UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2021-004
The Faculty of Sciences, Technology and Medicine

Dissertation

Defence held on the 7^{th} January 2021 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Jun Gao
Born on 16^{th} October 1982 in Shaanxi, China

Mining App Lineages: A Security Perspective

Dissertation Defense Committee

Dr. Jacques Klein, Dissertation Supervisor
Associate Professor, University of Luxembourg, Luxembourg

Dr. Tegawendé F. Bissyandé, Chairman
Assistant Professor, University of Luxembourg, Luxembourg

Dr. Li Li, Vice Chairman
Assistant Professor, Monash University, Australia

Dr. Riccardo Scandariato
Professor, Hamburg University of Technology, Germany

Dr. Antonino Sabetta
Senior Researcher, SAP Security Research, France

Abstract

Today's Android ecosystem is a growing universe of a few billion devices, hundreds of millions of users, and millions of applications targeting a wide range of activities where sensitive information is collected and processed. The security of Android apps is thus of utmost importance and needs to be addressed carefully. In the last decade, several studies have investigated Android applications from a security point of view, focusing on the detection of vulnerabilities or the appropriate usage of cryptography APIs. However, with the Android framework's rapid iteration, new issues are continuously popping up while some old issues may not have been detected. As a result, security studies on Android apps have never been stopped.

Meanwhile, Android applications, just like other software, are developed by following an iterative process. Indeed, applications are updated regularly to fix bugs or introduce new features. In practice, to release a new version of their applications, developers need to provide a brand new installation package, which is known as an apk file. Therefore, each of these apk files stands for one version of a specific application, and the evolution of an application can be obtained by collecting all these apks. Nevertheless, the collection of these apk files are not straightforward because Android markets such as GooglePlay do not preserve the history of apk files. Instead, only the latest version of an app, i.e., the most recent apk, is provided. This fact challenges studies focusing on Android application evolution. However, history and past experiences allow us to learn from past mistakes. That is why evolutionary studies can potentially benefit both developers and users in many ways, such as: discovering trends for security issue predictions or policy evaluations, unveiling fundamental causes of vulnerabilities for prevention.

In this dissertation, by leveraging AndroZoo, a popular Android application dataset made available to researchers, the versioned lineages of Android apps are re-constructed. Then several security-relevant aspects of Android applications are investigated from an evolutionary perspective. Our study begins with a wide-range investigation in which we take a deep insight into the evolution of several vulnerabilities of Android applications. Then we focus on the vulnerabilities related to crypto-API. We present our attempt to learn cryto-APIs usage from the crowd, i.e., by mining crypto-APIs usage rules from app lineages. Finally, we further narrow down the scale to a new security breach spotted by us. We elaborate on the mechanism of the breach and investigate its evolution patterns. The detailed contributions include:

- *Re-construction of app lineages*: Android developers update their apps by providing new apk files which are the installation packages, and these apks have to be published via relevant markets. Nevertheless, mainstream Android application markets including the official market GooglePlay provide applications as a fleeing data stream where only the latest version of an application is available. This causes one of the main difficulties to re-construct the lineage of Android applications. Moreover, to build an app lineage dataset of large scale, besides the collection of millions of apk files, it also requires a considerable amount of computation capacities for feature extraction and matching. In this dissertation, we take advantage of the *AndroZoo* dataset and the High Performance Computing (HPC) clusters of the University of Luxembourg to re-construct the first large scale app lineage dataset and publicly share it with the community. Furthermore, a primary study based on the lineage dataset has been done to investigate the evolution of Android app complexity by leveraging six well-established complexity metrics.

- *Understanding the evolution of Android app vulnerabilities*: The community is still lacking comprehensive studies exploring how vulnerabilities have evolved and how they evolve in a single app across developer updates. In this dissertation, we fill this gap by leveraging the re-constructed app lineages. We apply state-of-the-art vulnerability-finding tools and systematically investigate the reports produced by each tool. In particular, we study which types of vulnerabilities are found, how they are introduced in the app code, where they are located, and whether they foreshadow malware. We provide insights based on the quantitative data reported by the tools, but we further discuss the potential false positives. Our findings and study artifacts constitute tangible knowledge to the community.

- *Mining crypto-API usage rules by analyzing app updates*: Android app developers recurrently use crypto-APIs to provide data security to app users. Unfortunately, misuse of APIs only creates an illusion of security and even exposes apps to systematic attacks. It is thus necessary to provide developers with a statically-enforceable list of specifications of crypto-API usage rules. On the one hand, such rules cannot be manually written as the process does not scale to all available APIs. On the other hand, a classical mining approach based on typical usage patterns is not relevant in Android, given that a large share of usages include mistakes. In this dissertation, building on the assumption that *"developers update API usage instances to fix misuses"*, we propose to mine the app lineages dataset to infer API usage rules. Eventually, our investigations yield negative results on our assumption that API usage updates tend to correct misuses. Actually, it appears that updates that fix misuses may be unintentional: subsequent updates quickly re-introduce the same misuses patterns.

- *Direct inter-app code invocation in Android apps and its evolution*: The Android ecosystem offers different facilities to enable communication among app components and across apps to ensure that rich services can be composed through functionality reuse. At the heart of this system is the Inter-component communication (ICC) scheme, which has been largely studied in the literature. Less known in the community is another powerful mechanism that allows for *direct inter-app code invocation* which opens up for different reuse scenarios, both legitimate or malicious. In this dissertation, we expose the general workflow for this mechanism, which beyond ICCs, enables app developers to access and invoke functionalities (either entire Java classes, methods or object fields) implemented in other apps using official Android APIs. We experimentally showcase how this reuse mechanism can be leveraged to "*plagiarize*" supposedly-protected functionalities. Typically, we could leverage this mechanism to bypass security guards that a popular video broadcaster has placed for preventing access to its video database from outside its provided app. We further contribute with a static analysis toolkit, named `DICIDer`, for detecting direct inter-app code invocations in apps. An empirical analysis of the usage prevalence and evolution of this reuse mechanism is then conducted.

A happy family is but an earlier heaven.

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 Ph.D. studies. It is my pleasure to express my gratitude to them.

First of all, I would like to express my deepest thanks to my supervisor, Prof. Jacques Klein, 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 Ph.D. journey.

Also, I am equally grateful to my co-supervisor, Prof. Tegawendé Bissyandé, for his patience, valuable advice and careful guidance. He has taught me how to perform rigorous researches, shown me how to write wonderful academic papers.

Third, I am particularly thankful to my previous daily adviser, Prof. Li Li, who has introduced me to the field of static analysis and Android security, led me in most of my researches step by step and guided me whenever I met problems. His continuous support has made my Ph.D. journey smooth and fulfilling.

I would like to extend my thanks to my previous colleague, Prof. Alexandre Bartel, for his great help with the static analysis tool *Soot* and my office mates Pingfan Kong and Timothée Riom for their valuable discussions and suggestions.

I want to thank all the members of my Ph.D. defense committee, in particular, both external reviewers Prof. Riccardo Scandariato and Dr. Antonino Sabetta. It is my great honor to have them on my defense committee and I appreciate their efforts to examine my dissertation and evaluate my Ph.D. work.

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

Finally, and more personally, I would like to express my most profound appreciation to my dear wife and boy for their 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.

Jun Gao
University of Luxembourg
October 2020

Contents

List of figures

List of tables

1 Introduction

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

Contents

1.1 Motivation

Mobile software has been overtaking traditional desktop software to support citizens of our digital era in an ever-increasing number of activities, including for leisure, internet communication or commerce. In this ecosystem, the most popular and widely deployed platform is undoubtedly Android, powering more than 1.6 billion active users, taking over 74% of worldwide mobile operating system market share and contributing to over 3 million mobile applications, hereinafter referred to as *apps*, in online software stores[1]. Yet from a security standpoint, the Android stack has been pointed out as being flawed in several studies: among various issues, its permission model has been extensively criticized for increasing the attack surface [2, 3, 4]; the complexity of its message-passing system has led to various vulnerabilities in third-party apps allowing for capability leaks [5] or component hijacking attacks [6]. Moreover, as the Android framework evolves rapidly, new security issues have been continuously popping up while existing issues are still waiting for discovery. Therefore, attention is still required to be paid to the security of Android apps.

Meanwhile, Android has been attracting the interest of developers since its early days. This also creates the situation of high competition in Android app development. Consequently, to keep up, developers are engaged in a frenzy of updates [7, 8, 9, 10]. In general, developers update their apps for (1) keeping up with the evolution of Android APIs (e.g., discarding the use of deprecated ones [11] while accessing early-release ones [12]), (2) adapting to new requirements or providing new features to keep the app competitive, (3) fixing bugs that may cause runtime crashes, or that make the app vulnerable to security threats, (4) improving the performance or maintainability, either by removing unnecessary code or by refactoring existing functionalities. Hence, evolutionary studies of Android apps become essential. From the security point of view, the evolutionary studies are potentially beneficial in many ways, for example: 1) Trends of certain issues could be discovered and these trends can be used to measure the effectiveness of relevant security policies or taken as an indicator for focus shifting to certain security issues. 2) As solving security issues is one of the primary purposes of app updates, fixing patterns could be learned by mining from the updates. However, investigating the evolution of Android apps is challenging. In the quasi-totality of apps available in the marketplace, the history of development is a fleeing data stream: at a given time, only a single version of the app is available in the market; when the next updated version is uploaded, the past version is lost. A few works [13, 12, 10] involving evolution studies have proposed to "watch" a small number of apps for a period of time to collect historical versions, and the insight observed by such studies may not be representative of that of the whole Android ecosystem.

To sum up, there is still a lot to explore in Android app security, while evolutionary studies will provide us a new angle to look at these issues and learn from history. Because of the limited number of literature focusing on such a topic so far, we are highly motivated to take action on the evolutionary study of Android app security.

1.2 Challenges

In this section, we introduce the technical challenges we face when conducting evolutionary security analysis of Android apps. More specifically, we introduce these challenges from two different aspects, which are software engineering and static analysis.

[1] Published by Shanhong Liu, Jun 2, 2020. Android - Statistics & Facts. `https://www.statista.com/topics/876/android/#dossierSummary__chapter1`. Accessed: Oct. 2020

1.2.1 Software Engineering Challenges

From a software engineering point of view, dealing with tasks of large-scale is one of the main challenges. The purpose of this is to make our studies be representative of the whole Android ecosystem. Moreover, the lack of open source apps in Android markets also leads to difficulties in some cases. We discuss these challenges in detail as shown in the following:

- **App lineage re-construction**. Although the open ecosystem strategy is adopted by Android, which leads to the prosperity of Android apps as well as its markets. In the quasi-totality of apps available in the marketplace, the history of development is a fleeing data stream: at a given time, only a single version of the app is available in the market; when the next updated version is uploaded, the past version is lost. Thus, the re-construction of a large scale dataset of Android app lineages requires a continuous scraping of a decent amount of apps through a long time span. Consequently, so far, a few works [13, 12, 10] involving evolution studies have proposed to "watch" a small number of apps for a period of time to collect historical versions. However, the insight observed by such studies may not be representative of that of the whole Android ecosystem.

- **Large scale analysis**. As the results concluded from studies with small datasets could be potentially lacking statistical significance. Our empirical studies are all based on datasets of large scales, such as our app lineage dataset, which contains around half a million apks. Moreover, our studies are leveraged by self-developed as well as state-of-the-art analysis tools, and for some cases, several tools need to be used in sequence or parallel. Therefore, how to compute such big datasets within a limited time budget is a major challenge. Although, we are benefiting from our High Performance Computing (HPC) clusters. There are still several things that need to be considered. The first one is task distribution. For those tasks which can run in parallel, server nodes of the clusters need to cooperate. The second is task cooperation, which is to deal with tasks that need to run in sequence. Following are the storage considerations. Since analysis tools commonly generate reports for each apk file. The total size of the reports in both volume and number of files could be easily over the limitation. Last but not least, crashes and exceptions are not rare when executing analysis tools. Thus, we need to either avoid the global crashes caused by local exceptions or restore from such a global crash.

- **Verification of tools**. Android embraces an open system model, which makes the initiative of Android app development easier and costless. It is indeed probably one of the primary cause of Android apps' pervasiveness, which also fosters more open source app projects compared to markets of other systems (e.g., IOS). However, the majority of applications is still close-sourced. Thus, to check the correctness of a new tool, reverse-engineering is mostly required, and manual verification is based on the intermediate representation (IR) generated from the reverse-engineered sample apps. As IRs are commonly assembly-like languages, it is much more difficult for a human to understand. Therefore, verification is always a challenging task.

1.2.2 Static Analysis Challenges

On the one hand, as Android apps are mainly written in Java, most of its features are also inherited by Android apps. Although these features provide lots of flexibility to developers. From a static analysis point of view, they also introduce many challenges. On the other hand, Android apps are different from traditional software in many ways because of the different operation mechanism of Android framework. Some of these differences also lead to new challenges in static analysis of Android apps. Thus, we list some main challenges which we encounter in the studies of this dissertation below:

- **Call graph construction**. In this dissertation, we mainly use static analysis to study the security evolution of Android apps and call graph is one of the most crucial techniques of static analysis. Therefore, how to correctly construct the call graph of Android apps becomes critical to our studies. Although Android apps are primarily developed in Java. The construction of the call graph of Android apps is much more complicated than traditional Java programs. The reasons that lead to this difficulty are: 1) The entry point of an app is not definite. Traditional Java programs have a single entry point, which is the *main* method. However, Android apps do not have such a method. Instead, each app could indicate several methods as a candidate, and the Android framework decides the entry point at runtime. 2) Component life-cycle is managed by the Android framework. In Android, apps are mainly composed of 4 kinds of components: *Activity*, *Service*, *BroadcastReceiver* and *ContentProvider*. These components have their own life-cycle controlled via methods such as *onStart, onStop, onResume*. Since the life-cycle of each component is managed by the Android framework. All these life-cycle related methods are not directly connected to the execution flow, which could lead to missing parts in the call graph. 3) Inter-component communication (ICC). Android introduces a unique mechanism for messages exchanging between different components. However, this mechanism is based on the Android framework as the intermediate. Therefore, from the application call graph point of view, the execution flow is hindered again by the Android framework.

- **Libraries**. App development based on third-party libraries is a common practice. Nevertheless, an apk, the Android installation file, is a standalone package containing code from current developer as well as libraries, and libraries could contribute to a significant proportion of the app size. Even worse, there are no reliable methods to distinguish between developer code and library code. This situation leads to three major difficulties: 1) as many apps could share with the same libraries, analyzing these apps will wast a considerable amount of time on duplicated libraries. 2) Duplicated libraries could cause re-calculations, which could threaten the validation of statistical results. 3) Most of the time, libraries are partially used in the app. Thus, those unused "dead code" are also threats to the correctness of statistical results.

- **Native code**. By taking advantage of Java Native Interface (JNI) APIs, Android apps can be coded in both Java and C/C++, and the C/C++ part of the code is named as native code. However, reverse engineering and static analysis techniques for the native part are entirely different from the Java part. As the research community has been mainly focusing on the Java part so far, Few techniques and tools are available for native code analysis. Hence, dealing with native code in Android app studies becomes a thorny problem.

- **Reflection**. Reflection is a feature provided by Java, which is inherited by Android as well. It is used to load, inspect and manipulate Java code at runtime. While introducing flexibility to development, reflection also results in challenges in static analysis. Because of the nature of dynamic loading, the dynamically loaded part cannot be statically analyzed. Consequently, the call graph will be incomplete, for example. Moreover, points-to analysis is a static analysis technique to track reference passing. Since the tracking begins at an allocation site (i.e., a statement creating an object with *new* keywords in Java), while reflection creates objects without such a statement. Analyses of this kind will be completely failed in facing reflection.

- **Obfuscation**. The purpose of obfuscation is to make either source or machine code incomprehensible to humans. It becomes a common practice for more and more developers out of security consideration and also, creates many troubles for static analysis. Take the string constant as an example. It is not difficult to understand that to know constant strings are critical hints to understand programs. However, string obfuscation techniques transform comprehensive string constants into random character sequences, therefore concealing these constants' original meaning and deter specific analyses.

1.3 Contributions

The main contributions of this dissertation are listed as follows:

- **Re-construction of Android app lineages**. By leveraging (1) our *AndroZoo* dataset, which is so far the most prominent and continuous growing Android app repository, and (2) High Performance Computing (HPC) clusters of the University of Luxembourg, we carefully proceed to reconstruct the version lineages of Android apps at an unprecedented scale. These app lineages not only serves the purpose of this dissertation, but also are a valuable artefact for diverse research fields in our community. Meanwhile, we present a preliminary study with the newly re-constructed app lineages. We discuss insights of complexity evolution in Android apps and enumerate its implications as well as the limitations based on six well-established metrics. Last but not least, we make our toolset publicly available to compute complexity metrics for Android apps readily .

  This work has led to a research paper published to the 24th International Conference on Engineering of Complex Computer Systems (ICECCS 2019)

- **Understanding the Evolution of Android App Vulnerabilities**. We present a large scale study on the evolution of Android app vulnerabilities by applying state-of-the-art vulnerability finding tools on all app versions of our app lineage dataset and record the alerts raised as well as their locations. We investigate specifically 10 vulnerability types associated with 4 different categories related to common security features (e.g., SSL), its sandbox mechanism (e.g., Permission issues), code injection (e.g., WebView RCE vulnerability) as well as its inter-app message passing (e.g., Intent spoofing). Correlating the analysis results for consecutive app pairs in lineages, we extract a comprehensive dataset of reported vulnerable pieces of code in real-world apps, and, whenever available, the subset of changes that were applied to fix the vulnerabilities. Finally, we perform several empirical analyses to (1) highlight statistical trends on the temporal evolutions of vulnerabilities in Android apps, (2) capture the common locations (e.g., developer vs. library code) of vulnerable code in apps, (3) comprehend the vehicle (e.g., code change, new files, etc.) through which vulnerabilities are introduced in mobile apps, (4) investigate via correlation analysis whether vulnerabilities foreshadow malware in the Android ecosystem.

  This work has led to a research paper accepted by the IEEE Transactions on Reliability in 2020

- **Mining Crypto-API Usage Rules**. We proposed a novel approach for inferring the rules of using cryptography APIs based on an intuitively reasonable assumption: "API usage updates generally transform incorrect usages into correct usages". We report the negative result of such an approach, and in detail, we discuss why the assumption is invalid. We notice that most of the misuse-fixing updates may not have been made intentionally.

  This work has led to a technical paper published to the 16th International Conference on Mining Software Repositories (MSR 2019)

- **Code Reuse in Android via Direct Inter-app Code Invocation**. We expose a little-advertised reuse mechanism within the Android ecosystem. In particular, we demonstrate how it can be leveraged to perform stealthy functionality plagiarism that may not be covered by a standard licensing scheme. Then, we develop a static analysis tool, `DICIDer`, for the detection of DICIs in Android apps, and perform an empirical analysis on the prevalence of DICIs among a large dataset of apps as well as the app lineages to investigate its common usage and evolutionary characteristics. We further provide extensive discussions on how and why developers use DICIs through an analysis of sample cases. Eventually, we propose an example of a countermeasure that developer could used to protect their apps against DICI.

This work has led to a research paper published to 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)

1.4 Roadmap

The structure of this dissertation is organized as follows. A brief introduction on the necessary background information, including Android, app lineage and static analysis is given in Chapter 2. Followed by Chapter 3, we first introduce the re-construction of app lineages in detail and present a preliminary study of the evolution of Android app complexity by leveraging the newly constructed lineages. In the following three chapters, we take a width-to-depth approach to introducing our evolutionary study journey on Android app vulnerabilities. In Chapter 4, we systematically investigate the evolution of several vulnerabilities reported by state-of-the-art vulnerability-finding tools. Then we focus on vulnerabilities in using cryptography APIs. We propose a novel approach to infer the rules of cryptography API usages by mining the updates of real-world app lineages in Chapter 5. Chapter 6 reveals a new vulnerability caused by a less known mechanism named direct inter-app code invocation (DICI) and its evolution. Finally, in Chapter 7, 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 related research studies that we have conducted in this dissertation. Mainly, we introduce the primary dataset we used and the definition of app lineages, revisit some concepts of Android and static analysis, and go through some evolution studies, respectively.

Contents

2.1 Android

In this section, we introduce several background knowledge about Android. We first briefly go through the Android operating system (OS) about its different versions and architecture. Then we talk about the manifest file of Android apps, which contains important information and is essential for static analysis. Finally, we introduce the inter-component communication mechanism of Android apps.

2.1.1 Android OS

The Android mobile operating system is built on top of the Linux kernel and provides a framework to facilitate the development of Android apps. As the framework evolves, the provided Software Development Kit (SDK), including the Application Programming Interfaces (APIs), is regularly updated. To better track and reflect those changes, each major release of the Android framework is tagged with multiple names: (1) its version number (e.g., Android 4.4); (2) its API level (e.g., 19); and (3) a name of sweet (e.g., KitKat). Figure 2.1 presents an example of API levels with respect to their adoption by millions of Android-powered devices using the official Google Play store as of April 2020[1].

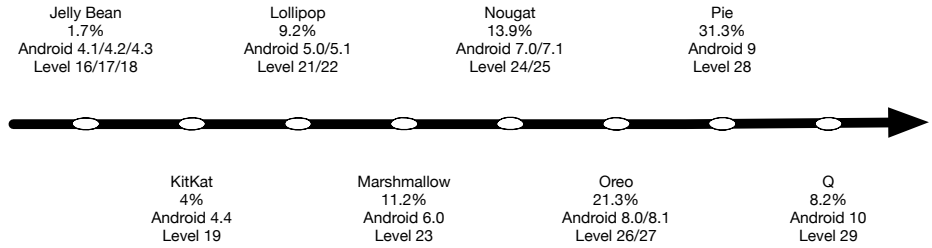


Figure 2.1: Distributions of API levels supported by current Android-powered devices (versions with less than 1% are not shown).

It is worth to mention that around every half year, the API level is upgraded. As every Android device only provides exactly one API level, it is commonly used to determine the compatibility of an application.

As we know that Android OS is developed based on the Linux kernel, it is actually composed of five layers of software as shown in Figure 2.2[2]. Hereafter, we briefly go through these layers from bottom to top:

- **Linux kernel.** This layer is the foundation of the Android OS. It provides functionalities such as threading and low-level memory management. All drivers here are allowed the system to access all relevant hardware as well.
- **Hardware Abstraction.** The purpose of this layer is to expose device hardware capabilities to the API framework layer in a standard way.
- **Android runtime and native libraries.** This layer, as it is named, is actually divided into two parts. Android Runtime, before Android 5.0 is known as Dalvik, runs each app in its own process and with its own runtime instance. While native libraries provide the ability to access functionalities in C/C++ libraries to android applications.
- **API framework.** The APIs in this layer is written in Java language. By using these APIs, Android applications can access all the features provided by the OS, such as utilize the camera to take photos as well as building UI via the Android view system.

[1] Data obtained from: `https://www.xda-developers.com/android-version-distribution-statistics-android-studio/` accessed on Oct. 2020

[2] Referred to: `https://developer.android.com/guide/platform`

- **Application layer.** This is the layer with a set of core apps for email, short message, contact, etc. These apps can be provided officially from Android as well as third-parties.

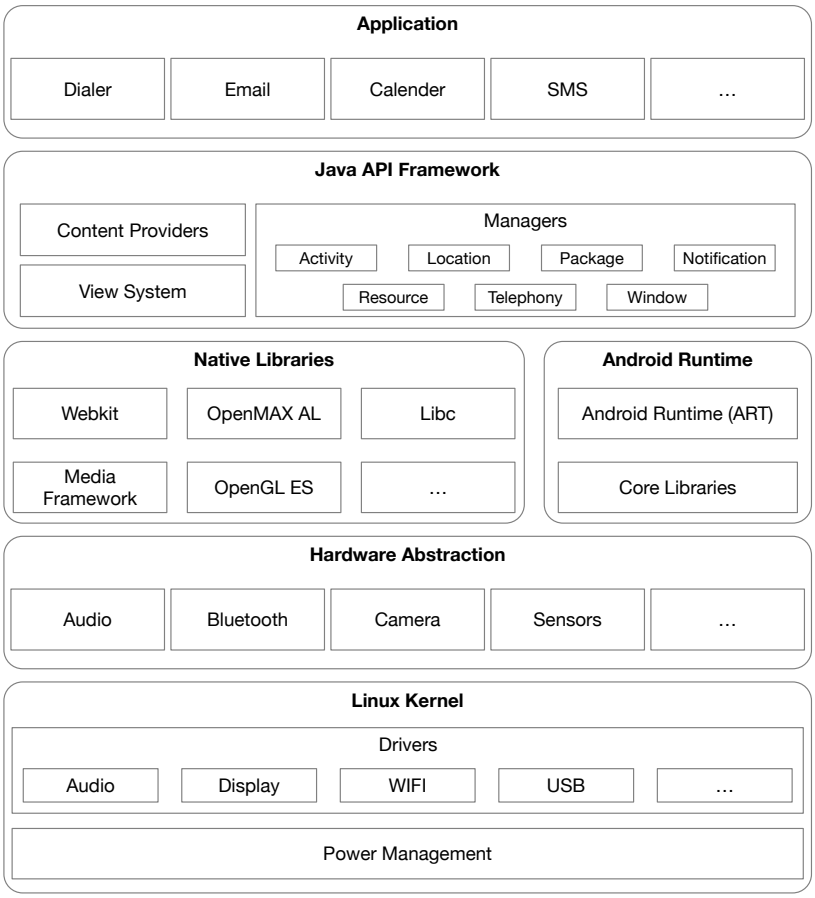


Figure 2.2: Android OS Architecture

2.1.2 App Manifest

An Android app is installed with an Android Application Package (APK) file, which is actually a zip archive file. The main content of this file is shown in Table 2.1. From the static analysis point of view, the most important files are the *DEX* files, the files under the *lib* directory and *AndroidManifest.xml*. The previous two contain the real code of the application, while the last one describes essential information about the app to build tools, Android OS as well as some markets. In this section, we introduce some of the key attributes of the manifest file.

Table 2.1: Content of Android Application Package

Content	Description
AndroidManifest.xml	Application configuration file.
classes.dex	Compiled bytecode, commonly generated from Java. Multiple DEX files is possible.
resources.arsc	Compressed resource file.
META-INF/	The directory contains relevant meta-data of the app, such as app certificate.
res/	The directory of resources such as images which not compiled into resources.arsc file.
assets/	The directory containing application assets.
lib/	The directory of native libraries.

Listing 2.1 illustrates a simplified manifest file of a real project, and we discuss the following attributes base on this example:

- **Package name.** The package name in the manifest file is the name of an application (cf. line 4). Android build tools use this to determine the location of code entities when building a project. Android OS and GooglePlay leverage this name to distinguish between apps.

- **Components.** Android components include activities (for UI), services (for background long-running operations), broadcast receivers (for listening to broadcast events and intents) and content providers (for data access), and Android apps are the composition of these components. Commonly, all components of an app need to be declared in the manifest file, and if any of these components need to be accessible by other apps, they have to be exported further. In the example, it contains 3 components which are 2 activity components and a service component. As declared in line 32 that the *exported* attribute is assigned with *false*, the service *ExoPlayerService* cannot be accessed from another application. For activity *MainActivity*, though the *exported* is not declared, since there are no *intent-filter*s declared for this activity, the default value for *exported* is *false* as well.

- **Permission.** In order to access protected parts of the system, certain permission requirements need to be declared in the manifest file. These permissions will be consented during installation by the user normally. As shown in the example, from line 8 to 9, it requires 2 permissions which are for the access of the internet and Bluetooth state.

- **Features.** The access to hardware and some software features has to be declared as well. Such as cameras, Bluetooth and microphones. In line 6 of the example, it asks for access to the Bluetooth device.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:tools="http://schemas.android.com/tools"
4      package="com.example.myapp">
5
6      <uses-feature android:name="android.hardware.bluetooth" />
7
8      <uses-permission android:name="android.permission.INTERNET" />
9      <uses-permission android:name="android.permission.BLUETOOTH" />
10
11     <application
12         android:allowBackup="true"
13         android:icon="@mipmap/ic_launcher"
14         android:label="@string/app_name"
15         android:theme="@style/AppTheme">
16        <activity
17             android:name=".activities.splash.SplashActivity"
18             android:finishOnTaskLaunch="true"
19             android:label="@string/app_name"
20             android:launchMode="singleTask">
21                 <intent-filter>
22                     <action android:name="android.intent.action.MAIN" />
23                     <category android:name="android.intent.category.LAUNCHER" />
24                 </intent-filter>
25         </activity>
26         <activity
27             android:name=".activities.main.MainActivity"
28             android:launchMode="singleTask" />
29         <service
30             android:name=".services.ExoPlayerService"
31             android:enabled="true"
32             android:exported="false" />
33     </application>
34 </manifest>
```

Listing 2.1: Android Manifest File Example

2.1.3 Inter-Component Communication

As we mentioned in the previous section, Android components are the essential elements of Android applications. Therefore, inter-component communication (ICC) becomes the fundamental way to communicate within an app as well as between different apps.

There are dedicated methods for this purpose to start a communication between components, such as *startActivity()* which is one way used to communicate with another activity component. Meanwhile, we need to emphasize that the Android framework is in charge of the coordination between components. Thus, the intention of communication is always passed to the framework as an **Intent** object, in which besides message information, the target component has to be declared. There are two ways of specifying the target and they define two types of ICC. The first one is known as *explicit* ICC which provides the exact class name of a component as the communication target. While the other only defines the action wanted to take and leaves the framework to decide what component to communicate to achieve the action. The alternative way is known as *implicit* ICC. As the example shown in Listing 2.2, from line 2 to 4 are the code snippet of explicit ICC. We can see that in line 2, it is specified the class of *NewRecipeActivity* as the target component when instantiating the Intent object. However, for the code snippet of implicit ICC from line 7 to 11, there is no specification of the target class. Instead, it specifies the action in line 8.

```
1  // Explicit ICC, to invoke NewRecipeActivity
2  Intent intent = new Intent(this, NewRecipeActivity.class);
3  intent.putExtra("recipeID", recipeID);
4  startActivity(intent);
5
6  // Implicit ICC, to invoke a component which can deal with action ACTION_SEND
7  Intent sendIntent = new Intent();
8  sendIntent.setAction(Intent.ACTION_SEND);
9  sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
10 sendIntent.setType("text/plain");
11 startActivity(sendIntent);
```

Listing 2.2: Code Snippet for Triggering ICC Calls

Moreover, for those components declared as exported in manifest files, ICC can be used to communicate with these exported components from other apps. This kind of communication is also known as Inter-App Communication (IAC). Hence, *implicit* ICC is commonly used in IAC scenarios since most of times, developers do not know the class names of other apps. On the contrary, *explicit* ICC is highly recommended to be used for component communications within an app. Otherwise, *implicit* ICC could lead to unexpected results and vulnerability to attacks such as *intent interception*.

2.2 Dataset

In this section, we introduce two datasets, which are the important building ingredients of this dissertation. More specifically, we present the AndroZoo dataset and our app lineage dataset, which is re-constructed based on apps from AndroZoo.

2.2.1 AndroZoo

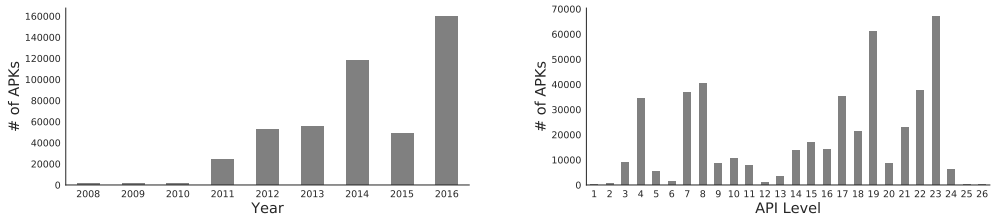
AndroZoo [14, 15] is so far one of the most extensive Android app collection and is continuously growing by collecting Android apps from various sources, including the official Google Play app market and third party alternative markets such as AppChina. So far, AndroZoo repository contains

over 13 million[3] Android apks and has been successfully leveraged to support the analysis of various research studies [16, 17].

2.2.2 App Lineage

An **apk** represents a released package of an app. So, app **version** refers to a specific *apk* released in the course of development of an app and the nth apk of an app is denoted as apk_n. Thus, we define an app **lineage** as the consecutive series of its *versions* that is: $L = \{apk_1, apk_2, \ldots, apk_n\}$.

Our app lineage dataset is re-constructed based on AndroZoo, and it includes 28,564 app lineages of app versions no less than 10, which contains 465,037 app versions. The details about the re-construction are presented in Section 3.3. Figure 2.3a shows the distribution of the releasing years of the apks. The releasing time is obtained from the last modification time of the "classes.dex" files decompressed from apks. While Figure 2.3b exhibits its target API level distribution.



(a) Year Distribution (b) API Level Distribution

Figure 2.3: Statistics of Lineage Dataset

Notice that, for this dataset, one *lineage* may include only a subset of the *apks* that the app developers have released since our dataset, although massive, is not exhaustive[4].

2.3 Static Analysis

Static analysis consists in examining the code of apps without execution but checking the code structure, logic between statements, etc. Static analysis techniques are widely used in the literature for scaling various investigations on Android apps to large datasets. As the majority of Android apps are not open-sourced, the analyses are commonly based on an intermediate representation (IR) produced via reverse engineering of the byte code of the apps. Meanwhile, since Android apps are primarily developed in Java, many tools initially designed for Java programs could also be applied to Android apps with or without modifications. Among most of the static analysis tools, we mainly leverage *Soot* [18] and *Apktool*[5] in this dissertation. They use *Jimple* and *smali* as the IR, respectively. In this section, we briefly go through two static analysis techniques which closely related to this dissertation.

2.3.1 Call Graph

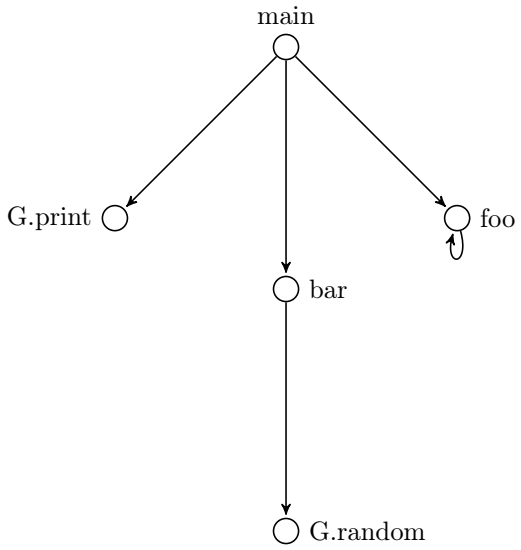
A call graph (CG) is a directed graph showing the invocation relationships between one method to other methods. To construct the CG of a traditional Java program, as it always starts with a single entry point which is the *main* method, the construction also starts with this method. By inspecting its method body, all the methods invoked by it can be found, and in the graph, the connections

[3] As of December 2020
[4] Therefore, for a certain app lineage, there could be missing versions here and there.
[5] `https://ibotpeaches.github.io/Apktool/`

between the *main* method and these methods can be constructed. Iteratively doing the same process with all methods found will lead to the whole CG of the program (e.g., see the example shown in Figure 2.4). However, in Android apps, there is no *main* method. Instead, Android apps can contain several entry points which are called by the Android framework at runtime. Also, invocations within an app can be done via ICC. Therefore, the construction of CG for Android apps becomes more complicated. In practice, original apps are firstly instrumented with a dummy *main* method and other relevant invocation methods to mimic conventional invocation relations. Then, CG can be constructed with the instrumented apps in the traditional way.

```
1  class Example {
2      public static void main() {
3          int x = foo(100);
4          int y = 0;
5          y = bar(x);
6          G.print(y)
7      }
8
9      private static int foo(int q) {
10         int r = q;
11         if (q % 7 != 0) {
12             r = foo(q - 1);
13         }
14         return r;
15     }
16
17     private static int bar(int p) {
18         int q = G.random();
19         return q;
20     }
21 }
22
```

(a) Source Code



(b) Call Graph

Figure 2.4: Call Graph Example

2.3.2 Points-to Analysis

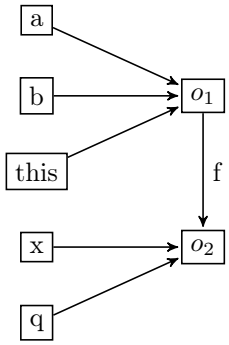
The points-to analysis is one of the most fundamental static analysis and enables a variety of other analyses. Mostly, it is also inter-related with the construction of CG and affects the accuracy of the final CG. Initially, it is designed for the C language, which provides pointer variables. As object-oriented languages became mainstream, the points-to analysis had adapted to languages such as Java. For Java, the analysis aims to determine the set of objects whose addresses may be stored in a given reference variable or reference object field. Therefore, an abstraction of the run-time memory states could be constructed by computing such a set for all app variables. It is also a core analysis for Android app security in finding information leaks, inferring ICC calls, etc. Figure 2.5 shows a simple example. In Figure 2.5b, o_1 and o_2 stand for objects initiated in line 8 and 9, respectively, this stands for this pointer used in line 5, and f is the field of class B which is declared in line 4. The arrows in the figure indicate the points-to relationships between reference variables and its pointing objects, such as variable a refers to object o_1.

2.4 Related Works of Evolution Study

Several researchers studied the general laws of software evolution [19, 20, 21], which show that software will continuously change and so does its complexity, demonstrating that software evolution analysis is essential in our community.

