Understanding the Evolution of Android App Vulnerabilities

Jun Gao, Li Li, Pingfan Kong, Tegawendé F. Bissyandé, and Jacques Klein

***Abstract*—The Android ecosystem today 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. Security of communication and privacy of data are thus of utmost importance in application development. Yet, regularly, there are reports of successful attacks targeting Android users. While some of those attacks exploit vulnerabilities in the Android operating system, others directly concern application-level code written by a large pool of developers with varying experience. Recently, a number of studies have investigated this phenomenon, focusing, however, only on a specific vulnerability type appearing in apps, and based on only a snapshot of the situation at a given time. Thus, the community is still lacking comprehensive studies exploring how vulnerabilities have evolved over time, and how they evolve in a single app across developer updates. Our work fills this gap by leveraging a data stream of 5 million app packages to reconstruct versioned lineages of Android apps, and finally, obtained 28 564 app lineages (i.e., successive releases of the same Android apps) with more than ten app versions each, corresponding to a total of 465 037 apks. Based on these app lineages, we apply state-of-the-art vulnerability-finding tools and investigate systematically 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 as reported by the tools, but we further discuss the potential false positives. Our findings and study artifacts constitute a tangible knowledge to the community. It could be leveraged by developers to focus verification tasks, and by researchers to drive vulnerability discovery and repair research efforts.**

***Index Terms*—Android, vulnerability, evolution.**

I. Introduction

MOBILE software has been overtaking the 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 [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]–[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 [7].

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 [8]–[12] 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 [13]–[16] and by proposing improvements to current security models [17]–[20]. Although advanced techniques have been employed by malicious developers such as packer and obfuscation, countermeasures such as [21]–[23] 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 [24]–[26] 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.

Manuscript received April 14, 2019; revised September 13, 2019; accepted November 15, 2019. Date of publication December 19, 2019; date of current version March 2, 2021. This work was supported in part by the Luxembourg National Research Fund (FNR) under Grant PRIDE15/10621687/SPsquared and project CHARACTERIZE C17/IS/1169386, in part by the Australian Research Council (ARC), under projects DE200100016 and DP200100020, and in part by the Oceania Cyber Security Centre (OCSC), under the 2019 ICFP scheme. Associate Editor: Dr. P. Laplante. *(Corresponding author: Li Li.)*

J. Gao, P. Kong, T. F. Bissyandé, and J. Klein are with the University of Luxembourg, 4365 Esch-sur-Alzette, Luxembourg (e-mail: jun.gao@uni.lu; pingfan.kong@uni.lu; tegawende.bissyande@uni.lu; jacques.klein@uni.lu).

L. Li is with the Monash University, Clayton VIC 3800, Australia (e-mail: li.li@monash.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2019.2956690

[1]For example, padlock and HTTPS in url input field.

This article: In this article, 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.

1) We carefully proceed to reconstruct the version lineages of Android apps at an unprecedented scale, based on a dataset of over 5 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 ten versions of a given app. Although this contribution serves the purpose of our study, it is a *valuable artifact* for diverse research fields in our community, notably software quality and its subfields of testing, repair, and evolution studies.
2) 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 ten vulnerability types associated to four different categories related to common security features (e.g., secure sockets layer (SSL)), its sandbox mechanism (e.g., Permission issues), code injection (e.g., WebView RCE vulnerability) as well as its interapp 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.
3) Finally, we perform *several empirical analyses* to highlight statistical trends on the temporal evolutions of vulnerabilities in Android apps; capture the common locations (e.g., developer versus library code) of vulnerable code in apps; comprehend the vehicle (e.g., code change, new files, etc.) through which vulnerabilities are introduced in mobile apps; and investigate via correlation analysis whether vulnerabilities foreshadow malware in the Android ecosystem.

And the main findings are as follows:

1) most vulnerabilities will survive at least three updates;
2) some third-party libraries are major contributors to most vulnerabilities detected by static tools;
3) vulnerability reintroduction occurs for all kinds of vulnerabilities with *Encryption*-related vulnerabilities being the mostly reintroduced type in this study;
4) 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 first 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 rest of this article is organized as follows. Section II describes the experimental setup, including the construction of app lineages, an introduction of the vulnerability finding tools, and the research questions. Section III unfolds the empirical analyses. Section IV-B enumerates threats to validity, while Section IV-A discusses some promising future works. Section V discusses related work, and finally, Section VI concludes this article.

II. Study Data and Design

In this section, we first define and clarify some terms used in this article. Second, we provide some background information on the scale of the app dataset that we adopted, as well as on the security vulnerability detection tools that we leveraged in Section II-B. Third, we describe the methodology developed in this work to reconstruct app lineages, which are required to perform the evolution study (cf., Section II-C). Finally, we outline the research questions as well as the motivations behind them (cf., Section II-D).

A. Terminology

An *apk* represents a released package of an app. All apks in our dataset are uniquely identified based on their hash. App *version* is used in our work to refer to a specific *apk* released in the course of development of an app and the *n*th apk of an app is denoted as apk_n. In this article, we use both terms (*apk* and *version*) interchangeably.

An app *lineage* is defined as the consecutive series of its *versions* that is: $L = \{\text{apk}_1, \text{apk}_2, \ldots, \text{apk}_n\}$. In this work, a *lineage* may include only a subset of the *apks* that the app developers have released since our dataset, although massive, is not exhaustive.[2]

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.

B. Datasets and Tools

Our investigation relies on the AndroZoo repository [27], [28], which currently provides the largest publicly available dataset of Android apps. For harvesting vulnerability issues,

[2]Therefore, for a certain app lineage, there could be missing versions here and there.

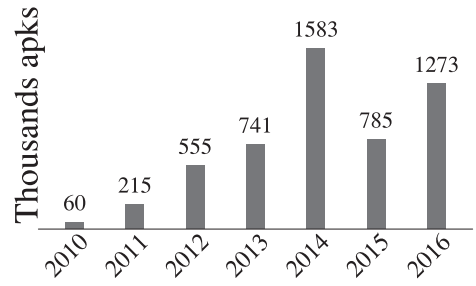

Fig. 1. apk packaging date.

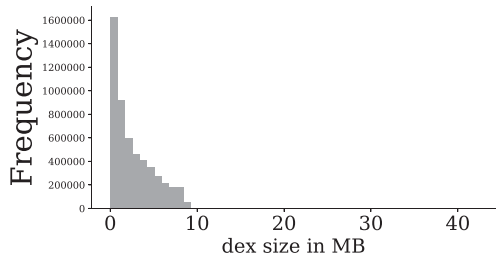

Fig. 2. Dalvik executable size.

we resort to state-of-the-art vulnerability detection tools from academia and the security industry.

1) App Dataset: AndroZoo currently hosts a dataset of over 5.5 millions distinct app packages (*apks*) collected from 14 representative markets (including the official Google Play store) and repositories (including the F-Droid open-source repository). According to the description of its crawlers [27], apks are continuously collected, opportunistically when they become visible to crawlers, to keep up with the evolution of apps in the Android ecosystem. This stream of apks is then immediately accessible to the research community via a download application programming interface (API).

As illustrated in Fig. 1, AndroZoo has collected apps that were packaged going back to 2010.[3] As reported by AndroZoo providers (cf., [27, p. 3]), the dataset does not represent an exact view of the apps ecosystem throughout time given that crawling was often challenged by various limitations including recurrent changes in store web page structures that require crawlers to be regularly reengineered, as well as occasional fails due to lack of storage space. Nevertheless, AndroZoo includes a diversity of apps: the barplot in Fig. 2 illustrates this in terms of app size distribution.[4]

2) Vulnerability Scanning: Vulnerabilities, also known as security-sensitive bugs [29], 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 article, 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.

