Understanding the Evolution of Android App Vulnerabilities

Jun Gao*, Li Li†, Pingfan Kong*, Tegawendé F. Bissyandé*, Jacques Klein*
*University of Luxembourg, Luxembourg
†Monash University, Australia
{jun.gao, pingfan.kong, tegawende.bissyande, jacques.klein}@uni.lu
li.li@monash.edu

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 OS, 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 re-construct versioned lineages of Android apps and finally obtained $28{,}564$ app lineages (i.e., successive releases of the same Android apps) with more than 10 app versions each, corresponding to a total of $465{,}037$ apks. 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.

1 Introduction

Mobile software has been overtaking traditional desktop software to support citizens of our digital era in an ever-increasing number of activities, including for leisure, internet communication or commerce. In this ecosystem, the most popular and widely deployed platform is undoubtedly Android, powering more than 2 billion monthly active users, and contributing to over 3 million mobile applications, hereinafter referred to as *apps*, in online software stores [1]. Yet from a security standpoint, the Android stack has been pointed out as being flawed in several studies: among various issues, its permission model has been extensively criticized for increasing the attack surface [2], [3], [4]; the complexity of its message passing system has led to various vulnerabilities in third-party apps allowing for capability leaks [5] or component hijacking attacks [6]; 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], [9], [10], [11], [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], [14], [15], [16] and by proposing improvements to current security models [17], [18], [19], [20]. Although advanced techniques have been employed by malicious developers such as packer and obfuscation, countermeasures such as [21], [22], [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

This work is supported by the Luxembourg National Research Fund (FNR) through grant PRIDE15/10621687/SPsquared and project CHARACTERIZE C17/IS/1169386.

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

few works [24], [25], [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.

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

- We 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 10 versions of a given app. Although this contribution serves the purpose of our study, it is a **valuable artefact** for diverse research fields in our community, notably software quality and its sub-fields of testing, repair and evolution studies.
- We apply state-of-the-art static vulnerability finding tools on all app versions and record the alerts raised as well as their locations. We investigate specifically 10 vulnerability types associated to 4 different categories related to common security features (e.g., SSL), its sandbox mechanism (e.g., Permission issues), code injection (e.g., WebView RCE vulnerability) as well as its inter-app message passing (e.g., Intent spoofing). Correlating the analysis results for consecutive app pairs in lineages, we extract a **comprehensive dataset of reported vulnerable pieces of code in real-world apps**, and, whenever available, the subset of changes that were applied to fix the vulnerabilities.
- Finally, we perform **several empirical analyses** to (1) highlight statistical trends on the temporal evolutions of vulnerabilities in Android apps, (2) capture the common locations (e.g., developer vs library code) of vulnerable code in apps, (3) comprehend the vehicle (e.g., code change, new files, etc.) through which vulnerabilities are introduced in mobile apps, (4) investigate via correlation analysis whether vulnerabilities foreshadow malware in the Android ecosystem.

And the main findings are:

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

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

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

https://avedroid.github.io

The remainder of this paper is organized as follows: Section 2 describes the experimental setup, including the construction of app lineages, an introduction of the vulnerability finding tools, and the research questions. Section 3 unfolds the empirical analyses. Section 4.2 enumerates threats to validity while Section 4.1 discusses some promising future works. Section 5 discusses related work and Section 6 concludes this work.

2 Study Data & Design

In this section, we first define and clarify some terms used in the paper. 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 2.2. Third, we describe the methodology developed in this work to re-construct app lineages, which are required to perform the evolution study (cf. Section 2.3). Finally, we outline the research questions as well as the motivations behind them (cf. Section 2.4).

2.1 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 nth apk of an app is denoted as apk_n. In this paper, we use both terms (*apk* and *version*) interchangeably.

An app **lineage** is defined as the consecutive series of its *versions* that is: $L = \{apk_1, apk_2, \ldots, apk_n\}$. 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.

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

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

2.2.1 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 API.

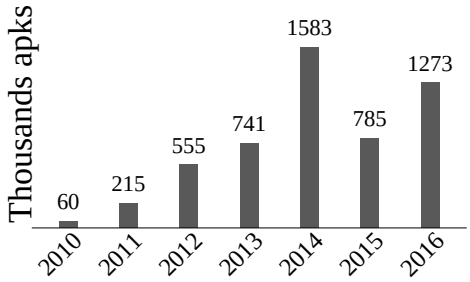


Fig. 1: APK Packaging Date

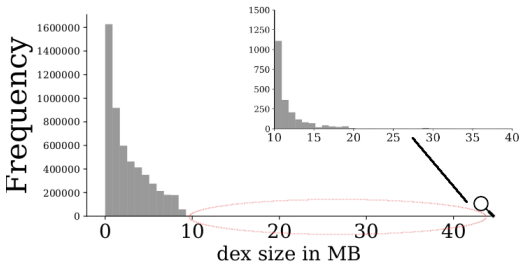


Fig. 2: Dalvik Executable Size.

