to as rider), and finally re-pack it back to a new app version (referred to as piggybacked app). The injected code will be triggered thanks to the so-called hooks, which connect the execution of carrier code to rider code.

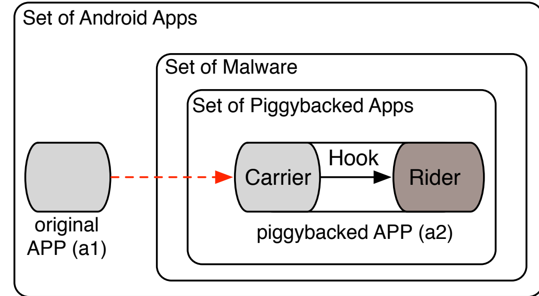

Fig. 5: Piggybacking Terminology.

Despite many research works have been conducted in the literature to detect piggybacked apps, the literature lacks a comprehensive study on the behavior of piggybacking. To this end, we construct a benchmark set of piggybacked app pairs (through pairwise similarity analysis) and investigate the characteristics of malicious piggybacked apps in comparison with their original counterparts. Thanks to these comparisons, we have eventually observed many interesting findings on the piggybacking process [24], [25]. The findings are summarized as follows:

1) The realization of malicious behavior is often accompanied by manipulation of app resource files.
2) Piggybacking changes app behavior mostly by tampering with carrier code.
3) Piggybacked apps are potentially built in batches.
4) Piggybacking often asks for new permissions to allow the realization of malicious behavior.
5) Piggybacking may recurrently request some specific permissions that are less requested by non-piggybacked apps.
6) Piggybacking is probably largely automated.
7) Piggybacking may overly request permissions that have been already declared by their original apps.
8) Piggybacking may introduce new user interfaces, implement new receivers and services, but will not add new database structures.
9) Piggybacking often consists in injecting a component that offers the same capabilities (i.e., Action, Category, etc.) as an existing component in the original app.
10) Piggybacking may change the launcher component so as to trigger the execution of rider code.
11) Piggybacking is often characterized by a naming mismatch between existing and newly injected components.
12) Piggybacking generally connects the malicious payloads to the benign carrier code via a single method call, making it possible to automatically locate grafted malicious payloads from piggybacked malicious apps.
13) Piggybacking hooks are generally placed within library code rather than in core app code.

14) Piggybacking often reuses the to-be injected malicious payloads.
15) Piggybacking adds code which performs sensitive actions, often without referring to device users.
16) Piggybacking operations are distributed over well-known malicious behavior types.
17) Piggybacking increasingly hides malicious actions via uses of reflection and dynamic class loading.
18) Piggybacking complicates app's overall call graph, while rider code can even largely exceed in size the carrier code.
19) Piggybacking are seldom conducted by authors of benign apps.
20) Piggybacking code brings more execution paths where sensitive data can be leaked.

C. Malware Analysis

Potential Component Leaks. Potential Component Leaks are such attacks that leak sensitive information through known ICC vulnerabilities such as Activity Hijacking Attack and Broadcast Injection Attack [26]. In this work, we have defined two types potential component leaks: 1) potential passive component leak, which leaks everything a component receives from other components; and 2) potential active component leak, which attempts to send sensitive information to other components. In order to detect the aforementioned attacks, we present a prototype tool named PCLeaks, which based on ICC vulnerabilities to perform data-flow analysis on Android apps to pinpoint potential component leaks that could potentially be exploited by other components (or apps) [27].

ML-based Malware Detection. As a follow-up work of PCLeaks, we find that potential component leaks are common in Android apps and that malicious apps have manipulated significantly more potential component leaks than benign apps. This evidence makes potential component leaks perfect candidates of features for machine learning (ML) based malware detection. Towards verifying this hypothesis, we take potential component leaks as features to train several classification models (with different settings such as different benign/malware ration, different ML algorithm, etc.) and perform 10-fold cross-validation to justify the ability of identifying malicious Android apps. Our experimental validations show high performance for identifying malware, demonstrating that potential component leaks are useful for discriminating malicious from benign apps [28].

Topic-Specific Data-Flow Analysis. State-of-the-art work has shown that both app descriptions [29] and sensitive data-flows [17] in standalone are capable of discriminating malicious from benign apps. In this work, we take both app descriptions (i.e., indicative of app topics) and sensitive data-flows (i.e., functionality implemented) into consideration for discriminating malware from benign apps. At beginning, we leverage adaptive LDA with GA, an advanced topic model, to cluster apps different categories based on their descriptions. Then, we use information gain ratio of sensitive data-flows to build so-called "topic-specific data-flow signatures". Finally, we leverage those signatures to characterize malicious Android apps. Our experiments on 3,691 benign and 1,612 malicious apps demonstrate that topic-specific data-flow signatures are useful and effective in highlighting malicious app behavior [30].