```
1  class A {void m(X p) {}}
2
3  class B extends A {
4      X f;
5      void m(X q) {this.f =
       q;}
6  }
7
8  B b = new B(); // o1
9  X x = new X(); // o2
10 A a = b;
11 a.m(x);
12
```

(a) Source Code



(b) Points-to Relationship Graph

Figure 2.5: Points-to Analysis Example

Software evolution analysis has been widely adopted to understand the evolutionary process of a software system and hence to predict its future evolution [22, 23, 24]. Generally, software evolution analysis investigates the evolution of a software system to identify potential shortcomings in its architecture. Those identified shortcomings can then be addressed specifically to improve the quality of the software system.

Neamtiu et al. [22], by studying nine open-source projects covering 705 official releases, they confirmed Lehman's two laws of software evolution (i.e., software continuing change and continuing growth). Behnamghader et al. [24] argued that studying software quality before and after each commit can reveal how each change impacts the overall quality.

However, Android apps are generally released as APKs which do not contain commit messages. Therefore, research leveraged the difference between two subsequent app releases to investigate the evolution of Android apps [25, 12, 13, 10]. Calciati et al. [13] have investigated the evolution of permissions. Taylor et al. [10] investigated the evolution of app vulnerabilities. Hecht et al. [26] investigated the evolution of Android poor design choices based on 3,568 versions of 106 Android apps.

3 App Lineage Re-construction and App Complexity Evolution

In this chapter, we overview our methodology to yield the app lineage dataset summarized in Chapter 2. The dataset includes 28,564 app lineages (i.e., successive releases of the same Android apps) with more than 10 app versions each, corresponding to a total of 465,037 apks. We then take the opportunity of this large-scale dataset of app lineages to attempt a comprehensive study on the evolution of apps, notably in terms of complexity. The investigation is based on six well-established, maintainability-related complexity metrics commonly accepted in the literature on app quality, maintainability etc. The result reveals that, overall, while Android apps become bigger in terms of code size as time goes by, the apps themselves appear to be increasingly maintainable and thus decreasingly complex.

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

- J. Gao, L. Li, T. F. Bissyandé, and J. Klein. On the evolution of mobile app complexity. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 200–209, 2019

Contents

3.1 Overview

Android has been attracting the interest of developers since its early days. This also creates the situation of high competition in Android app development. Consequently, to keep up, developers are engaged in a frenzy of updates [7, 8, 9, 10]. In general, developers update their apps for (1) keeping up with the evolution of Android APIs (e.g., discarding the use of deprecated ones [11] while accessing early-release ones [12]), (2) adapting to new requirements or providing new features to keep the app competitive, (3) fixing bugs that may cause runtime crashes, or that make the app vulnerable to security threats, (4) improving the performance or maintainability, either by removing unnecessary code or by refactoring existing functionalities.

Standing out among other apps requires app developers to guarantee a level of quality in their app code. Unfortunately, in the absence of a concrete guideline for maintaining quality, it is difficult to measure to what extent quality is taken into account with respect to update changes. Instead, and as the first step towards building such a guideline, it is important to investigate some quality properties of various app versions in order to draw insights from the practice of real-world app development. Our objective is thus to conduct a large-scale empirical study on the quality evolution of Android apps.

To that end, we focus on measuring *maintainability* of app code. Software maintainability is indeed considered today as one of the most important concerns in the software industry [28, 29]. Corbi, a recognized expert in the field, has even elevated maintainability as a major challenge for program understanding since the 1990s. Generally, *code complexity* is accepted to provide a good proxy for measuring maintainability [30]. Given the pervasiveness of mobile software in our daily life today, it is important to study how complexity has evolved in order to build knowledge towards improving quality in software development.

In this work, we first re-construct an unprecedented large dataset of 28,564 app lineages and investigate evolution trends of complexity, relying on six metrics proposed by Chidamber et al. [31]. We implement a process where each app is analyzed and six renowned maintainability-related complexity metrics are computed, trends are highlighted and outliers are summarized.

To summarize, this chapter focuses on the following contributions of our dissertation:

- We carefully proceed to reconstruct the version lineages of Android apps at an unprecedented scale, based on a dataset of over 10 million apps collected from a continuous crawling of Android markets (including the official Google Play). Since market scraping opportunistically follows links in online store webpages, no explicit identifier could be maintained to track app versions. Therefore, we rely on heuristics to conservatively link and order app versions to retrieve lineages, leading to the selection of 28,564 app lineages containing each at least 10 versions of a given app. Although this contribution serves the purpose of our study, it is a **valuable artefact** for diverse research fields in our community, notably software quality and its sub-fields of testing, repair and evolution studies.
- We share with the community all complexity metric values for a large dataset of Android apps where each app is associated with several of its release versions.
- We present an empirical study on the evolution of complexity in Android apps based on six well-established metrics (such as NOC, Number of Children or LCOM, Lack of cohesion in Methods), and from different perspectives such as median and standard deviation values.
- We discuss insights from our study and enumerate its implications as well as the limitations.
- We make our toolset publicly available to readily compute complexity metrics for Android app APKs.

The remainder of this chapter is organized as follows. Section 3.2 presents background information on the metrics leveraged in our work. Then, we introduce the re-construction of app lineages in Section 3.3. Section 3.4 overviews the experimental setup for answering the research questions. Section 3.5 details the results of our study while Section 3.6 discusses some insights as well as the limitations. Finally, we discuss related work in Section 3.7 and sum up this chapter in Section 3.8.

3.2 Background of Complexity Metrics

Chidamber et al. [31] have introduced six metrics to "measure the complexity in the design of classes". Since Android apps, as mentioned before, are written in Java and thereby have extended Java's object-oriented features, the proposed six metrics should also be able to reliably improve the development processes of Android apps. State-of-the-art studies such as Jost et al. [32, 33] have also leveraged those metrics for Android app developers to consider so as to write high-quality code. We note that these metrics are highly related to complexity concerns, and thus, we adopt them to measure the complexity of Android apps.

1. **Weighted methods per class (WMC)** is the sum of the complexities of all methods in a class. It is used to measure the effort required for developing and maintaining a particular class as well as the inheritability and reusability of a class. A high WMC score of a class means that the class is complex that hence is difficult to reuse and maintain. To simplify the calculation, in this work, we consider the complexity of all methods to be unity. Then WMC is simply a method counter of each class.

2. **Depth of inheritance tree (DIT)** is used to measure the depth of a given class based on the inheritance tree. Ideally, the value of DIT metric should be kept low as the complexity of developing, testing and maintaining a class would significantly increase if the depth of inheritance tree increases. As DIT defined, the inheritance tree of each class is calculated and the maximum length is set as the value of DIT.

3. **Number of children (NOC)** is another metric leveraged to measure the "width" of a given class (i.e., the number of direct sub-classes) based on the inheritance tree. The value of NOC approximately indicates the reuse degree of a given class. While the reusability of a class increases if more children are introduced, the responsibility required to maintain the class not to break the children's behavior also increases.

4. **Lack of cohesion in methods (LCOM)** is a metric used to measure the cohesiveness between methods and attributes of a given class. A higher LCOM value indicates a low cohesion between the methods and data, which hence increases the complexity of the class and subsequently increasing the possibility of introducing errors during the development of software. There are two ways to calculate the value according to Linda et al. [34] and in this work, we choose the first one which is based on the average percentage of each data field used by the methods of a class.

5. **Coupling Between Object classes (CBO)** measures the dependency of a class on other classes. High CBO value indicates excessive dependency which means lower reusability and higher testing complexity. It is calculated by counting the number of other classes used by a class.

6. **Response For a Class (RFC)** reflects the potential invocation of methods of a class on responding to a message. A low value of RFC is preferred since it indicates short possible invocation chain which makes debugging and testing easier. RFC is calculated by counting all the methods invoked in a class. For methods invoked more than once, only the first time will be counted.

Initially, we have considered the 22 quality metrics proposed by Mercaldo et al. [35]. However, our preliminary experiments have revealed that many of them are highly correlated with each other as demonstrated in Figure 3.1. Moreover, because of space limitations to present the results of all the 22 metrics, Therefore, for this study, we decide to focus only on the six classic metrics. We believe that the other metrics, especially the ones that are recently introduced, are also worth to explore and hence we will consider them in our future works.

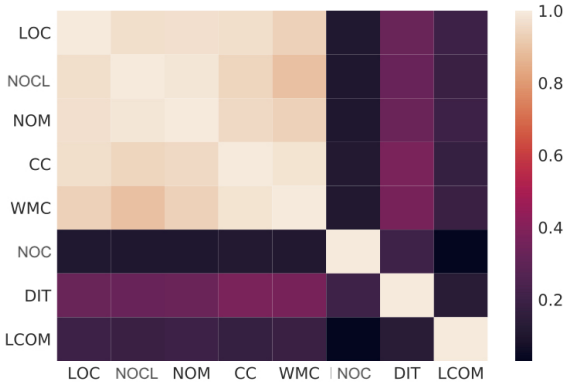


Figure 3.1: Metrics Correlation Map

3.3 Re-construction of App Lineages

We now describe the process (illustrated in Figure 3.2) that we followed to re-construct app lineages from AndroZoo's data heap.

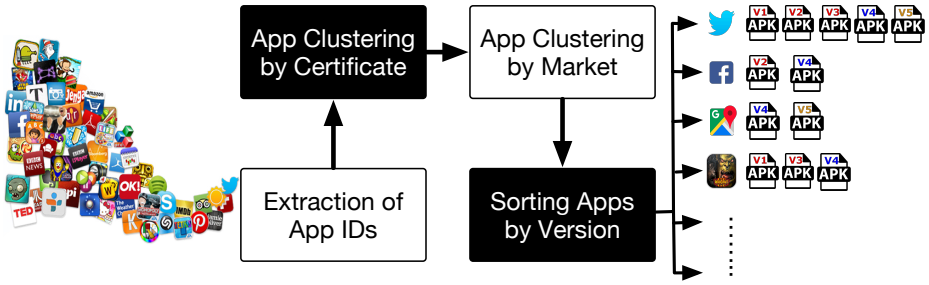


Figure 3.2: App Lineages Re-construction Process.

To re-construct app lineages, we need first conservatively identify unique apps and then link and order their app versions (i.e. apks) into a set of lineages. The objective is to maximize precision (i.e., a lineage will only contain apks which are actually different versions of the same app) even if recall may be penalized (i.e., not all apk versions might be included in a lineage). Indeed, missing a few versions will not threaten the validity of our study as much as linking together unrelated apps. Hence, we implement the following four steps:

1. *Application Id Extraction.* Google recommends [36] that each app should be named, in all its versions, following the usual Java package naming convention[1]. This avoids the collision in app names, which the market must avoid since two different apps with the same name cannot be installed on the same device. App name is indicated uniquely in the Manifest file with the attribute

[1]Developers should use their reversed internet domain name to begin package names

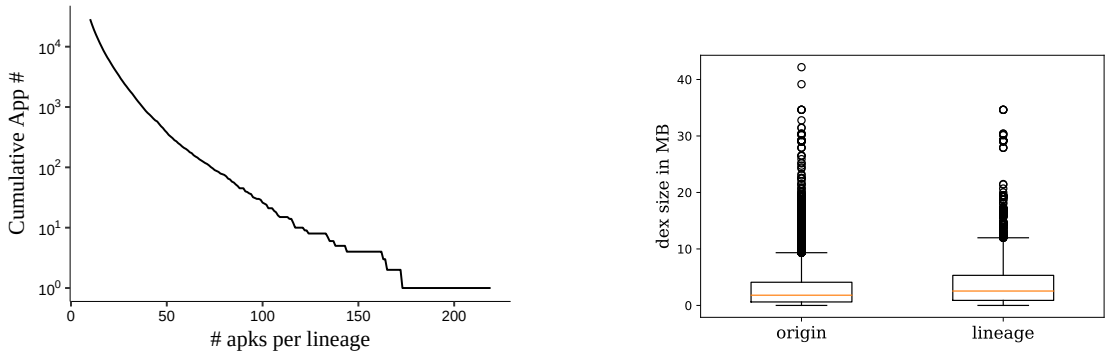
applicationId. We group together apks with the same application id as candidate versions of a given app.

2. *App Grouping by Certificate.* Since Android apps are prone to repackaging[2] attacks [16, 37], different apks in a group sharing the same app name may actually be different branches by different "developers". We do not consider in our study that repackaged apps should appear in a lineage since the changes that are brought afterward may not reflect the natural evolution of the app. Thus, we group apks in each group based on developer signatures. Meanwhile, during the implementation of this step we also noticed that most of the markets, even for Google Play (the official market), do not emphasize a unique certificate (i.e., only one certificate for each app). For these cases, we found that there are around 0.064% apks which include more than one certificates. Since, for these apks, we cannot uniquely distinguish their ownership, we dropped them in the final dataset.

3. *App Grouping by Market.* We further constrained our lineage construction by assuming that developers distribute their app versions regularly in the same market. From each group obtained in the previous step, we again separate the apks according to the market from which they were crawled. As a result, at the end of this step each group only includes apks that are (1) related to the same app (based on the name), (2) from the same development team (based on the signatures), and (3) were distributed in the same market (based on AndroZoo metadata). Each group is then considered to contain a set of apks forming an unique lineage.

4. *App Version Sorting.* In order to make our dataset readily usable in experiments, we proceed to sort all apks in each lineage to reflect the evolution process. We rely on the *versionCode* attribute which is set by developers in the Manifest file. We further preserve our dataset from potential noise by dropping all apks where no *versionCode* is declared.

To avoid toy apps, we adopted the strategy used in [38] to set a threshold of at least 10 apks before considering a lineage in our study. And during this step almost 92% of apks were filtered out.



(a) Lineage Sizes

(b) Dex Sizes

Figure 3.3: Statistics of App Lineages

Overall, we were able to identify 28,564 lineages and the five-number summary of lineage size are 10, 11, 13, 18 and 219 respectively. Figure 3.3a shows that the largest proportion of apks are included in smaller-size lineages. This also explained why large portion of apks were removed during toy app filtering. Table 3.1 presents the top 5 lineages w.r.t. the number of apks. In total, our lineage dataset includes 465,037 apks. Figure 3.3b compares dex sizes of APKs between the original dataset and the re-constructed lineage dataset. It can be noticed that apks of small size has been removed mostly during lineage re-construction, as the median value increased from around 2.6 to 3.3 MB.

For the toy apps we removed from our dataset, there are still chances that they are highly used by smartphone users. To further study such a possibility, we investigate the installation situations of

[2]An apk can be disassembled, slightly modified and reassembled into another apk

Table 3.1: Top Five App Lineages.

Lineage	#apks	Market	Developer
com.knightli.book.jokebookseries.m3	172	appchina	knightli
wp.wpbeta	164	google play	WP Technology
com.manle.phone.android.yaodian	162	appchina	manle
com.imo.android.imoimbeta	143	google play	imo.im
com.knightli.ebook.zyys	134	appchina	knightli

the apps removed and compare it with the situation of kept apps. Since there are almost 3 million removed apps, we randomly sampled 200 thousand Google Play apps for this investigation. We successfully crawled the "installs" metadata for 29,300[3] apps and the installation situation for both removed and kept apps are shown in Figure 3.4. We observe that compared to kept apps, the whole shape of removed apps is remarkably shifted to the left, which indicates that removed apps are much less installed by app users. Thus, we can confirm that our study focuses on apps that are more likely to be downloaded and installed by users.

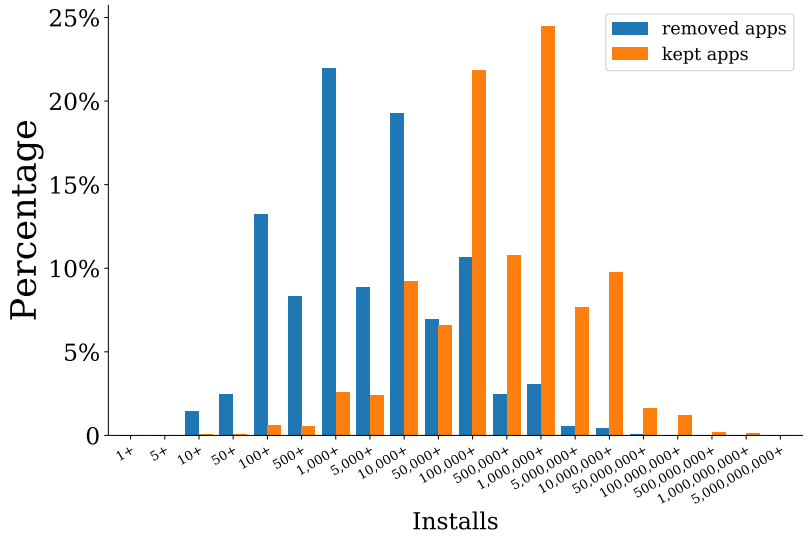


Figure 3.4: Intallation Comparison between Removed and Kept Apps

To assess the diversity of our lineage dataset, we first explore the categories of the concerned apps. Since this information is not available from the AndroZoo repository, we took on the task of crawling market web pages to collect meta-data information for each lineage. We focused on our study on the official Google Play. Out of the 16,074 lineages which were crawled from Google Play, we were able to obtain category information for 14,208 lineages. 1098 lineages were no longer available in the market while the market page for 768 lineages could not be accessed because of location restrictions. Figure 3.5 illustrates the high diversity in terms of category through a word cloud representation.

Second, we investigate the API levels (i.e., the Android OS version) that are targeted by the apps in our dataset. Since the Android ecosystem is fragmented with several versions of the OS being run on different proportions of devices, it is important to ensure that our study covers a comprehensive set of Android OS versions. Figure 3.6 presents the distribution of API level span of lineages of the dataset. The API level span of a lineage indicates the range between the minimum and maximum targeted API level found in the lineage. In the figure, the X-axis is the lower bound of the API level

[3]Because of app off-shelf and region-based access control of Google Play Store, the metadata for some apps cannot be collected.

Figure 3.5: Word Cloud Representation of Categories Associated with Our Selected Lineage Apps.

of a lineage while the Y-axis stands for the upper bound. Therefore, for each square, it indicates an API level span from the lower bound to upper bound. Meanwhile, the color of a square reflects the number of lineages of this API level span. According to the figure, it is easy to find out that most of the deep colored squares are located either on the diagonal or on the right lower corner. This phenomenon suggests that most apps tend to stay within one API level. For such app lineages that have their apps initiated with latest API levels, they are more likely to be upgraded with higher API levels. But still, apps of other API level spans can also be spotted in our dataset. Thus, our lineage dataset is quite diverse in terms of API level span.

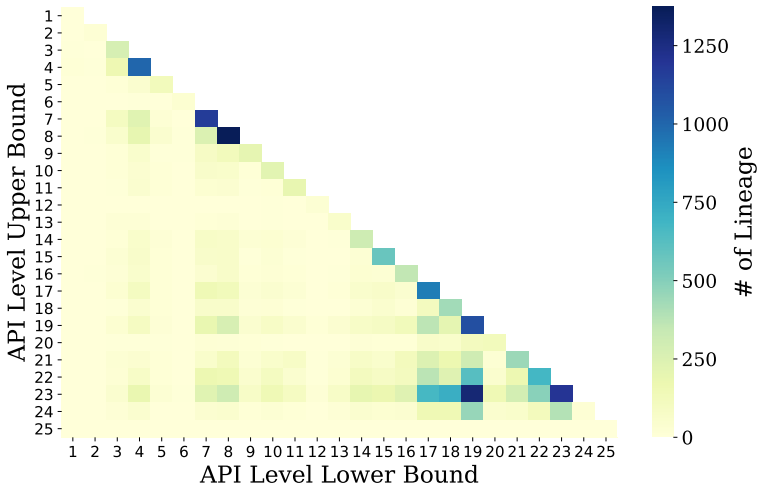


Figure 3.6: API Level Span Distribution of Lineages

3.4 Experiment Setup

To set up the empirical experiments related to the complexity evolution of Android apps, we present the main research questions this work explores and the computation of the metrics this work stands upon in Section 3.4.1 and Section 3.4.2, respectively.

3.4.1 Research Questions

Our objective is to understand the evolution of Android apps' complexity and hence to empirically observe practical insights for guiding the evolution of Android apps towards engineering more reliable apps. To fulfill this objective, we plan to perform an exploratory study to answer the following research questions:

- **RQ1:** How does the code of Android apps generally evolve? As the first research question, in order to have an overall understanding of the general evolution of Android apps, we empirically investigate the changes in terms of code size (i.e., DEX size and class number) of Android apps over time.
- **RQ2:** How does the complexity of Android apps evolve as time goes by? The complexity evolution within this research question will be investigated year by year. For each app lineage, we choose one app version for each year: the latest one released in that year. The chosen apps from the same year (different lineages) will be considered as a whole and the extracted metric values will be leveraged to represent app complexity of that year.
- **RQ3:** How do Android API level updates impact on app complexity? Android framework is recurrently updated to introduce new features or fix critical bugs. To benefit from these updates, Android apps need to be correspondingly changed. Hence, the complexity evolution within this research question will be investigated based on the targeted API levels of the considered apps.
- **RQ4:** What are the patterns of complexity evolution? By defining feature patterns, the evolution of complexity will be investigated in the manner of individual app lineage. Then, how Android apps evolves normally as well as what is the uncommon pattern during complexity evolution can be spotted.

3.4.2 Metrics Computing

In this work, the metric values are computed at the *smali* code level. All the considered lineage apps are disassembled by *Apktool*, a well-known static analysis tool for reverse engineering Android APK files.[4] Apktool will translate the executable part of an app, namely the *DEX* bytecode into the so-called *smali* code.

Because the considered complexity metrics are measured at class levels, while Android apps normally are made up of multiple classes, for a given metric, we regard its value for a given Android app as the median and standard deviation value among that of all the classes of the app. Statistically speaking, these two values have characterised the majority of the sample population (i.e., median) and their spreads (i.e., standard deviation). Indeed, for a certain app, the median value can represent the app in most of its classes while the standard deviation reflects the extent the complexity of the classes can go, e.g., either better or worse.

In this work, we rename these two values (median and standard deviation) as ***feature*** and ***variation***[5], which are explained as follows:

Given an app a, $C = \{c_1, c_2, \ldots, c_n\}$ is the set of its classes, for a certain metric m, the value of c_i is $v_m(c_i)$, where $c_i \in C$, then

- ***feature*** value: $feature(a) = M$, where M is the median value of $\{v_m(c_1), ..., v_m(c_n)\}$.
- ***variation*** value: $variation(a) = \sigma$, where σ is the standard deviation of $\{v_m(c_1), ..., v_m(c_n)\}$.

During our experiments, we have found that the *android.support* package has been widely presented in some Android apps. Since this package is provided by Google as an official library for resolving issues such as compatibility[6], we do not take this package into consideration when computing the values of metrics.

It is also worth to mention that not all lineage apps can be successfully reverse engineered by our tool for computing the values of our selected metrics. The main reasons led to the failures are 1) *Apktool* crashes due to exceptions such as no *smali code* generated, (2) null values are returned by

[4] https://ibotpeaches.github.io/Apktool/
[5] The rationale behinds this renaming is to avoid confusions about expressions such as "median of the median values".
[6] https://developer.android.com/topic/libraries/support-library/index.html

our tool because the number of classes is too small (e.g., less than three for some app versions) or there is no field defined by some classes (i.e., this will lead to null value for metric LCOM). Moreover, since date information is also important to this study, (e.g., we leverage it to perform the year-based evolution study), we further remove such app lineages that have incomplete DEX date associated, i.e., we cannot extract a validated assembly time from the app.

To conclude, among the 28,559 lineages (464,649 app versions in total), 1,389 app lineages (23,451 apps versions) that have confronted the aforementioned issues are ignored in this study. In other words, our study is conducted based on 27,170 app lineages (441,198 app versions).

3.5 Results

We now present our investigation details towards answering the aforementioned research questions.

3.5.1 RQ1: General Evolution of Android Apps

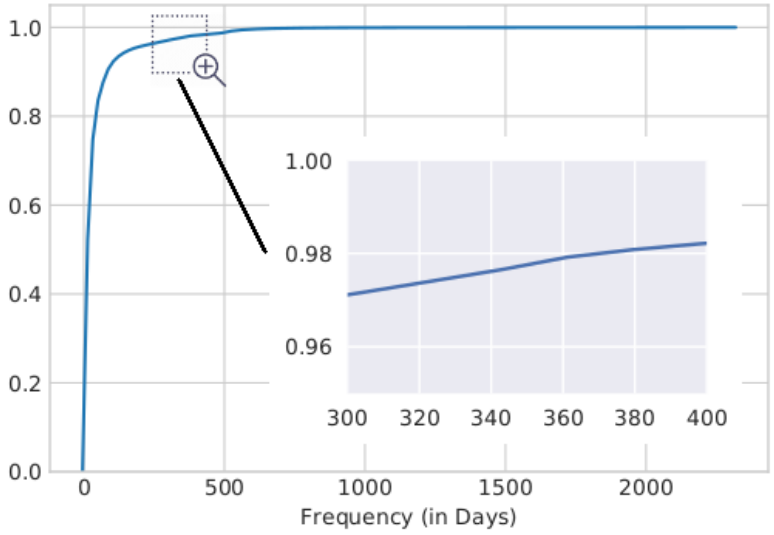
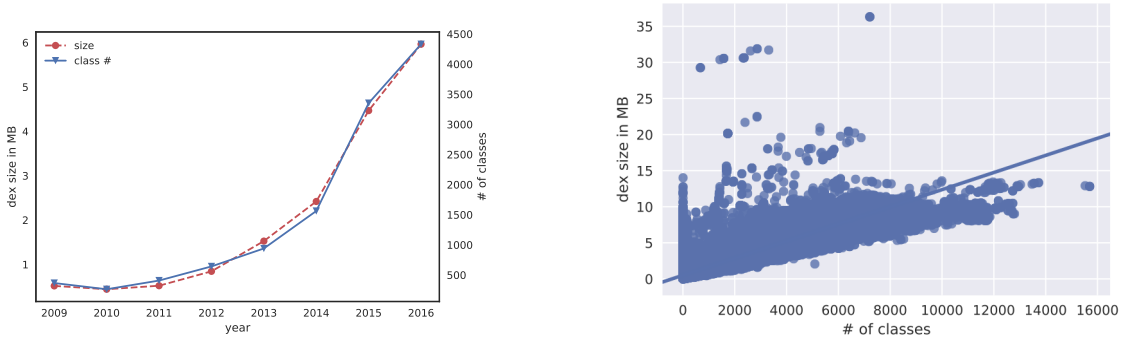


Figure 3.7: Cumulative distribution function (CDF) of the update frequency of selected lineage apps. Given a frequency (e.g., $x = 365$ days), the probability for an app to have an update within $x = 365$ days can be quickly observed from the CDF (i.e., the corresponding value in the Y-axis).

Since it is non-trivial to select the time interval for re-aligning lineage apps, we resort to a simple empirical study to select such time interval. The study looks into the update frequency of all the selected lineage apps. Figure 3.7 illustrates the Cumulative Distribution Function (CDF) of the update frequency, where the frequency is counted in days (as shown in the X-axis). For about a year (e.g., 365 or 366 days), more than 95% of considered apps have been updated at least once, presenting a great time interval to build our time-based evolution dataset. Therefore, we select a year as the time interval to investigate the complexity evolution of Android apps.

To understand the general evolution of Android apps, we first look into the evolution of size and the number of Java classes of Android apps. Figure 3.8a shows how are the median value of app size and the number of classes evolved as time goes by. For each median value, it is calculated based on all apps of that year.



(a) App Size and # Classes Evolution in Time

(b) Scatter Plot of App Size and # Classes

Figure 3.8: Statistics of App Size and # Classes

Quite clearly, both app sizes and class numbers were increasing, especially, from 2014 to 2015, the rise was dramatic. This evidence suggests that Android apps become bigger and bigger in both size and the number of classes.

Furthermore, as demonstrated in Fig. 3.8b, there is also a strong correlation between app size and the number of classes. This strong correlation is also confirmed to be statistically significant via the Pearson's correlation coefficient ($\rho > 0.9$, showing a strong positive correlation). This strong positive correlation implies that app updates are more likely to add new classes than simply add codes to existing classes. Indeed, in our selected app lineages, 80.73% of them have their apps eventually become bigger (comparing the last app version with the first one) in terms of the number of classes, while the percentage is even higher when talking about the app size: 83.4% of selected lineages.

As app size and the number of classes getting bigger and bigger, intuitively, apps are becoming more complex and more difficult to maintain. Consequently, a detailed study on app complexity evolution is expected. This research question actually motivated us to perform an in-depth analysis of the complexity evolution of Android apps.

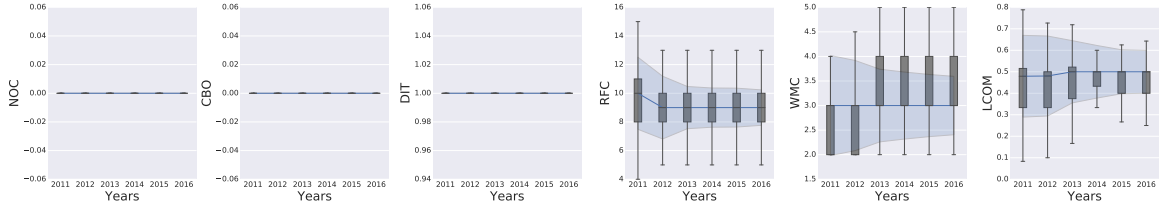
> During the evolution, app developers are more likely to introduce new classes rather than adding code to existing ones, as shown by the strong correlation of changing in app size and number of classes.

3.5.2 RQ2: Complexity Evolution via Time

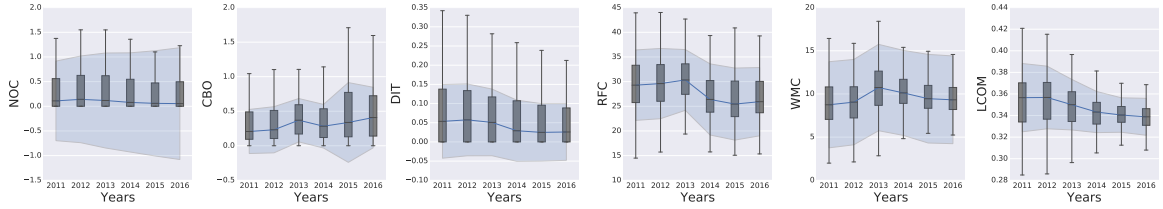
We investigate the complexity evolution of Android apps via their release time[7]. State-of-the-art approaches for time-based evolution normally choose random apps for different time-points. As a result, the apps chosen in different time-points could be different. On the contrary, our lineage based time evolution approach is expected to always select app versions from same app lineages. By doing this, the consistency of samples between different time-points can be well reserved, which makes the final result more reliable. To support this kind of investigation, we need to re-construct a fine-grained dataset where the considered lineage apps are aligned via time. To this end, we re-align our lineage apps by selecting the last app version of each year from each app lineage.

Figure 3.9a presents the evolution of the metrics feature value from 2011 to 2016. The median value of metrics NOC, DIT, WMC and CBO exhibit as horizontal lines with very low values, which indicates that app complexity in terms of these 4 metrics has kept very low and constant for past 6 years.

[7]Since AndroZoo does not collect the release date metadata for Android apps, and it is virtually impossible to retrieve such metadata for previous app versions, as these metadata have already been overwritten by the data of updated app versions, in this work, we consider the assemble DEX time as the app release time.



(a) Feature Values



(b) Variation Values

Figure 3.9: Lineage based evolution with the central lines depict the trend of median value while the ribbons show the changing of standard deviation and the boxplots illustrate the distribution of each year

These values tell us that for most classes of an app, they were not sub-classed (i.e., 0 of NOC), they had shallow inheritance trees (i.e., 1 of DIT), their method complexities are low (i.e., 3 of WMC) and they were not coupled with other classes (i.e., 0 of CBO). Regarding the standard deviations shown as the ribbons, NOC, DIT and CBO show no ribbon at all while WMC shows an observable ribbon with a narrow-down trend. Since the standard deviations reflect the difference between different apps, we can say that the vast majority of apps exhibit no difference of complexity in terms of NOC, DIT and CBO. Meanwhile, in terms of WMC, there are apps with different feature values, but the difference is not big (± 1 on average) and getting smaller.

On the other hand, RFC and LCOM show more changes as they evolving. The drop of RFC in 2012 indicates a clear improvement of this metric for most of the apps. While a slight deterioration of LCOM happened in 2013 can be observed as well. Furthermore, the difference between apps in these 2 metrics was getting narrower too.

Because the feature values only reflect app complexity in major situations as explained in Section 3.4.2. To have a more comprehensive understanding of apps complexity evolution, we further resort to an investigation into the evolution of variation values of Android apps.

Figure 3.9b shows the evolution of app variation values. From the median value perspective, 4 of the metrics show a clear decline trend which are NOC, DIT, RFC and LCOM. While for CBO and WMC, they were slightly increased over the years. As the variation of a metric measures the differences of the metrics among different classes of an app, Thus, a decreasing in trend is preferred. On the other hands, the differences of variation values between different apps are exhibited by the ribbons in the figure. Therefore, for past 6 years, the differences of NOC and CBO have been increased while DIT and LCOM have been decreased. For RFC and WMC, the differences kept almost the same.

As time goes by, the complexity in terms of RFC has been mitigated but deteriorated in LCOM. Out of the six metrics, nature updates (update via time) have only impacted these two metrics, although the impacts are quite limited. It is worth to highlight that the complexity difference between different apps is getting closer during the evolution.

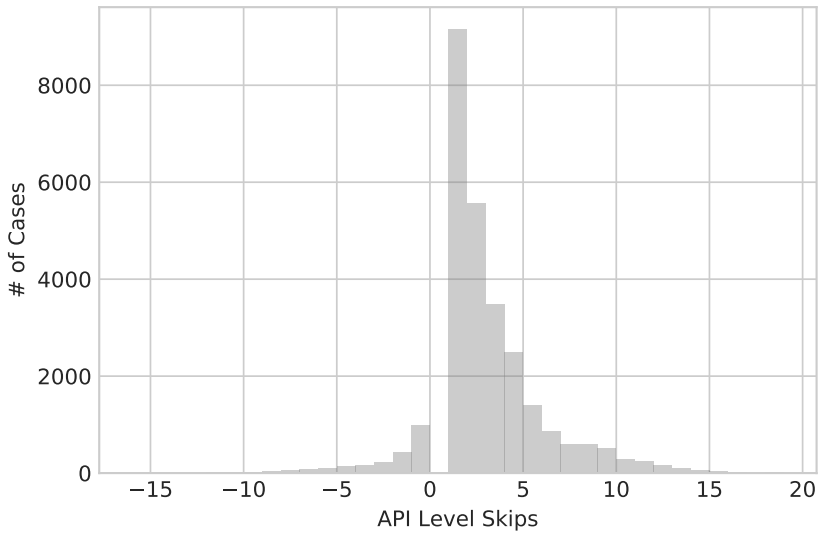


Figure 3.10: Distribution of API level skips for every two adjacent app versions while $x = 0$ is not shown as it means no skip

3.5.3 RQ3: Complexity Evolution via API Level

In order to understand the possible impact of API levels on the evolution of app complexity, we conduct another study that specifically looks into the complexity difference between such apps that target different Android API levels. To this end, for each app lineage, we pair up adjacent app versions, which target different API levels, for difference examination. Given a pair of app versions (a_i, a_{i+1}) and their targeted API levels (L_i, L_{i+1}), we define *level skip* as the difference between the two targeted API levels (i.e., $\textit{level skip} := L_{i+1} - L_i$). Figure 3.10 illustrates the distribution of API level skips summarized from our app lineage dataset. The level skips vary from -16 to 19 while the majority app pairs fall into the category of level skip equals to 1, followed by 2 and 3 skips respectively. The reasons causing minus level skips could be: (1) to support previous users or features requiring old API, (2) version code assigned reversely, (3) some other unknown purposes. As these cases are rare and abnormal, they will not be considered here. In this work, we take into account all the app pairs that have API level skip between 1 and 3. Based on this criterion, we form a new dataset containing three types of app pairs: S1, S2 and S3 for app pairs with one, two and three level skips, respectively,

Figure 3.11 illustrates the distribution of feature value differences of app pairs via level skip. Since metrics NOC, DIT, WMC and CBO are quite stable during the evolution of Android apps, as shown in Section 3.5.2, we only present the distribution of metrics RFC and LCOM in the figure.

Interestingly, the median values stay closely to 0 suggests that the changes are quite small despite the targeted API level is updated. The fact that the major parts of the boxes fall into the negative side of Y-axis and larger level skips seem to yield larger ranges of the negative parts indicates that the changes do not seem to increase the app complexity (at least for RFC and LCOM).

Figure 3.12 illustrates the distribution of variation value differences via level skip. Similarly, except for metric WMC, where the median values are generally decreasing when level skip increases, the median values of other metrics are very close to 0. Regarding the body of the boxes, the major parts for metrics DIT, WMC, RFC and LCOM fall into the negative side of Y-axis while for metrics NOC and CBO, they fall into the positive side. The body size is increased as level skip increasing. For

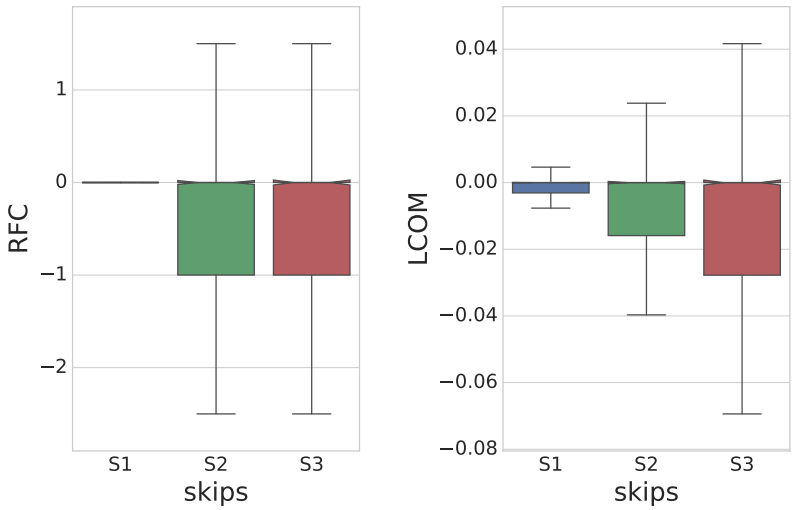


Figure 3.11: API Level based Evolution of Feature Values

NOC and CBO, they increase mainly on the positive side of Y-axis. But the rest four metrics increase mainly on the negative side.

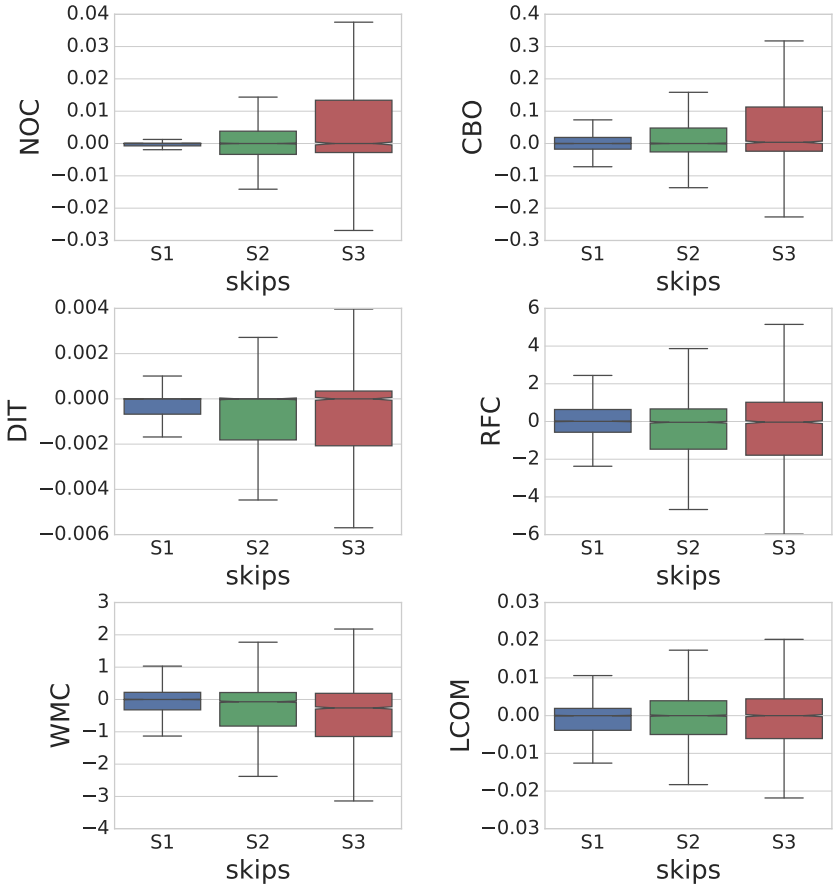


Figure 3.12: API Level based Evolution of Variation Values

As the interpretation of these patterns, updates of API levels may effect on different metrics differently. A bigger level skip normally causes a larger increase in the complexity difference within an app from the aspect of NOC and CBO (remind that the definition of variation value in Section 3.4.2). However, from the aspect of DIT, WMC, RFC and LCOM, the complexity difference shrinks mostly.

> API level updates could cause the complexity of Android apps to decrease, although the extent is quite limited. Also, for most of the metrics, API level updates shrink the complexity difference within apps.

3.5.4 RQ4: Patterns of Complexity Evolution

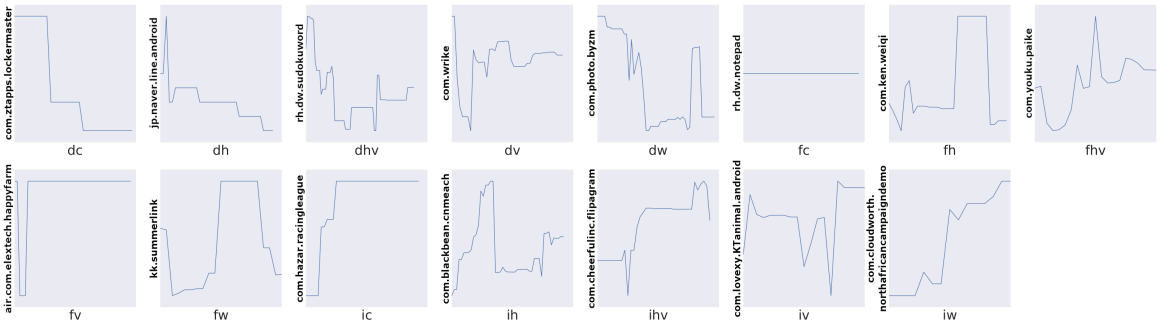


Figure 3.13: Real world examples of patterns of complexity evolution with the name of patterns as x-axis labels and application names as y-axis labels

We remind the readers that in this work we use lineage apps rather than randomly selected apps to investigate the complexity evolution of Android apps, where the dataset allows us to have a deeper look at how each app lineage evolves. Consider an app lineage with n app versions $app_1, app_2, \ldots, app_n$. Let set $M = \{m_1, m_2, \ldots, m_n\}$ stand for the feature or variation values of a metric of these app versions and σ be the standard deviation of set M. The possible evolution patterns that each app lineage may fall into are defined as follows:

- overall patterns
  - flat: $|m_1 - m_n| < \sigma$, which means the difference between the first and last app version is less than the standard deviation of the app lineage.
  - decrease: $m_1 - m_n \geq \sigma$, which means the value of the first app version is greater than the value of the last one and the difference is bigger than the standard deviation.
  - increase: $m_n - m_1 \geq \sigma$, which means the value of the last app version is greater than the value of the first one and the difference is bigger than the standard deviation.
- detail patterns
  - constant: patterns between adjacent app versions are consistent with the overall pattern. For flat pattern, it means $m_i = m_j$. For decrease pattern, $m_i \geq m_{i+1}$. For increase pattern, $m_i \leq m_{i+1}$. Where $i, j \in \{1, \ldots, n\}$.
  - hill: $\max M \in \{m_2, \ldots, m_{n-1}\}$ and $\max M - \max\{m_1, m_n\} > \sigma$, which means maximum value happens in an app version which is not the first or the last app version. Additionally, the maximum value needs to be greater than the maximum value of the first and the last app version and the difference need to be bigger than the standard deviation of the app lineage.
  - valley: this is the opposite situation of hill and it expresses as $\min M \in \{m_2, \ldots, m_{n-1}\}$ and $\min\{m_1, m_n\} - \min M > \sigma$

– wave: other cases where no constant, hill and valley patterns can be observed.

The overall patterns are the patterns defined by the starting and ending points. They are designed to give a brief concept of what is the evolution trend. While detail patterns are meant to reflect the feature patterns during the evolution. To give a complete evolution pattern, one of the overall patterns combined with one or two detail patterns is required and the possible combination patterns are shown in Table 3.2. Figure 3.13 further shows the real world examples of each defined pattern from our dataset.

Table 3.2: Possible Patterns & Abbreviation

Pattern	**Abbreviation**
Flat Constant	*fc*
Increase Constant	*ic*
Decrease Constant	*dc*
Flat Wave	*fw*
Increase Wave	*iw*
Decrease Wave	*dw*
Flat Hill	*fh*
Increase Hill	*ih*
Decrease Hill	*dh*
Flat Valley	*fv*
Increase Valley	*iv*
Decrease Valley	*dv*
Flat Hill & Valley	*fhv*
Increase Hill & Valley	*ihv*
Decrease Hill & Valley	*dhv*

According to the patterns defined, we analyze each app lineage to obtain their evolution patterns and then calculate the frequency of each pattern. The final result is displayed by a heat map in Figure 3.14. Likewise, frequencies of NOC, DIT and CBO feature values are removed from the figure because all their values keep constant (cf. Section 3.5.2).

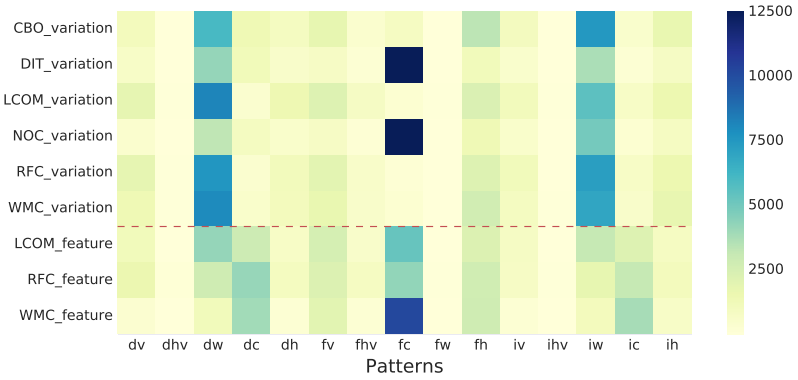


Figure 3.14: Frequency distribution of patterns of complexity evolution with feature and variation value parts divided by a red horizontal dash line

Regarding the evolution pattern summarized via feature values (the part under the red horizontal dash line), undoubtedly, *fc* is the most evident column, followed by column *dc* and *ic* sequentially. On the other hand, in the variation value part (the part above the red horizontal dash line), although the column of *fc* is still noticeable, there are only 2 tiles (which are DIT and NOC) with a very dark color, while the rest tiles are quite bright. Meanwhile, column *dw* and *iw* are also very distinguishable and with much even darkness. Moreover, 3 brightest columns are also spotted which are *dhv*, *fw* and *ihv* and both median value and standard deviation parts are consistent in these 3 brightest columns.

So far, we can observe that the complexity of major Android apps tends to stay constant (do not forget that the median values of the 3 removed metrics are even more constant). Nonetheless, there are still many apps tend to decrease constantly in complexity while others may increase constantly. However, except for metrics DIT and NOC, where the evolution pattern of most apps is still, the complexity difference within an app lineage is more likely to either decrease or increase wavily.

> According to the patterns of complexity evolution, wavily increasing and decreasing have dominated the trend of complexity difference during the evolution of Android apps. This empirical evidence suggests that app developers might not really be aware of controlling the complexity of their apps.

3.6 Discussion

In this section, we discuss several implications that this study can lead to and disclose the potential threats to validity.

3.6.1 Implication

Towards Engineering Better Metrics Our experimental results suggest that app complexity does not significantly change during app updates. This evidence can be explained by the fact that, as shown in Section 3.5.1, Android app updates usually do not simply add codes to existing classes but are more likely to add new classes. Unfortunately, the six metrics we used in this study are all based on classes. They might not be representative to fully capture the complexity of Android apps. Therefore, we argue that there is still a lot of space to improve towards engineering better metrics for characterizing the development of mobile apps. To this end, designing a new set of complexity-related metrics (e.g., to take into account invocation chains) is needed. Moreover, neglecting the complexity conducted by the interaction between classes is not reasonable, so comprehensive application level metrics are also needed.

Best Practice to Guide Future Quality Evolutions Generally, preserving and improving software quality is a long-time challenge that is difficult to resolve. Due to software aging, without active countermeasures, the quality of applications slowly degrades during their evolutions [20, 39]. As argued by Mens et al., there is a need to provide tools and techniques that preserve or even improve the quality characteristics of software systems [40]. In this study, around 9% of our selected app lineages are always in line with that of the mainstream. For our future work, good practices could be learned based on these apps. If so, we subsequently present automated tools to apply the obtained good practices, e.g., by instrumenting directly the bytecode of Android apps [41].

Observing differences between developer capabilities Since an Android app is likely developed by multiple developers, who might have different abilities to control the quality of their implemented code, we believe that standard deviation value could be a good means to capture the differences among developers in a team which can further provide insights to optimize development teams.

3.6.2 Threats to Validity

The study conducted in this work has presented several threats to validity.

First, the considered six metrics may not be fully representative of the quality of Android apps. For example, compared to the six metrics proposed by Jost [32], we have missed four of them although have additionally considered 2 metrics. Also, many metrics are highly correlated with others. Hence, as suggested by Mourad et al. [42], there is a need to invent new quality metrics that attempt to unify similar metrics so as to simplify further analysis and make interpretation concise. We consider this as our future work.

For an app lineage, some versions could be missing without our awareness. Also, the order of app versions may not be correct if the version code in the manifest is assigned randomly. However, the impact of missing versions on this study is limited while given random version is not common practice.

Finally, our time-based evolution study is at year level, although we have empirically shown that a year is actually a reasonable interval, it might still be too long for this study as in practice popular apps are updated more frequently. To mitigate this potential threat, we plan to design and implement a generic framework for supporting more advanced evolution analyses of mobile apps, where different parameters such as time interval, level skips and metrics can be easily configured and adjusted.

3.7 Related Work

Various studies have investigated the problem of observing reliable metrics for characterizing the quality of mobile apps. Chidamber and Kemerer [31] introduce six metrics for guiding the design of object-oriented programs and four of them have been considered by Jost et al. [32] and hence by this work. Thomas McCabe [43] further introduced Cyclomatic Complexity (CC) for measuring the complexity. Fenton and Neil [44] argued that the future for software metrics lies in using them to develop decision-support tools to support risk assessment.

Several researches have focused on quality metrics related to Android apps. Tian et al. [45] investigated the characteristics which make Android apps high-rated. They found that metrics such as app size, target SDK version are influential factors contributing to the success of Android apps. Protsenko et al. [46] also leverage software metrics to detect Android malware. Experimental results show that software metrics are reliable for distinguishing malware and resilient against common obfuscation.

3.8 Summary

Evolution studies are important for assessing software development process and measure the impact of different practices. However, such studies, to be meaningful, must scale to the size of the artifact. For Android apps, this was so far a challenge due to the lack of significant records on market apps. Our work first addresses these challenges by re-constructing 28,564 lineages formed in total by 465,037 apks. We have then conducted a large-scale empirical study of the complexity evolution of Android apps. We select six metrics that have been successfully leveraged by literature works for quantifying the complexity of Android apps. Based on the evolution of these six metrics, we eventually find that (1) Android apps usually become bigger during their evolutions and updates are tend to add new classes, (2) nature updates do not really impact on the complexity of Android apps, (3) the update of Android framework could mitigate app complexity but very limited, (4) complexity evolution is more like to wavily increase or decrease.

4 Understanding the Evolution of Android App Vulnerabilities

In this chapter, we explore the constructed app lineages to investigate the presence of vulnerabilities and their evolution across app versions. By leveraging state-of-the-art static vulnerability detection tools, we extract a comprehensive dataset of reported vulnerable pieces of code in real-world apps. Several findings from the analyses, such as some vulnerabilities reported by detection tools may foreshadow malware are also reported.

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

- J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, pages 1–19, 2019

Contents

4.1 Overview

Mobile software has been overtaking traditional desktop software to support citizens of our digital era in an ever-increasing number of activities, including for leisure, internet communication or commerce. In this ecosystem, the most popular and widely deployed platform is undoubtedly Android, powering more than 2 billion monthly active users, and contributing to over 3 million mobile applications, hereinafter referred to as *apps*, in online software stores [48]. Yet from a security standpoint, the Android stack has been pointed out as being flawed in several studies: among various issues, its permission model has been extensively criticized for increasing the attack surface [2, 3, 4]; the complexity of its message passing system has led to various vulnerabilities in third-party apps allowing for capability leaks [5] or component hijacking attacks [6]; furthermore, the lack of visible indicators[1] for (in)secure connections between apps and the internet is exposing user communication to Man-In-The-Middle (MITM) attacks [49].

Vulnerabilities of mobile apps, in general, and of Android apps, in particular, have been studied from various perspectives in the literature. Security researchers have indeed provided comprehensive analyses [50, 51, 52, 53, 54] of specific vulnerability types, establishing how they could be exploited and to what extent they are spread in markets at the time of the study. The community has also contributed to improve the security of the Android ecosystem by developing security vulnerability finding tools [55, 56, 57, 58] and by proposing improvements to current security models [59, 60, 61, 62]. Although advanced techniques have been employed by malicious developers such as packer and obfuscation, countermeasures such as [63, 64, 65] have also been proposed. Unfortunately, whether these efforts have actually impacted the overall security of Android apps, remains an unanswered question. Along the same line of questions, little attention has been paid to the evolution of vulnerabilities in the Android ecosystem: which vulnerabilities developers have progressively learned to avoid? have there been trends in the vulnerability landscape? Answering these questions could allow the community to focus its efforts to build tools that are actually relevant for developers and market maintainers to make the mobile market safer for users.

Investigating the evolution of vulnerabilities in Android apps is however challenging. In the quasi-totality of apps available in the marketplace, the history of development is a fleeing data stream: at a given time, only a single version of the app is available in the market; when the next updated version is uploaded, the past version is lost. A few works [13, 12, 10] involving evolution studies have proposed to "watch" a small number of apps for a period of time to collect history versions. However, the insight observed by such studies may not be representative of that of the whole Android ecosystem.

In this study, we set to perform a large scale investigation on how vulnerabilities evolve in Android apps. We fully rely on static vulnerability detection tools and report their results on consecutive versions of Android apps. We refer the reader to the discussion section on false positive detections by the state-of-the-art tools that were used. Our contributions are as follows:

- We apply state-of-the-art static vulnerability finding tools on all app versions and record the alerts raised as well as their locations. We investigate specifically 10 vulnerability types associated to 4 different categories related to common security features (e.g., SSL), its sandbox mechanism (e.g., Permission issues), code injection (e.g., WebView RCE vulnerability) as well as its inter-app message passing (e.g., Intent spoofing). Correlating the analysis results for consecutive app pairs in lineages, we extract a **comprehensive dataset of reported vulnerable pieces of code in real-world apps**, and, whenever available, the subset of changes that were applied to fix the vulnerabilities.
- We perform **several empirical analyses** to (1) highlight statistical trends on the temporal evolutions of vulnerabilities in Android apps, (2) capture the common locations (e.g., developer vs

[1] e.g., padlock and HTTPS in url input field

library code) of vulnerable code in apps, (3) comprehend the vehicle (e.g., code change, new files, etc.) through which vulnerabilities are introduced in mobile apps, (4) investigate via correlation analysis whether vulnerabilities foreshadow malware in the Android ecosystem.

And the main findings are:

- Most vulnerabilities will survive at least 3 updates.
- Some third-party libraries are major contributors to most vulnerabilities detected by static tools.
- Vulnerability reintroduction occurs for all kinds of vulnerabilities with *Encryption*-related vulnerabilities being the mostly reintroduced type in this study.
- Some vulnerabilities reported by detection tools may foreshadow malware.

Noticeably, this is the largest scale Android vulnerability study so far. Meanwhile, we novelly analyze vulnerabilities from the aspect of app lineages and certain patterns (e.g., vulnerability reintroduction) are firstly spotted in this study.

The artifacts of our study, including the constructed app lineages as well as the harvested vulnerability detection tool reports, are made publicly available to the community in the following anonymous repository:

`https://avedroid.github.io`

The remainder of this chapter is organized as follows: Section 4.2 describes the experimental setup, including an introduction of the vulnerability finding tools, and the research questions. Section 4.3 unfolds the empirical analyses. Section 4.4.1 discusses some promising future works. Section 4.4.2 enumerates threats to validity, while Section 4.5 discusses related work and Section 4.6 concludes this work.

4.2 Study Design

In this section, we first define and clarify some terms used in the study. Second, we provide some background information on the security vulnerability detection tools that we leveraged in Section 4.2.2. Then, we outline the research questions as well as the motivations behind them (cf. Section 4.2.3). Finally, we talk about the experimental setup in Section 4.2.4.

4.2.1 Terminology

A **vulnerability location** l is specified by the class and method in which the vulnerability is spotted in an apk by a vulnerability detection tool.

Vulnerability Reintroduction is to check whether fixed vulnerabilities reappear in app lineages. For a certain type of vulnerability v, we denote that a vulnerability v is found at location l as v_l. if $\exists i, j, k | 1 \leq i < j < k \leq n$, where v_l is found in apk_i and apk_k, but not in apk_j. v_l is said to be reintroduced at location l. Moreover, if v is found in apk_i and apk_k but not in apk_j, then we say vulnerability type v is reintroduced.

4.2.2 Vulnerability Scanning Tools

Vulnerabilities, also known as security-sensitive bugs [66], could be statically detected based on rules modeling vulnerable code patterns. They are typically diverse in the components that are involved, the attack vector that is required for exploitation, etc. In this work, we focus on selecting common vulnerabilities with a severity level that justifies that they are highlighted in security reports and in previous software security studies. Before enumerating the vulnerabilities considered in our work, we describe the vulnerability detection tools (detection tools for short hereafter) that we rely upon to statically scan Android apps.

We stand on three state-of-the-art, open source and actively used detection tools: FlowDroid, AndroBugs, and IC3.

- **FlowDroid** [55] – In the literature on Android, FlowDroid has imposed itself as a highly reputable framework for static taint analysis. It has been used in several works [56, 67, 68] for tracking sensitive data flows which can be associated with private data leaks[2]. The tool is still actively maintained [69]. Moreover, since the original version of FlowDroid can only analyze intra-component data flows. In order to further consider the inter-component data flows, in this work, we used an ICCTA [56] enhanced version of FlowDroid.
- **AndroBugs** [70] was first presented at the BlackHat security conference, after which the tool was open sourced [71]. This static detection tool was successfully used to find vulnerabilities and other critical security issues in Android apps developed by several big players [72]: it is notably credited in the security hall of fame of companies such as Facebook, eBay, Twitter, etc.
- **IC3** [57] is a state-of-the-art static analyzer focused on resolving the target values in *intent* message objects used for inter-component communication. The tool, which is maintained at Penn State University, can be used to track unauthorized Intent reception [50], Intent spoofing attacks [60], etc.

Table 4.1: List of Considered Vulnerabilities.

Type	Vulnerability checking description	Detection Tool
	Security features	
SSL_Security[73]	SSL Connection	AndroBugs
	SSL Certificate Verification	AndroBugs
	SSL Implementation (Hostname Verifier of ALLOW_ALL_HOSTNAME_VERIFIER)	AndroBugs
	SSL Implementation (Verifying Host Name in Custom Classes)	AndroBugs
	SSL Implementation (WebViewClient for WebView)	AndroBugs
	SSL Implementation (Insecure component)	AndroBugs
Encryption[74]	Base64 String Encryption	AndroBugs
KeyStore[75]	KeyStore Protection	AndroBugs
	Permissions, privileges, sandbox, access-control	
Permission[75]	App Sandbox Permission	AndroBugs
IntentFilter[3]	Unauthorized Intent Reception	IC3
	Injection flaws	
Command[76]	Runtime Command	AndroBugs
	Runtime Critical Command	AndroBugs
WebView[77]	WebView RCE Vulnerability	AndroBugs
Fragment[78]	Fragment Vulnerability	AndroBugs
	Data and Communication Handling	
Intent[79]	Intent Spoofing	IC3
Leak[55]	Sensitive Data Flow	FlowDroid

Table 4.1 summarizes the vulnerability checks that we focus on, in accordance with the capabilities of selected detection tools. Overall, we consider 10 vulnerability types. For AndroBugs, not like other detection tools, it reports on dozens of issues. To focus on those vulnerabilities having a high level of

[2]We remind the readers that FlowDroid is mainly designed for detecting sensitive data flows, which may not necessarily be privacy leaks (e.g., it can be intended behaviours). Nonetheless, since such sensitive data flows indeed send private data outside the device, and it is hard to know how these private data will be used, we consider in this work such sensitive data flows as privacy leaks.

criticality, we only considered the issues which are marked as critical by AndroBugs. Furthermore, several critical issues are also discarded, such as cases where exploitation scenario was not clearly defined (e.g., checking for SQLiteDatabase transaction deprecated) and a few other issues which were not explicitly about executable code (e.g., relevant to Manifest information), to eliminate the share of noise that they can bring to the study.

We now detail the different vulnerabilities and explicate their potential exploitation scenarios. Due to space constraints, we provide actual vulnerable code examples for only a few cases. For other cases, we provide references to the interested reader. Since all apps are collected from markets without source code, we use Soot [18], reverse engineering apps to obtain their code. So the code snippets in the following part are Jimple code, the default intermediate representation of Soot for representing decompiled dex code of real apps.

SSL Security Vulnerabilities related to SSL are a common concern in all modern software accessing the Internet [80, 81, 82, 83, 84]. In its basic form, any access to the Internet using the HTTP protocol without encryption (i.e. without using https), as in the code example in Listing 4.1 line 4, could be subjected to man-in-the-middle (MITM) attacks [49].