As illustrated in Figure 1, AndroZoo has collected apps which 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 re-engineered, as well as occasional fails due to lack of storage space. Nevertheless, AndroZoo includes a diversity of apps: the barplot in Figure 2 illustrates this in terms of app size distribution[4].

2.2.2 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 work, we focus on selecting common vulnerabilities with a severity level that justifies that they are highlighted in security reports and in previous software security studies. Before enumerating the vulnerabilities considered in our work, we describe the vulnerability detection tools (detection tools for short hereafter) that we rely upon to statically scan Android apps.

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

- **FlowDroid** [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 intra-component data flows. In order to further consider the inter-component data flows, in this work, we used an ICCTA [14] enhanced version of FlowDroid.
- **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.
- **IC3** [15] is a state-of-the-art static analyzer focused on resolving the target values in *intent* message objects used for inter-component communication. The tool, which is maintained at Penn State University, can be used to track unauthorized Intent reception [8], Intent spoofing attacks [18], etc.

Table 1 summarizes the vulnerability checks that we focus on, in accordance with the capabilities of selected detection tools. Overall, we consider 10 vulnerability types. For AndroBugs, not like other detection tools, it reports on dozens of issues. To focus on those vulnerabilities having a high level of criticality, we only considered the issues which are marked as critical by AndroBugs. Furthermore, several critical issues are also discarded, such as cases where exploitation scenario was not clearly defined (e.g., checking for SQLiteDatabase transaction deprecated) and a few other issues which were not explicitly about executable code (e.g., relevant to Manifest information), to eliminate the share of noise that they can bring to the study.

We now detail the different vulnerabilities and explicate their potential exploitation scenarios. Due to space constraints, we provide actual vulnerable code examples for only a few cases. For other cases, we provide references to the interested reader. Since all apps are collected from markets without source code, we use Soot [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.

2.2.2.1 SSL Security: Vulnerabilities related to SSL are a common concern in all modern software accessing the

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 which 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 behaviours). Nonetheless, since such sensitive data flows indeed send private data outside the device, and it is hard to know how these private data will be used, we consider in this work such sensitive data flows as privacy leaks.

TABLE 1: 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

Internet [44], [45], [46], [47], [48]. In its basic form, any access to the Internet using the HTTP protocol without encryption (i.e. without using https), as in the code example in Listing 1 line 4, could be subjected to man-in-the-middle (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 3 implemented methods are empty, which only ensures that the app compiles, but creates vulnerabilities for MITM attacks.

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

Listing 2: SSL Vulnerability Related to Certificate Verification.

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

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

2.2.2.2 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`'.

2.2.2.3 Permission.: The Android application sandbox security feature isolates each app data and code execution from other apps. However, the documentation explicitly recommends to avoid permissions `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE` for inter-process communication files (i.e., sharing data between applications using files), since, in this mode, Android cannot limit the access only to the desired apps [50]. Nevertheless, the secure alternative of implementing Content Provider may be too demanding for developers, leading to the development of many vulnerable apps.

2.2.2.4 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. [40] detailedly described this kind of attacks.

2.2.2.5 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 which can then be obtained by attackers.

2.2.2.6 Fragment.: A specific case of code injection can be implemented in apps running earlier versions of the Android OS: *fragment injection*, reported by researchers at IBM [41], exploits the fact that any UI class (i.e., *Activity* extending *PreferenceActivity*) can load any other arbitrary class in a *Fragment* (i.e., sub-Activity). When the UI class is exported (i.e., can be reused by other apps – for example, a mail app may directly allow viewing a PDF attachment by calling a reader app activity), malicious apps can break the sandbox mechanism by accessing information pertaining to the vulnerable apps or abuse its permissions. Roee Hey has demonstrated[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.