Malicious Payload Identification. As shown in Fig. 5, the malicious payloads of piggybacked apps are usually triggered by a small peace of code called hooks. If we are able to locate such hooks, we can accurately locate the malicious payloads, and thus reducing the examination space for security analysts to understand the malicious behavior. To this end, we propose in this work a tool-based approach called HookRanker [31], which provides rank lists of potential malicious packages based on the way malware behavior are revealed. With experiments on a ground truth of piggybacked Android apps, we demonstrate that HookRanker is helpful for locating malicious packages of piggybacked Android malware.

V. Conclusion and Future Work

In this paper, we have provided a retrospect on our works relating to mining Android apps, more specially on mining AndroZoo, our growing collection of Android apps. We have enumerated three research dimensions that we have explored in the field mining Android apps, including code analysis, app evolution analysis, and malware analysis.

Through almost five years of research in this field, we have learned several lessons: (1) constructing reliable dataset is the key to fulfill different mining purposes. (2) fundamental functionalities should be implemented in a reusable way, so that it does not need to be re-innovated.

Future Work: We plan to dig deeper into existing dimensions and also explore new dimensions in mining Android apps:

- We aim at enhancing AndroZoo by providing an even larger dataset (e.g., via crawling more alternative markets) and easing the access of interested AndroZoo apps (e.g., via pre-computing more meta-data of collected apps).
- We intent to summarize the common parts of conducting code analysis and thereby propose generic frameworks for promoting the reuse of the implementation of fundamental functionalities [32]. For example, we plan to design a Jimple-based Instrumentation Language (JIL) and a solver that interprets JIL descriptions and thus solves instrumentation problems in a generic way [33].
- We plan to explore the direction of mining app lineages, which provides a wealth of change information that can be leveraged in various research directions such as learning bug fix patterns or recommending new API usages.
- We plan to embrace Android testing techniques to examine Android apps dynamically. Since static and dynamic analysis are naturally complementary to each other, we plan to combine these two techniques together and thereby to propose hybrid approaches for resolving more advanced challenges in mining Android apps.

References

[1] Li Li. *Boosting Static Security Analysis of Android Apps through Code Instrumentation*. PhD thesis, University of Luxembourg, Luxembourg, 2016. http://orbilu.uni.lu/bitstream/10993/29387/1/thesis.pdf.

[2] 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*, pages 2–11. ACM, 2014.

[3] Md Yasser Karim, Huzefa Kagdi, and Massimiliano Di Penta. Mining android apps to recommend permissions. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 427–437. IEEE, 2016.

[4] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, 2016.

[5] 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. *Information and Software Technology*, 2017.

[6] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 468–471. ACM, 2016.

[7] 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 IFIP International Conference on ICT Systems Security and Privacy Protection (SEC 2015)*, 2015.

[8] 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.

[9] 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.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, 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. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[11] 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-Volume 1*, pages 77–88. IEEE Press, 2015.

[12] 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)*, 2016.

[13] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

[14] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *The 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016.

[15] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of android apps. In *The 31st IEEE/ACM International Conference on Automated Software Engineering, Demo Track (ASE 2016)*, 2016.

[16] 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*, pages 71–82. ACM, 2015.

[17] 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 37th International Conference on Software Engineering-Volume 1*, pages 426–436. IEEE Press, 2015.

[18] 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*, pages 431–444. ACM, 2013.

[19] 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*, pages 101–112. ACM, 2012.

[20] Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up technologies for reuse: automated extractive adoption of software product lines. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 67–70. IEEE Press, 2017.

[21] 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 *The 20th International Systems and Software Product Line Conference (SPLC 2016)*, 2016.

[22] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Simidroid: Identifying and explaining similarities in android apps. In *The 16th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom)*, 2017.

[23] Li Li, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. The multi-generation repackaging hypothesis. In *The 39th International Conference on Software Engineering, Poster Track (ICSE 2017)*, 2017.

[24] 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.

[25] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking. In *The 39th International Conference on Software Engineering, Poster Track (ICSE 2017)*, 2017.

[26] 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 *USENIX Security*, 2013.

[27] 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)*, 2014.

[28] 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 *The 2015 IEEE International Conference on Software Quality, Reliability & Security (QRS)*, 2015.

[29] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.

[30] Xinli Yang, David Lo, Li Li, Xin Xia, Tegawendé F Bissyandé, and Jacques Klein. Comprehending malicious android apps by mining topic-specific data flow signatures. *Information and Software Technology*, 2017.

[31] 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. In *The 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft 2017)*, 2017.

[32] 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 *The 31st ACM/SIGAPP Symposium on Applied Computing (SAC 2016)*, 2016.

[33] Li Li. Boosting static analysis of android apps through code instrumentation. In *The Doctoral Symposium of 38th International Conference on Software Engineering (ICSE-DS 2016)*, 2016.