1) *FlowDroid* [13]—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 [14], [30], [31] for tracking sensitive data flows, which can be associated with private data leaks.[5] The tool is still actively maintained [32]. Moreover, since the original version of FlowDroid can only analyze intracomponent data flows. In order to further consider the intercomponent data flows, in this article, we used an IccTA [14] enhanced version of FlowDroid.
2) *AndroBugs* [33] was first presented at the BlackHat security conference, after which the tool was open sourced [34]. This static detection tool was successfully used to find vulnerabilities and other critical security issues in Android apps developed by several big players [35]: it is notably credited in the security hall of fame of companies such as Facebook, eBay, Twitter, etc.
3) *IC3* [15] is a state-of-the-art static analyzer focused on resolving the target values in *intent* message objects used for intercomponent communication. The tool, which is maintained at Penn State University, can be used to track unauthorized intent reception [8], intent spoofing attacks [18], etc.

Table I summarizes the vulnerability checks that we focus on, in accordance with the capabilities of selected detection tools. Overall, we consider ten 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 criticality, we only considered the issues that are marked as critical by AndroBugs. Furthermore, several critical issues are also discarded, such as cases where the exploitation scenario was not clearly defined (e.g., checking for SQLiteDatabase transaction deprecated) and a few other issues that 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 [43], 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.

a) SSL security: Vulnerabilities related to SSL are a common concern in all modern software accessing the Internet [44]–[48]. In its basic form, any access to the Internet using the HTTP

[3]We use the Dalvik executable code compilation timestamp as the packaging date.

[4]We focus on the size of the actual executable code, since a given apk may include resource files such as images that may bias a global apk size metric.

[5]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 behaviors). 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 such sensitive data flows as privacy leaks in this work.

TABLE I
LIST OF CONSIDERED VULNERABILITIES

Type	Vulnerability checking description	Detection Tool
	Security features	
SSL_Security[36]	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[37]	Base64 String Encryption	AndroBugs
KeyStore[38]	KeyStore Protection	AndroBugs
	Permissions, privileges, sandbox, access-control	
Permission[38]	App Sandbox Permission	AndroBugs
IntentFilter[3]	Unauthorized Intent Reception	IC3
	Injection flaws	
Command[39]	Runtime Command	AndroBugs
	Runtime Critical Command	AndroBugs
WebView[40]	WebView RCE Vulnerability	AndroBugs
Fragment[41]	Fragment Vulnerability	AndroBugs
	Data and Communication Handling	
Intent[42]	Intent Spoofing	IC3
Leak[13]	Sensitive Data Flow	FlowDroid