2.2.2.7 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 Inter-Component Communication (ICC) for enabling interaction and information exchange between Android app components (within and across apps). Two Android concepts are key in these scenarios: the *intent* object, which is created by a component to hold the data and action request that must be transferred to another component, and the *intentfilter* attribute, which specifies the kind of intents that the declaring component can handle. When intents are *implicit*, i.e., they do not name a recipient component, they are routed by the system to the appropriate components with matching intent filters. Security of intents can then be compromised by malicious apps which may exploit vulnerabilities to intercept intents intended for another, or by sending malformed data to induce undesired behavior in a vulnerable app. These attacks, known as *intent interception* and *intent spoofing* attacks, have been studied in detail in the literature [8], [37], [42], [51], [52].

2.2.2.8 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 study, 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.

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

2.2.2.10 Leak.: Sensitive data flows across app components and outside an app have been extensively studied in the literature [13], [14], [53], [54], [55], [56]. When such flows depart from known sensitive *sources* (e.g., API meth-

6. https://goo.gl/zQnpTq - Retrieved August 17, 2017

ods 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.

2.3 Re-construction of App Lineages

We now describe the process (illustrated in Figure 3) that we followed to re-construct app lineages from AndroZoo's data heap.

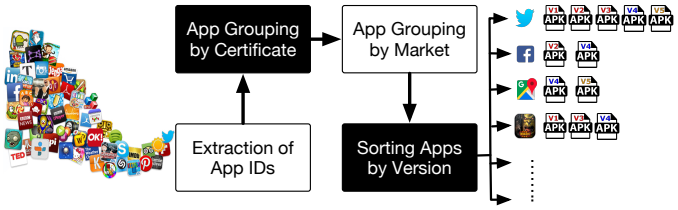


Fig. 3: App Lineages Re-construction Process.

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

1. *Application Id Extraction.* Google recommends [57] that each app should be named, in all its versions, following the 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 (1) related to the same app (based on the name), (2) from the same development team (based on the signatures), and (3) were distributed in the same market (based on AndroZoo metadata). Each group is then considered to contain a set of apks forming an unique lineage.
4. *App Version Sorting.* In order to make our dataset readily usable in experiments, we proceed to sort all apks in each lineage to reflect the evolution process. We rely on the *versionCode* attribute which is set by developers in the Manifest file. We further preserve our dataset from potential noise by dropping all apks where no *versionCode* is declared.

To avoid toy apps, we adopted the strategy used in [60] to set a threshold of at least 10 apks before considering a lineage in our study. And during this step almost 92% of apks were filtered out.

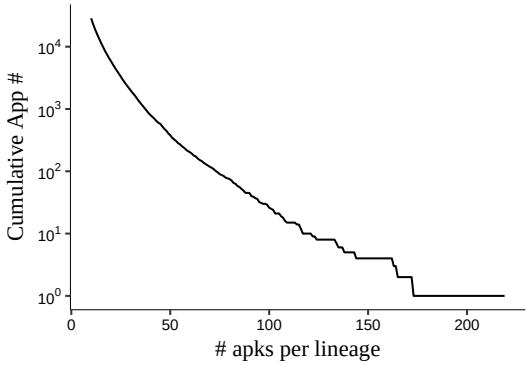


Fig. 4: Lineage Sizes

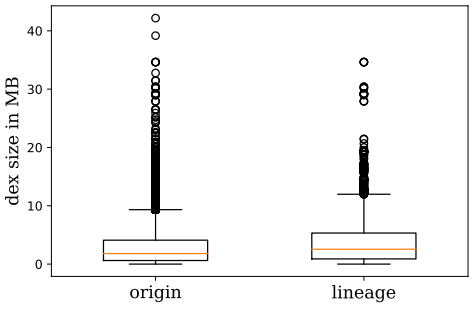


Fig. 5: Dex Sizes

Overall, we were able to identify 28,564 lineages and the five-number summary of lineage size are 10, 11, 13, 18 and 219 respectively. Figure 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 2 presents the top 5 lineages w.r.t. the number of apks. In total, our lineage dataset includes

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

465,037 apks. Figure 5 compares dex sizes of APKs between the original dataset and the re-constructed lineage dataset. It can be noticed that apks of small size has been removed mostly during lineage re-construction, as the median value increased from around 2.6 to 3.3 MB.