```
1 //SSL Connection Checking
2 private void c(Activity, Bundle, IUiListener) {
3     $r6 = new java.lang.StringBuffer;
4     specialinvoke $r6.<init>( "http://openmobile.qq.com/api/check?page=
      shareindex.html&style=9");
5     $r10 = virtualinvoke $r6.toString();
6     $z0 = staticinvoke Util.openBrowser($r1, $r10);
7 }
```

Listing 4.1: SSL Vulnerability Related to Insecure Connection.

In some cases, although the app code is using SSL, the *Certificate Verification* is sloppy, still presenting vulnerabilities. As the example shown in Listing 4.2, the app developer implements the required `X509TrustManager` interface in line 6. Nevertheless, from line 7 to 9, the 3 implemented methods are empty, which only ensures that the app compiles, but creates vulnerabilities for MITM attacks.

```
1 //SSL Certificate Verification Checking
2 class cn.domob.android.ads.r {
3  public void <init>(android.content.Context) {
4   $r3 = new cn.domob.android.ads.r$b;
5 }}
6 class r$b implements X509TrustManager {
7  public void checkClientTrusted(X509Certificate[],String) {}
8  public void checkServerTrusted(X509Certificate[],String) {}
9  public X509Certificate[] getAcceptedIssuers() {return null;}
10 }
```

Listing 4.2: SSL Vulnerability Related to Certificate Verification.

Vulnerability detection tools further ensure that hostnames are properly verified before an SSL connection is created. Vulnerable apps generally accept all hostnames, e.g., by setting hostname verifier with either an *SSLConnectionSocketFactory*.`ALLOW_ALL_HOSTNAME_VERIFIER` or implementation of *HostnameVerifier* interface which overrides *verify* method with a single "`return TRUE;`" statement, creating opportunities for attacks with redirection of the destination host.

Another reported vulnerability, specific to mobile apps, is related to the widespread use of *WebViewClient*. *WebViewClient* is an event handler for developers to customize how should a WebView react to events. For SSL connections, developer suppose to deal with SSL errors within method *onReceiveSslError()* of *WebViewClient*. However, if a developer chooses to ignore the errors when implementing this method, then it introduces a vulnerability to MITM attacks[85].

Finally, still with regards to SSL security, Listing 4.3 illustrates a classic vulnerability where developers bring development test code into production. The well-named *getInsecure* method in line 7 for creating unsafe sockets, when used in a market app, offers immediate paths to MITM attacks.

```
1 //E6: SSL Implementation Checking (Component)
2 class org.jshybugger.ji {
3  public void <init>(Context) {
4   $r2 = new android.net.SSLSessionCache;
5   $r1 = $r0.b;
6   specialinvoke $r2.<init>($r1);
7   $r3 = staticinvoke SSLCertificateSocketFactory.getInsecure(5000, $r2);
8 }}
```

Listing 4.3: SSL Vulnerability Related to Insecure Component.

Command. Android apps can be vulnerable to a class of attacks known as Command injection where arbitrary commands, e.g., passed via unsafe user-supplied data to the system shell, are executed using the *Runtime* API. Such vulnerabilities can appear in unsuspected scenarios: in a recent study, Thomas *et al.* [76] discussed a case where a remote attacker could use a WebView executing dynamic HTML content driven by JavaScript to reflectively call the Java *Runtime.exec()* method for executing underlying sensitive Shell commands such as '`id`' or even '`rm`'.

Permission. The Android application sandbox security feature isolates each app data and code execution from other apps. However, the documentation explicitly recommends to avoid permissions `MODE_WORLD_READABLE` and
`MODE_WORLD_WRITEABLE` for inter-process communication files (i.e., sharing data between applications using files), since, in this mode, Android cannot limit the access only to the desired apps [86]. Nevertheless, the secure alternative of implementing Content Provider may be too demanding for developers, leading to the development of many vulnerable apps.

WebView. The example vulnerability described for the *Command* case reflects a more generalized security issue with WebView's capability to render dynamic content based on JavaScript. Until Android Jelly Bean, i.e., API level 17 (included), JavaScript code reflectively access public fields of app objects. This is problematic since an attacker may leverage this security hole to remotely manipulate the host app into running arbitrary Java code. [77] detailedly described this kind of attacks.

KeyStore. Android relies on the KeyStore API to manage highly sensitive information such as cryptographic keys for banking apps, certificates for virtual private networks, or even pattern sequences or PINs used to unlock devices. Unfortunately, a recent study has confirmed that developers may not use the API very well, opening doors to attacks [75]. In any case, some developers continue to directly hard code certificate information in their app. Others who use the KeyStore end up exposing the so-far secured information by saving the keystore object into an unprotected file, or by loading it into as an ordinary byte array which can then be obtained by attackers.

Fragment. A specific case of code injection can be implemented in apps running earlier versions of the Android OS: *fragment injection*, reported by researchers at IBM [78], exploits the fact that any UI class (i.e., *Activity* extending *PreferenceActivity*) can load any other arbitrary class in a *Fragment* (i.e., sub-Activity). When the UI class is exported (i.e., can be reused by other apps – for example, a mail app may directly allow viewing a PDF attachment by calling a reader app activity), malicious apps can break the sandbox mechanism by accessing information pertaining to the vulnerable apps or abuse its permissions. Roee Hey has demonstrated[3] how this vulnerability could be exploited to attack the Android Settings app to enable an unauthorized and effortless change of device password. Fortunately, this vulnerability was patched starting with Android Kit Kat (API level 19), where all apps including the concerned activities must implement a specific behavior for properly checking the code to be run via the *isValidFragment()* API.

[3] `https://goo.gl/zQnpTq` - Retrieved August 17, 2017

Encryption. It is a standard practice to encrypt sensitive information when they are hard-coded within app code. Unfortunately, developers often confuse *encryption* with simple *encoding*: in both cases, the string may appear unreadable (e.g., in base 64 representation); however simply encoded strings can be decoded by anyone using the standard API without the need of a key. Listing 4.4 illustrates an example of vulnerable code. In line 4, where base 64 encoded information is hardcoded in the program which the developer believes it is safe as it is decoded on-the-fly at runtime, is actually accessible to any attacker.