protocol without encryption (i.e., without using https), as in the code example in Listing 1 line 4, could be subjected to MITM attacks [7].

```
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 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 2, the app developer implements the required `X509TrustManager` interface in line 6. Nevertheless, from line 7 to 9, the three 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 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 the hostname verifier with either an *SSLConnectionSocketFactory.* `ALLOW_ALL_HOSTNAME_VERIFIER` or implementation of *HostnameVerifier* interface, which overrides the *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, the 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 [49].

Finally, still with regards to SSL security, Listing 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 3. SSL vulnerability related to insecure component.

b) 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.* [39] 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`."

c) 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 interprocess communication files (i.e., sharing data between applications using files), since, in this mode, Android cannot limit the access only to the desired apps [50]. Nevertheless, the secure alternative of implementing *content provider* may be too demanding for developers, leading to the development of many vulnerable apps.

d) 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. Luo *et al.* [40] described this kind of attacks in detail.

e) 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 [38]. 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 that can then be obtained by attackers.

f) Fragment: A specific case of code injection can be implemented in apps running earlier versions of the Android operating system (OS): *fragment injection*, reported by researchers at IBM [41], exploits the fact that any user interface (UI) class (i.e., *Activity* extending *PreferenceActivity*) can load any other arbitrary class in a *Fragment* (i.e., Subactivity). 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. R. Hay has demonstrated[6] 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.

[6][Online]. Available: https://goo.gl/zQnpTq —Retrieved August 17, 2017.

g) 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 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. Vulnerable encryption using Base64 string encoding.

The next two vulnerability cases that we consider are related to the pervasive use of intercomponent 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 that 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 [8], [37], [42], [51], and [52].

h) Intent: Implicit intents, although they provide flexibility in run-time binding of components, are often reported to be overused or inappropriately used [42]. 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 article, following security recommendations in [42], 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.

i) 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 redelegation [3] of permissions to perform sensitive tasks. Best security practices require app developers to protect the offered capabilities with the relevant

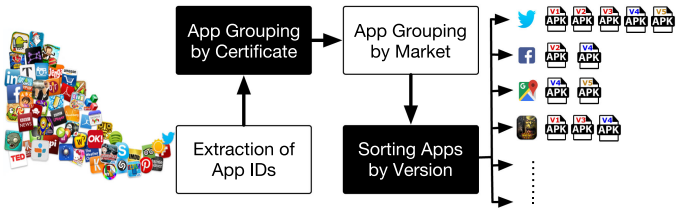

Fig. 3. App lineages reconstruction process.

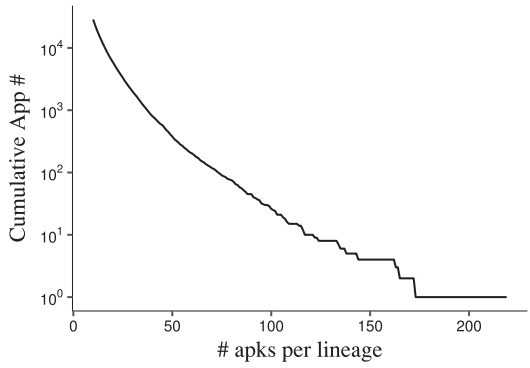

Fig. 4. Lineage sizes.

(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.

j) Leak: Sensitive data flows across app components and outside an app have been extensively studied in the literature [13], [14], [53]–[56]. 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.

C. Reconstruction of App Lineages

We now describe the process (illustrated in Fig. 3) that we followed to reconstruct app lineages from AndroZoo's data heap.

To reconstruct 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 that 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 [57] that each app should be named, in .he usual Java package naming convention.[7] 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 *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[8] attacks [58], [59], 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 related to the same app (based on the name), from the same development team (based on the signatures), and 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 that 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 [60] to set a threshold of at least ten apks before considering a lineage in our study. And during this step, almost 92% of apks were filtered out.

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. Fig. 4 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 II presents the top five lineages w.r.t. the number of apks. In total, our lineage dataset includes 465 037 apks. Fig. 5 compares dex sizes of apks between the original dataset and the reconstructed lineage dataset. It can be noticed that apks of small size has been removed mostly during lineage reconstruction, as the median value increased from around 2.6 to 3.3 MB.

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

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

TABLE II
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

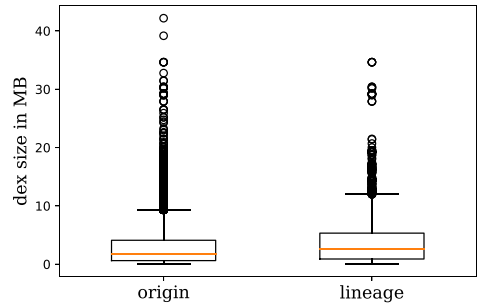

Fig. 5. Dex sizes.

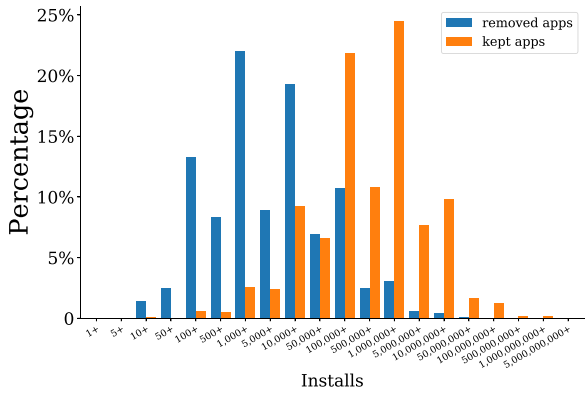

Fig. 6. Installation comparison between removed and kept apps.

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 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[9] apps and the installation situation for both removed and kept apps are shown in Fig. 6. 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.

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 metadata information for each lineage. We focused on our study on the official Google Play. Out of the 16 074 lineages that 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. Fig. 7 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. Fig. 8 presents the distribution of the 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 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.

D. 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 previously have been, at some periods, trending topics in the research community [61]–[65], we investigate whether they actually correspond to isolated issues in time.[10] 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 [26].

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

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

Fig. 7. Word cloud representation of categories associated with our selected lineage apps.

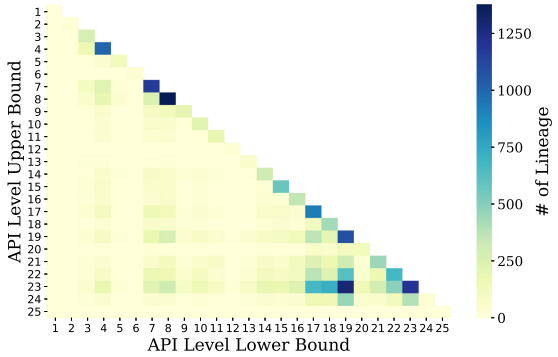

Fig. 8. API level span distribution of lineages.

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" [66] and "stagefright" [67]–[69] 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 [70]. A recent study by Watanabe *et al.* [71] 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.

E. Experimental Setup

1) 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 seven days and occupied 56 TB of local storage space. Among the vulnerability detection tools, FlowDroid and IC3, as previously reported in the literature [30], are heavy in terms of resource consumption. Fortunately, we were able to leverage a high performance computing (HPC) platform [72], using up to 80 nodes, to run as many analyses as possible. We use the *fully parallel* capability of the HPC platform and we automated the analysis scenarios with Python scripts: FlowDroid[11] 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 3 357 and 2 048 lineages, respectively.[12] The final raw results hold in about 40 GB of disk space. There are two 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 that 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. Toward 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 two Ph.D. students who have been working on Android and static analysis related topics to work on the reports of the three selected tools, respectively. One