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

TABLE 2: 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

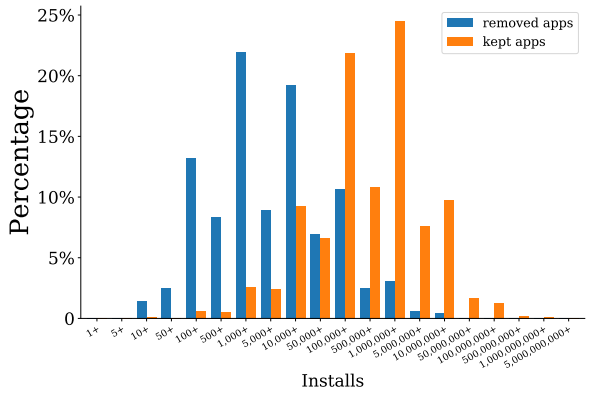


Fig. 6: Intallation Comparison between Removed and Kept Apps

To assess the diversity of our lineage dataset, we first explore the categories of the concerned apps. Since this information is not available from the AndroZoo repository, we took on the task of crawling market web pages to collect meta-data information for each lineage. We focused on our study on the official Google Play. Out of the 16,074 lineages which were crawled from Google Play, we were able to obtain category information for 14,208 lineages. 1098 lineages were no longer available in the market while the market page for 768 lineages could not be accessed because of location restrictions. Figure 7 illustrates the high diversity in terms of category through a word cloud representation.

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

Second, we investigate the API levels (i.e., the Android OS version) that are targeted by the apps in our dataset. Since the Android ecosystem is fragmented with several versions of the OS being run on different proportions of devices, it is important to ensure that our study covers a comprehensive set of Android OS versions. Figure 8 presents the distribution of API level span of lineages of the dataset. The API level span of a lineage indicates the range between the minimum and maximum targeted API level found in the lineage. In the figure, the X-axis is the lower bound of the API level 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.

2.4 Research Questions

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

RQ1: ***Have there been vulnerability "bubbles" in the Android app market?*** The literature of Android security appears to explore vulnerabilities in waves of research papers. Considering that many of the vulnerabilities described above have been, at some periods, trending topics in the research community [61], [62], [63], [64], [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]. 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,

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.

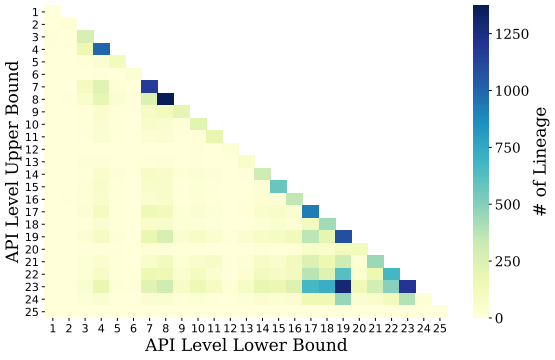


Fig. 8: API Level Span Distribution of Lineages

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], [68], [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.

2.5 Experimental Setup

2.5.1 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 7 days and occupied 56 terabytes (TB) of local storage space. Among the vulnerability detection tools, FlowDroid and IC3, as previously reported in the literature [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 3357 and 2048 lineages respectively[12]. The final raw results hold in about 40GB of disk space. There are 2 reasons that caused the different numbers in the result of different tools. One is because of the limited time budget for running analysis. As we know from the previous paragraph that AndroBugs is the lightest tool in resource requirement, we collected the most results from its analysis. On the contrary, IC3 is the heaviest tool which got the least results. Meanwhile, some apks could cause crashing of certain detection tools and, normally, different tools crash on different apks. This is another reason that leads to a different number of analysis results in different tools.

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

Specifically, we invited 2 PhD students who have been working on Android and static analysis related topics to work on the reports of the three selected tools, respectively. One student spent one day on sampled reports [13] of AndroBugs and another one spent two days on the reports of IC3 and FlowDorid respectively. They are able to check 711

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

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

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

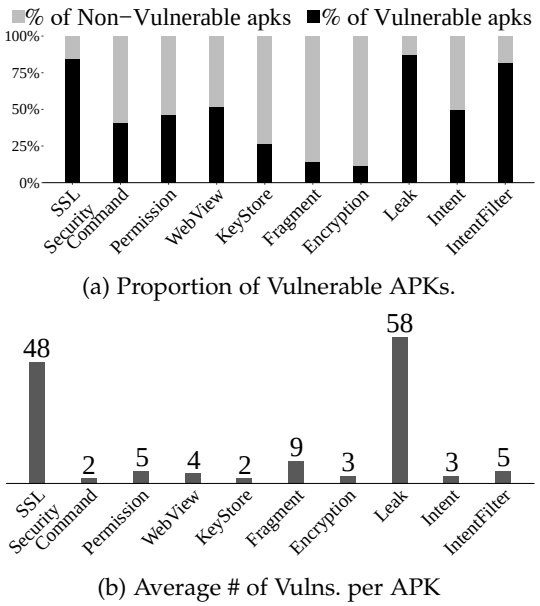
vulnerabilities[14] for AndroBugs, 275 vulnerabilities (98 of intent spoofing and 177 of unauthorized intent reception) for IC3 and only 78 leaks for FlowDroid. The manual verification process confirms that, at least from the syntactic point of view (i.e., these vulnerabilities are in conformance with 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 non-trivial to understand the intention behind the code if no prior knowledge is applied.