```
1 //Base64 String Encryption
2 public static byte[] b(byte[]) {
3  byte[] $r0, $r1;
4   $r1 = staticinvoke <android.util.Base64: byte[] decode(java.lang.String,int)>
      ("MDNhOTc2NTExZTJjYmUzYTdmMjY4MDhmYjdhZjNjMDU=", 0);
5   $r0 = staticinvoke <com.tencent.wxop.stat.b.g: byte[] a(byte[],byte[])>($r0,
      $r1);
6   return $r0;
7 }
```

Listing 4.4: Vulnerable Encryption Using Base64 String Encoding

The next two vulnerability cases that we consider are related to the pervasive use of Inter-Component Communication (ICC) for enabling interaction and information exchange between Android app components (within and across apps). Two Android concepts are key in these scenarios: the *intent* object, which is created by a component to hold the data and action request that must be transferred to another component, and the *intentfilter* attribute, which specifies the kind of intents that the declaring component can handle. When intents are *implicit*, i.e., they do not name a recipient component, they are routed by the system to the appropriate components with matching intent filters. Security of intents can then be compromised by malicious apps which may exploit vulnerabilities to intercept intents intended for another, or by sending malformed data to induce undesired behavior in a vulnerable app. These attacks, known as *intent interception* and *intent spoofing* attacks, have been studied in detail in the literature [50, 74, 79, 87, 88].

Intent. Implicit intents, although they provide flexibility in run-time binding of components, are often reported to be overused or inappropriately used [79]. For example, attackers may simply prepare malicious apps with intents matching the actions requested (e.g., PDF reader capability) by vulnerable apps, to divert the data as well as prevent other legitimate components to be launched. In our study, following security recommendations in [79], we consider an app to be vulnerable w.r.t. to *Intent* when it uses implicit intents to communicate with its own components: the developer should have used explicit intent, thus avoiding potential interception by unexpected parties.

IntentFilter. Android apps may declare their capabilities via intent filters. However, when faced with an incoming intent, a component cannot systematically identify which component (trusted or untrusted) sent it. In that case, a vulnerable app may actually be implementing a re-delegation [3] of permissions to perform sensitive tasks. Best security practices require app developers to protect the offered capabilities with the relevant (or some ad-hoc) permissions; thus, the attacker would need the user to grant permission for accessing the sensitive resources he was attempting to abuse. We otherwise consider the app to be vulnerable.

Leak. Sensitive data flows across app components and outside an app have been extensively studied in the literature [55, 56, 89, 90, 91, 92]. When such flows depart from known sensitive *sources* (e.g., API methods for obtaining user private data) and end up in known unsafe *sinks* (e.g., methods allowing to transfer data out of the device by logging, HTTP transferring, etc.), these are privacy leakages. When such data flow paths are found in an app, a vulnerability alert should be raised.

4.2.3 Research Questions

The goal of this work is to explore the evolution of vulnerabilities in the ecosystem of Android apps. Our purpose is to highlight trends in the vulnerability landscape, gain insights that the community can build on, and provide quantitative analysis for support the research and practice in addressing vulnerabilities. We perform this study in the context of Android app lineages, and investigate the following research questions:

RQ1: ***Have there been vulnerability "bubbles" in the Android app market?*** The literature of Android security appears to explore vulnerabilities in waves of research papers. Considering that many of the vulnerabilities described above have been, at some periods, trending topics in the research community [93, 94, 95, 96, 97], we investigate whether they actually correspond to isolated issues in time[4]. In other words, we expect to see the disappearing of some vulnerabilities just like the explosion of bubbles. This question also indirectly investigates whether measures taken to reduce vulnerabilities have had a visible impact in markets.

RQ2: ***What is the impact of app updates w.r.t. vulnerabilities?*** Few studies have shown that Android developers regularly update their apps for various reasons (including to keep up with users' expectations). A recent paper by Taylor *et al.* has concluded that apps do not get safer as they are updated [10]. We do not only investigate the same question with a significantly larger and more diversified dataset, but also find detailed patterns of the survivability of vulnerabilities.

RQ3: ***Do fixed vulnerabilities reappear later in app lineages?*** One of the main reason for software updates is to patch security flaws (i.e., vulnerabilities). Nevertheless, there could be chances for updates to introduce vulnerabilities as well, especially for those that had been fixed in previous updates. With RQ3, we study the phenomenon of vulnerability reintroductions in Android apps.

RQ4: ***Where are vulnerabilities mostly located in programs and how do they get introduced into apps?*** The recent "heartbleed" [98] and "stagefright" [99, 100, 101] vulnerabilities in the SSL library and the media framework have left the majority of apps vulnerable and served as a reminder on the unfortunate reality of insecure libraries [102]. A recent study by Watanabe *et al.* [103] has even concluded that over 50% of vulnerabilities of free/paid Android apps stem from third-party libraries. We partially replicate their study at a larger scale. Furthermore, to help researchers narrow down searching range for vulnerabilities we investigate whether vulnerabilities get introduced while developers perform localized changes (e.g., code modification to use new APIs), or whether they come in with entirely new files (e.g., an addition of new features).

RQ5: ***Do vulnerabilities foreshadow malware?*** Although vulnerabilities do not represent malicious behavior, they are related since attackers may exploit them to implement malware. We investigate whether some vulnerability types can be associated more with some malware types than others. Considering evolution aspects, we study whether some malware apps appear to have been "prepared" with the introduction of specific vulnerabilities.

4.2.4 Experimental Setup

Execution Environment

Experiments at the scale considered in our study are challenging, requiring a significant amount of memory, storage disk as well as computing power. The retrieval of apks from the AndroZoo repository alone took 7 days and occupied 56 terabytes (TB) of local storage space. Among the vulnerability detection tools, FlowDroid and IC3, as previously reported in the literature [67], are heavy in terms of resource consumption. Fortunately, we were able to leverage a high performance computing (HPC) platform [104], using up to 80 nodes, to run as many analyses as possible. We use

[4]We use the Dalvik executable code compilation timestamp as the packaging date to implement this study.

the *fully parallel* capability of the HPC platform and we automated the analysis scenarios with Python scripts: FlowDroid[5] analyses occupied 500 cores and consumed 240 CPU hours to scan 223,474 apks; IC3 occupied 200 cores and consumed 360 CPU hours to scan 72,983 apps. AndroBugs light scanning only took 13 CPU hours with 500 cores to go through 458,814 apks.

Overall, we obtained results for 454,799 apks of 27,974 lineages by AndroBugs, 37,736 apks by FlowDroid and 30,042 apks by IC3 with 3357 and 2048 lineages respectively[6]. The final raw results hold in about 40GB of disk space. There are 2 reasons that caused the different numbers in the result of different tools. One is because of the limited time budget for running analysis. As we know from the previous paragraph that AndroBugs is the lightest tool in resource requirement, we collected the most results from its analysis. On the contrary, IC3 is the heaviest tool which got the least results. Meanwhile, some apks could cause crashing of certain detection tools and, normally, different tools crash on different apks. This is another reason that leads to a different number of analysis results in different tools.

False positives of the selected static analysis tools. It is known that static analyzers will likely yield false positive results. Towards evaluating the severity of this impact, we resort to a manual process to verify some of the results. Because manual verification is time-consuming and may require training in understanding vulnerability types, we restrict ourselves within a working day to conduct the manual verification of a sampled set of vulnerabilities.

Specifically, we invited 2 PhD students who have been working on Android and static analysis related topics to work on the reports of the three selected tools, respectively. One student spent one day on sampled reports [7] of AndroBugs and another one spent two days on the reports of IC3 and FlowDorid respectively. They are able to check 711 vulnerabilities[8] for AndroBugs, 275 vulnerabilities (98 of intent spoofing and 177 of unauthorized intent reception) for IC3 and only 78 leaks for FlowDroid. The manual verification process confirms that, at least from the syntactic point of view (i.e., these vulnerabilities are in conformance with the definition of vulnerabilities as proposed in the tools documentation), the results reported by the adopted static analyzers are all true positive results. The students, however, admit that they are only able to focus on checking simple syntactic rules for validating the results. It is time-consuming and sometimes very hard to follow the semantic data flows within the disassembled Android bytecode. Indeed, Android apps are commonly obfuscated, making it difficult to understand the code manually. Even without obfuscation, it is also non-trivial to understand the intention behind the code if no prior knowledge is applied.

Moreover, in addition to checking real-world Android apps via disassembled bytecode, which is known to be difficult, we conducted another experiment with a set of open-source apps, in the hope that these apps could help us better validate the reported static analysis results. To this end, we randomly selected 200 apps from F-Droid and conducted the same experiments as for the close-source apps. Interestingly, the results of this experiment are more or less the same to that of close-source apps. We have only observed one false positive for FlowDroid. Among the 200 open-source apps, FlowDroid reported that 45 of them contain sensitive data flows. We manually investigated 15 of them (i.e. developers' code14 were manually checked) accounting for 29 leaks. Out of 29 reported leaks from these apps, we spot one false positive, which was found in app *idv.markkuo.ambitsync*. The false positive is caused by an incorrectly generated *dummyMainMethod*. For FlowDroid to construct call graphs for Android apps, a *dummyMainClass* containing several *dummyMainMethods* is required to be instrumented. However, in this case, the *dummyMainMethod* is incorrectly generated which further leads to a non-exist path, and hence a false-positive result. Similar validations were done for Androbugs and IC3 as well, while no false positives were spotted.

[5]Default sources and sinks configuration file provided with FlowDroid source code was used in this study. It can be obtained from the GitHub repository under directory "soot-infoflow-android".

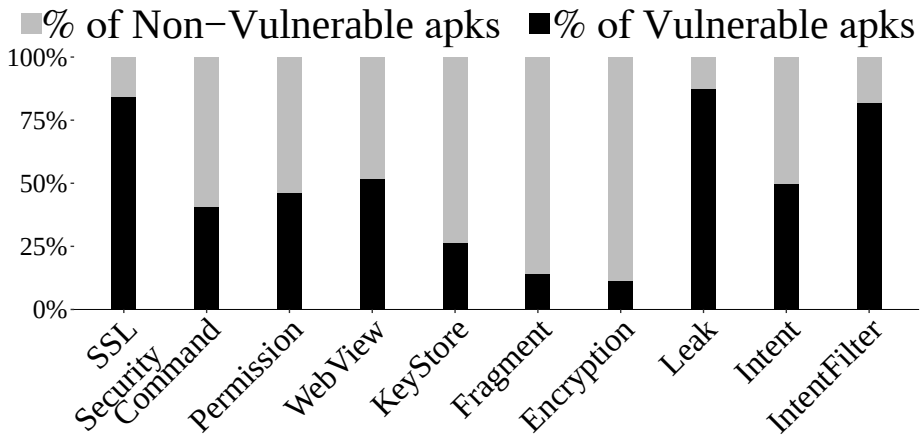
[6]Since the obtained results for different vulnerabilities (i.e. tools) are different, the percentages calculated afterwards are based on the analyzed apks of a certain vulnerability.

[7]Sampling by using *find path/to/reports -type f | shuf -n sampleNumber*

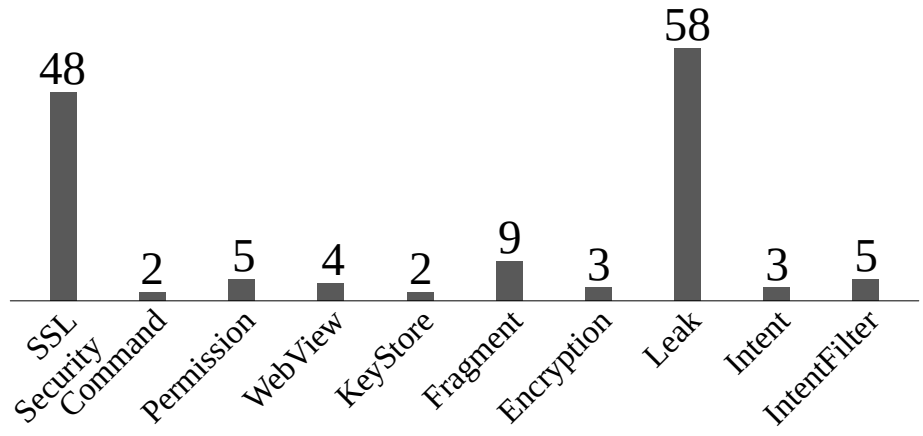
[8]Vulnerabilities of non-HTTPS links are not considered since these vulnerabilities are pervasive in our dataset and are relatively straightforward to identify statically (hence, less likely to be false positives).

Furthermore, it is worth to mention that the three static analyzers we selected in this work have been recurrently leveraged by a significant number of state-of-the-art approaches to achieve various purposes. For example, FlowDroid's results have been leveraged by Avdiienko et al. [67] to mine abnormal usage of sensitive data, Cai et al. [105] leverage IC3 to understand Android application programming and security, while Taylor et al. [10] have leveraged AndroBugs to investigate the evolution of app vulnerabilities. Moreover, there are studies focusing on analyzing and comparing analysis tools too. Qiu et al. [106] compared the three most prominent tools which are FlowDroid, AmanDroid and DroidSafe and discussed their accuracy, performances, strengths and weaknesses etc. Meanwhile, Ibrar et al. [107] studied vulnerability detectors of Mobile Security Framework (MobSF), Quick Android Review Kit Project (QARK) and AndroBugs Framework with banking apps. In the aforementioned two works, they all discussed the false positive issues of the tools. According to their results, FlowDroid and Androbugs both performed the best among their kind of tools in terms of false positives. Therefore, from the false positive point of view, we can conclude that the tools we have chosen are the most reliable among other counterpart tools.

Study Protocol



(a) Proportion of Vulnerable APKs.



(b) Average # of Vulns. per APK

Figure 4.1: Distributions of Vulnerable APKs and of Vulnerabilities

Each of the vulnerability detection tools outputs its results in an ad-hoc format. We build dedicated parsers to automatically extract relevant information for our study. Figure 4.1 provides quantitative details on the distributions of vulnerable apks in the lineages dataset. SSL vulnerabilities are widespread among Android apps and across several apk locations. We also note that a large majority

of apps may include a large number of sensitive data flows. As these leaks reveal private information, although for most of them, how the sensitive data will be used is unknown, we should consider them as vulnerabilities.

For the evolution study, Vulnerable pieces of code are extracted from the location l of an apk indicated by the vulnerability detection tools. These vulnerable pieces of code are collected and released as a valuable artifact for the community. Real-world examples from this artifact were presented in Section 4.2.2.

Finally, we monitor and record how vulnerabilities change at these locations that is: given the analysis results for an apk v_1 and its successor v_2 of a lineage, we track the differences in terms of vulnerability locations; when a given vulnerability type is identified in a location but is no longer reported at the same location, we compute the change diff between the two apk versions and refer to it as *potential vulnerability fix changes*[9].

4.3 Results

We now investigate the evolution of Android app vulnerabilities. Our objective in this work is to understand the evolution of Android app vulnerabilities and thereby to recommend actionable countermeasures for mitigating the security challenges of Android apps.

4.3.1 Vulnerability "Bubbles" in App Markets

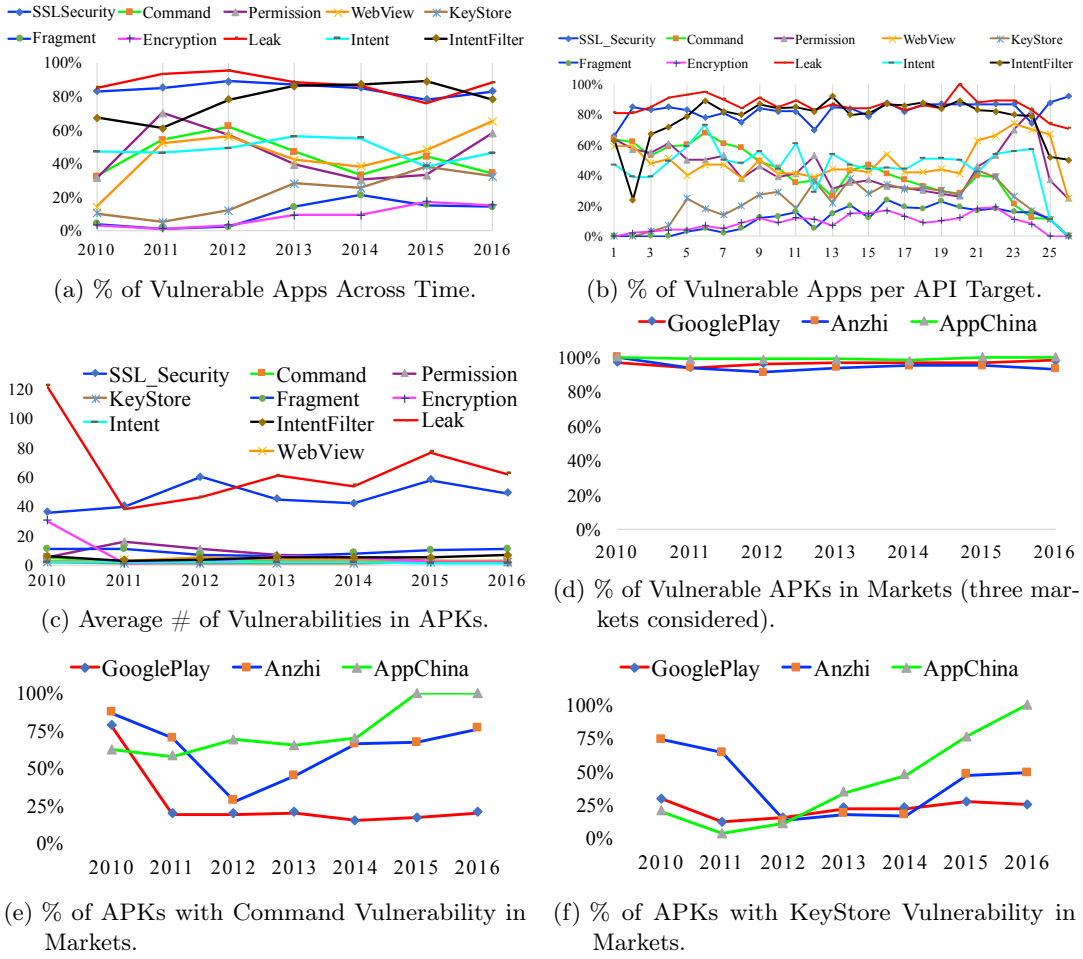
To answer RQ1: ***Have there been vulnerability "bubbles" in the Android app market?*** We first compute, for each vulnerability type, the percentage of apks which are infected in a given year. Figure 4.2a outlines the evolution of vulnerable apks in the space of 6 years. Clearly, we do not see any steady trend towards less and less proportions of vulnerable apks. A more specific investigation is conducted to further explore the expected pattern. The same computation is repeated with apps only debuted on year 2010. This limits to a dataset containing only 3109 apks of 141 app lineages. Nevertheless, very similar patterns have been observed[10]. Since apks are built to target specific Android OS versions (i.e., API level targets), the availability of specific features and programming paradigms may influence the share of vulnerable apks. Thus we present in Figure 4.2b the evolution of the percentage of vulnerable apks across different API level targets. We note an interesting case with the *Command* vulnerability: the percentage of vulnerable apks has steadily dropped from 60% in apks targeting first OS versions to about 10% for the more recent OS version. This evolution is likely due to the various improvements made in the OS as well as in the app markets towards preventing capability and permission abuse.

We further investigate (1) whether the overall evolutions depicted previously break down differently in specific markets, given that markets do not implement the same security checking policies; and (2) whether evolution trends are visible inside the apps, since developers may make efforts to at least reduce their numbers. Figure 4.2d illustrates the evolution of three dominant markets, namely the official Google Play store, and the alternative markets AppChina and Anzhi. We note that the rate of vulnerable apks in all three markets has remained high throughout the considered history[11]. Evolution trends in Figure 4.2c reveal how Leak vulnerabilities have significantly dropped in 2011: from an average 120 vulnerabilities per apk, it came to about 40 before slowly increasing again. We remind the reader that these vulnerabilities found using FlowDroid are computed as possible paths from sensitive data to sinks such as log files. Such a drop in the number of leak vulnerabilities per apk

[9] Since the change could be only related to program refactoring, we cannot say if the change is a real fix or not.

[10] vulnerabilities of *Intent* and *IntentFilter* contains missing data in certain years. Therefore, these 2 vulnerabilities are not discussed here.

[11] A given apk is considered to be vulnerable if it includes any case of our selected vulnerability types.



(a) % of Vulnerable Apps Across Time.

(b) % of Vulnerable Apps per API Target.

(c) Average # of Vulnerabilities in APKs.

(d) % of Vulnerable APKs in Markets (three markets considered).

(e) % of APKs with Command Vulnerability in Markets.

(f) % of APKs with KeyStore Vulnerability in Markets.

Figure 4.2: Evolution of Android App Vulnerabilities.

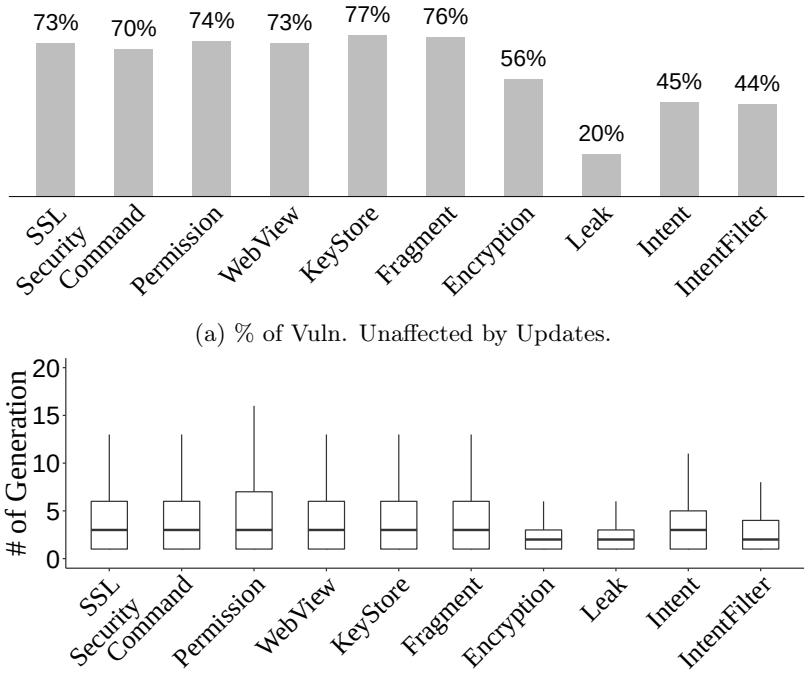
may be explained by the wide interest of the community. TaintDroid [108], the first state-of-the-art tool for tracking data flows has just been proposed, and the first comprehensive study on Android security issues (which put leaks as a priority concern) was made available [109]. MIT technology review had also realized on the wave of apps leaking private info [110].

Figures 4.2e and 4.2f further depict interesting evolution cases for the *Command* and *KeyStore* vulnerability types between markets. While the official Google Play has seen Command-vulnerable apks drop and KeyStore-vulnerability remain low, alternative markets have accepted more and more Command-vulnerable apks, and still include a large share of KeyStore-vulnerable apks. These findings may suggest that the security mechanisms implemented by some markets might be effective against frequently exploited vulnerabilities. Indeed, let us take Google Play as an example, Google has introduced Google Play Protect[12] for continuously pinpointing potentially harmful applications (such as apps with SMS fraud, phishing, or privilege escalation, etc.). As revealed in the Android Security 2017 year in review report, Google Play Protect had actually disabled potentially harmful apps from roughly 1 million devices with approximately 29 million apps removed.

Insights from RQ1: Our analyses did not uncover any vulnerability bubble in the history of app markets. Instead, we note that vulnerabilities have always been widespread among apps and across time. Nevertheless, the case of *Leaks* suggests that wide and intense researching focus can significantly impact the number of vulnerabilities in apps.

[12]https://www.android.com/play-protect/

4.3.2 Survivability of Vulnerabilities



(a) % of Vuln. Unaffected by Updates.

(b) # of Updates before Fix

Figure 4.3: Survivability of Vulnerabilities in APKs

To answer RQ2: ***What is the impact of app updates w.r.t. vulnerabilities?*** We investigate whether a given vulnerability type identified at a location remains or is removed from the successor apk in the lineage. Similarly, we investigate whether new vulnerability types appear in updated versions of the app. Figure 4.3a summarizes the impact that app updates have on the vulnerabilities in a lineage. On average, for most vulnerabilities, more than 50% of vulnerable locations remain. The number of vulnerabilities related to Encryption and Inter-component communication (i.e., Leaks, Intent and IntentFilter) has evolved substantially across app versions (e.g., only 20% of Leak vulnerabilities kept untouched). Figure 4.3b presents the distribution of delay (in terms of apk versions) before a vulnerability is removed from its location. Survivability appears to be similar across vulnerability types. Furthermore, the median delays indicate that most vulnerabilities will not be fixed until three version updates later.

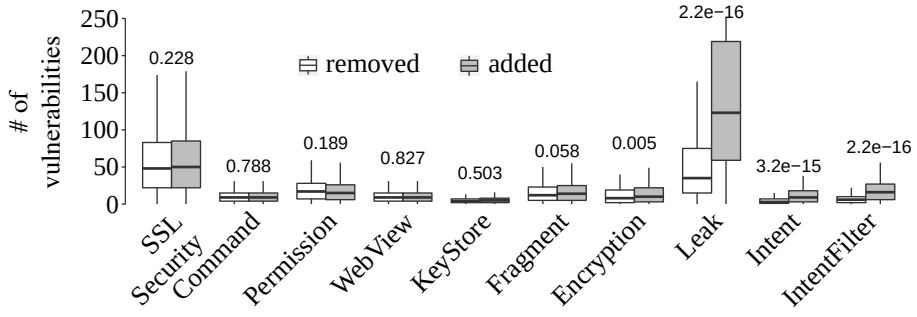
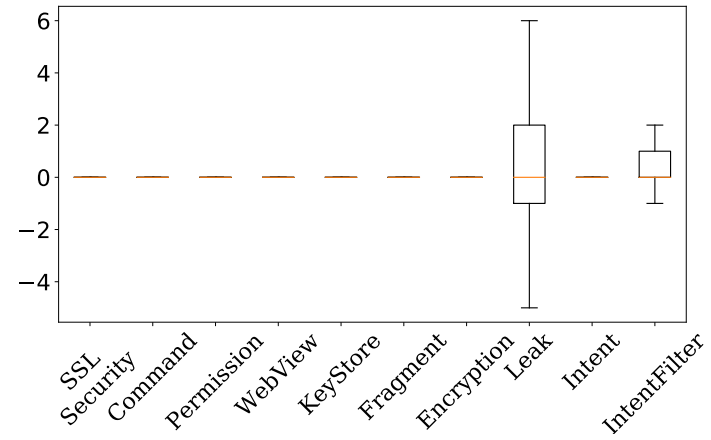
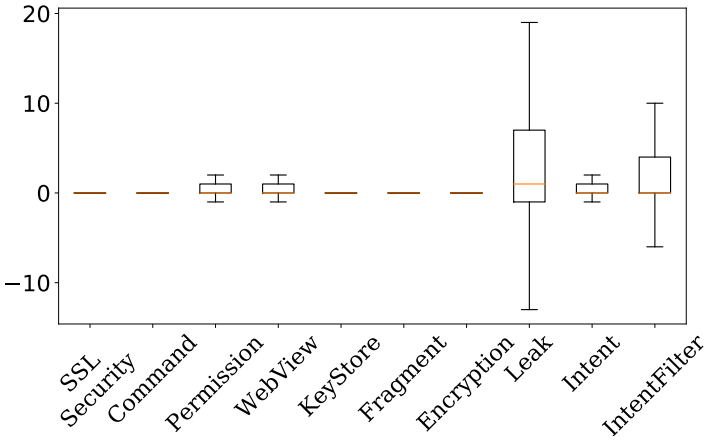


Figure 4.4: Distribution of Added and Removed in a Numbers of Vulnerabilities between Consecutive APK Versions. Numbers represent p-values from MWW tests on the statistical significance of the differences.

We further detail in Figure 4.4 the distribution of the numbers of vulnerabilities added and removed in apps. Except for the *Permission* case, we note that the median number of vulnerabilities added is equal or higher to the number of vulnerabilities that are removed. This confirms a finding in a recent study by Taylor *et al.* [10] on a smaller set of apps: "Android apps do not get safer as they are updated". The p-values of Mann-Whitney-Wilcoxon (MWW) test with null hypothesis of equal distribution, alternative hypothesis of not equal distribution and confidence level of 0.95, indicated above each box plot pair, however, show that the difference is statistically significant only for the three ICC-related vulnerabilities.



(a) Between Consecutive APK Pairs.



(b) Between Initial and Latest Versions.

Figure 4.5: Variations in # of Vulnerabilities Following Updates.

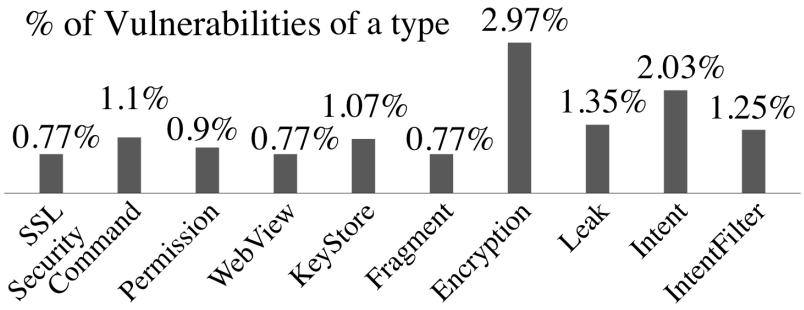
We now investigate the general trend in vulnerability evolutions, comparing the impact of updates between consecutive pairs and the impact of all updates between the beginning and the end of a lineage. We expect to better highlight the overall evolution of vulnerabilities as several changes have been applied. The box plots in Figures 4.5 highlight a simple reality: commonly vulnerabilities are neither removed nor introduced during app updates (i.e., all median values equal to 0 in Figure 4.5a) and when they happen, their chances are quite equal as well (i.e., all mean values are very close to 0 too). When looking at the distribution obtained based on the initial/latest versions shown in Figure 4.5b, the major pattern stays similarly (i.e., all median values are still 0 only except for *Leak* which is 1 and for most of the mean values, they increased slightly but still between around 0.5 to 0. the exceptions are *Leak* and *IntentFilter* which are 3.8 and 2.3 respectively), but observable differences are exhibited as well. Several vulnerabilities expand in size and the scale increases obviously. The

main parts of all boxes are on the positive side of y-axis, which indicates that there are more cases of adding vulnerabilities than removing.

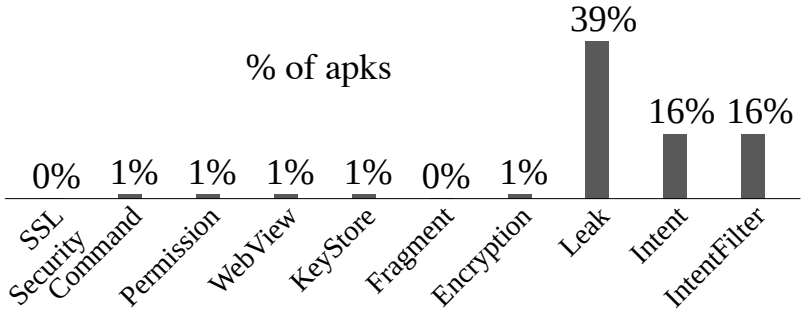
Insights from RQ2: As more than 50% of vulnerabilities stay untouched during 1 update and the possibility of fixing and introducing vulnerabilities during updates does not show a significant difference, app updates indeed do not make apps safer. Moreover, vulnerabilities can normally survive 3 updates and even longer, this suggests developers haven't been paying enough attention on vulnerability issues.

4.3.3 Vulnerability Reintroductions

To answer RQ3: ***Do fixed vulnerabilities reappear later in app lineages?*** We track all vulnerability alerts (associated with their locations) and cross-check throughout the lineages. We found 342,809 distinct cases of location-based vulnerability reintroductions (i.e., vulnerable code removed and reappeared in the same method of the same class of an app, as specified in Section 4.2.1) for 15,375 distinct apps. On average, a given app is affected by 6.7 vulnerability reintroductions. Figure 4.6a further breaks down reintroduction cases and their proportions among all vulnerability alerts. *Encryption*-related vulnerabilities (2.97%) are the most likely to be reintroduced, in contrast to *SSL Security*-related vulnerabilities (0.77%).



(a) Vulnerability Reintroduction

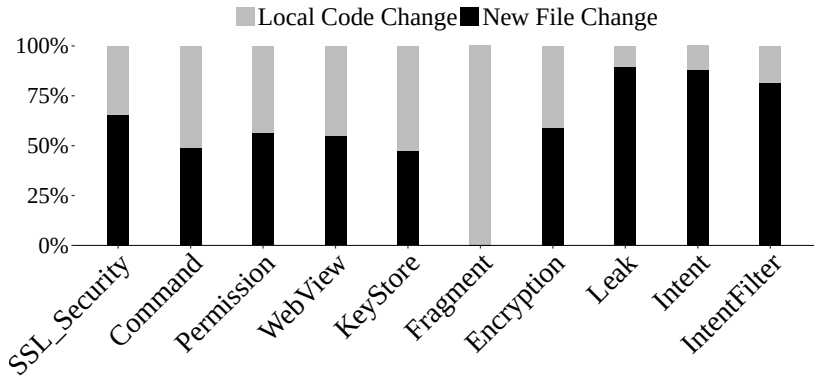


(b) Vulnerability Type Reintroduction

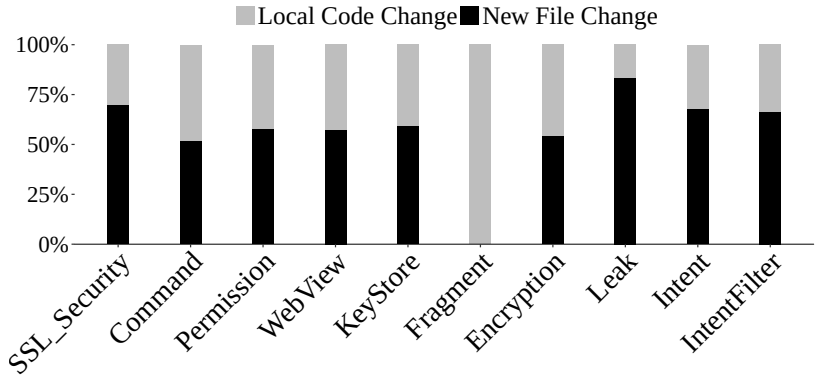
Figure 4.6: Statistics on Reintroduction Occurrences.

We investigate whether, in some lineages, a **vulnerability type** may completely disappear at some point and later re-appear. Figure 4.6b provides statistics on proportions of lineages where a given vulnerability type is reintroduced (note that this type-based vulnerability reintroduction is only discussed here, for the rest of this chapter, without specification, the reintroduction should be location-based).

Figure 4.7 details, for each vulnerability type, the proportion of cases where a vulnerability was removed in an apk version following a complete deletion of its location file, or following code changes in its location (at method level or file level depending on the vulnerability type). File deletion and



(a) Vulnerability Removal



(b) Vulnerability Introduction.

Figure 4.7: Statistics on How Vulnerabilities Are Removed (during file deletion or code change within vulnerability location file) or Introduced (during new file insertion or code change within vulnerability location file).

new file insertion occupy a big portion suggests that vulnerabilities are probably fixed or introduced with third-party code. Our in-depth analysis reveals that those deleted and inserted files are indeed mostly from libraries. For example, we have found that file *com.tencent.open.SocialApiIml* has been deleted and newly inserted 3,730 and 6,012 times respectively.

Insights from RQ3: Vulnerability reintroductions occur in Android apps and *Encryption*-related vulnerabilities are the most like to be reappeared with the possibility of around 3%.

4.3.4 Vulnerability Introduction Vehicle

We answer RQ4: ***Where are vulnerabilities mostly located in programs and how do they get introduced into apps?*** By first providing a characterization of code locations where vulnerabilities are found. We focus on two main location categories: *library* code and *developer* code. We attempt to provide a fine-grained view on vulnerable-prone code by distinguishing between:

- *Developer code*, approximated to all app components that share the same package name with the app package (i.e., app id).
- *Official libraries*, which we reduce in this work to only Android framework packages (e.g., that start with `com.google.android` or `android.widget`).
- *Common libraries*, which we identify based on whitelists provided in the literature [17].

- *Reused or other Third-party code*, which we defined as all other components that do not share the app package name, but are neither commonly known library code.

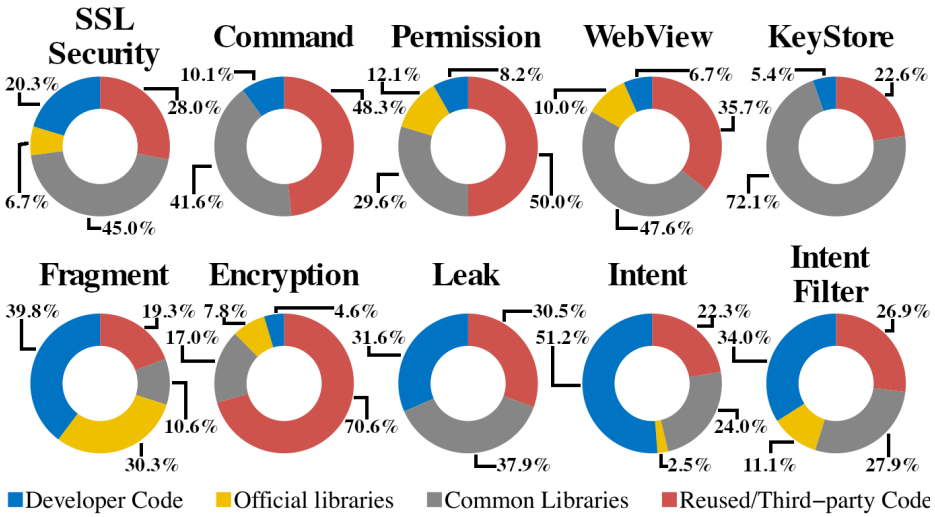


Figure 4.8: Distribution of Vulnerable Code in *Developer Code*, *Official Libraries*, *Common Libraries* and *Reused/Third Party Code*.

Figure 4.8 details the distribution of vulnerable code in different locations. For most vulnerability types, it stands out that third-party code (including common libraries) is the main carriers of app vulnerabilities. Developer code is more affected by ICC data handling vulnerabilities. Android "official"[13] libraries are however affected by Fragment vulnerabilities. This could be explained by the fact that several of such libraries are widely used to implement ads display in app foreground UIs. Although such vulnerabilities may be fixed by Google library maintainers, it is commonly known that update propagations can be slow in Android [76].

We investigate the correlation between the size of apps and the number of vulnerabilities to assess a literature intuitively-acceptable claim that larger apps are more vulnerable. Then, we study how this reflects in evolution via apk updates, by checking whether the number of new code packages added in an app during an update correlates with the number of newly appearing vulnerabilities. Table 4.2 provides Spearman correlation computation results. All correlation appear to be 'Negligible'. *IntentFilter* shows the highest correlation close to being categorized as 'Moderate' w.r.t. the size of the apps.

Table 4.2: Spearman Correlation Coefficient (ρ) Values. With experiments **Exp.1:** # of packages vs. # of vulns. per apk; **Exp.2:** # of new packages vs. # of vulns. per update.

Type	SSL Security	Command	Permission	WebView	KeyStore
Exp.1	0.08	0.07	-0.11	0.08	0.08
Exp.2	0.14	0.06	-0.02	0.07	0.02
Type	**Fragment**	**Encryption**	**Leak**	**Intent**	**IntentFilter**
Exp.1	0.19	-0.02	0.05	0.06	0.22
Exp.2	0.17	-0.00	0.04	0.10	0.12

Interestingly, computation of LOESS regression [111] shown in Figure 4.9 further highlights that while a positive correlation, although 'negligible', may exist between added packages and the number of added vulnerabilities, no correlation can be observed between removing packages and variations in vulnerability numbers.

[13]Our heuristics are solely based on package name and thus may actually include abusively named packages. See package list on artefact release page.

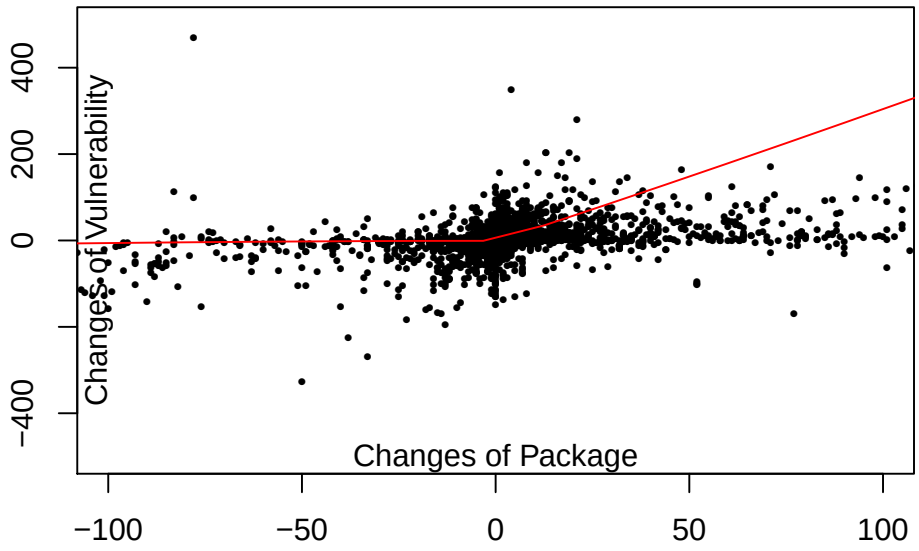


Figure 4.9: Overall Regression.

As numbers of the vulnerabilities are not correlated with both apk size and package numbers. We can deduce that not all packages commonly introduce vulnerabilities yet only for certain packages.

Insights from RQ4: Although third-party libraries are the main contributor of vulnerabilities, it is quite possible that the major contribution is only from part of these libraries. Therefore more focus should be given on the analysis of libraries and market maintainers could draw policies rejecting apps using *non-vetted* libraries. Moreover, the claim made in [103] that more code and libraries imply more vulnerabilities may not be always true.

4.3.5 Vulnerability and Malware

To answer RQ5: ***Do vulnerabilities foreshadow malware?*** We investigate relationships between app vulnerabilities and malware. One way for malware to achieve their malicious behaviors is by leveraging vulnerabilities. Reasonably, malware can deliberately implement vulnerabilities for their own use as presented in [112]. Moreover, to distinguish malware from benign apks, the common practice is using anti-virus(AV) flagging reports. AndroZoo provides these reports [14] as metadata for all its apks and in this study, we treat an apk as malware as long as one or more AVs gave positive reports.

Table 4.3: Benign and Malware in Vulnerable APK Sets

	SSL Security	Command	Permission	WebView	KeyStore
Malware	42.17%	56.00%	62.90%	44.73%	35.79%
Benign	57.83%	44.00%	37.10%	55.27%	64.21%
	Fragment	Encryption	Leak	Intent	IntentFilter
Malware	14.28%	58.82%	38.12%	37.96%	43.85%
Benign	85.72%	41.18%	61.18%	62.04%	56.15%

Table 4.3 reports the proportion of benign and malware which are detected as vulnerable for each vulnerability type. Malware are not more likely to contain a given vulnerability than benign apps. We further perform a correlation study on these malware to assess whether the number of vulnerabilities in an apk can be correlated to the number of AVs that flag it. This is important since AVs are

[14] AndroZoo provides, for each apk, AV reports of dozens of AV engines hosted by VirusTotal (`https://www.virustotal.com`)

known to lack consensus among themselves [113, 114]. For every vulnerability type we found that the Spearman's ρ was below 0.30, implying negligible correlation.

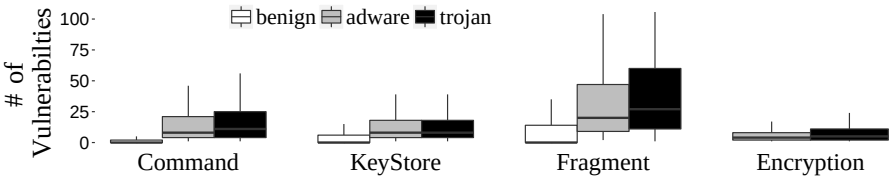


Figure 4.10: Vulnerability VS Malware[15]

We now carry an evolution study to investigate whether certain vulnerabilities may foreshadow certain *type* of Android malware. To this end, we rely on type information provided in AndroZoo based on the Euphony tool [115]. A malware can be labeled with various types, including *trojan*, *adware*, etc. Given an app lineage, when a single apk is flagged by AVs, we consider its non-flagged predecessors in the lineage and count cumulatively how many vulnerabilities were included in them. For each lineage where all apks are benign we also count the number of vulnerabilities per vulnerability types. We then perform the MWW test, for each vulnerability type, to assess whether the difference between, on the one hand, the median number of vulnerabilities for malware of a given type, and, on the other hand, the median number of vulnerabilities in benign apps, is statistically significant. In most cases, this difference is not statistically significant, suggesting that most types of Android malware cannot be readily characterized by vulnerabilities within the malware itself. Nevertheless, we find four interesting cases of vulnerability types (namely, *Command*, *KeyStore*, *Fragment* and *Encryption*), where vulnerabilities are suggestive of malicious behavior. Figure 4.10 illustrates the distribution of vulnerabilities across benign, trojan, and adware. Vulnerabilities of these types are significantly less in *benign* lineages than in earlier apks of lineages where malware of type *trojan* or *adware* will appear as shown in the figure and further proved by the 0 valued p-values between *benign* and *adware* and between *benign* and *trojan* of all 3 types except *Encryption*, while the absence of *benign* of type *Encryption* in the figure reflects that *benign* apks does not contain any of such vulnerability.

Insights from RQ5: Our study finds similar rates of vulnerabilities in malware as well as benign apps. However, we uncovered cases where vulnerable apks were updated into malicious versions later in the app lineage.

4.4 Discussion

We now discuss the potential implications and future works, as well as the possible threats to the validity of this study.

4.4.1 Implication and Future Work

The datasets and empirical findings in this work suggest a few research directions for implications and future works in improving security in the Android ecosystem.

- **Understanding the genesis of mobile app vulnerabilities.** Since app lineages represent the evolution of apps, they could contain the information about "when" and "how" is a given vulnerability initially introduced. This information could then be leveraged to understand the genesis of the vulnerability and thereby help researchers and developers invent better means to locate and defend such vulnerabilities.

[15]Other vulnerabilities are ignored due to insignificant differences observed between benign, adware and malware samples. Hence, they are omitted from the figure to give a clear exhibition.

- **Tools to address vulnerability infections.** By leveraging app lineages, the corrected pieces of code of a vulnerability happened in a certain app version could be spotted and extracted from its subsequent app versions with the fixes. Indeed, the vulnerable code snippets disclosed in this work could be leveraged to mine fix patterns for certain vulnerabilities and subsequently enable the possibility of automated vulnerability fixes.

- **Reintroduction analysis for app updates.** As revealed in the answer to RQ3, the fact that vulnerabilities can be reintroduced into apps during their updates, it is essential to perform reintroduction analysis (either statically or dynamically) for Android apps. These analyses later could be immediately adopted by app markets to guarantee that app updates do not introduce more (known) vulnerabilities.

- **Library screening strategies.** As concluded in Section 4.3.4, third-party libraries are the main contributor of vulnerabilities in an app. Thus, when libraries containing serious vulnerabilities get to be popular, the aftermath will be difficult to estimate. Such incident happened once on August 21, 2017, Bauer and Hebeisen [116], from the Lookout Security Intelligence team, have reported that their investigation of a suspicious ad SDK (i.e., ad library) revealed a vulnerability that could allow the SDK maintainer to introduce malicious spyware into apps. After it was alerted, Google has then removed from the market over 500 apps containing the affected SDK: those apps were unfortunately already downloaded over 100 million times across the Android ecosystem. Therefore, strategies of selective screening of libraries could be investigated to clean app markets with apps which unnecessarily ship vulnerable libraries.

- **Understanding the pervasiveness of vulnerabilities.** According to this study, each apk contains more than 60 vulnerabilities on average. Although intensive studies have been done on different kinds of vulnerabilities, no vulnerability "bubble" explosions have been observed as we studied in RQ1. However, detection tools targeting on these vulnerabilities have been made publicly available and free for quite a long time such as the tools we used in this study. Therefore, why developers did not using these tools to protect their apps could be an interesting question to answer in future work.

- **A correlation study of vulnerabilities and malware.** In this study, the cases where APKs containing certain vulnerabilities were updated into malware have been spotted. Khodor *et al.* [112] also observed similar cases that malware deliberately implements vulnerabilities for its malicious purpose. This phenomenon implies that there could be correlations between certain vulnerabilities and malware. Nonetheless, more thoroughly defined experiments are needed to confirm this hypothesis. We believe that app lineages, introduced in this work, can be leveraged to implement such studies.

4.4.2 Threats to Validity

Like most empirical investigations, our study carries a number of threats to validity. We now briefly summarise them in this subsection.

Threats to external validity are associated with our study subjects as well as to the vulnerability detection tools that are selected. To provide reasonable confidence in the generalizability of our findings, this study leverages the most comprehensive dataset of Android apps. Threats to external validity are further minimized by considering a variety of vulnerability detection tools (hence of vulnerability types) for our study.

The main threat to internal validity is related to the process that we have designed for re-constructing app lineages. To minimize this threat, we have implemented constraints that are conservative in including only relevant APKs in a lineage.

In terms of threats to construct validity, our analyses assume that all vulnerability types are of the same importance and that every APK can be successfully analyzed. Yet, since the IC3 and FlowDroid successfully analyzed fewer APKs than AndroBugs, the scale of the significance of the findings may vary. Nevertheless, we have focused more on assessing proportions related to available data per vulnerability types (instead of immediate averages).

Also, code obfuscation is not considered in this study. As it is more and more common for developers to obfuscate their code because of security or malicious consideration. This could introduce some impacts on our results. However, many of our analyses are naturally obfuscation immune (e.g., leak analysis checks the data flows from sources to sinks, while sources and sinks are normally Android API calls which cannot be obfuscated.). Therefore, the impact of code obfuscations should be limited.

Furthermore, the experimental results may be impacted by the validity of the results of the selected vulnerability detection tools. Given that these are static analysis tools, it is known that they may yield false positives. We attempt to mitigate this impact by performing a manual verification to some of the randomly selected vulnerabilities yielded by the three analyzers. As discussed in Section 4.4.1, the naive verification process does not spot any clearly false positive results (i.e., the vulnerabilities are at least in conformance with the definition of vulnerabilities as proposed in the tools documentation). However, since the verification was implemented by 2 PhD students, their experiences could have a direct impact on the verification result. Thus, lack of proof of the authenticity of the vulnerabilities is the main threats to the validity of this study. Moreover, during the manual verification, vulnerabilities of non-HTTPS links are not considered. The main reasons are: 1) they are quite straightforward to be identified, and 2) these links can be changed over time and thereby are difficult to be verified (e.g., for an HTTP link, if an HTTPS page and redirect were added just before the verification, should we consider it as a false positive?). The pervasiveness of such vulnerabilities also makes it hard to be manually checked. But we still have to be aware of the possibility of the impact on the results.

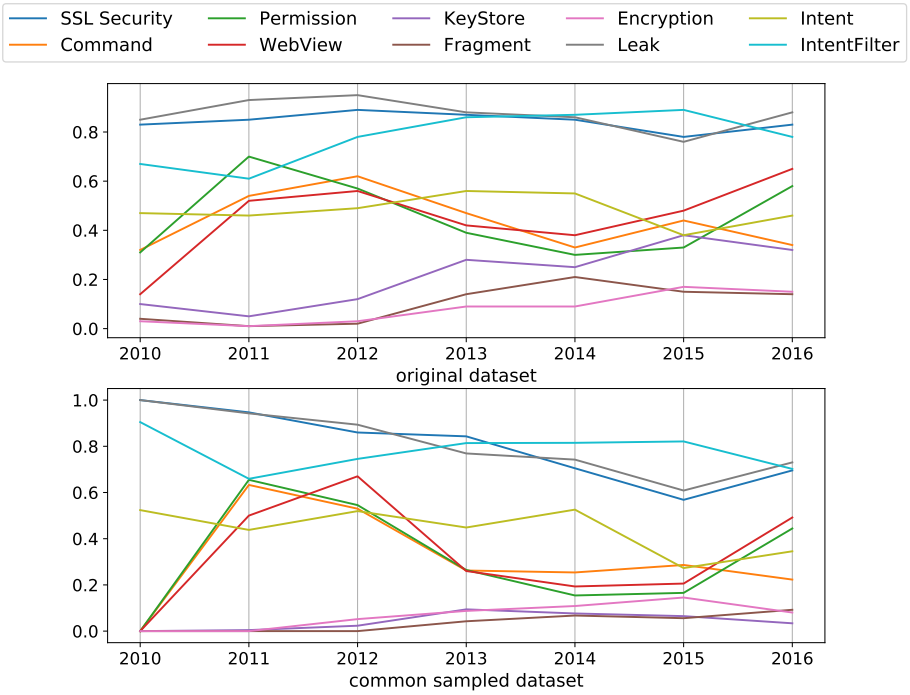


Figure 4.11: Trends Comparison between the Original Dataset (imbalanced) and the Common Sampled Dataset (balanced).

Finally, due to constraints such as time budgets and computation resources, the results yielded by the three selected static analyzers are for a different number of apps. Since the results obtained by

the static analyzers are not always from the same samples, different vulnerabilities could be collected from different datasets. Therefore, our empirical observations could have been impacted by such inconsistent datasets. However, as we have not attempted to compare the results between different static analyzers, we believe that such an impact should be negligible Nevertheless, to empirically demonstrate this, we go one step deeper to revisit the aforementioned studies with a common corpus. Specifically, we conduct our revisit study on 356 app lineages, which correspond to 3984 APKs, having all these apps successfully analyzed by the three tools. Our revisit study reveals that the empirical findings observed from a common app lineage set are more or less similar to that of imbalanced datasets. For example, regarding the evolving patterns of vulnerable apps, as illustrated in Figure 4.11, the results observed from the imbalanced dataset (above sub-figure) and the common corpus (bottom sub-figure) more or less follow similar trends, indicating that the empirical results observed will unlikely be impacted by the dataset chosen in this study.

4.5 Related Work

Our work is related to several contributions in the literature. In previous sections, we have discussed the case of data leaks vulnerability in Android investigated by the authors of TaintDroid [108], FlowDroid [55] and IccTA [56]. Other analyzers have been proposed based on static analysis [6, 89, 117], dynamic analysis [118, 119, 120, 121, 122] or a combination of both [54] to find security issues in apps. In view of the amount of literature that relates to our work, we focus on three main topics:

Android security studies Subsequent to the launch of Android, several comprehensive studies have been proposed to sensitize on security issues plaguing the Android ecosystem. Enck *et al.* [109] have provided the first major contribution to understanding Android application security in general with all potential issues. However, compared to the dataset used in this study, the number of their samples was limited. Felt *et al.* [3] have then focused on permission re-delegation attacks while Grace *et al.* [5] focuses on capability leaks. They unveiled vulnerabilities related to permissions. While this work studied different kind of vulnerabilities from the evolution aspect. Zhou *et al.* [37] have later focused on manually dissecting malicious apps to characterize them and discuss their evolution. The MalGenome dataset produced in this study has since been used as a reference dataset by the community. We mainly focus on the vulnerabilities of Android. More recently, Li *et al.* [16] have performed a systematic study for understanding Android app piggybacking: they notably pointed out libraries as a primary canal for hooking malicious code. Although piggybacking is different from updating, as it is a tempering by other developers, there are some similar mechanisms and we borrowed some ideas from their study.

Vulnerability studies Vulnerabilities, also known as security-sensitive bugs, have been extensively studied in the literature [123] for different systems [124, 125, 126, 127, 128] and languages [80, 129, 130, 131]. Camilo et [66] have recently investigated the Chromium project to check whether bugs foreshadow vulnerabilities. Researchers have also proposed approaches to automatically patch them [132, 133].

In the Android literature, several studies have already been performed: Bagheri *et al.* [134] have recently analyzed the vulnerabilities of the permission system in Android OS; Huang *et al.* [135] have studied so-called Stroke vulnerabilities in the Android OS which can be exploited for DoS attacks and for inducing OS soft-reboot; Similarly Wang *et al.* [136] have analyzed Android framework and found 6 until-then unknown vulnerabilities in three common services and 2 shipped apps, while Cao *et al.* [137] focused on analyzing input validation mechanisms. Qian *et al.* [58] have developed a new static analysis framework for vulnerability detection. Thomas *et al.* [76] have analyzed 102k+ apks to study a CVE reported vulnerability on the JavaScript-to-Java interface of the WebView API. Jimenez

et al. [138] have attempted to profile 32 CVE vulnerabilities by characterizing the OS components, the issues, the complexity of the associated patches, etc. Linares-Vásquez *et al.* [139] have then presented a larger-scale empirical study on Android OS-related vulnerabilities. OS vulnerabilities have also been investigated by Thomas *et al.* [140] to assess the lifetime of vulnerabilities on devices even after OS updates are provided. Closely related to our work is the study by Watanabe *et al.* [103] where authors investigated the location of vulnerabilities in mobile apps. Our work extends and scales their study to a significantly larger dataset. Finally, Mitra and Ranganath recently proposed the *Ghera* [141] repository with a benchmark containing artifacts on 25 vulnerabilities. Our work is complementary to theirs as we systematically collect thousands of pieces of code related to a few vulnerabilities, from which researchers can extract patterns, and help validate detection approaches.

Table 4.4: Related Works in Vulnerability Study

Work	**APK #**	**Type #**	**Detail**	**Year**
Fahl *et al.* [52]	13,500	1	Studied only SSL security vulnerabilities	2012
Jiang *et al.* [53]	62,519	1	2 vuls stem from content provider components which are called passive content leaks and content pollutions	2013
Sounthiraraj *et al.* [54]	23,418	1	Studied SSL security by using both static and dynamic analysis	2014
Watanabe *et al.* [103]	30,000	4	3 vuls of information disclosure, 6 vuls of SSL security, 5 vuls of inter-component communication and 4 vuls of webview	2017
Taylor *et al.* [10]	30,000	5	Studied 3 vuls of information disclosure, 3 vuls of insecure network communication, 2 vuls of cryptography, 2 vuls related to intent spoofing and debuggability and 1 vuls of binary protection and did and evolutionary study based on 1 update comparison.	2017

Table 4.4 lists the works which are similar to this study. It is noteworthy that the number of APKs considered in these reported studies is much less (by an order of magnitude) than the number of APKs considered in this study. Moreover, most of the studies focused on one specific vulnerability type. Although, the latest two works studied Android vulnerabilities more generally and the last one even considered about app updates. None of them studied vulnerabilities from the aspect of app lineages. Therefore, some evolution patterns of vulnerabilities can only be found in this study such as vulnerability reappearing.

4.6 Summary

We have run computationally expensive vulnerability scanning experiments on app lineages providing a view vulnerability evolution, which is so far the largest scale of this kind studies. Moreover, investigating Android vulnerabilities from app lineage point of view is the major novelty of this study, which allows us to yield several newly spotted findings: 1) most vulnerabilities can survival at less 3 updates; 2) part of third-party libraries are the major contributors of the most vulnerabilities; 3) vulnerability reintroduction occurs for all kinds of vulnerabilities while *Encryption*-related vulnerabilities are the

most reintroduced within all types of this study; and 4) some vulnerabilities may foreshadow malware. In addition to new findings, this large scale study also confirms most of the conclusions from previous studies with relatively small datasets. However, the result of this study also suggests that the recent claim made by Watanabe *et al.* [103] that more code and libraries imply more vulnerabilities may not be always true. Finally, 2 valuable artifacts produced by this study: 1) the complete dataset of **vulnerability scanning reports**, 2) recorded **vulnerable pieces of code**, are shared.

5 Negative Results on Mining Crypto-API Usage Rules in Android Apps

A common vulnerability in Android apps is related to mishaps with cryptography and associated failure to ensure confidentiality and authentication. Relevant APIs are indeed commonly misused by developers. Therefore, having clear specifications about how crypto APIs should be used is essential for both training developers during development and for systematizing checks to validate code in production. Since such rules are not fully available in the documentation, there is a need to extract them from code artefacts. We propose, in this dissertation, considering code changes associated with API usages are a potential source for capturing API usage rules. While we see that mining API usage rules from app updates is intuitively reasonable, the hypothesis needs to be confirmed. In this chapter, we test this hypothesis by performing a large-scale investigation with the app lineage dataset. Although we produce a negative result, we elaborate our analysis with insight and make available the artefact of the study.

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

- J. Gao, P. Kong, L. Li, T. F. Bissyandé, and J. Klein. Negative results on mining crypto-api usage rules in android apps. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 388–398, 2019

Contents

5.1 Overview

Although software systems have greatly impacted the efficiency of transactions and communications in our digital world, the security and privacy issues that they carry have been raising concerns among all stakeholders. In this context, the software development community is now urged to implement means to protect user assets, most notably by using cryptography for ensuring confidentiality of data and transactions, as well as the authenticity of information. Unfortunately, several recent studies [143, 144, 145] have revealed that developers often make mistakes when using cryptography APIs (hereafter, crypto-APIs is used for short), even those APIs implemented in widely used libraries such as the Java Cryptography Architecture (JCA). These misuses, which may lead to security mishaps [146, 147], actually carry an illusion of safety for users and developers. Consequently, the research community has started a new effort towards improving the analysis and fix of crypto-APIs usage [148, 149].

To properly use crypto-APIs, developers must learn the API usage rules. Similarly, to validate code, the research and practice communities must build tools for checking API calls against a database of the associated API usage rules. To the best of our knowledge, there are three strategies commonly adopted in the literature for the inference of general API usage rules:

- **manual specification:** Recent literature on static analysis for verifying crypto-API usages propose approaches that are based on manually-written specifications [148, 150]. Although such approaches offer a high degree of reliability, they may require extensive security expertise, and do not scale to the sheer number of cryptography libraries (and their associated recurrent API updates).
- **majority contest of usage patterns:** A trivial approach for systematically finding and updating API protocols is to mine usage patterns in a representative dataset of developer code [151, 152]. Most recurrent patterns are considered as the correct protocol. Such approaches have been shown effective in operating system code [153] where the majority of developers have a significant level of expertise [154, 155]. In the Android community, however, most developers are novice and their usages of crypto-APIs are generally incorrect [156].
- **commit log mining:** Recently, Paletov *et al.* [149] have proposed to mine commit messages from software version tracking systems to identify fixes of API usages and infer the "correct" usages based on static code analysis. Theoretically, this strategy is reliable (in contrast to simple popularity voting of usage patterns). In practice, however, developers often make uninformed updates, and it is now accepted that commit messages are often less informative than what researchers expect [157].

This study. Our work is set in the context of the Android development community where millions of apps are built and regularly updated on markets. Our objective is to present and investigate the suitability of an approach to infer crypto-API usage rules based on developer updates. Although most of Android apps are not associated with public source code management systems, their different apk releases can be readily reverse-engineered into intermediate representations (e.g., *smali* or *Jimple*) by using frameworks such as *Apktool* and *Soot* [158]. These representations can then be statically analyzed in a straightforward way for extracting usage instances, and comparing usages across updates. Our approach will leverage the AndroZoo [15] dataset where successive apk releases are continuously crawled for the research community.

> *Our main assumption for inferring crypto-API usage rules by mining code updates is that "***API usage updates generally transform incorrect usages into correct usages***". Although this assumption is intuitively reasonable, our investigations have yielded contradictory results. We thus report on the negative results of mining crypto-API usage rules by mining Android app updates. We focus in this study on the widespread JCA APIs used in 598,875 apk releases associated with 39,213 lineages of real world Android apps.*

5.2 Background

We now provide details on the crypto-APIs studied in this work as well as the tool chain leveraged for statically checking API usages (i.e., to build the ground truth for the study).

5.2.1 Crypto-APIs

Cryptography is the science that yields algorithms for hiding information from third-parties. Encryption and decryption mechanisms are used to support authentication as well as to guarantee the confidentiality of transactions and information integrity. In software development, crypto-APIs are provided as part of programming toolkits to accelerate the inclusion of cryptography functionalities in developer code. For example, in the Java realm, the Java Cryptography Architecture (JCA) APIs, which are officially provided by Oracle [159], are widely used by developers. Given that most Android apps are built in Java, the use of JCA APIs is also widespread in the Android community. Although some alternate crypto-APIs, such as Apache Commons Crypto [160] APIs, do exist, they are substantially less widespread. Therefore, our work is focused on JCA to investigate the potential API misuses among real-world apps.

Table 5.1 enumerates the 23 API classes implemented in the JCA library, where each class was designed to address a specific cryptography functionality: for example, the *MessageDigest* class includes algorithm implementations for computing the digest of some information (e.g., text message) which can be used to check its integrity after transmission.

Table 5.1: Java Cryptography Architecture (JCA) APIs.

API Class: *Description*
java.security.AlgorithmParameters: maintainer for security parameters for specific algorithms
javax.crypto.Cipher: provide encryption and decryption functionality
javax.crypto.spec.DHGenParameterSpec: parameters for generating **D**iffie-**H**ellman parameters for DH key agreement
javax.crypto.spec.DHParameterSpec: parameters used for DH algorithm
java.security.spec.DSAGenParameterSpec: parameters for DSA parameter generation
java.security.spec.DSAParameterSpec: parameters used for DSA algorithm
javax.crypto.spec.GCMParameterSpec: parameters for cipher using **G**alois/**C**ounter **M**ode
javax.xml.crypto.dsig.spec.HMACParameterSpec: parameters for the XML signature HMAC algorithm
javax.crypto.spec.IvParameterSpec: Initialization **V**ector for block cipher
javax.crypto.KeyGenerator: generate keys for encryption-decryption
java.security.KeyPair: holder of a publicprivate key pair
java.security.KeyPairGenerator: create publicprivate key pairs
java.security.KeyStore: a memory storage to maintain keys and certificates for later usage
javax.crypto.Mac: Message **A**uthentication **C**ode for message integrity protection
java.security.MessageDigest: a one-way hash for messages
javax.crypto.spec.PBEKeySpec: specification of a **P**assword **B**ased **E**ncryption key
javax.crypto.spec.PBEParameterSpec : parameters for password based encryption
java.security.spec.RSAKeyGenParameterSpec: parameters for RSA key pair generation
javax.crypto.SecretKey: a symmetric secret key
javax.crypto.SecretKeyFactory: convert key into key specification and vice-versa
javax.crypto.spec.SecretKeySpec: specification of a symmetric secret key
java.security.SecureRandom: generate secured pseudo-random numbers
java.security.Signature: digital signature

Implementation-wise, to perform a cryptography-related task, an object associated with the relevant JCA class must first be instantiated. Subsequently, a sequence of the object methods is invoked in a specific order of steps. Listing 5.1 shows a usage example of API *PBEKeySpec* retrieved from a human resource management app named *com.successfactors.android*. The code snippet is written in *Jimple*, the intermediate representation of *Soot*, which we leveraged in this work to reverse engineering Android apps.

First, the *password* (i.e., $r3) and *salt* (i.e., $r0) parameters of the constructor of *PBEKeySpec* are initialized with the passed-in arguments (lines 7-10). Then, an object of *PBEKeySpec* is constructed with the *password* and *salt* (lines 11-12). There are 2 extra constants of type *int* used when instantiating the object (line 12): the first one, (1 000 in this example), is used to specified the iteration number, the second one (i.e., 256) is used to specify the key length. The *PBEKeySpec* object is used to further generate a *SecretKey* object (line 13-14). After using the *PBEKeySpec* object, for security consideration, the password is cleared from the memory (line 15). The rest part of the example is to create a *SecretKeySpec* by using the previously generated objects and return it for other utilizations.