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

[12]Since the obtained results for different vulnerabilities (i.e., tools) are different, the percentages calculated in the rest of this article are based on the analyzed apks of a certain vulnerability.

student spent one day on sampled reports[13] of AndroBugs and another one spent two days on the reports of IC3 and FlowDorid, respectively. They are able to check 711 vulnerabilities[14] 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 nontrivial 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 nonexist path, and hence, a false-positive result. Similar validations were done for Androbugs and IC3 as well, while no false positives were spotted.

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.* [30] to mine abnormal usage of sensitive data, Cai *et al.* [73] leverage IC3 to understand Android application programming and security, while Taylor *et al.* [26] 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.* [74] 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.* [75] 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.

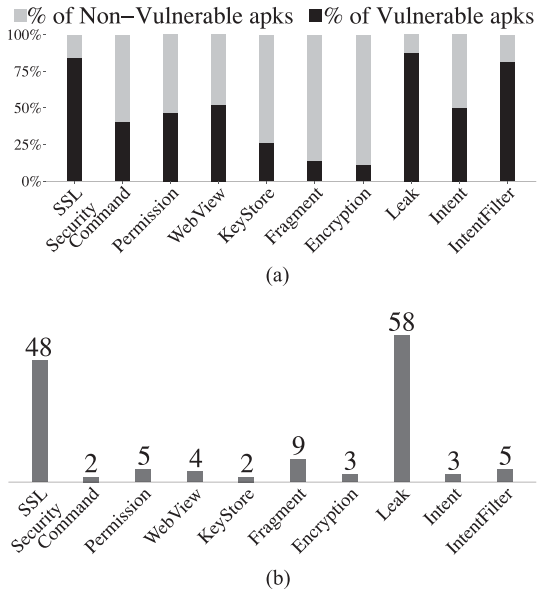

Fig. 9. Distributions of vulnerable apks and of vulnerabilities. (a) Proportion of vulnerable apks. (b) Average # of vulnerabilities per apk.

2) Study Protocol: 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. Fig. 9 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 II-B2.

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

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

[14] 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).

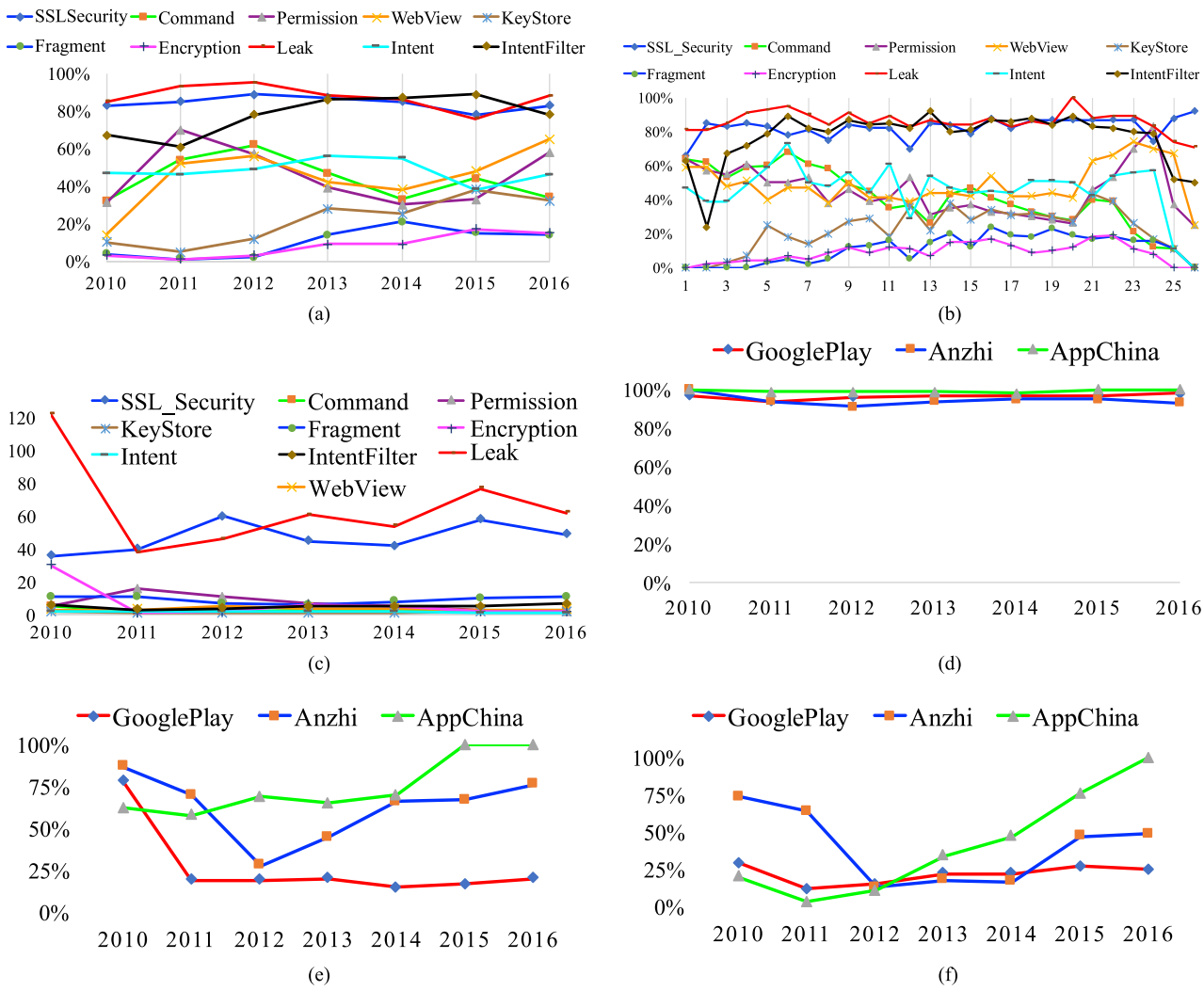

Fig. 10. Evolution of Android app vulnerabilities. (a) Percentage of vulnerable apps across time. (b) Percentage of vulnerable apps per API target. (c) Average # of vulnerabilities in apks. (d) Percentage of vulnerable apks in markets (three markets considered). (e) Percentage of apks with Command vulnerability in markets. (f) Percentage of apks with KeyStore vulnerability in markets.

same location, we compute the change diff between the two apk versions and refer to it as *potential vulnerability fix changes*.[15]

III. 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.

A. 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. Fig. 10(a) outlines the evolution of vulnerable apks in the space of six years. Clearly, we do not see any steady trend toward 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.[16] 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 Fig. 10(b) 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

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

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

to the various improvements made in the OS as well as in the app markets toward preventing the capability and permission abuse.

We further investigate whether the overall evolutions depicted previously break down differently in specific markets, given that markets do not implement the same security checking policies; and whether evolution trends are visible inside the apps, since developers may make efforts to at least reduce their numbers. Fig. 10(d) 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.[17] Evolution trends in Fig. 10(c) 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 may be explained by the wide interest of the community. TaintDroid [76], 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 [77]. MIT technology review had also realized on the wave of apps leaking private info [78].

Fig. 10(e) and (f) further depicts 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[18] 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.

B. Survivability of Vulnerabilities

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. Fig. 11(a) 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 Intercomponent communication (i.e., Leaks, Intent, and IntentFilter) has evolved substantially across app versions (e.g., only 20% of Leak vulnerabilities kept untouched). Fig. 11(b) 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.

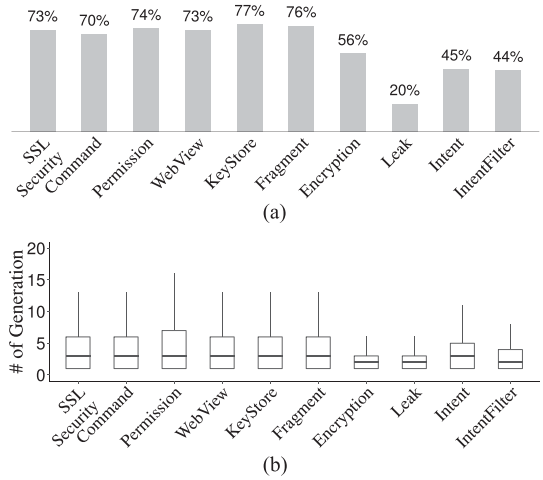

Fig. 11. Survivability of vulnerabilities in apks. (a) Percentage of vulnerabilities unaffected by updates. (b) # of updates before fix.

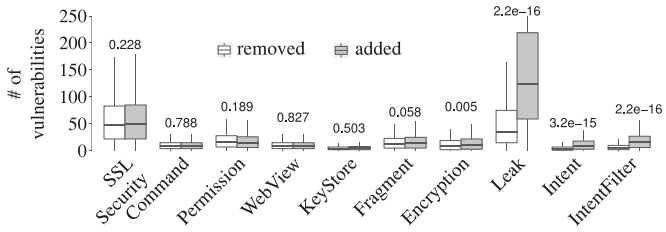

Fig. 12. Distribution of added and removed in a numbers of vulnerabilities between consecutive apk versions. Numbers represent p-values from MWW tests on the statistical significance of the differences.

We further detail in Fig. 12 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 the following finding in a recent study by Taylor *et al.* [26] on a smaller set of apps: "Android apps do not get safer as they are updated." The p-values of the 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

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

[18] [Online]. Available: https://www.android.com/play-protect/

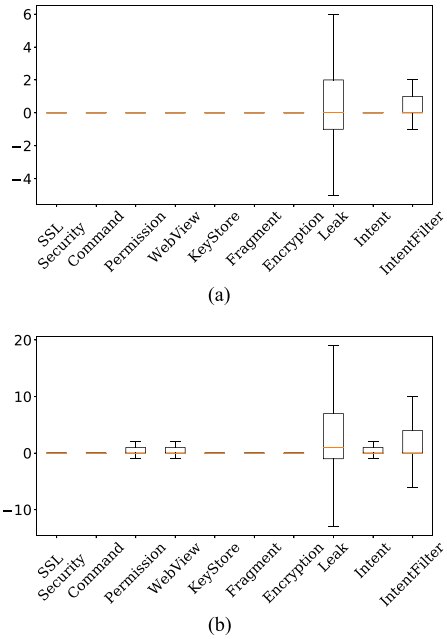

Fig. 13. Variations in # of vulnerabilities following updates. (a) Between consecutive apk pairs. (b) Between initial and latest versions.

pair, however, show that the difference is statistically significant only for the three ICC-related vulnerabilities.

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 Fig. 13 highlight the following simple reality: Commonly vulnerabilities are neither removed nor introduced during app updates [i.e., all median values equal to 0 in Fig. 13(a)] 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 Fig. 13(b), 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 and 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 one 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 three updates and even longer, this suggests developers have not been paying enough attention on vulnerability issues.

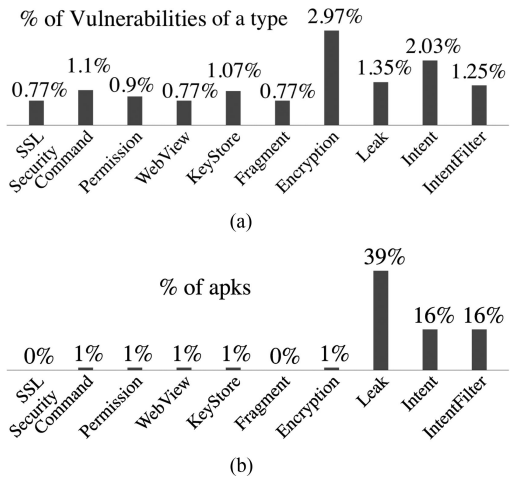

Fig. 14. Statistics on reintroduction occurrences. (a) Vulnerability reintroduction. (b) Vulnerability type reintroduction.

C. 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 II-A) for 15 375 distinct apps. On average, a given app is affected by 6.7 vulnerability reintroductions. Fig. 14(a) 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%).

We investigate whether, in some lineages, a *vulnerability type* may completely disappear at some point and later reappear. Fig. 14(b) 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 article, without specification, the reintroduction should be location based).

Fig. 15 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 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%.

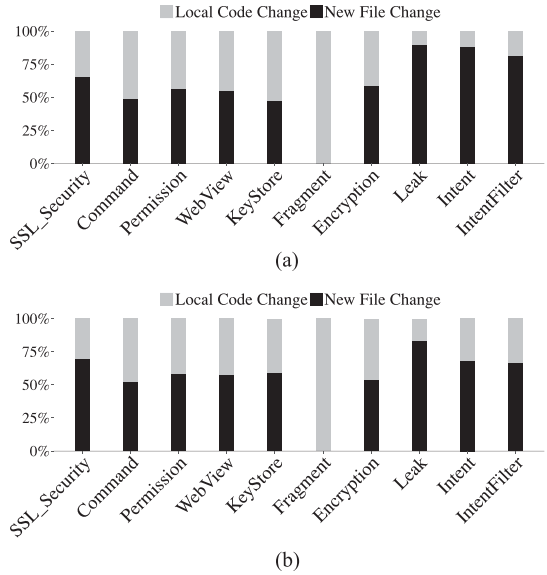

Fig. 15. Statistics on how vulnerabilities are removed (during file deletion or code change within vulnerability location file) or introduced (during new file insertion or code change within vulnerability location file). (a) Vulnerability removal. (b) Vulnerability introduction.