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

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.

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).

2.5.2 Study Protocol



(a) Proportion of Vulnerable APKs.

(b) Average # of Vulns. per APK

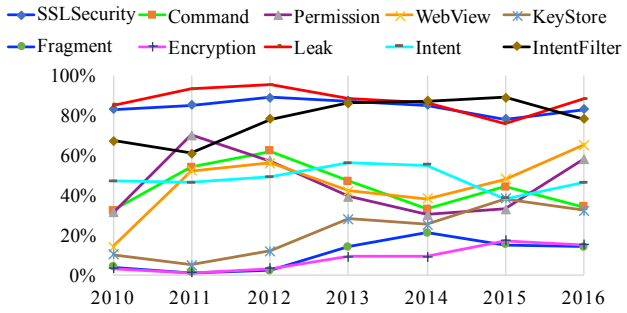
Fig. 9: Distributions of Vulnerable APKs and of Vulnerabilities

Each of the vulnerability detection tools outputs its results in an ad-hoc format. We build dedicated parsers to automatically extract relevant information for our study. Figure 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.

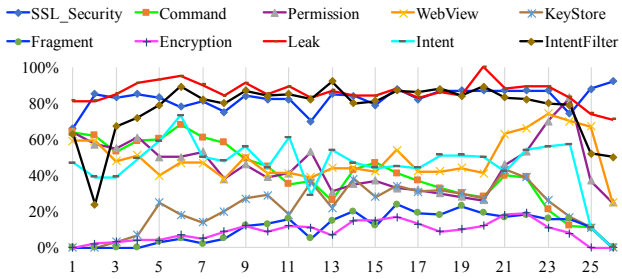
For the evolution study, Vulnerable pieces of code are extracted from the location l of an apk indicated by the vulnerability detection tools. These vulnerable pieces of code are collected and released as a valuable artifact for the community. Real-world examples from this artifact were presented in Section 2.2.2.

Finally, we monitor and record how vulnerabilities change at these locations that is: given the analysis results for an apk v_1 and its successor v_2 of a lineage, we track the differences in terms of vulnerability locations; when a given vulnerability type is identified in a location but is no longer reported at the same location, we compute the change diff between the two apk versions and refer to it as *potential vulnerability fix changes*[15].

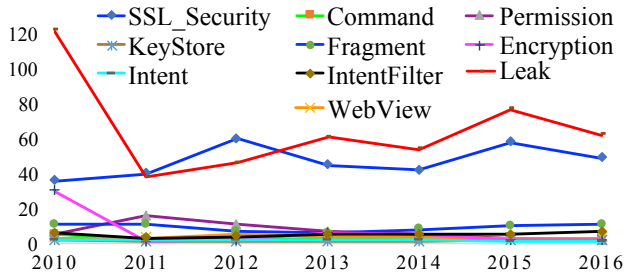
15. Since the change could be only related to program refactoring, we cannot say if the change is a real fix or not.



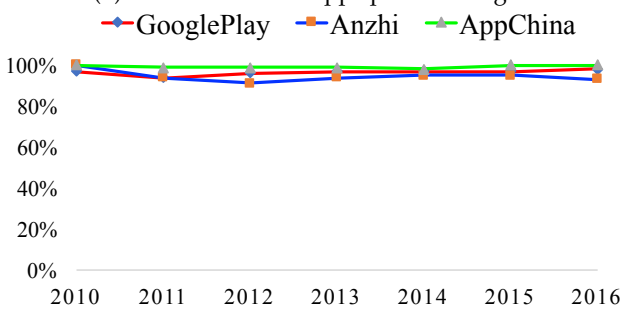
(a) % of Vulnerable Apps Across Time.



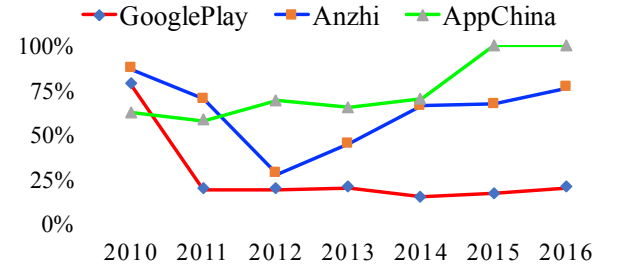
(b) % of Vulnerable Apps per API Target.