```
1  private static javax.crypto.spec.SecretKeySpec deriveEncryptrionKey(char[],
       byte[])
2      {
3          javax.crypto.spec.PBEKeySpec $r2;
4          javax.crypto.SecretKeyFactory $r5;
5          javax.crypto.SecretKey $r6;
6          javax.crypto.spec.SecretKeySpec $r7;
7          byte[] $r0;
8          char[] $r3;
9          $r0 := @parameter1: byte[];
10         $r3 := @parameter0: char[];
11         $r2 = new javax.crypto.spec.PBEKeySpec;
12         specialinvoke $r2.<javax.crypto.spec.PBEKeySpec: void
       <init>(char[],byte[],int,int)>($r3, $r0, 1000, 256);
13         $r5 = <com.sybase.persistence.SharedDataVault:
       javax.crypto.SecretKeyFactory secretKeyFactory>;
14         $r6 = virtualinvoke $r5.<javax.crypto.SecretKeyFactory:
       javax.crypto.SecretKey generateSecret(java.security.spec.KeySpec)>($r2);
15         virtualinvoke $r2.<javax.crypto.spec.PBEKeySpec: void clearPassword()>();
16         $r7 = new javax.crypto.spec.SecretKeySpec;
17         $r0 = interfaceinvoke $r6.<javax.crypto.SecretKey: byte[] getEncoded()>();
18         specialinvoke $r7.<javax.crypto.spec.SecretKeySpec: void
       <init>(byte[],java.lang.String)>($r0, "AES");
19         return $r7;
20     }
```

Listing 5.1: JCA API Usage Example (Jimple code representation)

In this example, there are several code locations where developers can make mistakes that would lead to misuses of the *PBEKeySpec* API:

- (line 9) - Security strength of a password is heightened when *salt* is properly generated in a random way. In practice, however, developers commonly hard code their *salt* value. In the example code, *salt* is specified as parameter of method *deriveEncryptrionKey* (line 1) and then stored in $r0. So, a misuse could happen if a constant is passed to *deriveEncryptrionKey* in the second parameter.

- (line 10) - Often, developers use a *String* object to hold the password and then use *toCharArray()* to convert to the required type (i.e., *char[]*) when necessary. However, the intention of designing *PBEKeySpec* constructor to only accept *char[]* instead of *String* is to avoid using *String*, since *String* object is immutable, therefore, they cannot be destroyed or modified after instantiation until *garbage collection* revokes the memory. Given that *garbage collection* occurs randomly and is out of the control of developers, the password can survive in memory for a long time, increasing the risk of being exploited by attacks.

- (line 12) - Documentation of JCA recommends an iteration number above 1 000. It is however common to have cases where developers, with little expertise, assign a smaller iteration rate.
- (line 15) - Password information should be kept in memory only for the duration it is needed, in order to minimize attack opportunities. Thus it should not be held in a *String* object and must be cleared immediately after the use of the *PBEKeySpec* object. Developers unfortunately often overlook the call to the *clearPassword()* of *PBEKeySpec*. In this example code, the method *clearPassword* in line 15 is correctly called, so there is no misuse.