D. 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 the following two main location categories: *library* code and *developer* code. We attempt to provide a fine-grained view on the vulnerable-prone code by distinguishing between the following.

1) *Developer code*, approximated to all app components that share the same package name with the app package (i.e., app ID).
2) *Official libraries*, which we reduce in this work to only Android framework packages (e.g., that start with `com.google.android` or `android.widget`).
3) *Common libraries*, which we identify based on whitelists provided in the literature [79].
4) *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.

Fig. 16 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. The *developer code* is more affected by ICC data handling vulnerabilities. Android "official"[19] 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 [39].

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

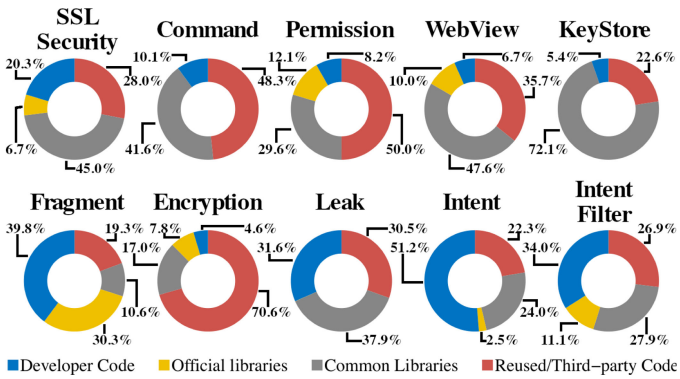

Fig. 16. Distribution of vulnerable code in *developer code*, *official libraries*, *common libraries*, and *reused/third party Code*.

TABLE III
SPEARMAN CORRELATION COEFFICIENT (ρ) VALUES

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

With experiments **Exp.1:** # of packages versus # of vulns. per apk; **Exp.2:** # of new packages versus # of vulns. per update.

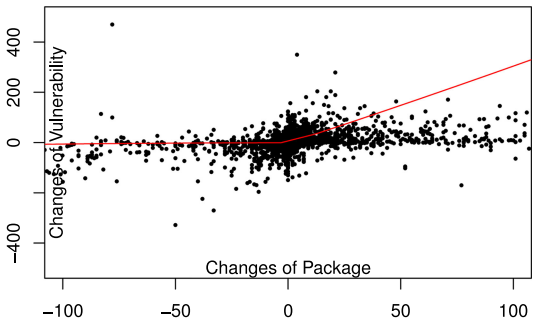

Fig. 17. Overall regression.

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 III 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.

Interestingly, computation of locally estimated scatterplot smoothing (LOESS) regression [80] shown in Fig. 17 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.

As numbers of the vulnerabilities are not correlated with both apk size and package numbers. We can deduce that not

TABLE IV
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%

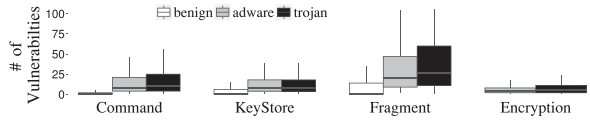

Fig. 18. Vulnerability versus malware. (Other vulnerabilities are ignored due to insignificant differences observed between benign, adware, and malware samples. Hence, they are omitted from the figure to give a clear exhibition.)

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 *nonvetted* libraries. Moreover, the claim made in [71] that more code and libraries imply more vulnerabilities may not be always true.

E. 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 [81]. Moreover, to distinguish malware from benign apks, the common practice is using antivirus (AV) flagging reports. AndroZoo provides these reports[20] as metadata for all its apks, and in this article, we treat an apk as malware as long as one or more AVs gave positive reports.

Table IV reports the proportion of benign and malware that 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 known to lack consensus among themselves [82], [83]. For every vulnerability type, we found that the Spearman's ρ was below 0.30, implying negligible correlation.

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 [84]. 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 nonflagged 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. Fig. 18 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 three 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.

IV. DISCUSSION

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

A. 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, which are as follows.

1) *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.
2) *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

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

work could be leveraged to mine fix patterns for certain vulnerabilities, and subsequently, enable the possibility of automated vulnerability fixes.

3) *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.
4) *Library screening strategies:* As concluded in Section III-D, 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, Bauser and Hebeisen [85], from the Lookout Security Intelligence team, have reported that their investigation of a suspicious ad software development kit (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 that unnecessarily ship vulnerable libraries.
5) *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.
6) *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.* [81] 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.

B. Threats to Validity

Like most empirical investigations, our study carries a number of threats to validity. We now briefly summarize 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 the internal validity is related to the process that we have designed for reconstructing 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 article. 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 that 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 IV-A, 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 two Ph.D. 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 as follows: 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.

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

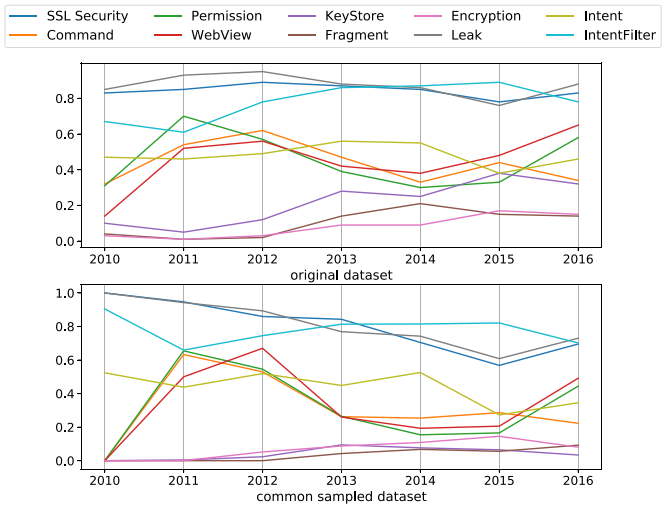

Fig. 19. Trends comparison between the original dataset (imbalanced) and the common sampled dataset (balanced).

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 Fig. 19, the results observed from the imbalanced dataset (top subfigure) and the common corpus (bottom subfigure) more or less follow similar trends, indicating that the empirical results observed will unlikely be impacted by the dataset chosen in this article.

V. 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 [76], FlowDroid [13], and IccTA [14]. Other analyzers have been proposed based on static analysis [6], [53], [86], dynamic analysis [87]–[91], or a combination of both [12] to find security issues in apps. In view of the amount of literature that relates to our work, we focus on the following three main topics.

1) 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.* [77] 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 redelegation attacks, while Grace *et al.* [5] focused on capability leaks. They unveiled vulnerabilities related to permissions. While this article studied different kind of vulnerabilities from the evolution aspect. Zhou *et al.* [59] 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.* [58] 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.

2) Vulnerability Studies: Vulnerabilities, also known as security-sensitive bugs, have been extensively studied in the literature [92] for different systems [93]–[97] and languages [44], [98]–[100]. Camilo *et al.* [29] have recently investigated the Chromium project to check whether bugs foreshadow vulnerabilities. Researchers have also proposed approaches to automatically patch them [101], [102].

In the Android literature, several studies have already been performed, which are as follows. Bagheri *et al.* [103] have recently analyzed the vulnerabilities of the permission system in Android OS; Huang *et al.* [104] 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.* [105] have analyzed Android framework and found six until-then unknown vulnerabilities in three common services and two shipped apps, while Cao *et al.* [106] focused on analyzing input validation mechanisms. Qian *et al.* [16] have developed a new static analysis framework for vulnerability detection. Thomas *et al.* [39] have analyzed 102k+ apks to study a common vulnerabilities and exposures (CVE) reported vulnerability on the JavaScript-to-Java interface of the WebView API. Jimenez *et al.* [107] 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.* [108] have then presented a larger-scale empirical study on Android OS-related vulnerabilities. OS vulnerabilities have also been investigated by Thomas *et al.* [109] 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.* [71], 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* [110] 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 V lists the works that 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 article. 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.

TABLE V
RELATED WORKS IN VULNERABILITY STUDY

Work	APK #	Type #	Detail	Year
Fahl *et al.* [10]	13,500	1	Studied only SSL security vulnerabilities	2012
Jiang *et al.* [11]	62,519	1	2 vuls stem from content provider components which are called passive content leaks and content pollutions	2013
Sounthiraraj *et al.* [12]	23,418	1	Studied SSL security by using both static and dynamic analysis	2014
Watanabe *et al.* [71]	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.* [26]	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

3) Software Evolution: The software engineering literature includes a large body of work on software maintenance and evolution [111]–[115]. We focus our discussion on recent Android-related work. In one of the latest work, Coppola *et al.* have investigated the fragility in Android GUI testing by utilizing 21 metrics to measure the adoption tools and their evolution [116]. Differently, our work is more focused on Android app vulnerability analysis. Calciati and Gorla [24] have leveraged the AndroZoo dataset to investigate the evolution of permission requests by Android apps. Taylor and Martinovic [26] also investigated how permission usage by apps, as well as security and privacy issues, have changed over a two-year period. Our work scales their approach with a larger set of apps, spanning a larger timeline, and considering entire lineages instead of only a pair of app versions. Evolution of the Android OS code has also been investigated. McDonnell *et al.* have empirically studied the stability of Android APIs [117], while Li *et al.* focused on the evolution of inaccessible Android APIs [25]. Thanks to the lineage dataset that we have collected, further studies can be performed to check the alignment between OS API evolution and API usage evolution in app code.

VI. CONCLUSION

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 addressed these challenges by reconstructing 28 564 lineages formed in total by 465 037 apks. We have then run computationally expensive vulnerability scanning experiments on these app lineages providing a view vulnerability evolution, which was so far the largest scale of this kind of studies. Moreover, investigating Android vulnerabilities from app lineage point of view was the major novelty of this study, which allowed us to yield several newly spotted findings, which are as follows:

1) most vulnerabilities could survive at least three updates;
2) part of third-party libraries were the major contributors of the most vulnerabilities;
3) vulnerability reintroduction occurs for all kinds of vulnerabilities, while *Encryption*-related vulnerabilities were the most reintroduced within all types of this study;
4) some vulnerabilities may foreshadow malware.

In addition to new findings, this large-scale study also confirmed most of the conclusions from previous studies with relatively small datasets. However, the result of this study also suggested that the recent claim made by Watanabe *et al.* [71] that more code and libraries imply more vulnerabilities may not be always true. Finally, the following three valuable artifacts produced by this study: *app lineages*; the complete dataset of *vulnerability scanning reports*; and recorded *vulnerable pieces of code*, were shared.

REFERENCES

[1] C. Smith, *75 Amazing Android Statistics and Facts*, Aug. 2017. [Online]. Available: http://expandedramblings.com/index.php/android-statistics/. Accessed: Aug. 20, 2017.
[2] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon, "Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing Android," *IEEE Trans. Softw. Eng.*, vol. 40, no. 6, pp. 617–632, Jun. 2014.
[3] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. USENIX Secur. Symp.*, 2011, vol. 30, p. 22.
[4] T. Vidas, D. Votipka, and N. Christin, "All your droid are belong to us: A survey of current Android attacks." in *Proc. 5th USENIX Conf. Offensive Technol.*, San Francisco, CA, USA, 2011, p. 1.
[5] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, vol. 14, 2012, p. 19.
[6] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting Android apps for component hijacking vulnerabilities," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 229–240.
[7] Y. Desmedt, "Man-in-the-middle attack," in *Encyclopedia of Cryptography and Security*. Boston, MA, USA: Springer, 2011, p. 759.
[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. 9th Int. Conf. Mobile Syst., Appl., Services*, 2011, pp. 239–252.
[9] J. Clark and P. C. van Oorschot, "SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements," in *Proc. IEEE Secur. Privacy Symp.*, 2013, pp. 511–525.
[10] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love Android: An analysis of Android SSL (in) security," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 50–61.
[11] J. Xuxian and Y. Zhou, "Detecting passive content leaks and pollution in Android applications," in *Proc. 20th Netw. Distrib. Syst. Secur. Symp.*, 2013.
[12] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "SMV-hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps," in *Proc. 21st Annu. Netw. Distrib. Syst. Secur. Symp.*, 2014.
[13] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.
[14] L. Li *et al.*, "ICCTA: Detecting inter-component privacy leaks in Android apps," in *Proc. IEEE 37th Int. Conf. Softw. Eng.*, vol. 1, 2015, pp. 280–291.

[15] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *Proc. IEEE 37th Int. Conf. Softw. Eng.*, vol. 1, 2015, pp. 77–88.

[16] C. Qian, X. Luo, Y. Le, and G. Gu, "Vulhunter: Toward discovering vulnerabilities in Android applications," *IEEE Micro*, vol. 35, no. 1, pp. 44–53, Jan. 2015.

[17] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in Android and its security applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 356–367.

[18] D. Kantola, E. Chin, W. He, and D. Wagner, "Reducing attack surfaces for intra-application communication in Android," in *Proc. 2nd ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2012, pp. 69–80.

[19] M. Xu *et al.*, "Toward engineering a secure Android ecosystem: A survey of existing techniques," *ACM Comput. Surveys*, vol. 49, no. 2, pp. 38:1–38:47, 2016.

[20] W. You *et al.*, "Reference hijacking: Patching, protecting and analyzing on unmodified and non-rooted Android devices," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 959–970.

[21] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of Android apps," in *Proc. IEEE 39th Int. Conf. Softw. Eng.*, Piscataway, NJ, USA, 2017, pp. 358–369.

[22] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: Toward extracting hidden code from packed Android applications," in *Computer Security—ESORICS 2015*, G. Pernul, P. Y. A. Ryan, and E. Weippl, Eds. Cham, Switzerland: Springer, 2015, pp. 293–311.

[23] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of Android applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, 2016, pp. 343–355.