```
1  Findings in Java Class: com.umeng.common.util.h
2
3   in Method: java.lang.String a(java.lang.String)
4      ConstraintError violating CrySL rule for MessageDigest
5          First parameter (with value "MD5") should be any of {SHA-256, SHA-384,
       SHA-512}
6              at statement: $r2 = staticinvoke <java.security.MessageDigest:
       java.security.MessageDigest getInstance(java.lang.String)>("MD5")
7
8      TypestateError violating CrySL rule for MessageDigest
9          Unexpected call to method reset on object of type
       java.security.MessageDigest. Expect a call to one of the following methods
       digest,update
10             at statement: virtualinvoke $r2.<java.security.MessageDigest: void
       reset()>()
```

Listing 5.2: CogniCrypt_SAST Report Example

5.2.2 Static API usage checker

We leverage DICIDER [161], a static analyzer of the *CogniCrypt* [162] framework, for detecting JCA API misuses in Java programs. This analyzer was selected as it has been extended to be compatible with Android apps as well [161], a static analyzer of the [163]. DICIDER checks JCA APIs against a set of rules that were manually specified by security experts using *CrySL* [148] (CogniCrypt Specification Language). We consider CrySL rules as a reliable and accurate oracle for deciding whether a JCA API is misused or not. Concretely, given an Android apk, DICIDER identifies all instances of the 23 JCA API classes and checks the usage against the CrySL rules to generate a report on all detected misuses. Listing 5.2 showcases an example of a report generated by DICIDER, where misuses are hierarchically grouped by classes and methods. In this example, 2 misuses are found in method *java.lang.String* of class *com.umeng.common.util.h* of app *com.lovinc.radio*:

- The first misuse, a *ConstraintError* of API *Messagedigest*, is reported in line 4, with details indicating that argument *"MD5"* is not recommended when invoking API method *getInstance(java.lang.String)*.
- The second misuse (*TypestateError* error) also relates to the same API object *$r2* and occurs in the same method. It specifies that the misuse is caused by the fact that the *MessageDigest* object had not been in the state to call method *reset()*, instead, method *digest* or *update* should be invoked before.

As demonstrated by this example, a single JCA API usage instance can suffer from multiple misuse errors. Table 5.2 enumerates and provides brief explanations on 6 misuse types detected by DICIDER. Actually DICIDER's reports may include *ImpreciseValueExtractionError* notifications, which indicate that DICIDER cannot obtain all the information for the analysis, and thus no clear conclusion can be given. We do not discuss such cases in this study. Nevertheless, given that we must be able to assess whether a misuse has actually been fixed in an app update, we must be able to enumerate all usage locations. To that end, we have developed on top of *Soot* [158] a dedicated tool for supporting the extraction of JCA API usage instances.

Table 5.2: CogniCrypt Misuse Types.

Type	**Explanation**
Example	
ConstraintError	Unrecommended arguments are given.
e.g., *MD5* as hashing algorithm.	
RequiredPredicateError	Arguments are not properly created.
e.g., constant values are used while values are required to be randomly generated.	
TypestateError	A JCA API object is not in the right state to invoke a certain method
e.g., a method *reset()* of a MessageDigest object is invoked before passing any information into it via calls to methods *digest* or *update*.	
IncompleteOperationError	Tasks are not completed using JCA API objects.
e.g., a *MessageDigest* object is instantiated by invoking the *getInstance* method, but no further method invocation on this object is performed. The digest task will therefore not be achieved.	
ForbiddenMethodError	Unrecommended API methods are invoked.
e.g., *PBEKeySpec(char[] password)* is one of the constructor of JCA API *PBEKeySpec* for deriving cryptographic keys from a given password. Since a key generated without *salt* has been proven to be weak, this constructor should be used in specific scenarios.	
NeverTypeOfError	Certain types are forbidden when storing sensitive information.
e.g., password value for *PBEKeySpec* should never be store as type *String*, but *char[]*. Since object *String* is immutable in Java, password information in this type cannot be explicitly freed from memory. The garbage collector is in charging of deleting it, yet it is unpredictable from a user standpoint, opening opportunities for password leakage.	

Finally, we have implemented a crawler for collecting on Google Play some metadata (e.g., category, rate, etc.) associated to AndroZoo apks. These metadata are leveraged in criteria for comprehensively dividing the dataset into relevant subsets for further investigations.

5.3 Scope of the Study

In this section, we state the study problem along with the research questions that we intend to investigate, and describe the data preparation for the study.

5.3.1 Problem Statement and Research Hypothesis

Given a crypto-API usage location, it is possible, with security expertise, to assess whether there is a misuse or not. Manual specification of usage rules however is tedious to collect over a large set of APIs. Previous studies have also shown that using a majority voting on usage patterns to conclude on the correctness of crypto-API usages will lead to poor results given that wrong usages are widespread in Android apps. Our intuition however is that, as time goes by, developers of a given app learn to fix API misuses. Thus, it should be possible, by analysing code updates in an app lineage (i.e., the series of apk versions released for a given app), to infer crypto-API usage rules.

Our hypothesis is thus that: *"updates in API usages across an app lineage will tend to fix misuses"*. Consequently, if an extensively large set of app lineages can be collected in the wild, it would be possible to retrieve a substantially large and diverse set of crypto-API usage fixes. Then, by assessing recurring patterns, we could infer API usage rules.

This work is about empirically assessing the validity of our hypothesis for the case of the JCA APIs within Android apps.

5.3.2 Research Questions

The empirical study mainly aims at (re)investigating the following questions:

1. *To what extent do Android developers misuse crypto-APIs?* The literature claims, often based on few example apps, that developers regularly make mistakes in using crypto-APIs. We attempt to provide a thorough picture of the state of crypto-API usages across a representative dataset of real-world Android apps.

2. *Are crypto-API usage updates fixing misuses?* Investigating actual API usages with an oracle, based on security expert manual specifications, will eventually help to conclude on the validity of our research hypothesis. We ensure in this study, that the cases of specific app categories (e.g., high rating apps, financially sensitive apps, etc.) are also analysed in comparison with the general trends.

3. *What are the impacts caused by API usage updates?* This question investigates how crypto-API usages get updated, in an attempt to derive explanations on the statistical results obtained in the previous question. Concretely, we study the proportions of updates that either successfully fix misuses, or (re)introduce mistakes in correct usages, or that fail to fix misuses.

5.3.3 Misuse detection

We assess crypto-API misuses based on the reports of the $CogniCrypt_{SAST}$ static checker. This tool, which implements analysis based on expert manual specifications of API usage rules, is used to collect the oracle to support our empirical assessment. Analyses are performed on a High-Performance Computing (HPC) platform [164]. Overall, we leveraged 142 HPC instances, each utilizing 24GB of memory, to successfully parse all 745 thousands apks in 5 days.

5.3.4 Dataset Curation

We took steps to remove from our study all irrelevant cases of apks or misuses reported by $CogniCrypt_{SAST}$.

- Apk releases with no JCA API usages are excluded from our study. Thus, 18% of our initial dataset is left out.

- Apk releases on which $CogniCrypt_{SAST}$ fails to generate a final report are also dropped from the study. Such cases often occur when the process runs out of memory. While the recommended memory size for $CogniCrypt_{SAST}$ is 8GB, we allocate 24GB in our experiments. Nevertheless, a few apk analyses are not able to be completed.

- Obfuscated apk releases are left out from the study. Since developers recurrently rely on obfuscation techniques to prevent reverse engineering of their apps, static checkers such as $CogniCrypt_{SAST}$ are challenged in their analyses: $CogniCrypt_{SAST}$ reports '?' for erratic character series that appear as class names. Given that our study of API misuses leverages class names as the basic unit for localisation, such unidentified class names constitute noise. Thus, after analysis, when the generated report contains any unlocalised class name, the corresponding apk is dropped.

Eventually, 146,226 apks are excluded from the dataset of apks. Table 5.3 summarizes some statistical details about our dataset. An apk is removed when the one of the situations above occurs. However, an app lineage is only removed when all its app versions are excluded.

[1] e.g., an app lineage of 10 app versions, 5 app versions could be removed because of tool failure while the rest could be caused by obfuscation

Table 5.3: Number of APKs and lineages in the Dataset

	# **lineages**	# **APKs**
initial dataset	43 365	745 101
remove because JCA API is not used	-3 882	-135 752
remove due to CogniCrypt failures	-108	-9 374
remove because of obfuscation	-7	-1 100
remove due to combination of the 3 conditions[1]	-155	
final dataset	39 213	598 875

Fig. 5.1 shows the distribution of the dex size of apks for the initial and the final dataset. Our statistical tests indicate no difference between the two distributions, implying that our final dataset is still representative of the initial dataset (at least w.r.t app sizes).

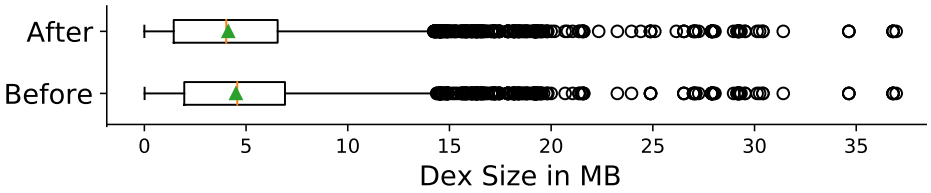


Figure 5.1: Distribution of Dex Size before-after dataset curation

Metadata Collection. Some of our investigations require up-to-data metadata (e.g., category, rating, number of installs) from markets. Because *GooglePlay* implements location restrictions (only apps targeting a country's users are made visible in that country), we were able to collect metadata for around 60% of the lineages.

5.4 Methodology of the Study

We carry experiments to assess the hypothesis behind relying on usage updates to mine crypto-API usage rules. We explore usage updates:

- with an analysis of *pairwise* comparisons among apks successive releases within the lineages in our dataset;
- with an overall *lineage-wise* study of the recurrence of misuses in an app across its entire lineage.

We further investigated updates for *selected subsets* of apps to confront the general trends against specific cases for popular apps, or sensitive apps.

5.4.1 Pairwise comparisons of apks from the same lineage

Given an app lineage l_i, its associated apk releases are combined into pairs (apk_{j-1}, apk_j), where $1 < j \leq n$, for the purpose of checking differences between misuses in apk_j and apk_{j-1}. The comparison takes into account only usage instances that are found at the same *code location* in both apps and that are relevant to the same *API*. In this study, a *code location* (lo) is represented by both the class and method in which the API usage is found. We consider the following cases which may occur:

- *misuse fixing (MF)* update: the $CogniCrypt_{SAST}$ analysis flags an issue with a usage in apk_{j-1} but not with the usage at the same location in apk_j. We conclude that the misuse has been fixed by the update.

- *misuse introducing (MI)* update: the $CogniCrypt_{SAST}$ analysis flags an issue with a usage in apk_j but not with the usage at the same location in apk_{j-1}. In contrast to a *MF* update, such an update introduces misuses.

- *misuse fixing and introducing (MFI)* update: the $CogniCrypt_{SAST}$ analysis flags an issue with a usage at a given location in both apk_j and apk_{j-1}, but the misuses are different. This suggests that developers corrected the previous misuse during the update, but somehow made another mistake.

- *none update*: the $CogniCrypt_{SAST}$ analysis flags the same issue with a usage in apk_{j-1} and apk_j at the same location. We conclude that the developers did not notice the issure during app updates.

To distinguish among different usages of the same API at the same location, one should take into account variables names associated to instantiated objects from API classes. In practice, however, the reverse-engineering of apks assigns random names to variables, making these names differ across the pair of apps. We use simple heuristics to match relevant pair of usage instances and iteratively start with identifying *none update* cases, then *MFI* updates, before *MF* updates and *MI* updates.

MI update and *MFI update* constitute two cases of *mis-updates*, as they result in API misuses in the most recent version of the app.

5.4.2 Investigations of updates across lineages

We investigate the overall evolution of a given app w.r.t. its misuses of crypto-APIs. We then study the trends of usage issues in app lineages. To that end, first, for each app version apk_j in an app lineage l_i, we compute a *misuse ratio* r_j defined in equation 5.1.

$$\begin{aligned} m_j &\leftarrow total\_number\_API\_misuses(apk_j) \\ u_j &\leftarrow total\_number\_API\_usages(apk_j) \\ r_j &:= \tfrac{m_j}{u_j} \end{aligned} \tag{5.1}$$

We consider the ordered ratio list R_i of app lineage l_i as $R_i := \{r_1, r_2, \cdots, r_n\}$, a list of misuse ratio of all apk releases included in l_i with r_1 being the misuse ratio of the first apk of l_i, r_2 the misuse ratio of the second apk, etc. We then compute the slope $s_i = linear\_regress(R_i)$ of the regressed between all points of R_i. In this study, s_i is used to characterize the *misuse trend*: a negative s_i indicates that lineage l_i is evolving towards a better usage of crypto-APIs, while a positive s_i suggests that the usage of crypto-APIs is worsening with app updates. A null s_i indicates a status quo. The bigger the value of $|s_i|$, the faster the evolution in a lineage.

5.5 Study Results

We now provide experimental results obtained while investigating the research questions enumerated in Section 5.3.2. In particular, we show statistics on crypto-API misuses in the wild, summarize the success rates in API misuse updates between apk releases, reveal the evolution of misuses across lineages, and eventually discuss the difference between the whole dataset and selected categories of apps.

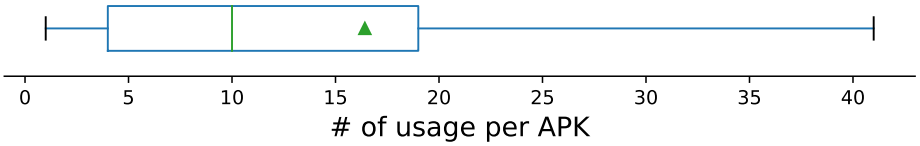


Figure 5.2: Distribution of JCA API usages in the study dataset

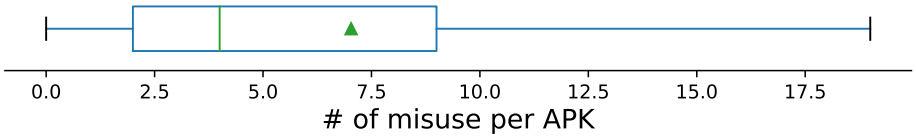
5.5.1 RQ1: Crypto-API misuses in Android apps

Crypto-APIs are widely used in Android apps. Statistics presented earlier (Table 5.3) indicated that over 80% (or 598 thousands apks) of the app versions in our sample dataset of 745 thousands apks include code with JCA API usages. Fig. 5.2 further shows the distribution of JCA API usages in our dataset of 598k apks. The usage statistics are collected from the analysis reports of $CogniCrypt_{SAST}$. On median average, 10 JCA API usage instances can be identified per app, while 75% of apks include at least 5 usage instances.

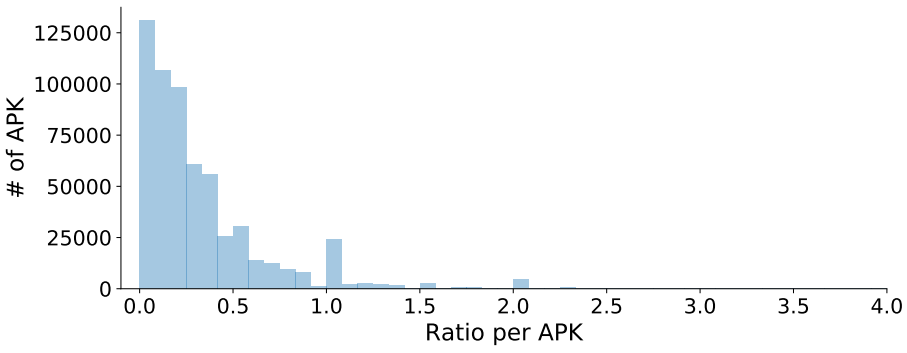
Table 5.4: Statistics on API Misuses in the dataset apks

Types	**Numbers**	**Related Information**
Percentage of APKs with misuse	96%	number of APKs without misuses is 24,880
Percentage of app lineages with misuse	97.6%	number of lineages without misuses is 942
misuse to usage ratio	1 : 5.53	total number of API usages is 23,281,216 and misuses is 4,210,667

Table 5.4 summarizes the statistics on API misuses among the apks that include JCA API usages. 96% of apks include misuses, and 97.6% of lineages include at least an apk version with a misuse. On average 1 misuse is found among 6 usages of JCA APIs.



(a) Distribution of # of Misuse per APK



(b) Distribution of Misuse to Usage Ratio per APK

Figure 5.3: API Misuse Distribution

We detail the misuse spread in Fig. 5.3. As shown by the misuse distributions in apks represented in Fig. 5.3a, a given apk contains commonly between 2 to 9 misuse cases. On median average, 4 misuses can be found per app. We further show in Fig. 5.3b the distribution of the ratio (as defined in Eq.1) between misuses and usages within apks. Interestingly, there are cases with the ratio is bigger than 1,

i.e., there are more misuse instances than usages. This suggests that developers can make several mistakes when using a single crypto-API.

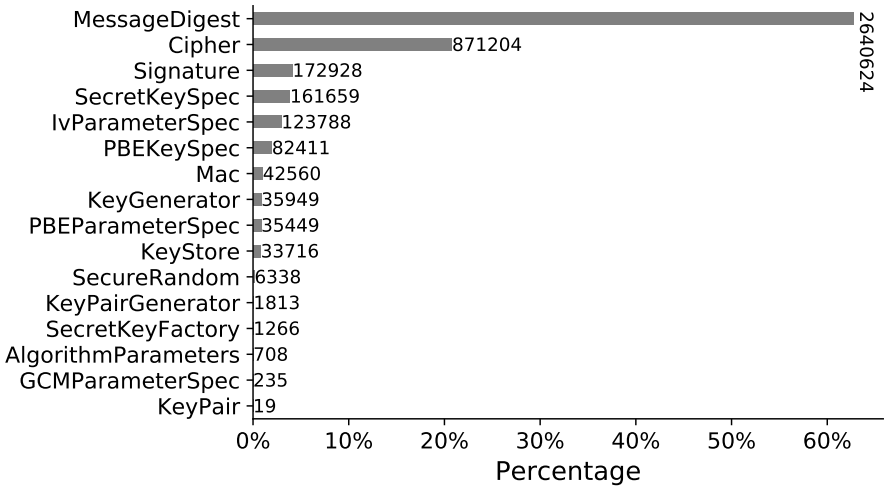


Figure 5.4: Misuse Ranking based on JCA APIs

The preponderance of the different APIs to be affected by misuses is detailed in Fig. 5.4. Over 60% of apks include instances of the *MessageDigest* crypto-API with a misuse. This is due to the widespread issue of using weak hashing algorithms. While the percentage for *Cipher*, the API in the second place, is around 21% for which the main misuse is incompleted operation indicating that some further method calls (e.g., *update*, *doFinal*) are expected but never achieved.

Table 5.5: Ranking of Misuses

Ranking by Types		
Type	**%**	**Misuse #**
ConstraintError	60.93%	2,565,892
RequiredPredicateError	14.63%	615,921
TypestateError	12.27%	516,758
IncompleteOperationError	11.64%	490,157
ForbiddenMethodError	0.51%	21,565
NeverTypeOfError	<0.01%	374
Top 10 by API Methods		
API Method	**%**	**Misuse #**
MessageDigest.getInstance(java.lang.String)	48.43%	2,039,428
Cipher.getInstance(java.lang.String)	8.80%	370,354
MessageDigest.reset()	8.06%	339,305
SecretKeySpec.<init>(byte[],java.lang.String)	3.79%	159,594
MessageDigest.digest()	3.78%	159,205
Cipher.init(int,java.security.Key, ... [2]	2.95%	124,274
IvParameterSpec.<init>(byte[]) [3]	2.92%	122,749
Cipher.init(int,java.security.Key)	2.50%	105,208
Signature.getInstance(java.lang.String)	1.99%	83,704
Signature.initVerify(java.security.PublicKey)	1.80%	75,705

[2]init(int,java.security.Key,java.security.spec.AlgorithmParameterSpec)
[3]Note: method <*init*> is the constructor of the class while method *init* is a common method of the class

Table 5.5 further provides statistical details on the types of errors that are raised by $CogniCrypt_{SAST}$ as well as on the top API methods that are concerned by misuses. Method *getInstance(java.lang.String)* of crypto-API *MessageDigest* takes as argument a String specifying the hashing algorithm. This algorithm must offer a strong protection and thus should be one of the recommended algorithms (e.g., *SHA-256*, *SHA-384* and *SHA-512*). When weak algorithms are used (e.g., *MD5* or *SHA-1*), it represents a *ConstraintError* which makes the app exposed to attacks.

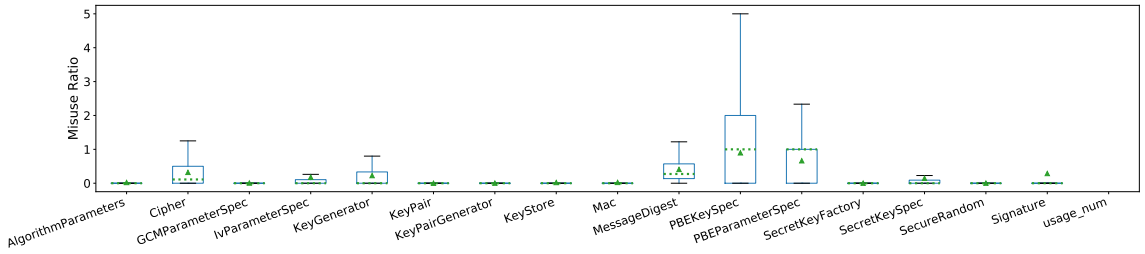


Figure 5.5: Misuse Ratio Distribution based on APIs

Nevertheless, we find that although *MessageDigest* is the API with the overall highest number misuses, it is not the most misuse-prone API. Crypto-API *PBEKeySpec* has a higher misuse ratio. Table 5.6 ranks the JCA APIs based on such a ratio. We provide details of misuse ratio distributions for each API in Fig. 5.5. The median value of misuse ratio for most of APIs is 0. For these APIs, only *IvParameterSpec*, *KeyGenerator* and *SecretKeySpec* are showing clear boxes and whisker lines while the rests only show outliers (which is not shown in the figure for clearity reason). This suggests that mistakes are quite rare for such APIs. In contrast, 5 APIs show relatively high misuse ratios. Based on their median values, *PBEParameterSpec* is the most error-prone (0.67), followed by *PBEKeySpec* (0.5), *MessageDigest* (0.24) and *Cipher* (0.9).

Table 5.6: JCA API Misuse to Usage Ratio Ranking. The misused apk % is calculated by number of apks containing the misuse divided by number of apks containing the relevant API usage.

API	Misuse Ratio	Misused APK %
PBEKeySpec	0.462098	55.50%
MessageDigest	0.287114	93.02%
Cipher	0.271808	50.77%
PBEParameterSpec	0.251944	57.26%
Signature	0.204835	23.76%
KeyGenerator	0.171380	27.74%
IvParameterSpec	0.102394	27.58%
SecretKeySpec	0.070372	27.81%
Mac	0.067249	3.05%
KeyStore	0.028322	9.12%
GCMParameterSpec	0.008141	0.79%
KeyPairGenerator	0.005865	0.60%
SecretKeyFactory	0.003974	0.35%
AlgorithmParameters	0.003307	3.46%
SecureRandom	0.003244	0.94%
KeyPair	0.000019	0.01%

We further investigated *PBEKeySpec* and *PBEParameterSpec* as they stand out in terms of misuse ratio distributions. For *PBEKeySpec*, we found that the main misuse type is *IncompleteOperationError* which is mainly caused by missing of calling method *clearPassword()* to clear the password from the

memory. *PBEParameterSpec* misuses only occur with type *ConstraintError*(80%) or *RequiredPredicateError* (20%). *ConstraintError* generally refers to the small iteration numbers as discussed in Listing 5.1.

Three JCA APIs (namely, *SecretKey*, *HMACParameterSpec* and *DSAGenParameterSpec*) are missing from the API usage list of our dataset. Four other APIs (namely *DHGenParameterSpec*, *DHParameterSpec*, *DSAParameterSpec*, and *RSAKeyGenParameterSpec*) although they have usage cases in our dataset, no misuses have been reported for them. Further investigations suggest that these four APIs are actually seldom used, and their usage rules are in any case straightforward. Indeed, except *DHParameterSpce*, these APIs are ranked at the bottom of the API usage ranked list. Their usage rules are only about the invocation of constructors with a number of constraints. Although other APIs, such as *GCMParameterSpec* and *KeyPair*, having similar simple rules, can be found with misuse cases, the occurrences are very low. In conclusion, these findings suggest that, since developers cannot avoid using crypto-APIs, simplifying the usage rules during API design could be an effective way to avoid misuses.

> *The JCA APIs for implementing cryptography are widely misused across Android apps. Usage mistakes range from issues with parameter initialisation to mishaps with the sequence of API method invocations. Nevertheless, all crypto-APIs are not similarly affected by misuse cases.*

5.5.2 RQ2: Impact of crypto-API usage updates on misuse cases

From our dataset of 39,213 lineages accounting for about 598K apks, we collected 559,662 apk pairs (apk_{j-1}, apk_j), where apk_j is the updated version of apk_{j-1}. From these apk pairs, we were able to extract 3,291,723 crypto-API usage pairs that fall into the four categories defined in Subsection 5.4.1: *MF* update, *MI* update, *MFI* update, and *none* update. Among 559,662 apk pairs, around 75% (or 410,587) of them fall into the *none* update category. This situation is even worse if we count the *none* update rate at the crypto-API uasge pair level, over 95% of the 3,291,723 misuses are not touched by app developers during the evolution of Android apps. This finding suggests that app developers are unlikely to update crypto-API usages when updating their apps or may not be even aware of the misuses in their app code.

For the remaining 162 970 crypto-API usage pairs involved with developer updates (e.g., not in the *none* update category), surprisingly, only 76,341 of them (less than 47%) have successfully fixed the misuse issues (i.e., falling into the *MF* update category). Over half of the crypto-API updating attempts fall into either *MI* update category (e.g., 72,143, around 44%) or *MFI* update category (e.g., 14,486, around 9%). At the apk level, the *MF*, *MI*, and *MFI* attempts are 53,030, 50,183, and 4,483, respectively, resulting in still over 50% of mis-update rate. This surprising result indicates that even in the cases that app developers are aware of crypto-API misuse and are attempting to fix such misuses, most of them do not have the right knowledge to properly fix the misuse issues. As a result, our previous assumption on mining crypto-API usage rules from the evolution of Android apps cannot be easily realised in practice.

Because the overall dataset leveraged, although very big, is collected from various sources, we hypothesise that the selected datasets (or app lineages) contain a broad set of apps with varying quality. Consequently, the large mis-update rate might be impacted directly by the selection of poor quality apps. If we focus our experiments on high-quality apps only, we might be able to observe clear trends that app developers are recurrently and successfully fix crypto-API misuse issues. To this end, we resort to two specific subsets to reconduct our empirical experiments.

- **Reputed Apps.** Updates of reputed apps are selected only within lineages where the app has high rates (i.e., ≥ 4.5) and large installs (i.e., $\geq 1\,000\,000$). With this subset, we assess whether widely used apps are similarly affected by crypto-APIs misuses.

- **Finance Apps.** Updates of finance apps are focused on app lineages tagged in *GooglePlay* as being for financial services. In this case, we again constrain this selection to apps with high rates and large installs as the case of Reputed apps. Because finance apps are critical to security issues, with this subset, we assess whether app developers of finance apps have special treatment to crypto-API misuse.

Table 5.7: Misuse Update Statistics

		MF	**MI**	**MFI**	**Mis-update Rate**
Overall	APK level	53,030	50,183	4,483	50.76%
	Usage level	76,341	72,143	14,486	53.16%
Reputed	APK level	4,300	3,934	446	50.46%
	Usage level	5,809	5,204	1,862	54.88%
Finance	APK level	179	153	30	50.55%
	Usage level	232	192	195	62.52%

Table 5.7 summarises the experimental results we observed for the different sub-datasets. For comparison purpose, we also present the *Overall* results representing the statistics computed for the whole dataset of apks. Following the same strategy, we also provide data on the number of *MF*, *MI* and *MFI* updates. The statistics are computed at the apk level (i.e., given a pair of successive apks in a lineage, how many of these pairs contain at least one *MFI* update, at least one *MF* update, etc.) and at the usage level (how many updates turned out to be a *MF*, a *MFI* case, etc., i.e., we count the number of *MF* update, *MFI* update, etc.). Unfortunately, compared to the mis-update rates of the *Overall* set, the results observed on that of the selected subsets do not suggest any substantial difference, only about 1 out of 2 updates will yield a correct fix for a crypto-API misuse.

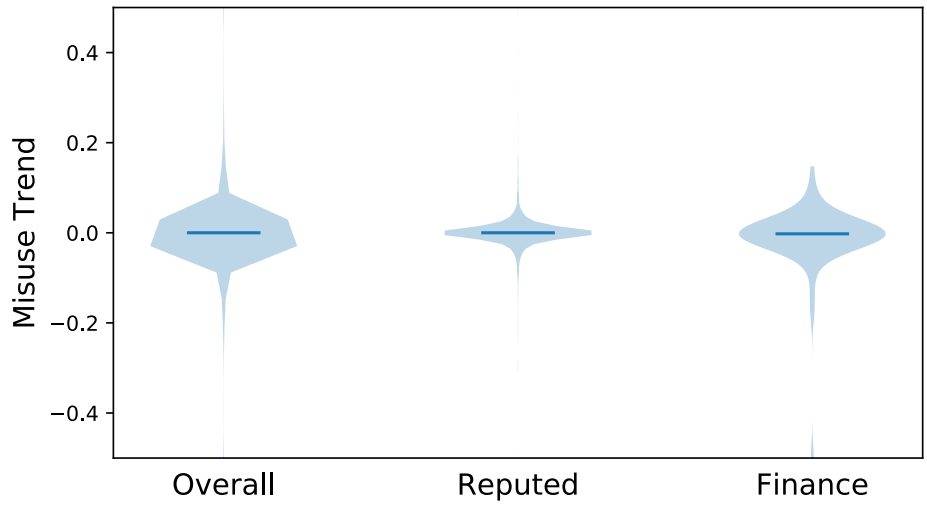


Figure 5.6: Distribution of Slopes (Misuse Trend)

Fig. 5.6 further illustrates the distribution of slopes (i.e., misuse trends as defined in Subsection 5.4.2) across the different subsets. We consider the evolution of misuse rate across each app lineage. When the mis-update rate is stable across the lineage, the slope metric evaluates to 0. A negative slope implies that the situation is getting better along the lineage: latest updates in the lineage are more successful. In contrast, a positive slope suggests that the situation is getting worse along the lineage.

We observe that, for all three datasets, the distribution of trend slopes presents more or less a symmetric pattern around a median value at 0. The fact that the distribution spans are relatively

narrow further suggests that for most lineages, crypto-API usage updates are rather stable. In the overall dataset, we can find 10,316 (26.31%) lineages with a negative trend of misuse update. Nevertheless, the successful cases in recent updates are still not more numerous: the mis-update rate amounts to 53.66% when we consider these lineages apps altogether.

Based on Table 5.7, it is noteworthy that the *MI* update (i.e., where a mistake is made on a usage that was previously correct) cases are the major contributors to the *mis-update* rate. There are often as many *MI* updates as *MF* updates in each dataset. We further investigate the recurrence of misuses across the different datasets and found 19,392, 1,332 and 65 recurrent misuse cases respectively in the overall, reputed and finance datasets. Concretely, we consider a misuse to be recurrent when, within an app lineage, an update has eliminated it (i.e., *MF* update), and then it has been later introduced (i.e., *MI* update). Overall, we find that 25% of *MI* updates actually represent recurring misuses, suggesting that the associated *MF* updates may have been unintentional (which made them likely to be reintroduced).

Table 5.8 details the misuse update results for the different APIs with their rankings. Even at the level of each API, *MF* and *MI* updates appear to compensate each other. This can be observed by the similar numbers of updates as well as their ranks across the API list, with APIs *Cipher* and *MessageDigest* leading the statistics (consistently with the misuses preponderance shown in Table 5.5).

Table 5.8: Misuse Update Ranking by APIs

API	**MF**		**MI**		**MFI**	
	#	**Rank**	#	**Rank**	#	**Rank**
Cipher	20,574	1	21,751	1	2,549	2
MessageDigest	17,668	2	20,451	2	1,283	3
IvParameterSpec	14,300	3	14,150	3	17	10
SecretKeySpec	14,085	4	14,126	4	71	7
KeyGenerator	3,372	5	3,465	5	65	8
Mac	878	6	860	6	9,382	1
Signature	504	7	516	7	864	4
KeyStore	278	8	268	9	153	5
PBEKeySpec	155	9	304	8	83	6
SecureRandom	131	10	180	11	0	12
PBEParameterSpec	123	11	212	10	0	12
AlgorithmParameters	47	12	22	12	2	11
KeyPairGenerator	14	13	20	13	0	12
GCMParameterSpec	13	14	13	14	0	12
SecretKeyFactory	1	15	3	15	17	10
KeyPair	0	16	0	16	0	12

Statistical data in Table 5.8 show that none of these three update kinds happened with *KeyPair* as shown in the last line of the table. This is likely due to its low misuse occurrences (i.e., 19 according to Table 5.5). Moreover, API *SecureRandom*, *PBEParameterSpec*, *KeyPairGenerator* and *GCMParameterSpec* do not show any cases of *MFI* updates. Nevertheless, we still cannot learn from their changes as they can either be *MF* or more like *MI* updates.

Android app developers are generally unaware of crypto-API misuses and hence will unlikely fix such issues. Unexpectedly, usage updates are evenly distributed between successful fixes and failures. The recurrence of misuses further imply that most of the successful updates may have not been made intentionally.

5.5.3 RQ3: Errors and methods impacted by misuse updates

We now investigate the types of errors that are concerned by *MF*, *MI* and *MFI* updates. Table 5.9 indicates that *MF* updates are dominated by misuses cases of type *RequiredPredicateError* (67%) followed by *ConstraintError* (29%). Consistently with the previous finding on recurrence of misuses, we note that *MI* updates follow a similar pattern. We recall that both *RequiredPredicateError* and *ConstraintError* generally concern the initialization or the selection of proper arguments to API methods.

Table 5.9: Misuse Update Ranking by Types

API	MF		MI		MFI	
	%	Rank	%	Rank	%	Rank
RequiredPredicateError	66.78%	1	69.99%	1	3.86%	4
ConstraintError	29.10%	2	25.73%	2	7.08%	3
TypestateError	3.35%	3	3.38%	3	12.07%	2
IncompleteOperationError	0.71%	4	0.84%	4	76.67%	1
NeverTypeOfError	0.05%	5	0.05%	5	0.05%	6
ForbiddenMethodError	0.02%	6	0.01%	6	0.27%	5

MFI updates show a different pattern, where *IncompleteOperationError* becomes the major misuse type: in 77% of cases, the update does not properly fix the errors. *TypeState* misuses then account for 12% of misuses that are difficult to fix. Both are about the sequence of API method invocations, either about missing certain method calls or related to a wrong order of invocation. For example, we note that in several cases, developers invoke the method *doFinal* immediately after getting an object instance of the *Mac* API class. However, they are supposed to start with the invocation of method *init* and, occasionally perform an *update* before calling *doFinal*: this leads to a misuse of type *TypestateError*. During the updates, it appears that most developers are trying to fix the problem by adding invocations of method *update*. Nevertheless, they generally still do not call *init* which leaves the issue improperly addressed. More strangely, we noted that in many cases, instead of further completing the fix, developers simply reverted back to the previous misuse version. *IncompleteOperationError* is seen an opposite scenario: after generating the instance of *Mac*, method *doFinal* is never called. Developers update the usage by adding method calls such as *init* or *update* but never add *doFinal* call, which again leaves this issue unresolved.

Table 5.10: The Most Common MFI Update Methods

API	Missing Method		#	%	Explanation
	From	To			
Mac	*update* or *doFinal*	*init*	4,182	28.87%	before update, *init* was call but missing method call of *update* or *doFinal*. However, update even removed method call of *init*.
Mac	*init*	*update* or *doFinal*	3,971	27.41%	expected method call of *init* was added during update but still require further method call of *update* or *doFinal* which is missing.

From the perspective of crypto-API methods, Table 5.11 exhibits details about the top API method invocations which are involved in *MF* or *MI* updates. Note that the top 9 methods for the two kinds of updates are exactly the same. The last two lines of the table are the 10th methods for the two update kinds. As the most commonly misused method (cf., Table 5.5), *getInstance* of *MessageDigest* is also the most often updated method for fixing but also for introducing misuses. The recurrent misuse with this method is about the parameter specifying the hashing algorithm. Weak algorithms such as *SHA1* and *MD5* or sometimes even wrong values, like *SHA*, are used.

Finally, *MFI* updates are generally related to *IncompleteOperationError* and *TypestateError* types, which are all caused by incorrect API method invocation sequences. Due to space limitation, Table 5.10

only shows a couple of example cases of how *MFI* updates occur. These examples show that misuses can be bounced back and forth when API usages require several steps in the invocation sequence.

Table 5.11: Top 10 MF & MI Update Methods

API Method	MF			MI		
	#	%	Rank	#	%	Rank
MessageDigest.getInstance(java.lang.String)	19,183	25.13%	1	16,331	22.64%	1
Reason: parameter with value: MD5, SHA1, SHA						
IvParameterSpec.<init>(byte[])	14,070	18.43%	2	14,234	19.73%	2
Reason: parameter is not randomized						
SecretKeySpec.<init>(byte[],java. lang.String)	13,681	17.92%	3	13,636	18.90%	3
Reason: first parameter is not randomized						
Cipher.init(int,java.security.Key, java.security.spec.AlgorithmParameterSpec)	12,207	15.99%	4	12,097	16.77%	4
Reason: second parameter is not properly generated						
Cipher.init(int,java.security.Key)	6,559	8.59%	6	6,274	8.70%	5
Reason: second parameter is not properly generated						
KeyGenerator.init(int,java.security. SecureRandom)	3,363	4.41%	6	3,285	4.55%	6
Reason: second parameter is not properly randomized (e.g., fixed seed)						
Cipher.getInstance(java.lang.String)	2,553	3.34%	7	1,888	2.62%	7
Reason: parameter with not recommended algorithm (e.g., DES), unappropriated combination of algorithm and feedback mode (e.g., AES/EBC) or with without padding scheme						
MessageDigest.reset()	604	0.79%	8	630	0.87%	8
Reason: missing method call of *digest* or *update*						
SecretKeySpec.<init>(byte[],int,int, java.lang.String)	445	0.58%	9	449	0.62%	9
Reason: first parameter is not randomized						
Mac.doFinal()	366	0.48%	10			
Reason: expected to call method *init* before.						
Signature.initSign(java.security.PrivateKey)				391	0.54%	10
Reason: private key (first parameter) is not properly generated						

Misuses caused by missing steps when using crypto-APIs appear to be difficult to fix. In turn this difficulty is manifested by recurrent failures in API usage updates.

5.5.4 Discussion

Initially, we planned to build on the assumption that app developers are likely to fix crypto-API misuse issues during app evolution. Hence, by mining lineages of a large set of Android apps, one can summarise the crypto-API usage rules. Unfortunately, and also surprisingly, our investigations reveal that :

- crypto-API misues are very common in Android apps.
- app developers are not likely to fix misuses when they update app code.
- For the cases where developers try to fix such misuses, they are often not able to make correct fixes.
- some misuses are recurrently fixed and reintroduced, implying that most of the successful updates might not be performed intentionally to fix the relevant misuses.
- some APIs are more impacted by misuses than others. Misuse updates are however likely to fail as much as to succeed.

We showed that even reputable or sensitive apps are substantially suffering from crypto-API misuses, suggesting that our community is still lacking reliable means to address this problem. Therefore, immediate actions are needed. From app developer side, the recurrence of misuses suggests a need to provide better developer education on how to correctly use crypto-APIs. Similarly, crypto-API providers also need to find better ways to design crypto-APIs to reduce the error margins. Finally, app markets must pay special attention to such apps with misused crypto-APIs, which will create a momentum of developers addressing them seriously.

More concretely, developers should pay extra attention when using *MessageDigest*, *PBEKeySpec* and *Mac*, as they are the widest misused, most misuse-prone and most difficult to be corrected APIs respectively. Choosing algorithms like *SHA-256* instead of *SHA-1* or *MD5* can quickly avoid most of the misuses in *MessageDigest*. While generating salt randomly and remembering to call method *clearPassword()* at the end can make usage of *PBEKeySpec* safe and sound. Finally, the key to use *Mac* correctly is the order of method invocations. Methods *getInstances* and *init* should be always used in the first 2 steps. Method *update* could be invoked more than once afterwards. And *doFinal* should always be called once at last.

5.6 Threats to Validity

For internal threats to validity, our results may be impacted by the dataset selected, which might not be representative. We attempt to mitigate this impact by considering a large set of Android apps. Furthermore, the app versions in a lineage are sequenced based on their version code, which however may not be always true as the version code is configured by app developers. We did not however find any false positives by sampling lineages.

Regarding external threats to validity, our results may be impacted by the false alarms of $CogniCrypt_{SAST}$. We have actually benchmarked a set of apps to check the false positive rates of $CogniCrypt_{SAST}$. Our manual verification confirms that $CogniCrypt_{SAST}$ is effective. We have only observed one case where $CogniCrypt_{SAST}$ may yield false positives, which is related to the artificially created *dummyMainMethod* method (because Android apps do not have a single *main()* like traditional Java code). We have reported this issue to the authors of $CogniCrypt_{SAST}$ and excluded such cases from consideration in this work.

Furthermore, negative result normally refers to a statistical null hypothesis is accepted or an approach is not better than the baseline. While, in this work, we use this term to emphasise the difference between the assumption and the surprising empirical results. Meanwhile, although we investigated our assumption from serval different angles, we did not exhaust all possible ways. Therefore, there are still chances for the assumption to be ture for certain elaborate sub-datasets.

Finally, we have only conducted our experiments on APIs and a few sub-datasets of apps. It might be still possible to mine usage rules on other datasets or other crypto-APIs. More sophisticated approaches may be successful for mining crypto-API usage patterns from the evolution of Android apps.

5.7 Related Work

Crypto-APIs have become a major feature in modern programming languages for encrypting/decrypting sensitive messages, while the misuses of such APIs have also been extensively studied in our community. In this section, we briefly discuss the representative ones.

Misuses of crypto-APIs. CryptoLint is a tool that performs lightweight syntactic analyses for pinpointing violations of hard-coded crypto-API usage rules in Android apps [165]. Similar to

CryptoLint, Crypto Misuse Analyzer (CMA) [166] is also based on hard-coded rules to flag misuses of crypto-APIs. In this work, we leverage $CogniCrypt_{SAST}$ to detect misuses of crypto-APIs in Android apps. To the best of our knowledge, $CogniCrypt_{SAST}$ is so far the most advanced tool for detecting misuses of crypto-APIs. Indeed, the rules hard-coded in CryptoLint and CMA are also contained in the rules of $CogniCrypt_{SAST}$. Therefore, the misuses of crypto-APIs leveraged in this work should be representative and suitable for this study. Also, by manually exploring 49 Android apps, Chatzikonstantinou et al. [167] confirm that at least 88% of the studied apps have misused at least one crypto-API. The ratio obtained in this work is even slightly higher, showing that misuses of crypto-APIs are indeed very common in Android apps.

Mining Usage Patterns in Android Apps. Researchers have reported various pattern mining approaches in the field of Android analyses. For example, Linares-Vasquez et al. [168] have conducted an empirical investigation to mine the energy-greedy API usage patterns in Android apps as well as mine the app usages for generating actionable GUI-based execution scenarios [169]. Similarly, Karim et al. [170] mine Android apps for recommending permissions while Moonsamy et al. [171] aim at mining permission patterns for contrasting clean and malicious Android apps.

5.8 Summary

Crypto-API misuses are common in Android apps. Mining usage rules is thus challenging given the noise in developer code. We hypothesise in this study that usage updates are likely fixing misuses, and may thus be efficiently leveraged for mining usage rules. We perform a large-scale investigation of thousands of Android app lineages and fail to confirm our initial hypothesis. We report these negative results to the community and make available the artefacts of the study.

Availability: `https://negative-crypto-api-mining.github.io/`

6 The Case of Code Reuse in Android via Direct Inter-app Code Invocation

While in previous chapters, we focused on existing vulnerabilities and attempted to study them. This last chapter presents a mechanism for code reuse that is less invasive and thus most challenging to code with by developers, including by industry practitioners. Code reuse is the foundation of software development acceleration, and Inter-Component Communication (ICC) is a standard way to achieve the reuse in Android. However, in this study, we unveil a less known mechanism named Direct Inter-app Code Invocation (DICI) which also can be used to access the code of other apps. We first showcase 2 proof-of-concept apps developed by us to elaborate the mechanism as well as verify its practicality. Furthermore, we develop a tool to detect the use of the mechanism in real world apps. We analyze a large-scale of Android apps with this tool and conclude the situation nowadays. Last but not least, we propose several countermeasures to protect apps from unwanted code access via this mechanism.

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

- Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Borrowing your enemy's arrows: the case of code reuse in android via direct inter-app code invocation. In *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020

Contents

6.1 Overview

Code reuse, a.k.a. *software reuse*, is a form of knowledge reuse in software development that is fundamental to accelerate innovation. Its practice in software engineering is as old as programming itself [173], and has been exacerbated recently within mobile programming frameworks to respond to the needs for keeping up with market requirements of up-to-date functionalities. The facilities offered by Android in this respect have even enabled a large number of software authors to contribute to the application ecosystem, often with little professional training [174].

Reusability is at the core of the Android ecosystem, which builds on the popular Linux kernel and the Java and XML languages to benefit of the extent of device drivers and software libraries to bootstrap functionality development. Unfortunately the staged compilation process as well as the packaging model makes Android apps straightforward to reverse engineer and copy. This has led to a situation where *cloning* (a.k.a., *repackaging*) is commonplace [175, 16, 176, 177, 178]. At an inner-level, Android intents and intent-filters facilitate decoupling and assembling of app components, providing opportunities for reuse of existing components to interact with new components. For example, malware writers are extensively exploring these reuse facilities to *piggyback* malicious code on legitimate app by leveraging events (e.g., SMS incoming broadcast) to trigger malware execution. More generally, *component hijacking* in Android has been largely investigated in the literature [133, 6]: by evading permission checks, an Android app may access resources that it is not allowed to. In this respect, Inter-Component Communication (ICC) analyses [179, 180, 181, 56, 68, 182, 183] have been proposed to track data leaks as well as to detect permission redelegations attacks [3]. Further investigations were performed in the literature towards uncovering potential *app collusion* [184, 185], i.e., cases where a set of apps are able to carry out a threat in a collaborative fashion. App collusion is indeed generally associated to information leakage and inter-app communication where developers leverage Android implicit and explicit messaging services to orchestrate legitimate rich scenarios or devise sophisticated attacks.

In this study, we dissect a less-advertised reuse mechanism that is available in the Android framework, through which developers can invoke a given functionality code implemented in another app. We refer to it as **Direct Inter-app Code Invocation** (DICI). To the best of our knowledge, this mechanism was never mentioned in the Android research literature. DICI is often used in legitimate contexts such as across Google Mobile Services[1] to enable functionality reuse among apps. Nevertheless, as we will demonstrate in Section 6.2, DICI can be used maliciously to plagiarize other-wise protected functionalities and by-pass standalone app analysis.

The DICI mechanism achieves inter-app interactions without leveraging the Android standard inter-component communication primitives. This mechanism builds on Java reflection and a set of dedicated API methods that are provided within the Android framework. DICI differs from existing reuse mechanisms in various ways: (1) In contrast to cloning of entire apps or copy/paste of code fragments, DICI does not require including the targeted functionality code in the attacking app. This property is leveraged by attackers to bypass security assessment where the attacking app is analysed. It also results in an app of smaller size, while avoiding potential decompilation issues (e.g., some code cannot be decompiled properly) with the attacked app. Finally, the attacking app is easy to maintain when the attacked app is updated. Last but not lest, DICI also requires little understanding of the implementation details of the leveraged apps since developers only need to know and invoke the entry method for a given functionality: relevant methods and classes will be loaded and invoked automatically even for native methods. (2) Unlike ICC, DICI can allow the invocation of functionality that is not implemented within an Android component, such as a library function in another app. In other words, DICI widens the reuse surface: with DICI any code can be invoked, not only code that is in specific components such as with ICC. (3) Finally, DICI can be leveraged to implement stealthy code reuse. Indeed, while with ICC the developer of the reuse target may be aware that her code

[1] GMS are the apps by Google that often come pre-installed on Android devices. They are not part of the Android Open Source project

could be reused, it is not necessarily the case for DICI. In Android, a component, such as an Activity or a Service, has its "`exported`" attribute set as "`True`" when the developer wishes to allow ICC from another app. Such a developer may then take appropriate measures to ensure component security. In the case of DICI, a developer of an app is not aware that her code will be invoked by a third party.

The main contributions of this work are:

- We expose a little-advertised reuse mechanism within the Android ecosystem. In particular we demonstrate how it can be leveraged to perform stealthy functionality plagiarism that may not be covered by standard licensing scheme.
- We develop a static analysis tool, `DICIDer`, for the detection of DICIs in Android apps. We perform an empirical analysis on the prevalence of DICIs among a large dataset of apps retrieved from the AndroZoo repository [186]. We further provide extensive discussions on how and why developers use DICIs through an analysis of sample cases.
- We propose an example of countermeasure that could be used by developers to protect their apps against DICI.

6.2 Dissection of the DICI Mechanism

We provide a problem statement for the direct inter-app code invocation mechanism (Section 6.2.1) and showcase some motivating examples of reuse based on this mechanism (Section 6.2.2).

6.2.1 Direct Inter-App Code Invocation in Android

Given the lack of related information on the mechanism of Direct Inter-app Code Invocation within the Android research literature, we contribute to the body of knowledge by presented an overview of the mechanism. DICI is a mechanism for inter-app communication (i.e., the possibility for one app to leverage resources, either functionality or information, from another app during its execution). Figure 6.1 summarizes how inter-app communication works in Android by illustrating DICI in comparison with the standard ICC (i.e., inter-component communication).

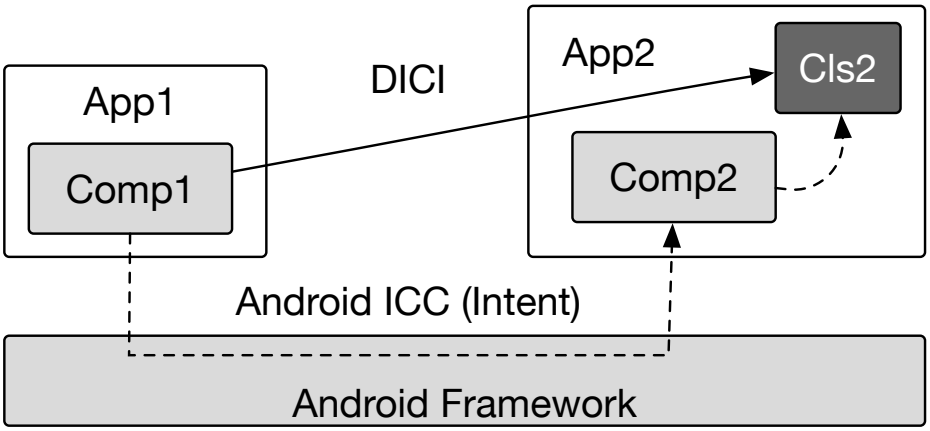


Figure 6.1: The two types of inter-app communication: Android ICC and DICI.

ICC is the standard mechanism used by developers (as recommended in the Android documentation) to achieve inter-app communication. Its capabilities and challenges have been (and still remain) intensively studied in the research literature. ICC is an Android-specific mechanism that was implemented to enable interaction among Android components, i.e., the basic units that are composed to form apps. Indeed, the four types of components, namely Activity, Service, Broadcast Receiver,

and Content Provider, which are responsible for different tasks, cannot directly invoke each other's functionalities. Developers must then rely on specific ICC methods, such as *startActivity()*, to achieve this interaction. As illustrated in Figure 6.1, an Android ICC is triggered by the Android framework. Thus, there is no direct connection between the source and target components *at the code implementation level*. When the target component of an ICC belong to a different app than the source component, an inter-app communication is implemented.

Beyond ICC, we have come across cases in the practice of Android app development where inter-app communication can be achieved through direct code invocation. We refer to such a mechanism as DICI. As illustrated in Figure 6.1, DICI can not only (1) bypass the Android ICC mechanism to realize inter-app communication, but also (2) directly invoke non-component code (e.g., standard Java class *Cls2* in Figure 6.1). The latter capability is not available through the recommended ICC mechanism.

Problem statement. To date, ICC-based inter-app communication has been widely investigated by the research community [187, 185, 188, 179, 56, 189]. The literature provides extensive results on tracking information flow through ICC, statically flagging how such inter-app communications are leveraged by attackers to achieve malicious behaviours (e.g., privacy leaks). Studies of how malware is written in bulk through reusing legitimate apps (i.e., piggybacking [16]) have mainly focused on investigating how ICC is relied upon to trigger malicious payload.

Our hypothesis is that DICI, yielding a larger reuse surface, poses challenges that are at least as acute as for ICC. Indeed, DICI may be used by attackers to achieve malicious code invocations while bypassing the security analysis tools which have been focused on ICC-based scenarios. Furthermore, our work raises awareness among the developer community on the possibility of having their functionalities reused without their knowledge by either plagiarists or malicious attackers. Finally, for the research community, exposure to this reuse mechanism in the Android realm re-opens a variety of research directions.

6.2.2 DICI in Action

We highlight the possibilities that offer the DICI mechanism with two motivational use cases. Both cases involve the development of apps that reuse code available in other apps.

1. `StealthApp` is designed to orchestrate an app collusion scenario for private data leak. In this example, we focus on the possibility of leveraging DICI to hide malicious code in order to increase the chances of escaping detections that are attempted via static analyses.
2. `TikTokDownloader` showcases the critical possibilities that DICI provides in terms of plagiarism. In this example, functionalities (including backend infrastructure) of one of the most popular apps in the world, namely TikTok [190], are reused to build at no-cost a video-sharing app. In particular, we show that (1) we can reuse several TikTok functionalities, (2) we can provide additional functionalities that are initially forbidden by TikTok, (3) we can even leverage the full infrastructure of TikTok.

Malicious Code Hiding

The International Mobile Equipment Identity (*IMEI*) is a number used as a standard to identify mobile phones. It is considered to be a key private information and should be kept private [191, 192, 193]. Consequently, APIs to obtain the IMEI are classically considered in the list of "sources" for data-flow analyses [55, 194], thus facilitating detection of leaks, even when ICC are used. We propose to leverage DICI to orchestrate the leakage of the *IMEI* via SMS: the goal is to build exclusively on code that is implemented from other apps (1) to retrieve then (2) to leak the IMEI. We consider this use case to be reasonable since on a given device it is highly likely to identify other apps that implement

a code fragment for sending SMS and another app that has code where the IMEI is retrieved. By doing so, we ensure that there is no explicit code in our developed app (i.e., `StealthApp`) where neither IMEI collection can be matched (e.g., via tracking calls to API) nor leaking via SMS can be identified. Therefore our implementation of such a collusion, with DICI, challenges the detection of security leaks in `StealthApp`.

Listing 6.1 provides an excerpt of the code used in `StealthApp` to invoke a method (*getDeviceID* at line 13) from another app (`org.communicorpbulgaria.bgradio` at line 4). Note that in this code the actual API provided by the framework (i.e., `android.telephony.TelephonyManager`) is hidden. DICI is implemented in this case through reflection after obtaining the context of the app that implements the code to reuse (lines 3-7). Based on this context, the class loader of the app can be obtained (line 8) and used to load relevant classes in the app (line 9-11). The method object is acquired with the class object containing it (line 12-15). Since *getDeviceID*, which is implemented by the target app, is a static method, it is invoked directly to finally get the *IMEI* number (lines 16).

```
1  private String getImei() {
2      String imei = null;
3      Context invokee = this.createPackageContext(
4          "org.communicorpbulgaria.bgradio",
5          Context.CONTEXT_INCLUDE_CODE |
6          Context.CONTEXT_IGNORE_SECURITY
7      );
8      ClassLoader loader = invokee.getClassLoader();
9      Class util = loader.loadClass(
10         "org.ccb.radioapp.components.Utils"
11     );
12     Method getDeviceId = util.getDeclaredMethod(
13         "getDeviceID", Context.class);
14     imei = (String) getDeviceId.invoke(null, this);
15     return imei;
16 }
```

Listing 6.1: Retrieval of IMEI through reflection for third-party code reuse

Similarly, a method from another third-party app is invoked to send the obtained *IMEI* via SMS as shown[2] in Listing 6.2. This method is named *sendSms* (line 12), but it cannot be confused with framework APIs for sending APIs. Instead, this method is contained in class `CommonUtils` (line 9) from app `com.globalcanofworms.android.simpleweatheralert` (line 3).