[24] P. Calciati and A. Gorla, "How do apps evolve in their permission requests? A preliminary study," in *Proc. IEEE 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 37–41.

[25] L. Li, T. F. Bissyandé, Y. L. Traon, and J. Klein, "Accessing inaccessible android APIs: An empirical study," in *Proc. 32nd Int. Conf. Softw. Maintenance Evolution*, 2016, pp. 411–422.

[26] V. F. Taylor and I. Martinovic, "To update or not to update: Insights from a two-year study of Android app evolution," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 45–57.

[27] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Androzoo: Collecting millions of Android apps for the research community," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, Austin, TX, USA, May 2016, pp. 468–471.

[28] AndroZoo. [Online]. Available: http://androzoo.uni.lu, Accessed on: 2017.

[29] F. Camilo, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities? A study of the chromium project," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories/*, 2015, pp. 269–279.

[30] V. Avdiienko *et al.*, "Mining apps for abnormal usage of sensitive data," in *Proc. IEEE 37th Int. Conf. Softw. Eng.—Vol. 1*, 2015, pp. 426–436.

[31] J. Burket, L. Flynn, W. Klieber, J. Lim, and W. Snavely, "Making DidFail succeed: Enhancing the CERT static taint analyzer for android app sets," Mar. 2015, doi: 10.1184/R1/6575201.v1.

[32] FlowDroid. [Online]. Available: https://blogs.uni-paderborn.de/sse/tools/flowdroid/, Accessed on: 2017.

[33] Y.-C. Lin, "Androbugs framework: An android application security vulnerability scanner," in *Proc. Blackhat Eur.*, 2015.

[34] AndroBugs. [Online]. Available: https://github.com/AndroBugs/AndroBugs_Framework, Accessed on: 2017.

[35] AndroBugs, "Hall of fame." [Online]. Available: https://www.androbugs.com/#hof, Accessed on: 2017.

[36] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A view to a kill: Webview exploitation," in *Proc. USENIX Workshop Large-Scale Exploits Emergent Threats*, 2013, p. 7.

[37] Y. Cifuentes, L. Beltrán, and L. Ramírez, "Analysis of security vulnerabilities for mobile health applications," in *Proc. 7th Int. Conf. Mobile Comput. Netw.*, 2015, pp. 1067–1072.

[38] M. Sabt and J. Traoré, "Breaking into the keystore: A practical forgery attack against Android keystore," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2016, pp. 531–548.

[39] D. R. Thomas, A. R. Beresford, T. Coudray, T. Sutcliffe, and A. Taylor, "The lifetime of Android API vulnerabilities: Case study on the javascript-to-java interface," in *Cambridge Int. Workshop Secur. Protocols*, 2015, pp. 126–138.

[40] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on webview in the Android system," in *Proc. ACM Annu. Comput. Secur. Appl. Conf.*, 2011, pp. 343–352.

[41] R. Hay, "Android collapses into fragments," in *IBM Security Systems*, 2013. [Online]. Available: https://securityintelligence.com/wp-content/uploads/2013/12/android-collapses-into-fragments.pdf, Accessed on: 2017.

[42] A. Cozzette, "Intent spoofing on Android," Aug. 20, 2017. [Online]. Available: http://blog.palominolabs.com/2013/05/13/android-security/index.html

[43] S. Arzt, S. Rasthofer, and E. Bodden, "The soot-based toolchain for analyzing Android apps," in *Proc. IEEE 4th Int. Conf. Mobile Softw. Eng. Syst.*, 2017, pp. 13–24.

[44] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State-of-the-art: Automated black-box web application vulnerability testing," in *Proc. IEEE Secur. Privacy, Symp.*, 2010, pp. 332–345.

[45] M. Bland, "Finding more than one worm in the apple," *Commun. ACM*, vol. 57, no. 7, pp. 58–64, 2014.

[46] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating SSL certificates in non-browser software," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 38–49.

[47] C. Meyer and J. Schwenk, "SoK: Lessons learned from SSL/TLS attacks," in *Proc. Int. Workshop Inf. Secur. Appl.*, 2013, pp. 189–209.

[48] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, "When private keys are public: Results from the 2008 Debian openSSL vulnerability," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas. Conf.*, 2009, pp. 15–27.

[49] C. Zuo, J. Wu, and S. Guo, "Automatically detecting SSL error-handling vulnerabilities in hybrid mobile web apps," in *Proc. 10th ACM Symp. Inf., Comput. Commun. Secur.*, New York, NY, USA, 2015, pp. 591–596.

[50] OWASP. "Mobile top 10 2014-m2: Insecure data storage," Aug. 20, 2017. [Online]. Available: https://www.owasp.org/index.php/Mobile_Top_10_2014-M2

[51] A. Cozzette *et al.*, "Improving the security of Android inter-component communication," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage.*, 2013, pp. 808–811.

[52] D. Sbîrlea, M. G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar, "Automatic detection of inter-application permission leaks in Android applications," *IBM J. Res. Develop.*, vol. 57, no. 6, pp. 10:1–10:12, 2013.

[53] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 235–245.

[54] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale," *Trust*, vol. 12, pp. 291–307, 2012.

[55] H. T. Ly, Tan C. Nguyen, and V.-H. Pham, "eDSDroid: A hybrid approach for information leak detection in Android," in *Proc. Int. Conf. Inf. Sci. Appl.*, 2017, pp. 290–297.

[56] S. Yovine and G. Winniczuk, "Checkdroid: A tool for automated detection of bad practices in Android applications using taint analysis," in *Proc. IEEE 4th Int. Conf. Mobile Softw. Eng. Syst.*, 2017, pp. 175–176.

[57] Google, "App id." Jul. 16, 2017. [Online]. Available: https://developer.android.com/studio/build/application-id.html

[58] L. Li *et al.*, "Understanding Android app piggybacking: A systematic study of malicious code grafting," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 6, pp. 1269–1284, Jun. 2017.

[59] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 95–109.

[60] L. Li, T. Bissyand, and J. Klein, "Moonlightbox: Mining Android API histories for uncovering release-time inconsistencies," in *Proc. IEEE 29th Int. Symp. Softw. Rel. Eng.*, Oct. 2018, pp. 212–223.

[61] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Comput. Surveys*, vol. 49, no. 4, pp. 76:1–76:41, 2017.

[62] P. Faruki *et al.*, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 2, pp. 998–1022, Secondquarter 2015.

[63] D. J. J. Tan *et al.*, "Securing Android: A survey, taxonomy, and challenges," *ACM Comput. Surveys*, vol. 47, no. 4, pp. 58:1–58:45, 2015.

[64] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE Trans. Softw. Eng.*, vol. 43, no. 9, pp. 817–847, Sep. 2017.

[65] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software," *IEEE Trans. Softw. Eng.*, vol. 43, no. 6, pp. 492–530, Jun. 2017.

[66] Z. Durumeric *et al.*, "The matter of heartbleed," in *Proc. Conf. Internet Meas. Conf.*, 2014, pp. 475–488.

[67] J. Drake, "Stagefright: Scary code in the heart of android," in *Proc. BlackHat USA*, 2015.

[68] M. Burgess, "Millions of Android devices vulnerable to new stagefright exploit," Aug. 20, 2017. [Online]. Available: http://www.wired.co.uk/article/stagefright-android-real-world-hack

[69] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for art," in *Proc. 26th USENIX Conf. Secur. Symp.*, Berkeley, CA, USA, 2017, pp. 289–306.

[70] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," Aspect Security Inc., Columbia, MD, USA, pp. 1–26, 2012.

[71] T. Watanabe *et al.*, "Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 14–24.