```
1  private void sendMsg(String num, String msg) {
2      Context invokee = this.createPackageContext(
3          "com.globalcanofworms.android.simpleweatheralert",
4          Context.CONTEXT_INCLUDE_CODE |
5          Context.CONTEXT_IGNORE_SECURITY
6      );
7      ClassLoader loader = invokee.getClassLoader();
8      Class util = loader.loadClass(
9          "com.globalcanofworms.android.coreweatheralert.CommonUtils"
10     );
11     Method sendSms = util.getDeclaredMethod(
12         "sendSms", Context.class, String.class, String.class);
13     sendSms.invoke(null, this, num, msg);
14 }
```

Listing 6.2: Data transfer via SMS through reflection for third-party code reuse

`StealthApp` performs a malicious behavior, through app collusion, without explicitly implementing any malicious code. As long as all the apps targeted for reuse are available on the users device, its *IMEI* can be leaked and yet, both dynamic and static scanning techniques will systematically fail to spot this leak if apps are analyzed individually. In any case, even when the apps are available, it is

[2]The aforementioned code snippets are simplified with absence of exception handling.

important to note that the use of reflection makes the use static analysis techniques challenging. While some techniques (e.g., code instrumentation of reflective calls into direct calls with DroidRA [195]) have been proposed in the literature to overcome limitations raised by reflection, these techniques generally target in-app code (e.g., dynamically loaded classes from an extra dex file) Such method would not work for DICI since the method that should be called is not present in the analyzed app.

⚠ **Limitations of the `StealthApp` use case:** We have developed a naive app collusion system with `StealthApp` as a proof-of-concept of hiding malicious code with DICI. The goal, with this use case, is not to implement a sophisticated attack. Besides its simplicity, this use case presents several limitations:

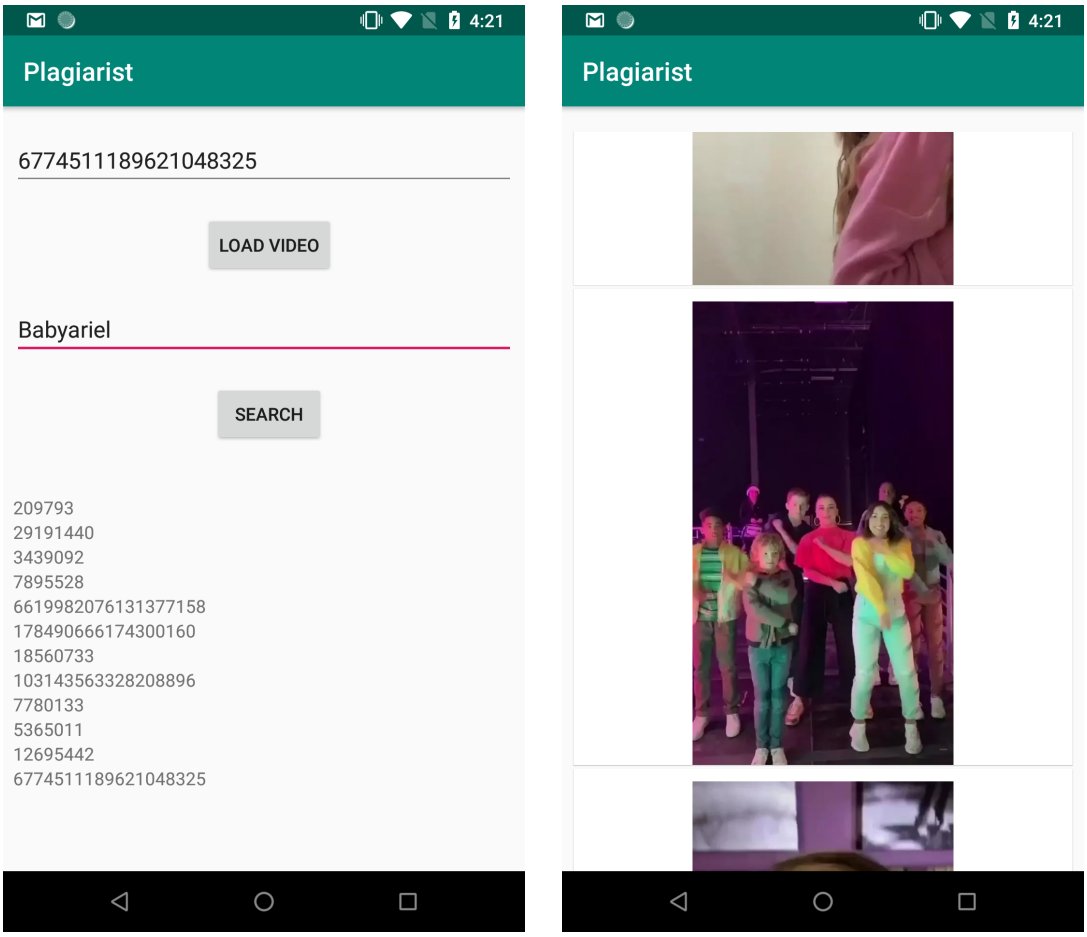
- *Availability of target apps.* The implementation of the malicious behavior depends on the installation status of other apps to orchestrate the app collusion. Their probability of availability on the device could lower the possibility of the execution of the malicious code. Nevertheless, we can expect attackers to leverage the diversity of apps that are shipped with new devices. For example, hackers could list all the functionalities offered by the apps that are already installed on all devices from a specific manufacture, or consider only focusing on popular apps to increase the probability of being able to realise the scenario on millions of devices. Finally, note that the official API `PackageManager.queryIntentActivities` with *Intent* category set to `CATEGORY_LAUNCHER` can be used to retrieve at runtime the relevant information on installed apps on the current device.
- *Permissions.* Another limitation is that permissions of other apps will not be granted to `StealthApp` when `StealthApp` is invoking their code. Therefore, when a method is protected by a permission, this permission must be granted as well to the app before invoking the third-party code. For our *IMEI* leakage example, the *READ_PHONE_STATE* and *SEND_SMS* permissions are required in `StealthApp`. Nevertheless, because of the recurrence of permission over-privilege (i.e., apps ask for more permissions than they need) in the Android ecosystem [196, 197], attacks such as the one perpetrated by `StealthApp` can go unnoticed.
- *Process access.* Finally, it is noteworthy that in the case of ICC, when an app A is "calling" a component of an app B, that component is launched in the process of B, i.e., the target code that is run in B can access the internal data of B. With DICI, when an app A invokes code from B, this code is launched in the process of A, meaning that this code cannot actually access internal data of B. Nevertheless, despite this limitation for accessing more resources, accessing functionality implementation poses different threats as we will show in the second use case.

Functionality Plagiarism

TikTok is a highly popular video-sharing app. It has more than 500 million installs on *Google Play* alone. In line with the necessity to control copyrights of video submitters as well as due to commercial needs to strongly bind users, all the shared videos can only be viewed and downloaded through the single TikTok app. Among other constraints, TikTok does not allow allow batch downloading (i.e., the possibility to download all of the videos of a single user at once). In order to block download requests originating from third-party interfaces, each request need to be appended with a one-time "signature" for the Tiktok server to verify the legitimacy of the request. This "signature" is calculated by an algorithm implemented within the user app with certain information such as user ID, time stamps, etc. These mechanisms are rather effective against the typical cloning (i.e., repackaging attack) or the reverse engineering of the TikTok app in order to exploit the backend infrastructure and resources of Tiktok, notably the database of videos.

We will show now that, with DICI, it is actually possible to *reuse* the code of TikTok to achieve the objective of exploiting the TikTok infrastructure. Typically, we were able to implement our own batch downloader, that we call `TikTokDownloader`, to download *TikTok* videos by accessing

and plagiarizing the signing algorithm implementation in the *TikTok* app. As shown in Figure 6.2, the developed app will require just to input a user ID to specify the videos of which user must be downloaded.



(a) Search User ID (b) Downloaded Videos

Figure 6.2: Batch Downloader Snapshots

TikTok implements video search and download via REST endpoints (i.e., an URL where requests can be specified for actions or resources). However, as endpoints are accessed via requests that are transmitted in plain text with logical structures, they can be obtained, manipulated and used easily by third parties. Thus, to reserve the exclusive use of these endpoints to *TikTok* itself, a one-time "signature" is required to be appended to each endpoint when requesting the server. After investigating the DEX[3] code of TikTok, we identified a method whose obfuscated name is "*a*" within class `com.ss.android.ugc.aweme.app.a.c`, which computes the signature for the app to access the TikTop server resources. Listing 6.3 presents the implementation code of method "a" which is the target of our plagiarism scenario.

`TikTokDownloader` implements the DICI mechanism to invoke the method illustrated in Listing 6.3 in order to sign the endpoints and then request the server with the signed endpoints. It is worth to mention that all of the relevant classes, such as *NetworkUtils* in line 6, will be automatically loaded as well. This constitutes a powerful capability of the DICI mechanism since even native libraries (generally preserved from reverse-engineering due to their machine binary format) can also be reached: for example, in the sample code, *getUserInfo* (line 32), *byteArrayToHexStr* (line 38) and *e* (line 39) are all sensitive code that are embedded in native code.

[3] *JADX* is used here for the decompiling, it can be found at `https://github.com/skylot/jadx`

```
1  private String a(String str) {
2      int i;
3      String userInfo;
4      String str3;
5
6      int serverTime = NetworkUtils.getServerTime();
7      if (serverTime < 0) {
8          i = 0;
9      } else {
10         i = serverTime;
11     }
12     String str4 = str + "&ts=" + i;
13
14     HashMap hashMap = new HashMap();
15     d.a(hashMap, true);
16     String[] strArr = new String[(hashMap.size() * 2)];
17     int i2 = 0;
18     for (String str5 : hashMap.keySet()) {
19         String str6 = (String) hashMap.get(str5);
20         if (str5 == null) {
21             str5 = "";
22         }
23         if (str6 == null) {
24             str6 = "";
25         }
26         int i3 = i2 + 1;
27         strArr[i2] = str5;
28         strArr[i3] = str6;
29         i2 = i3 + 1;
30     }
31
32     userInfo = UserInfo.getUserInfo(i, URLDecoder.decode(str4), strArr, "");
33
34     int length = userInfo.length();
35     String substring = userInfo.substring(0, length >> 1);
36     str3 = (str4 + "&as=" + substring + "&cp=" +
37         userInfo.substring(length >> 1, length)) + "&mas=" +
38         com.ss.android.common.applog.i.byteArrayToHexStr(
39             com.ss.sys.ces.a.e(substring.getBytes()));
40     return str3;
41 }
```

Listing 6.3: Simplified Signing Method from TikTok

The usage scenario of our `TikTokDownloader` app is that it is installed on a device where the user already has an account on TikTok. The DICI mechanism in this case has led to the implementation of copyright infringement attacks (since video uploaders did not provide any rights to `TikTokDownloader` to access their content). Another critical point is that DICI allowed to easily plagiarized the TikTok code in a stealthy: `TikTokDownloader` did not copy the code, nor did it rewrite in some way; instead it just invokes it at runtime, a case that is not comprehensively studied in the literature of code plagiarism.

Apps availability and responsible disclosure: We provide on GitHub the source code of both use-case apps as artefacts for further research: *https://github.com/gaojun0816/FSE-anonymous-artefact*. Both apps have been tested on a *Nexus* 5 device running Android version 8.1.0. We have also responsibly informed TikTok owner company about the risk posed by DICI with respect to the possibility to bypass their security infrastructure to access users copyrighted videos.

6.3 Tool Design

Aiming at automatically inferring the usage of DICIs in Android apps, we design and implement a prototype tool called DICIDer, which takes as input an Android APK file and outputs a list of DICI paths that trace how direct inter-app code invocations are planned in the analyzed app. An overview of the working process of DICIDer is presented in Figure 6.3. Overall, DICIDer follows four steps to pinpoint DICI instances. We now briefly introduce these steps.

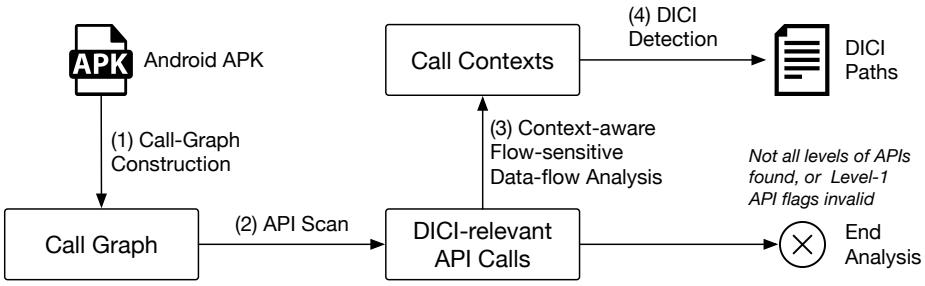


Figure 6.3: Static Analysis for Uncovering DICIs.

6.3.1 Step (1): Call-Graph Construction

DICIs are implemented following a sequence of API calls (e.g., to obtain the third party app context, load the relevant class, invoke the target code, etc.). We thus propose to construct the call graph of the input Android app to facilitate further analyses. To that end, DICIDer relies on the Soot Framework [198] as well as the FlowDroid [55] precise taint analysis tool. To realize this analysis, the apk is first disassembled and the app Dalvik bytecode is transformed into Jimple, the intermediate representation that is leveraged in Soot. Then, this Jimple code is analyzed by Soot to yield a call graph of the app.

We recall that Android apps do not come with a single entry-point (e.g., *main* in classical Java applications) to start app execution. Instead, the app can be started from different entry points, from any app components, which complexifies the construction of a single call graph. To address this problem, FlowDroid constructs a dummy main method for analysis. This dummy main method takes into account all the possible entry-points of the app (i.e., components) and their lifecycle methods (e.g., *onStart()*, *onStop()*) as well as all the leveraged callback methods (*onClick()*). The reason why lifecycle methods and callback methods are needed to be explicitly included is that these methods are not explicitly connected at the code level. The prepared dummy main method then enables the Soot to construct the call graph of the app and subsequently to traverse all the app code in their possible execution contexts. Note that Soot implements several in-house call graphs construction algorithms such as CHA and SPARK. While the CHA algorithm is faster than SPARK, it is rather more imprecise [198]. Given that in our work, precision of call graph is a key property to ensure that DICIDer yields good performance, we choose to leverage the SPARK algorithm to build the call graph (with the correct API calling sequences modeled).

6.3.2 Step (2): API Scan

Once the call graph is constructed, the second step that is unfolded is to identify the relevant APIs that contribute to the realization of DICIs. Then, one must assess the parameters of these API calls to further confirm potential code reuse scenarios.

API presence detection: DICIDer performs a quick scan over the call graph to check if DICI-relevant APIs of the Android framework are leveraged by the app. The presence of such APIs is a

primary condition for the presence of DICIs in the analyzed app. If such APIs do not exist, there is no need to proceed further, and the analysis of the app is safely halted.

Table 6.1: DICI-relevant APIs

Level	Class	Signature	Return	Description
1	android.content.Context	createPackageContext(java.lang.String,int)	android.content.Context	used to create a new context object of a specified application. The arguments are application name and creation flags. With flag *CONTEXT_INCLUDE_CODE* and *CONTEXT_IGNORE_SECURITY*, the code of another application can be loaded.
2	android.content.Context	getClassLoader()	java.lang.ClassLoader	get the class loader.
3	java.lang.ClassLoader	loadClass(java.lang.String)	java.lang.Class	get a specified class object by passing the name of the class.
4	java.lang.Class	getConstructor(java.lang.Class[])	java.lang.reflect.Constructor	get the constructor of a class with the argument specifying the signature.
		getDeclaredConstructor(java.lang.Class[])		
4	java.lang.Class	getDeclaredMethod(java.lang.String,java.lang.Class[])	java.lang.reflect.Method	get the method of a class with the arguments specifying the signature.
		getMethod(java.lang.String,java.lang.Class[])		
4	java.lang.Class	getDeclaredField(java.lang.String)	java.lang.reflect.Field	get the field of a class by passing the name.
		getField(java.lang.String)		
5	java.lang.Class	newInstance()	java.lang.Object	instantiate a class with its zero-argument constructor.
5	java.lang.reflect.Constructor	newInstance(java.lang.Object[])	java.lang.Object	instantiate a class with the specified constructor.
5	java.lang.reflect.Method	invoke(java.lang.Object,java.lang.Object[])	java.lang.Object	invoke the method. The first argument specifies the instance of the class and passing *null* indicates a class method.
5	java.lang.reflect.Field	set(java.lang.Object, java.lang.Object)	void	set the field with a certain value. The first argument indicates the object to which the field belongs and *null* means the field is static. For set*(java.lang.Object, *), the asterisk can be replaced with boolean, byte, char, double, float, int, long and short. For example, setInt(int).
		set*(java.lang.Object, *)		
		get(java.lang.Object)	java.lang.Object	get the value of a field. The asterisk stands for the same primary types mentioned in set*(java.lang.Object, *).
		get*(java.lang.Object)	*	
5	java.lang.reflect.Method java.lang.reflect.Field java.lang.reflect.Constructor	setAccessible(boolean)	void	set the accessibility of the method, field or constructor.

Which are the DICI-relevant APIs? In Section 6.2.2, our use-case description highlighted a sample sequence call of specific Android APIs. Following up on this example, we have carefully investigated APIs that are used in the same principles, and tag them as DICI-relevant. Table 6.1 enumerates all DICI-relevant APIs considered by DICIDer, along with their implementation class, signature, return type and a textual description. Since DICIs are performed through a sequence of calls of several DICI-relevant APIs, each API may be necessary at different position/level within the sequence. We indicate for each DICI-relevant API the **level** of that API, which represents the position of its call within an instance of DICI call sequence. Generally, a successful DICI needs to involve at least one API in each of the five levels' APIs.

- **Level 1:** Obtain the *context* of another app.
- **Level 2:** Obtain the corresponding *class loader* using the obtained *context* of the other app.
- **Level 3:** Load the *class* (to be directly invoked) of the other app through the obtained class loader.
- **Level 4:** Locate the *constructor*, *method*, or *field* (to be directly reused) from the loaded class.
- **Level 5:** Finally, access the previously located constructor, method, or field reflectively. If the method or field is not declared as *static*, an additional step is needed to instantiate an object of the class.

DICIDer uses the list of DICI-relevant APIs to check whether the analyzed apk contains such APIs. In particular, if the analyzed apk does not contain at least one DICI-relevant API of each of the five levels enumerated previously, the API call sequence is ignored at this stage and DICIDer terminates with no DICI paths detected.

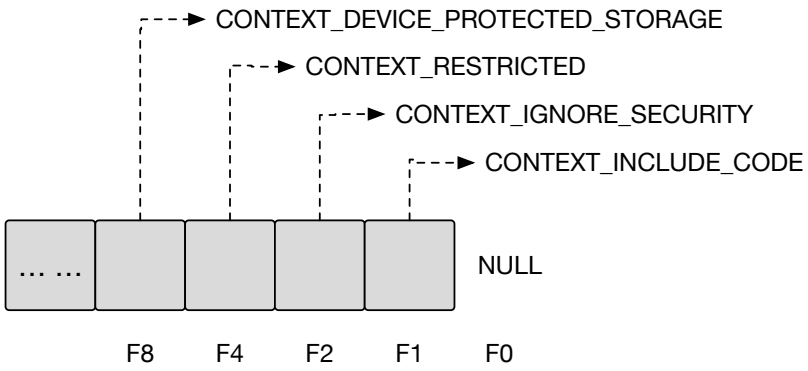


Figure 6.4: Example options that can be applied when creating contexts via package names.

API parameter checking: There is another constraint that may keep DICI from happening in practice. This constraint is brought by the second parameter of the *createPackageContext()* API. This second parameter, known as *flags*, allows developers to specify (via bitwise operators) how should the package context be created. Some of the options that developers can specify are highlighted in Figure 6.4 and briefly explained below.

- *F0 (or 0000).* The default option. None of the other options are enabled.
- *F1 (or 0001).* If enabled, **it allows the context to access the code implemented in the loaded package**. Otherwise, only resource files are allowed to be accessed.
- *F2 (or 0010).* This option will ask the context to ignore any security restrictions. When enabled along with *CONTEXT_INCLUDE_CODE*, it will allow code to be loaded into a process even when it is not safe to do so. As recommended by Google, developers should use this option with extreme care[4].
- *F4 (or 0100)*: This option will allow the context to disable specific features of its accessed resources.
- *F8 (or 1000).* This option allows the context to access APIs even at device-protected storage.

In order to invoke the code of other apps, when creating the context via *createPackageContext()*, the F1 option has to be enabled. Therefore, in this step, we further take efforts to trace the value of the *flags* parameter (through backward constant propagation) of located *createPackageContext()* usages. If the F1 option is not enabled, the corresponding API call will not be considered, so as to avoid false-positive results.

6.3.3 Step (3): Context-Aware Flow-Sensitive Data-Flow Analysis

While a call-graph is relevant to spot call sequences of DICI-relevant APIs, this sequence may actually not be about implementing a DICI. Indeed, APIs may be invoked under different contexts, leading to a situation where there is no actual inter-app interaction. Thus, we need to ensure that the code block that is eventually invoked is indeed reused from another app. Let us consider the example provided in Listing 6.4. This Listing is similar to the beginning of Listing 6.1 for the case of malicious code hiding: there is a difference in that the `loader` is instantiated by calling `this.getClassLoader()` rather than `invokee.getClassLoader()`. As a result, although the APIs of the first two levels are invoked following the ideal sequence (i.e., Level-1 method is called before the Level-2 method), they

[4] `https://developer.android.com/reference/android/content/Context`

cannot jointly form a DICI as the *loader* is not obtained from the context *invokee* but the current context (i.e., *this*).

```
1 Context invokee = this.createPackageContext(
2   "org.communicorpbulgaria.bgradio",
3   Context.CONTEXT_INCLUDE_CODE |
4   Context.CONTEXT_IGNORE_SECURITY);
5 ClassLoader loader = this.getClassLoader();
6 //ClassLoader loader = invokee.getClassLoader();
```

Listing 6.4: An example code showing the necessity of taking context into consideration.

We address the challenging of keeping track of the data-flow between API calls by performing a *context-aware data-flow analysis* to ensure that the APIs are all called under the same context. Nevertheless, instead of performing a generic context-aware data-flow analysis, which tracks all the flow of all the variables and hence could be compute-intensive, DICIDer implements a dedicated context-aware data-flow analysis for which only the contexts related to the DICI-relevant APIs are tracked.

6.3.4 Step (4): DICI Usage Identification

Finally, in the last step, DICIDer leverages the results of the previous steps to pinpoint DICI paths. We recall that the output of step 3 is a DICI path which is a sequence of API calls with a least one API for each defined level and called under the same context. However, at this stage, it is still not established which app, class and method are invoked via DICI, i.e., what is the target code for reuse. We introduce a lightweight *constant string propagation module* in DICIDer, which goes one step deeper to infer what are the methods/fields that are accessed via DICI. To that end, given a fifth-level API, such as *java.lang.reflect.Method.invoke()*, we perform a backward string analysis to infer which is the reflectively-accessed artifact. Regarding the example shown in Listing 6.1, for the *invoke* method illustrated in Line 16, our backward string analysis aims at inferring that the method, which is called via reflection, is *getDeviceID()* of the class *org.ccb.radioapp.components.Utils* in app *org.communicorpbulgaria.bgradio*.

6.4 Evaluation

We empirically assess DICIDer, and investigate the use of DICIs in the real-world.

Research questions: The study is driven by the following research questions (RQs).

- **RQ1:** *Can* DICIDer *spot DICIs in real-world Android apps?* To answer this RQ, we investigate on the one hand the recurrence of DICIs in apps collected from various markets. On the other hand, we study the prevalence of DICIs among goodware and malware apps respectively.
- **RQ2:** *How DICI usages evolve over time?* To answer this RQ, we consider both the evolution of number of apps leveraging DICIs within markets, as well as the evolution of DICIs usages within app lineages (i.e., based on their updates).
- **RQ3:** *For what purposes do developers implement DICIs?* We consider a number of real-world examples to dissect the purposes of DICI usages.

Dataset: The evaluation is conducted on apps collected from AndroZoo [186], a continuously growing repository of Android apps. At the time of writing, the dataset size was over 10 million apks crawled from the Google Play official store as well as from alternative markets and repositories. Some metadata on the apps are also collected via the toolkits provided by Li et al. [199].

Implementation: DICIDER is built based on the Soot Framework and leverages FlowDroid taint analysis implementation. DICIDER provides reasonable performance on a commodity computer (2.9 GHz quad core Intel Core i7 CPU with 16GB memory): the average time consumption for analysing a single apk is about 62.72 seconds.

6.4.1 RQ1: DICIs in Real-World Apps

The goal is to run DICIDER in order to attempt the detection of DICIs in real-world Android apps. To that end, we sample Android apps following their market provenance.

Comparison among Markets. Currently, the top-4 sources ranked based on the number of apps crawled in AndroZoo are Google Play, PlayDrone, Anzhi and AppChina. However, since PlayDrone is a specific subset of apps originally crawled from Google Play, we do not consider PlayDrone as a distinct provenance. Thus, we consider mainly the remaining 3 sources and randomly select 25,000 apps from each provenance[5] leading to a total of 75,000 Android apps.

Table 6.2: DICI Comparison among Markets

	Google	**Anzhi**	**AppChina**	**Total**
# of successfully analyzed apps	25,000	25,000	25,000	75 000
# of apps with DICIs	4,344	100	135	4579
Percentage of apps with DICIs	17.38%	0.40%	0.54%	6.11%
# of detected DICIs	4,396	1,051	227	5674
Median # of DICIs per App[6]	1	13	1	1

Table 6.2 provides statistics of the execution of DICIDER on the 75k real-world apps. Overall, DICIDER is able to detect a significant number of DICIs. At the market level, we notice that apps from the official market, *Google Play*, are much more likely to contain DICIs than apps from the alternative markets. A priori, this is reassuring since alternate markets are known to include more malicious samples than the official markets [200]. Nevertheless, when looking at the median number of DICIs per app, apps from market *Anzhi* exhibit a remarkably higher number of DICIs than for other markets when considering apps that implement this reuse mechanism. Figure 6.5 gives a more concrete understanding of the difference between *Anzhi* and the other markets from the perspective of DICI per app. Further statistical investigations of *Google Play* cases reveal that about 97% of DICIs are from a class named *com.google.android.gms.dynamite.DynamiteModule*.

Google Mobile Services and DICIs. We focus on the *DynamiteModule* class that is recurrently involved with DICIs of GooglePlay apps. Based on its package name, we suspect that it is may be part of the official *Google Mobile Service* (GMS) APIs. The official documentation does not however mention such a package. We postulate that such a package may have been intentionally omitted from the documentation to avoid uses by third-party developers. Nevertheless, we undertake to confirm the presence of this class within GMS by explicitly requesting Gradle dependency management to find the GMS libraries and included them in a toy/demo app. Afterwards, we manually analyzed the content of the class to further check what it does through DICI. According to the analysis report of DICIDER, *DynamiteModule* code instantiates a class named *com.google.android.gms.dynamite.IDynamiteLoader* from the app named *com.google.android.gms*. To further check how this instance is used, we proceed to reverse-engineer an app that contains GMS APIs the code of such APIs are not open-sourced. According to the decompiled code, this class implements the interface *android.os.IBinder* which is designed for in- and cross- process calls[7] and is used to query a local interface here. Although this class is also under the package of GMS *dynamite* according to its name. It cannot be found in the GMS libraries. Since there is quite little information about these libraries. We can only infer that

[5]Since DICIDER may fail to analyze some apps due to unexpected corner cases such as the given APK does not contain DEX file, in practice, we have randomly listed all apps and sequentially tested them until the quota of 25000 is reached for each provenance.

[7]According to official document at `https://developer.android.com/reference/android/os/IBinder`

the app *com.google.android.gms* is the GMS framework which is supposed to be embedded into the Android OS, and direct inter-app code invocation is the way to access the framework.

We also consider class *org.xwalk.core. XWalkCoreWrapper* which contributes to most DICIs in Anzhi dataset. Class *XWalkCoreWrapper* is from a project called *CrossWalk* which was once founded by *Intel's Open Source Technology Center*[8]. It is a web app runtime to provide manipulability to browser. The class uses DICIs to access functionalities of its own app.

While we studied the recurrent cases in this RQs, we will consider the remaining 3% in Google Play datasets for answering RQ3.

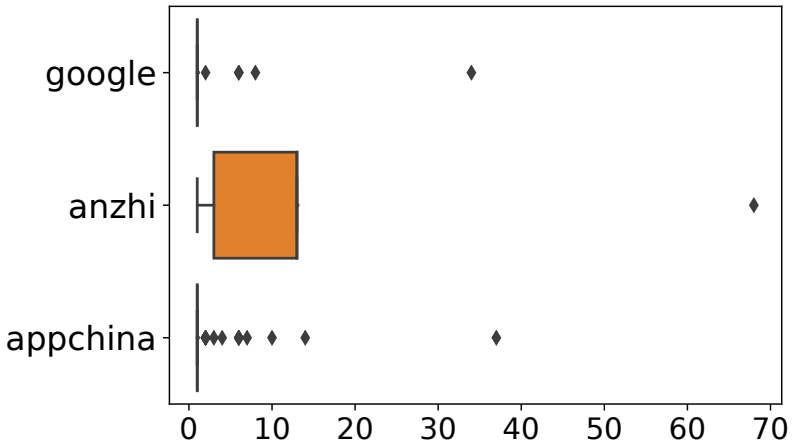


Figure 6.5: Distribution of DICI per App

Comparison between Benign Apps and Malware. AndroZoo not only crawls Android apps, but also the antivirus reports from VirusTotal [201] for each app. For a given app, AndroZoo indicates the number of anti-virus products which flag the app as a malware (among a total of about 60 anti-virus products). We rely on this information to build our dataset of goodware and malware. For goodware, we consider 25,000 apps selected from AndroZoo with no flag from any anti-virus product. For malware, we consider 25,000 apps selected from AndroZoo with at least 30 flags (i.e., at least half of the anti-virus have a consensus on app maliciousness).

Table 6.3: DICI Comparison between Goodware and Malware

	Benign	**Malware**
# of successfully analyzed apps	25,000	25,000
# of apps with DICIs	5,836	52
Percentage of apps with DICIs	23.34%	0.2%
# of detected DICIs	5,964	101
Median # of DICIs per App[9]	1	1

Table 6.3 presents the comparison between Goodware and Malware. Surprisingly, overall, malware actually use much less DICIs compared to benign apps. However, by further checking the source package of DICIs, we find that for benign apps, the dominated class is again *com.google.android.gms.dynamite.DynamiteModule* While, for malware, this class only contributes a quarter of DICI usages. This has two implications: the scope of using DICI for benign scenarios is still limited, although many instances of benign apps, because of their reliance on GMS, are actually hosting code that use the DICI mechanism; malicious apps on the other hand may have indeed been leveraging DICIs.

[8]See project page at `https://crosswalk-project.org/`

Answer to RQ1: DICIDER is able to detect DICIs in real-word apps. This reuse mechanism is actually seen in many apps, although mostly due to the use of the GMS libraries where class *com.google.android.gms.dynamite.DynamiteModule* heavily relies on DICI. There are however cases of malware leveraging DICI outside the scope of GMS libraries.

6.4.2 RQ2: Evolution of DICI Usages

Table 6.4: # APKs considered from the Lineage dataset [1]

2011	2012	2013	2014	2015	2016	2017	2018
3,950	6,252	13,191	23924	17,505	30,690	6,995	3,583

As shown in RQ1, *com.google.android.gms.dynamite.DynamiteModule* is the major source of DICIs. We noticed that this class was not present in the Android ecosystem since the beginning of Android. Thus, we propose to study the evolution in time of the number of DICIs within Android apps. To perform this experiment, we consider app lineages (i.e., different versions of apps over time). To that end, we consider a large lineage dataset proposed by Gao et al. [1] based on the AndroZoo repository. Most of the lineages are spread over several years, but for each year, we consider only the latest apk version in that year for a given lineage. The statistics of apks per year from the lineage dataset in the literature is listed in Table 6.4.

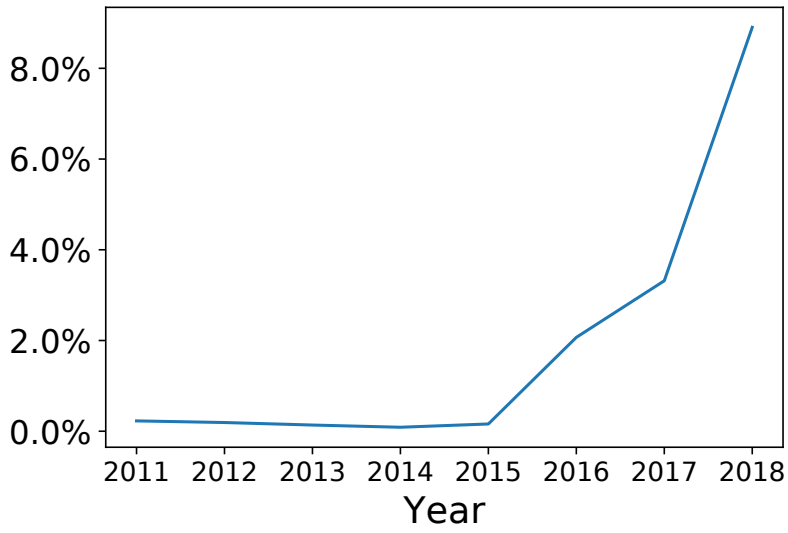


Figure 6.6: Percentage of APKs Contain DICI

We run DICIDER on this dataset and compute the percentage of apks containing DICIs for each year. The result is presented in Figure 6.6. A clear increasing trend can be observed after year 2015. By further investigating the DICI contributors, we notice that the main reason is still the GMS libraries. We find that before 2016, there are no contributors from GMS libraries. However, starting from 2016, the main contributors are all from GMS libraries, although the source packages shift from *internal* (mainly in 2016) to *dynamite* (after 2016). Unfortunately, these library not being part of the Android Open Source Project, we cannot find any information about the APIs publication and update time. We infer however that, starting from 2016, GMS libraries start to be more and more used Android applications.

By leveraging lineage, we also have the chance to investigate "update" patterns through checking the status of DICI usages in different versions of a same app. For this experiment, we consider all

Table 6.5: # APKs of Each Year

% of lineages in which a DICI has been introduced by an update	89.38%
% of lineages in which a DICI has been removed after an update	8.30%
% of lineages in which a DICI has been in all versions	2.07%
% of lineages with DICI added, then removed, and then added	0.26%

lineages with at least one DICI present in at least one apk of the lineage, and lineage with at least 4 apks. Table 6.5 presents the distribution of different update patterns. In a large majority of the cases, DICIs are not implemented in the initial version of the app. Instead it is added during an update.

Answer to RQ2: Over time, the use of the DICI mechanism has progressively become boomed among Android apps, mainly due to the availability of the GMS library. Nevertheless, it is noteworthy that DICIs can be implemented during app updates, which may cause concerns for update attacks, since it is well-known that users are less wary of apps during updates [202, 203].

6.4.3 RQ3: Purposes of Using DICIs

Table 6.6: Top DICI Contributor Packages

Rank	Package	Number
1	com.google.android.gms	7,462
2	com.lbe.doubleagent.client	128
3	com.jb.gosms.util	37
4	net.pierrox.lightning_launcher.b	30
5	com.dokdoapps.utility.GoogleServiceManager	20
5	com.handcent.common.v	20
7	kl.ime.oh.H	14
8	com.google.android.apps	13
9	org.xwalk.core.ReflectionHelper	12
10	cn.longmaster.common.pluginfx	10

Tables 6.6 and 6.7 enumerate respectively the top 10 DICI contributor packages where DICIs are invoked and the top 9 apps that are targeted by DICI reuse cases. These statistics are based on the lineage dataset described above. We note that, besides the fact that a GMS package is a top contributor for DICI usage, the GMS app is also the top app whose code is largely targeted for direct inter-app invocation. From the package and app names, we can notice the connection between some of them such as package *com.jb.gosms.util* tries to access code in app *com.jb.gosms.emoji*. Actually, by manually checking all the top 10 contributors, we confirm that they all try to access apps with similar names. For these cases, we can infer that they try to access the code from the app owned by same developers, to perform *app collusion* scenarios.

Table 6.7: Top DICI Invoked Apps

Rank	Package	Number
1	com.google.android.gms	7,469
2	com.jb.gosms.emoji	37
3	klye.hanwriting	14
4	org.xwalk.core	12
5	net.pierrox.lightning_locker_p	2
5	com.pansi.msg.plugin.custom_notify	2
6	com.pansi.msg.plugin.regins	1
6	com.shocktech.guaguahappy	1
6	com.pansi.msg.plugin.emoji	1

We take one more step to reveal the purpose of apps using DICIs by deeply exploring some apps.

Plugin implementation through DICI. The app with package name *kl.ime.oh*[10] is a multi-language keyboard app developed by *Honso* with more than 1 million installs on *Google Play*. It is found using 5 methods from another app with package name *klye.hanwriting*, which is a Chinese keyboard plugin app also can be found on *Google Play* with the same developer[11]. By searching apps from the same developer, more plugins for other languages can be found as well. We further notice that when the code loading failed, the app will return a string to ask to "Download Chinese plugin". Thus, we can infer that this app implemented a plugin functionality by using the DICI code reuse mechanism. We found several apps performing similar plugin behaviour. App named *com.jb.gosms* is found with loading 13 methods from an app with package name *com.jb.gosms.emoji* and a *BroadcastReceiver* is registered to check if the app is newly installed or uninstalled. App *com.pansi.msg* loads code from 3 different apps which are *com.pansi.msg.plugin.custom_ notify*, *com. pansi. msg. plugin. emoji* and *com.pansi.msg.plugin.regins*. And app *net. pierrox. lightning_ launcher* loaded code from *net.pierrox.lightning_ locker_ p*. We further notice that for the value of flags when creating the app context, some of them used value 1 instead of 3 which omits the flag of *CONTEXT_ IGNORE_ SECURITY*. This could be because all these apps are from the same developer (i.e., containing same signatures). Nevertheless, it is an interesting feature to be considered by static analyzers when assessing the security risks.

Keeping up with best practices through GMS. We find another use of the DICI code reuse mechanism which is to load up-to-date functionalities matching the best practices of Android programming available via *com.google.android.gms*. *GMS* stands for *Google Mobile Services* and *com.google.android.gms* is the *Google Play Services Packages*. The loader apps try to invoke the method *insertProvider* from class *com.google.android.gms.common.security.ProviderInstallerImpl*. According to the official documentation from *Google*[12], the purpose is to update the security provider to protect against SSL exploits. However, the relevant method mentioned in the document is *installIfNeeded* which is different from the one we found. By further checking the source code of method *installIfNeeded*, we find that the fundamental method is also *insertProvider*. Some developers may also notice this. Thus, instead of invoking the documented method and include all relevant libraries, they chose to directly invoke the fundamental method. All these loader apps are with package name of *com.monese.monese.live*, *nya.miku.wishmaster* and *com.levelup.touiteur* respectively[13].

6.5 Countermeasures

We now discuss possible countermeasures that could be leveraged by app developers to protect their app code from being reused in a stealthy way through DICI.

We found a straightforward countermeasure that, until now, we could not find a way to break. The idea is to check, with the code to protect, what is the "instance" of the application that is executing it. Indeed, to the best of our knowledge, there is no means to get or generate an instance of another app. Listing 6.5 presents the code that a developer could use to protect her app. First, to record the app's instance, a slight modification to the Application class is required (here `AntiTheftApp`). Specifically, we create a private instance field (line 2) and assign the current instance to this field (line 7). Then, we create an empty method called *verify* (lines 14 to 15). The purpose of this method is to check the availability of the stored application instance (i.e., *theInstance* at line 2), and an

[10] It can be downloaded from *AndroZoo* by using the SHA256: `04E37D1CE54C7E326A7714F56B35F922DF9EAF5AAD190FC5FD61716F84176D3E` and the app can be found on *Google Play* with link: `https://play.google.com/store/apps/details?id=kl.ime.oh`

[11] The page link is: `https://play.google.com/store/apps/details?id=klye.hanwriting`

[12] `https://developer.android.com/training/articles/security-gms-provider`

[13] The SHA256s are: `E953776572E4E84CB64D0ABF442211FC9A5EDDF0BDEF7E5DFC47C94756C714AB`, `8B16DBD2D4951BDAB16F2AA9AABCBC8BEE91264DAB78F2BCAFD3EE317B84E27C`, `06E280B615D5CF68E6BD3F89E27FF11FDBCFCE7D038AACCFE87C486F47159EB6`

empty method is already enough. Indeed, when `getInstance().verify()` is called in the original app, nothing happens. However, when this method is called in the plagiarist app, the app will crash because `theInstance` has not been initialized, and yet it cannot be instantiated or overwritten by attacking code in other apps. Finally, to protect from being reused (via DICI), app developer can simply write `AntiTheftApp.getInstance().verify()` at the beginning of each method, she wants to protect.