[72] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic HPC cluster: The UL experience," in *Proc. IEEE Intl. Conf. High Perform. Comput. Simul.*, Bologna, Italy, Jul. 2014, pp. 959–967.

[73] H. Cai and B. G. Ryder, "Understanding Android application programming and security: A dynamic study," in *Proc. IEEE Int. Conf. Softw. Maintenance Evolution*, 2017, pp. 364–375.

[74] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, New York, NY, USA, 2018, pp. 176–186.

[75] F. Ibrar, H. Saleem, S. Castle, and M. Z. Malik, "A study of static analysis tools to detect vulnerabilities of branchless banking applications in developing countries," in *Proc. 9th Int. Conf. Inf. Commun. Technol. Develop.*, New York, NY, USA, 2017, pp. 30-1–30-5.

[76] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. USENIX Secur. Symp.*, 2010, pp. 393–407.

[77] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proc. USENIX Secur. Symp.*, 2011, vol. 2, p. 21.

[78] R. Lemos, "Your apps could be leaking private info," MIT Technol. Rev., Aug. 21, 2017. [Online]. Available: https://www.technologyreview.com/s/420062/your-apps-could-be-leaking-pr% ivate-info/

[79] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in Android apps," in *Proc. 23rd IEEE Int. Conf. Softw. Anal., Evolution, Reeng.*, 2016, pp. 403–414.

[80] W. S. Cleveland and S. J. Devlin, "Locally weighted regression: An approach to regression analysis by local fitting," *J. Amer. Stat. Assoc.*, vol. 83, no. 403, pp. 596–610, 1988.

[81] K. Hamandi, A. Chehab, I. H. Elhajj, and A. Kayssi, "Android SMS malware: Vulnerability and mitigation," in *Proc. 27th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, Mar. 2013, pp. 1004–1009.

[82] M. Hurier, K. Allix, T. Bissyandé, J. Klein, and Y. Le Traon, "On the lack of consensus in anti-virus decisions: Metrics and insights on building ground truths of Android malware," in *Proc. 13th Conf. Detection Intrusions Malware Vulnerability Assessment*, 2016, pp. 142–162.

[83] A. Kantchelian *et al.*, "Better malware ground truth: Techniques for weighting anti-virus vendor labels," in *ACM Workshop Artif. Intell. Secur.*, 2015, pp. 45–56.

[84] M. Hurier *et al.*, "Euphony: Harmonious unification of cacophonous anti-virus vendor labels for Android malware," in *Proc. IEEE 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 425–435.

[85] A. Bauser and C. Hebeisen, "Igexin advertising network put user privacy at risk," Aug. 2017. [Online]. Available: https://blog.lookout.com/igexin-malicious-sdk

[86] D. Wu and R. K. C. Chang, "Analyzing Android browser apps for file://vulnerabilities," in *Proc. Int. Conf. Inf. Secur.* 2014, pp. 345–363.

[87] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in Android applications," in *Proc. Int. Workshop Inf. Secur. Appl.*, 2013, pp. 138–159.

[88] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in Android," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 118–128.

[89] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic security analysis of smartphone applications," in *Proc. 3rd ACM Conf. Data Appl. Secur. Privacy*, 2013, pp. 209–220.

[90] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices," in *Proc. Int. Conf. Secur. Cryptography*, 2013, pp. 461–468.

[91] J. Schütte, R. Fedler, and D. Titze, "Condroid: Targeted dynamic analysis of Android applications," in *Proc. IEEE 29th Int. Conf. Adv. Inf. Netw. Appl.*, 2015, pp. 571–578.

[92] I. V. Krsul, "Software vulnerability analysis," Ph. D. dissertation, Purdue Univ., West Lafayette, IN, USA, 1998.

[93] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proc. IEEE DARPA Inf. Survivability Conf. Expo.*, 2000, vol. 2, pp. 119–129.

[94] C. DOrazio and K.-K. R. Choo, "A generic process to identify vulnerabilities and design weaknesses in iOS healthcare apps," in *Proc. 48th Hawaii Int. Conf. Syst. Sci.*, 2015, pp. 5175–5184.

[95] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proc. USENIX Secur. Symp.*, 2005, vol. 14, p. 18.

[96] B. Schwarz *et al.*, "Model checking an entire linux distribution for security violations," in *Proc. IEEE 21st Annu. Comput. Secur. Appl. Conf.*, 2005, pp. 1–10.

[97] C. Zuo, Q. Zhao, and Z. Lin, "AUTHSCOPE: Towards automatic discovery of vulnerable authorizations in online services," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, 2017, pp. 799–813.

[98] S. Jain, D. S. Tomar, and D. R. Sahu, "Detection of javascript vulnerability at client agent," *Int. J. Sci. Technol. Res.*, vol. 1, no. 7, pp. 36–41, 2012.

[99] A. Kieyzun, P. J Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 199–209.

[100] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities," in *Proc. IEEE 34th Int. Conf. Softw. Eng.*, 2012, pp. 1293–1296.

[101] F. Yu, M. Alkhalaf, and T. Bultan, "Patching vulnerabilities with sanitization synthesis," in *Proc. ACM 33rd Int. Conf. Softw. Eng.*, 2011, pp. 251–260.

[102] M. Zhang and H. Yin, "AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 45–61.

[103] 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.*, vol. 30, pp. 525–544, 2016.

[104] H. Huang, S. Zhu, K. Chen, and P. Liu, "From system services freezing to system server shutdown in Android: All you need is a loop in an app," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1236–1247.

[105] K. Wang, Y. Zhang, and P. Liu, "Call me back!: Attacks on system server and system apps in Android through synchronous callback," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 92–103.

[106] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards analyzing the input validation vulnerabilities associated with Android system services," in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, 2015, pp. 361–370.

[107] M. Jimenez, M. Papadakis, T. F. Bissyandé, and J. Klein, "Profiling Android vulnerabilities," in *Proc. IEEE Int. Conf. Softw. Quality, Rel. Secur.*, 2016, pp. 222–229.

[108] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velásquez, "An empirical study on Android-related vulnerabilities," in *Proc. IEEE 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 2–13.

[109] D. R Thomas, A. R. Beresford, and A. Rice, "Security metrics for the Android ecosystem," in *Proc. 5th Annu. ACM CCS Workshop Secur. Privacy Smartphones Mobile Devices*, 2015, pp. 87–98.

[110] J. Mitra and V.-P. Ranganath, "Ghera: A repository of Android app vulnerability benchmarks," in *Proc. 13th Int. Conf. Predictive Models Data Analytics Softw. Eng.*, Toronto, Canada, 2017, pp. 43–52.

[111] E. J. Barry, C. F. Kemerer, and S. A. Slaughter, "On the uniformity of software evolution patterns," in *Proc. IEEE 25th Int. Conf. Softw. Eng.*, 2003, pp. 106–113.

[112] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of API-level refactorings during software evolution," in *Proc. ACM 33rd Int. Conf. Softw. Eng.*, 2011, pp. 151–160.

[113] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. IEEE*, vol. 68, no. 9, pp. 1060–1076, Sep. 1980.

[114] T. Mens, "Introduction and roadmap: History and challenges of software evolution," in *Software Evolution*. Berlin, Germany: Springer, 2008, pp. 1–11.

[115] Q. Tu *et al.*, "Evolution in open source software: A case study," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2000, pp. 131–142.

[116] R. Coppola, M. Morisio, and M. Torchiano, "Mobile GUI testing fragility: A study on open-source Android applications," *IEEE Trans. Rel.*, vol. 68, no. 1, pp. 67–90, Mar. 2019.

[117] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *Proc. IEEE 29th Int. Conf. Softw. Maintenance*, 2013, pp. 70–79.