```
1 public class AntiTheftApp extends Application {
2     private static AntiTheftApp theInstance;
3     @Override
4     public void onCreate() {
5         super.onCreate(); theInstance = this;
6     }
7     public static AntiTheftApp getInstance() {
8         return theInstance;    }
9     public void verify() {  }
```

Listing 6.5: Implementation of Application Instance based Verification

Other Possibilities: So far, we have only attempted to protect DICIs at the app code level. Yet we believe many other features could also be leveraged to protect DICIs. Indeed, on the one hand, native libraries (or Javascript code for WebKit-based apps) could be leveraged, as they increase significantly the complexity of the code, making it non-trivial to be bypassed by attackers. On the other hand, some system features could be leveraged as well. For example, each Android app will be allocated with a private directory that cannot be accessed by other apps, and thereby could be leveraged to check the identity of the active app. Last but not least, from the Android framework point of view, there are various countermeasures could be applied. For example, Android OS could provide a mechanism similar to permission and component management for the inter-app code invocation functionality. It could limit the code access within apps from the same developers, or allow a declaration in the AndroidManifest file to specify which part of the code (i.e., classes) can be accessed publicly by other apps through DICIs. It also can be limited based on the privileges of Linux users.

6.6 Limitations

The fact that our prototype tool has revealed various DICI usages in real-world Android apps shows that our tool is useful to pinpoint them. Nonetheless, the implementation of our tool has come with various limitations. First of all, since the dummy main method construction approach is borrowed from FlowDroid, all the relevant limitations reported by FlowDroid also apply to our approach. For example, unsoundness can arise if certain callbacks in Android lifecycles are overlooked when building the dummy main method. Second, DICIDer directly adopts the constant propagation approach provided by Soot which unfortunately only supports intra-procedural analysis. As a result, although it is not our main focus of this work, certain reflectively accessed methods or fields could be missed by our approach. We keep this for future work. At the moment, DICIDer does not take into account native libraries and is oblivious to multiple-threading implementations, which may result in unsound results as well.

Not only the implementation of our prototype tool comes with limitations, the validity of our experimental results may also be threatened by the experimental setup we designed in this work. The major threat to the validity lies in the choice of selected Android apps. Although we rely on a random selection from AndroZoo to prepare the real-world apps for analysis, since the distributions of apps in different markets available in AndroZoo vary significantly, we cannot guarantee the representativeness of these apps. Furthermore, we leverage the app assembly time to build app lineages in this work. The app assembly time, as experimentally revealed by Li et al. [38], may not be accurate to represent the app release time. Hence, the app lineages we leverage to study the evolution of DICI usages may not be reliable as well. In this work, we try to mitigate this by following the same approach of our

fellow researchers to build the app lineages, which have been demonstrated to be useful to support app evolution studies.

6.7 Related Work

To the best of our knowledge, this work presents the first work disclosing the possibility of direct inter-app code invocation among Android apps and subsequently detecting DICI usages in Android apps. As a result, there is no related work specifically focusing on this problem. However, the research community has proposed various contributions in the domain of static analysis of Android apps. Moreover, some works focused on the problems of Inter-Component Communication (ICC) and Inter-App Communication (IAC), which are closely related to DICI. We now discuss the representative ones.

Static Analysis of Android Apps: Many state-of-the-art works have adopted static analysis, as one of their fundamental parts, to perform their research investigations. As presented in a recent survey done by Li et al. [204], there are over 100 papers, published mainly in the software engineering and security community, proposed to analyze Android apps statically. As revealed in their survey, static analysis has been largely conducted to uncover security and privacy issues such as privacy leaks detection [56, 55, 205], advertisement violations [206, 207, 208], and malware detection [209, 210, 211]. Also, the survey discloses that the well-known Soot framework is the most adopted basic support tool in the community to implement static analysis approaches. We remind that the Soot framework is also leveraged by DICIDer to detect the usage of DICIs. Static analysis has also been used by researchers to scan for (1) app defects including energy issues [212, 213], (2) fix runtime crashes [214, 215], (3) improve the realization of dynamic testing appraoches [216, 217, 218, 219].

Focus on Inter-Component Communication: Android apps differ from traditional Java apps in that there is no single entry point, e.g., the main method, in the apps. Apps are composed of multiple basic components. To pass on data among these components, Android has a special Inter-component Communication (ICC) mechanism. However, malware may also use this mechanism to achieve their malicious behaviors, e.g., steal users' private data. To this end, our community has proposed various approaches to mitigate the attacks related to Android ICCs. As an example, Epicc [181] is proposed to reduce the ICC problem to an instance of the Interprocedural Distributive Environment (IDE) problem, and finds ICC vulnerabilities with far fewer false positives. IccTA [56] is a static taint analyzer to detect privacy leaks among components in Android applications. It goes beyond existing ICC leaks detection tools like [68, 182, 183].

Focus on Inter-Application Communication: Android's Inter-application communication (IAC) mechanism allows for reuse of functionality across apps via *Intent*s. Contrary to the technique described in this study, IAC is intended for functionality sharing. However, this mechanism also raises concerns for vulnerabilities crossing Android apps. Thereby, the research community has also proposed various approaches to mitigate possible vulnerabilities brought by IAC. For example, Li et al. [189] have proposed a tool called ApkCombiner aiming to combine multiple apps together to a single app, so as to reduce an IAC problem to an ICC problem. As a result, this tool allows all the aforementioned ICC-aware approaches to resolving IAC problems without modifications. PermissionFlow [88] can reliably and accurately detect vulnerable information flows among Android applications. IntentDroid is a cloud-based testing algorithm for Android apps for automated discovery of Android IAC vulnerabilities. ComDroid [119] is another tool to detect ICC related malicious behaviors in Android apps, e.g., sniffing message contents and injecting forged messages.

Unfortunately, since DICI leverages a totally different channel to implement inter-app communication, all the aforementioned existing works cannot be directly applied to detect DICI usages in Android apps. Our prototype tool DICIDer fills this gap by providing a means to statically pinpoint DICI usages in Android apps, which could be considered as a complement to the state-of-the-art.

6.8 Summary

In this study, we disclose to the software engineering community a novel mechanism allowing direct inter-app code invocation (DICI) among installed Android apps on mobile devices. Through concrete motivating examples, we demonstrate that DICI can be leveraged to successfully perform malicious attacks and plagiarize the core function of the competitor's apps. We then introduce to the community a static analyzer called DICIDer to automatically locate the usage of DICIs in Android apps. Experiments on a large set of Android apps reveal that DICIDer is indeed capable of detecting DICIs in Android apps, and the usage of DICIs tends to increase over time, which may cause concerns for update attacks since users might be less wary of apps during updates.

7 Conclusions and Future Work

In this chapter, we summarize the main contributions of our dissertation and present potential future research directions.

Contents

7.1 Conclusions

This dissertation presented the re-construction of Android app lineages and several Android app security-related studies based on mining the app lineages. These studies follow a general-to-specific path. We chose such a topic because 1) re-construction of large-scale Android app lineages is challenge work. So far, there is no such dataset available, which leads to few works of evolution studies in Android app security. 2) Attentions are still required on Android app security since new issues have never been stopping showing up while old issues remain unfixed. 3) Evolutionary studies in Android app security are potentially beneficial in many ways, such as providing a fundamental understanding of specific issues. More specifically, we scheduled this dissertation in three parts: 1) Re-construction of Android app lineages. 2) Evolution study on existing Android app vulnerabilities to learn trends as well as mine rules. 3) Unveiled a less known code access mechanism that could cause new vulnerabilities and checked its evolution. We now detail them.

In the first part, by leveraging our *AndroZoo* dataset, which is so far the largest Android app repository and HPC clusters, we carefully proceeded to re-construct the dataset of Android app lineages at an unprecedented scale. This dataset not only served the following studies of this dissertation, but also is a valuable artefact for various research in our community. We also implemented a primary study on the evolution of Android app complexity with the newly re-constructed app lineages.

For the second part, we implemented evolution studies on known vulnerabilities. We first investigated specifically 10 vulnerability types associated with 4 different categories by leveraging three state-of-the-art, open-source and actively used detection tools to have a general concept of how Android app vulnerabilities evolve and what we can learn from the evolution. Among all those findings, we even noticed that some vulnerabilities might foreshadow malware. Afterward, we focused on particular vulnerabilities caused by the misuse of cryptography APIs in Android apps. Instead of merely investigating its evolution, we tried to mining the usage rules of such APIs from the app evolution course. This study is based on an intuitively reasonable hypothesis that API usage updates generally transform incorrect usage into correct usage. Although we produced a negative result in this study, we profoundly elaborated the reason behind.

As the final part, we unveiled a less known mechanism used to invoke code of other Android apps. We exhibited the details of the mechanism and proved its practicality by showcasing two proof-of-concept applications developed by us. A tool named `DICIDer` was also developed to detect the use of this mechanism in real-world applications. We further implemented an empirical study on how this mechanism was used, and proposed countermeasures to protect apps from undesired inter-app code access. The evolution of the mechanism uses was also investigated and a progressive booming was spotted.

7.2 Future Work

We now summarize potential future directions that are in line with this dissertation.

1. **Re-constructing App Lineages in Real Time with Metadata.** The Android framework has been rapidly evolving, as well as all Android apps. So for our *AndroZoo* dataset has also been continuously growing by non-stopping crawling from different markets. However, the re-construction of the Android app lineages has not been implemented incrementally yet. Thus, to follow up with the fast evolution, a real-time updating of the app lineages is also necessary. Moreover, the metadata of an application (e.g., installs, update time, update log) provides extra context information about the current version of the application. It can expand the horizons of Android apps evolution studies. Nevertheless, the collection of these metadata has not been considered yet. Hence, such a collection should be implemented as soon as possible.

2. **Mining Updates with Knowledge from Update Logs.** From the negative result on mining cryptography API usage rules, we realized that inferring the purpose of app updates is challenging. However, update logs (i.e., the logs provide detailed information about what have been fixed or improved in the new version. For apps published on GooglePlay, such information are normally provided in the description section of the apps' page.) are normally provided by the developer to specify the improvements in the latest version of the apps. Thus, mining updates with relevant knowledge from the corresponding logs instead of hypotheses should produce more promising results.

3. **The Evolution in Android Native Code.** Android apps can be developed in Java as well as native code (i.e., C/C++). Because of the platform-independence of Java, Android apps are mainly written in Java, while native code is only used when apps need to access platform-specific features or improve performance. Therefore, security studies have heavily focused on the Java part of Android apps while left the native code far more behind. Nonetheless, such a situation causes much less native vulnerabilities spotted and makes malicious behaviors tend to hide in the native code. Hence, more security studies on the native code and its evolution are needed.

List of Papers

Papers included in this dissertation:

- J. Gao, L. Li, T. F. Bissyandé, and J. Klein. On the evolution of mobile app complexity. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 200–209, 2019
- J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, pages 1–19, 2019
- J. Gao, P. Kong, L. Li, T. F. Bissyandé, and J. Klein. Negative results on mining crypto-api usage rules in android apps. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 388–398, 2019
- Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Borrowing your enemy's arrows: the case of code reuse in android via direct inter-app code invocation. In *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020

Papers not included in this dissertation:

- Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Knowledgezooclient: Constructing knowledge graph for android. In *The 3rd International Workshop on Advances in Mobile App Analysis (A-Mobile 2020), co-located with ASE 2020*, 2020
- Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Cda: Characterising deprecated android apis. *Empirical Software Engineering (EMSE)*, 2020
- Pingfan Kong, Li Li, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. Mining android crash fixes in the absence of issue-and change-tracking systems. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 78–89, 2019
- Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66, 2018
- Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018

Bibliography

[1] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, pages 1–19, 2019.

[2] Alexandre Bartel, Jacques 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. *IEEE Transactions on Software Engineering*, 40(6):617–632, 2014.

[3] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, volume 30, page 88, 2011.

[4] Timothy Vidas, Daniel Votipka, and Nicolas Christin. All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, page 10, USA, 2011. USENIX Association.

[5] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, volume 14, page 19, 2012.

[6] 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*, pages 229–240, 2012.

[7] Yuan Tian, Bin Liu, Weisi Dai, Blase Ur, Patrick Tague, and Lorrie Faith Cranor. Supporting privacy-conscious app update decisions with user reviews. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 51–61. ACM, 2015.

[8] Rahul Potharaju, Mizanur Rahman, and Bogdan Carbunar. A longitudinal study of google play. *IEEE Transactions on Computational Social Systems*, 4(3):135–149, 2017.

[9] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200. ACM, 2017.

[10] Vincent F Taylor and Ivan Martinovic. To update or not to update: Insights from a two-year study of android app evolution. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 45–57. ACM, 2017.

[11] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[12] 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.

[13] Paolo Calciati and Alessandra Gorla. How do apps evolve in their permission requests?: a preliminary study. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 37–41. IEEE Press, 2017.

[14] 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 Workshop on Mining Software Repositories*, pages 468–471. ACM, 2016.

[15] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In *MSR '16 Proc. of the 13th Intl. Conference on Mining Software Repositories*, pages 468–471, Austin, Texas, May 2016.

[16] 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 and Security*, 12(6):1269–1284, 2017.

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

[18] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. The soot-based toolchain for analyzing android apps. In *Proc. of the 4th Intl. Conference on Mobile Software Engineering and Systems*, pages 13–24. IEEE Press, 2017.

[19] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proc. of the IEEE*, 68(9):1060–1076, 1980.

[20] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32. IEEE, 1997.

[21] Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesus M Gonzalez-Barahona. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys (CSUR)*, 46(2):28, 2013.

[22] Iulian Neamtiu, Guowu Xie, and Jianbo Chen. Towards a better understanding of software evolution: an empirical study on open-source software. *Journal of Software: Evolution and Process*, 25(3):193–218, 2013.

[23] Stephen W Thomas, Bram Adams, Ahmed E Hassan, and Dorothea Blostein. Studying software evolution using topic models. *Science of Computer Programming*, 80:457–479, 2014.

[24] Pooyan Behnamghader, Reem Alfayez, Kamonphop Srisopha, and Barry Boehm. Towards better understanding of software quality evolution through commit-impact analysis. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pages 251–262. IEEE, 2017.

[25] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

[26] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the software quality of android applications along their evolution (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 236–247. IEEE, 2015.

[27] J. Gao, L. Li, T. F. Bissyandé, and J. Klein. On the evolution of mobile app complexity. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 200–209, 2019.

[28] E Yourdon. The rise and fall of the american programmer. *Yourdon Press Computing Series*, 1992.

[29] Frederick P Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India, 1995.

[30] Vard Antinyan, Miroslaw Staron, and Anna Sandberg. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering*, 22(6):3057–3087, 2017.

[31] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[32] GREGOR JOŠT, JERNEJ HUBER, and MARJAN HERIČKO. Using object oriented software metrics for mobile application development. In *Second Workshop on Software Quality Analysis, Monitoring, Improvement and Applications SQAMIA 2013*, 2013.

[33] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[34] Linda H. Rosenberg and Lawrence E. Hyatt. Software quality metrics for object-oriented environments. Technical report, NASA, 1995.

[35] Francesco Mercaldo, Andrea Di Sorbo, Corrado Aaron Visaggio, Aniello Cimitile, and Fabio Martinelli. An exploratory study on the evolution of android malware quality. *Journal of Software: Evolution and Process*, 0(0):e1978, 2018. e1978 smr.1978.

[36] Google. App id. `https://developer.android.com/studio/build/application-id.html`. Accessed: 2017-07-16.

[37] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[38] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Moonlightbox: Mining android api histories for uncovering release-time inconsistencies. In *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*, 2018.

[39] David Lorge Parnas. Software aging. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 279–287. IEEE, 1994.

[40] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Principles of Software Evolution, Eighth International Workshop on*, pages 13–22. IEEE, 2005.

[41] Li Li. *Boosting Static Security Analysis of Android Apps through Code Instrumentation*. PhD thesis, University of Luxembourg, Luxembourg, 2016.

[42] Mourad Badri, Linda Badri, and Fadel Touré. Empirical analysis of object-oriented design metrics: Towards a new metric using control flow paths and probabilities. *Journal of Object Technology*, 8(6):123–142, 2009.

[43] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[44] Norman E Fenton and Martin Neil. Software metrics: roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 357–370. ACM, 2000.

[45] Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E Hassan. What are the characteristics of high-rated apps? a case study on free android applications. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 301–310. IEEE, 2015.

[46] Mykola Protsenko and Tilo Müller. Android malware detection based on software complexity metrics. In *International Conference on Trust, Privacy and Security in Digital Business*, pages 24–35. Springer, 2014.

[47] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, pages 1–19, 2019.

[48] Craig Smith. 75 amazing android statistics and facts (august 2017). `http://expandedramblings.com/index.php/android-statistics/`. Accessed: 2017-08-20.

[49] Yvo Desmedt. Man-in-the-middle attack. In *Encyclopedia of cryptography and security*, pages 759–759. Springer, 2011.

[50] 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*, pages 239–252, 2011.

[51] Jeremy Clark and Paul C van Oorschot. Sok: Ssl and https: Revisiting past challenges and evaluating certificate trust model enhancements. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 511–525. IEEE, 2013.

[52] 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 *Proc. of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[53] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *In Proceedings of the 20th Annual Symposium on Network and Distributed System Security, NDSS '13*, 2013.

[54] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *In Proc. of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. Citeseer, 2014.

[55] 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. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[56] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291, May 2015.

[57] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proc. of the 37th Intl. Conference on Software Engineering-Volume 1*, pages 77–88. IEEE Press, 2015.

[58] C. Qian, X. Luo, Y. Le, and G. Gu. Vulhunter: Toward discovering vulnerabilities in android applications. *IEEE Micro*, 35(1):44–53, Jan 2015.

[59] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM, 2016.

[60] David Kantola, Erika Chin, Warren He, and David Wagner. Reducing attack surfaces for intra-application communication in android. In *Proc. of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 69–80. ACM, 2012.

[61] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, et al. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Computing Surveys (CSUR)*, 49(2):38, 2016.

[62] Wei You, Bin Liang, Wenchang Shi, Shuyang Zhu, Peng Wang, Sikefu Xie, and Xiangyu Zhang. Reference hijacking: Patching, protecting and analyzing on unmodified and non-rooted android devices. In *Proc. of the 38th Intl. Conference on Software Engineering*, pages 959–970. ACM, 2016.

[63] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. Adaptive unpacking of android apps. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 358–369, Piscataway, NJ, USA, 2017. IEEE Press.

[64] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: Toward extracting hidden code from packed android applications. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, pages 293–311, Cham, 2015. Springer International Publishing.

[65] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 343–355, New York, NY, USA, 2016. ACM.

[66] Felivel Camilo, Andrew Meneely, and Meiyappan Nagappan. Do bugs foreshadow vulnerabilities? a study of the chromium project. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 269–279. IEEE, 2015.

[67] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proc. of the 37th Intl. Conference on Software Engineering-Volume 1*, pages 426–436. IEEE Press, 2015.

[68] Jonathan Burket, Lori Flynn, William Klieber, Jonathan Lim, Wei Shen, and William Snavely. Making didfail succeed: Enhancing the cert static taint analyzer for android app sets. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA PITTSBURGH United States, 2015.

[69] FlowDroid. Flowdroid website. `https://blogs.uni-paderborn.de/sse/tools/flowdroid/`.

[70] Yu-Cheng Lin. Androbugs framework: An android application security vulnerability scanner. In *Blackhat Europe 2015*, 2015.

[71] AndroBugs. Open source repository. `https://github.com/AndroBugs/AndroBugs_Framework`.

[72] AndroBugs. Hall of fame. `https://www.androbugs.com/#hof`.

[73] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. A view to a kill: Webview exploitation. In *LEET*, 2013.

[74] Y Cifuentes, L Beltrán, and L Ramírez. Analysis of security vulnerabilities for mobile health applications. In *2015 Seventh Intl. Conference on Mobile Computing and Networking (ICMCN 2015)*, 2015.

[75] Mohamed Sabt and Jacques Traoré. Breaking into the keystore: A practical forgery attack against android keystore. In *European Symposium on Research in Computer Security*, pages 531–548. Springer, 2016.

[76] Daniel R Thomas, Alastair R Beresford, Thomas Coudray, Tom Sutcliffe, and Adrian Taylor. The lifetime of android api vulnerabilities: case study on the javascript-to-java interface. In *Cambridge Intl. Workshop on Security Protocols*, pages 126–138. Springer, 2015.

[77] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on webview in the android system. In *Annual Computer Security Applications Conference*, pages 343–352. ACM, 2011.

[78] Roee Hay. Android collapses into fragments. In *IBM Security Systems*, 2013.

[79] Adam Cozzette. Intent spoofing on android. `http://blog.palominolabs.com/2013/05/13/android-security/index.html`. Accessed: 2017-08-20.

[80] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332–345. IEEE, 2010.

[81] Mike Bland. Finding more than one worm in the apple. *Communications of the ACM*, 57(7):58–64, 2014.

[82] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proc. of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[83] Christopher Meyer and Jörg Schwenk. Sok: Lessons learned from ssl/tls attacks. In *Intl. Workshop on Information Security Applications*, pages 189–209, 2013.

[84] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 debian openssl vulnerability. In *Proc. of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 15–27. ACM, 2009.

[85] 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*, ASIA CCS '15, pages 591–596, New York, NY, USA, 2015. ACM.

[86] OWASP. Mobile top 10 2014-m2: Insecure data storage. `https://www.owasp.org/index.php/Mobile_Top_10_2014-M2`. Accessed: 2017-08-20.

[87] Adam Cozzette, Kathryn Lingel, Steve Matsumoto, Oliver Ortlieb, Jandria Alexander, Joseph Betser, Luke Florer, Geoff Kuenning, John Nilles, and Peter Reiher. Improving the security of android inter-component communication. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE Intl. Symposium on*, pages 808–811. IEEE, 2013.

[88] 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*, 57(6):10–1, 2013.

[89] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proc. of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[90] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. *Trust*, 12:291–307, 2012.

[91] Hoang Tuan Ly, Tan Cam Nguyen, and Van-Hau Pham. edsdroid: A hybrid approach for information leak detection in android. In *Intl. Conference on Information Science and Applications*, pages 290–297. Springer, 2017.

[92] Sergio Yovine and Gonzalo Winniczuk. Checkdroid: a tool for automated detection of bad practices in android applications using taint analysis. In *Proc. of the 4th Intl. Conference on Mobile Software Engineering and Systems*, pages 175–176. IEEE Press, 2017.

[93] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):76, 2017.

[94] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022, 2015.

[95] Darell JJ Tan, Tong-Wei Chua, Vrizlynn LL Thing, et al. Securing android: a survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR)*, 47(4):58, 2015.

[96] 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*, 43(9):817–847, 2017.

[97] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6):492–530, 2017.

[98] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proc. of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[99] Joshua Drake. Stagefright: Scary code in the heart of android. *BlackHat USA*, 2015.

[100] MATT BURGESS. Millions of android devices vulnerable to new stagefright exploit. `http://www.wired.co.uk/article/stagefright-android-real-world-hack`. Accessed: 2017-08-20.

[101] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. Malton: Towards on-device non-invasive mobile malware analysis for art. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 289–306, Berkeley, CA, USA, 2017. USENIX Association.

[102] Jeff Williams and Arshan Dabirsiaghi. The unfortunate reality of insecure libraries. *Asp. Secur. Inc*, pages 1–26, 2012.

[103] Takuya Watanabe, Mitsuaki Akiyama, Fumihiro Kanei, Eitaro Shioji, Yuta Takata, Bo Sun, Yuta Ishi, Toshiki Shibahara, Takeshi Yagi, and Tatsuya Mori. Understanding the origins of mobile app vulnerabilities: a large-scale measurement study of free and paid apps. In *Proc. of the 14th Intl. Conference on Mining Software Repositories*, pages 14–24. IEEE Press, 2017.

[104] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an academic hpc cluster: The ul experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, pages 959–967, Bologna, Italy, July 2014. IEEE.

[105] Haipeng Cai and Barbara G Ryder. Understanding android application programming and security: A dynamic study. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 364–375. IEEE, 2017.

[106] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 176–186, New York, NY, USA, 2018. ACM.

[107] Fahad Ibrar, Hamza Saleem, Sam Castle, and Muhammad Zubair Malik. A study of static analysis tools to detect vulnerabilities of branchless banking applications in developing countries. In *Proceedings of the Ninth International Conference on Information and Communication Technologies and Development*, ICTD '17, pages 30:1–30:5, New York, NY, USA, 2017. ACM.

[108] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Security Symposium*, 2010.

[109] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.

[110] Robert Lemos MIT Technology Review. Your apps could be leaking private info. `https://www.technologyreview.com/s/420062/your-apps-could-be-leaking-private-info/`. Accessed: 2017-08-21.

[111] William S Cleveland and Susan J Devlin. Locally weighted regression: an approach to regression analysis by local fitting. *Journal of the American statistical association*, 83(403):596–610, 1988.

[112] K. Hamandi, A. Chehab, I. H. Elhajj, and A. Kayssi. Android sms malware: Vulnerability and mitigation. In *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pages 1004–1009, March 2013.

[113] Médéric Hurier, Kevin Allix, Tegawendé Bissyandé, Jacques Klein, and Yves Le Traon. On the lack of consensus in anti-virus decisions: Metrics and insights on building ground truths of android malware. In *13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, pages 142–162. Springer, 2016.

[114] A. Kantchelian, M.C. Tschantz, S. Afroz, B. Miller, V. Shankar, R. Bachwani, A.D. Joseph, and J.D Tygar. Better malware ground truth: Techniques for weighting anti-virus vendor labels. In *AISec '15*, pages 45–56. ACM, 2015.

[115] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. Euphony: harmonious unification of cacophonous anti-virus vendor labels for android malware. In *Proc. of the 14th Intl. Conference on Mining Software Repositories*, pages 425–435. IEEE Press, 2017.

[116] Adam Bauser and Christoph Hebeisen. Igexin advertising network put user privacy at risk. `https://blog.lookout.com/igexin-malicious-sdk`, August 2017.

[117] Daoyuan Wu and Rocky KC Chang. Analyzing android browser apps for file://vulnerabilities. In *Intl. Conference on Information Security*, pages 345–363. Springer, 2014.

[118] Erika Chin and David Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *Intl. Workshop on Information Security Applications*, pages 138–159. Springer, 2013.

[119] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128, 2015.

[120] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proc. of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.

[121] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, pages 461–468, 2013.

[122] Julian Schütte, Rafael Fedler, and Dennis Titze. Condroid: Targeted dynamic analysis of android applications. In *Advanced Information Networking and Applications (AINA), 2015 IEEE 29th Intl. Conference on*, pages 571–578. IEEE, 2015.

[123] Ivan Victor Krsul. *Software vulnerability analysis*. Purdue University West Lafayette, IN, 1998.

[124] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proc.*, volume 2, pages 119–129. IEEE, 2000.

[125] Christian DOrazio and Kim-Kwang Raymond Choo. A generic process to identify vulnerabilities and design weaknesses in ios healthcare apps. In *System Sciences (HICSS), 2015 48th Hawaii Intl. Conference on*, pages 5175–5184. IEEE, 2015.

[126] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[127] Benjamin Schwarz, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. Model checking an entire linux distribution for security violations. In *Computer Security Applications Conference, 21st Annual*, pages 10–pp. IEEE, 2005.

[128] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 799–813, New York, NY, USA, 2017. ACM.

[129] Saurabh Jain, Deepak Singh Tomar, and Divya Rishi Sahu. Detection of javascript vulnerability at client agent. *Intl. Journal of Scientific & Technology Research*, 1(7):36–41, 2012.

[130] Adam Kieyzun, Philip J Guo, Karthick Jayaraman, and Michael D Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Software Engineering, 2009. ICSE 2009. IEEE 31st Intl. Conference on*, pages 199–209. IEEE, 2009.

[131] Lwin Khin Shar and Hee Beng Kuan Tan. Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities. In *Proc. of the 34th Intl. Conference on Software Engineering*, pages 1293–1296. IEEE Press, 2012.

[132] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Patching vulnerabilities with sanitization synthesis. In *Proc. of the 33rd Intl. Conference on software engineering*, pages 251–260. ACM, 2011.

[133] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS*, 2014.

[134] H Bagheri, E Kang, S Malek, and D Jackson. A formal approach for detection of security flaws in the android permission system. *Formal Aspects Comput*, 2016.

[135] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1236–1247. ACM, 2015.

[136] Kai Wang, Yuqing Zhang, and Peng Liu. Call me back!: Attacks on system server and system apps in android through synchronous callback. In *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 92–103. ACM, 2016.

[137] Chen Cao, Neng Gao, Peng Liu, and Ji Xiang. Towards analyzing the input validation vulnerabilities associated with android system services. In *Proc. of the 31st Annual Computer Security Applications Conference*, pages 361–370. ACM, 2015.

[138] Matthieu Jimenez, Mike Papadakis, Tegawendé F Bissyandé, and Jacques Klein. Profiling android vulnerabilities. In *Software Quality, Reliability and Security (QRS), 2016 IEEE Intl. Conference on*, pages 222–229. IEEE, 2016.

[139] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An empirical study on android-related vulnerabilities. In *Proc. of the 14th Intl. Conference on Mining Software Repositories*, pages 2–13. IEEE Press, 2017.

[140] Daniel R Thomas, Alastair R Beresford, and Andrew Rice. Security metrics for the android ecosystem. In *Proc. of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 87–98, 2015.

[141] Joydeep Mitra and Venkatesh-Prasad Ranganath. Ghera: A repository of android app vulnerability benchmarks. *arXiv preprint arXiv:1708.02380*, 2017.

[142] J. Gao, P. Kong, L. Li, T. F. Bissyandé, and J. Klein. Negative results on mining crypto-api usage rules in android apps. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 388–398, 2019.

[143] 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*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM.

[144] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 75–80, Aug 2014.

[145] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. Evaluation of cryptography usage in android applications. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (Formerly BIONETICS)*, BICT'15, pages 83–90, ICST, Brussels, Belgium, Belgium, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[146] 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*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

[147] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.

[148] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. In *10: 1-10: 27*, July 2018.

[149] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Inferring crypto api rules from code changes. *SIGPLAN Not.*, 53(4):450–464, June 2018.

[150] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 711–722, New York, NY, USA, 2016. ACM.

[151] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 57–72, New York, NY, USA, 2001. Association for Computing Machinery.

[152] Zhenmin Li and Yuanyuan Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 306–315, New York, NY, USA, 2005. Association for Computing Machinery.

[153] Julia L Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. Wysiwib: A declarative approach to finding api protocols and bugs in linux code. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 43–52. IEEE, 2009.

[154] Tegawendé F Bissyandé, Laurent Réveillère, Julia L Lawall, and Gilles Muller. Diagnosys: automatic generation of a debugging interface to the linux kernel. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 60–69. ACM, 2012.

[155] Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Impact of tool support in patch construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 237–248. ACM, 2017.

[156] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 935–946, New York, NY, USA, 2016. ACM.

[157] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 709–712, Piscataway, NJ, USA, 2015. IEEE Press.

[158] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.

[159] Oracle. Java cryptography architecture (jca). `https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[160] Apache. Apache commons crypto. `https://commons.apache.org/proper/commons-crypto/index.html`.

[161] CogniCrypt Developers. $cognicrypt_{SAST}$. `https://github.com/CROSSINGTUD/CryptoAnalysis`.

[162] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. Cognicrypt: Supporting developers in using cryptography. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 931–936, Piscataway, NJ, USA, 2017. IEEE Press.

[163] CogniCrypt Developers. Cognicrypt for android. `https://github.com/CROSSINGTUD/CryptoAnalysis-Android`.

[164] Anonymised for blind review. Anonymised for blind review. In *Anonymised for blind review*, pages xx–yy. xxx, xxxx.

[165] 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*, pages 73–84. ACM, 2013.

[166] 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*, pages 75–80. IEEE, 2014.

[167] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. Evaluation of cryptography usage in android applications. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*, pages 83–90. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

[168] 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.

[169] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 111–122. IEEE, 2015.

[170] 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.

[171] Veelasha Moonsamy, Jia Rong, and Shaowu Liu. Mining permission patterns for contrasting clean and malicious android applications. *Future Generation Computer Systems*, 36:122–132, 2014.

[172] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Borrowing your enemy's arrows: the case of code reuse in android via direct inter-app code invocation. In *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

[173] William B Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE transactions on Software Engineering*, 31(7):529–536, 2005.

[174] Israel J Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. Understanding reuse in the android market. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 113–122. IEEE, 2012.

[175] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering*, 2019.

[176] 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)*, 2017.

[177] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communications*, 73(4):1421–1437, 2013.

[178] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. Appink: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 1–12, 2013.

[179] 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.

[180] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, Washington, D.C., 2013. USENIX.

[181] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 543–558, 2013.

[182] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android. Technical report, University of Maryland, 2009.

[183] 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*, pages 623–634, 2013.

[184] 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*, pages 1–10, 2014.

[185] Amiangshu Bosu, Fang Liu, Danfeng Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85, 2017.

[186] 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*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[187] Karim O Elish, Haipeng Cai, Daniel Barton, Danfeng Yao, and Barbara G Ryder. Identifying mobile inter-app communication risks. *IEEE Transactions on Mobile Computing*, 19(1):90–102, 2018.

[188] Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, Mohamed Mosbah, and Mauro Conti. Android inter-app communication threats and detection techniques. *Computers & Security*, 70:392–421, 2017.

[189] 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.

[190] TikTok. Web site: https://www.tiktok.com.

[191] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12(110):1, 2012.

[192] 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*, pages 1457–1462, 2012.

[193] Siddharth Prakash Rao, Silke Holtmanns, Ian Oliver, and Tuomas Aura. Unblocking stolen mobile devices using ss7-map vulnerabilities: Exploiting the relationship between imei and imsi for eir access. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 1171–1176. IEEE, 2015.

[194] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, volume 14, page 1125. Citeseer, 2014.

[195] 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 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 318–329, New York, NY, USA, 2016. Association for Computing Machinery.

[196] P. Calciati and A. Gorla. How do apps evolve in their permission requests? a preliminary study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 37–41, May 2017.

[197] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 31–40, New York, NY, USA, 2012. Association for Computing Machinery.

[198] 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)*, volume 15, page 35, 2011.

[199] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F. Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community, 2017.

[200] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In *The 2018 Internet Measurement Conference (IMC 2018)*, 2018.

[201] VirusTotal. Web site: https://www.virustotal.com/.

[202] Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Download malware? no, thanks: how formal methods can block update attacks. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering*, pages 22–28, 2016.

[203] David Barrera, Jeremy Clark, Daniel McCarney, and Paul C Van Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of android. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 81–92, 2012.

[204] 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.

[205] 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.

[206] Tianming Liu, Haoyu Wang, Li Li, Xiapu Luo, Feng Dong, Yao Guo, Liu Wang, Tegawendé F Bissyandé, and Jacques Klein. Maddroid: Characterising and detecting devious ad content for android apps. In *The Web Conference 2020 (WWW 2020)*, 2020.

[207] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. Frauddroid: Automated ad fraud detection for android apps. In *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*, 2018.

[208] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Guoai Xu, and Shaodong Zhang. How do mobile apps violate the behavioral policy of advertisement libraries? In *The 19th Workshop on Mobile Computing Systems and Applications (HotMobile 2018)*, 2018.

[209] 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.

[210] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294, 2012.

[211] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69. IEEE, 2012.

[212] Haowei Wu, Shengqian Yang, and Atanas Rountev. Static detection of energy defect patterns in android applications. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 185–195, 2016.

[213] Luis Cruz, Rui Abreu, John Grundy, Li Li, and Xin Xia. Do energy-oriented changes hinder maintainability? In *The 35th IEEE International Conference on Software Maintenance and Evolution (ICSME 2019)*, 2019.

[214] Pingfan Kong, Li Li, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. Mining android crash fixes in the absence of issue-and change-tracking systems. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 78–89, 2019.

[215] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. Repairing crashes in android apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 187–198. IEEE, 2018.

[216] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66, 2018.

[217] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[218] Hailong Zhang, Haowei Wu, and Atanas Rountev. Automated test generation for detection of leaks in android applications. In *Proceedings of the 11th International Workshop on Automation of Software Test*, pages 64–70, 2016.

[219] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256, 2017.

[220] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Knowledge-zooclient: Constructing knowledge graph for android. In *The 3rd International Workshop on Advances in Mobile App Analysis (A-Mobile 2020), co-located with ASE 2020*, 2020.

[221] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Cda: Characterising deprecated android apis. *Empirical Software Engineering (EMSE)*, 2020.