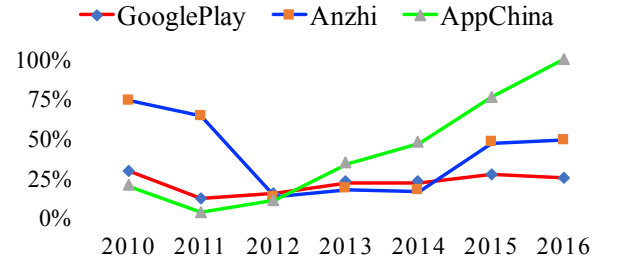
(c) Average # of Vulnerabilities in APKs.



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



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



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

Fig. 10: Evolution of Android App Vulnerabilities.

3 Results

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

3.1 Vulnerability "Bubbles" in App Markets

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

We further investigate (1) whether the overall evolutions depicted previously break down differently in specific markets, given that markets do not implement the same security checking policies; and (2) whether evolution trends are visible inside the apps, since developers may make efforts to at least reduce their numbers. Figure 10d 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

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

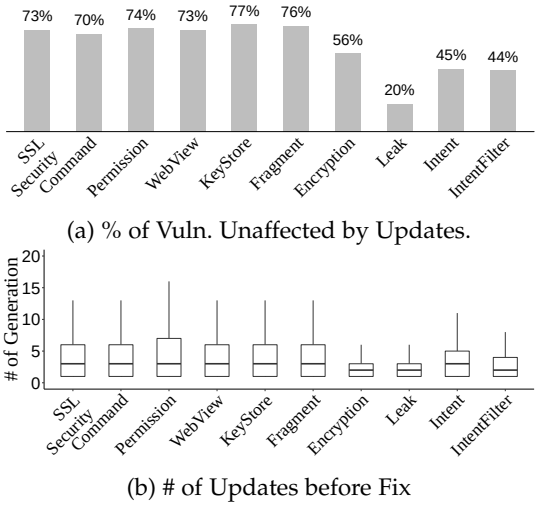
history[17]. Evolution trends in Figure 10c reveal how Leak vulnerabilities have significantly dropped in 2011: from an average 120 vulnerabilities per apk, it came to about 40 before slowly increasing again. We remind the reader that these vulnerabilities found using FlowDroid are computed as possible 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].

Figures 10e and 10f further depict interesting evolution cases for the *Command* and *KeyStore* vulnerability types between markets. While the official Google Play has seen Command-vulnerable apks drop and KeyStore-vulnerability remain low, alternative markets have accepted more and more Command-vulnerable apks, and still include a large share of KeyStore-vulnerable apks. These findings may suggest that the security mechanisms implemented by some markets might be effective against frequently exploited vulnerabilities. Indeed, let us take Google Play as an example, Google has introduced Google Play Protect[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.

3.2 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. Figure 11a summarizes the impact that app updates have on the vulnerabilities in a lineage. On average, for most vulnerabilities, more than 50% of vulnerable locations remain. The number of vulnerabilities related to Encryption and Inter-component communication (i.e., Leaks, Intent and IntentFilter) has evolved substantially across app versions (e.g., only 20% of Leak vulnerabilities kept untouched). Figure 11b 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.



(a) % of Vuln. Unaffected by Updates.

(b) # of Updates before Fix

Fig. 11: Survivability of Vulnerabilities in APKs

We further detail in Figure 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 a 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 Mann-Whitney-Wilcoxon (MWW) test with null hypothesis of equal distribution, alternative hypothesis of not equal distribution and confidence level of 0.95, indicated above each box plot pair, however, show that the difference is statistically significant only for the three ICC-related vulnerabilities.

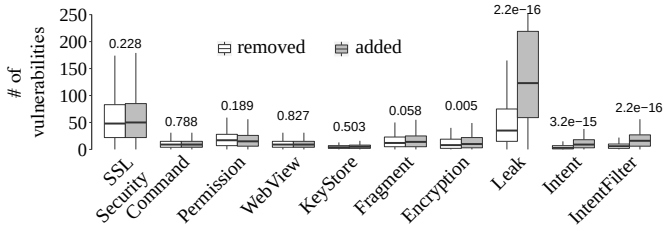


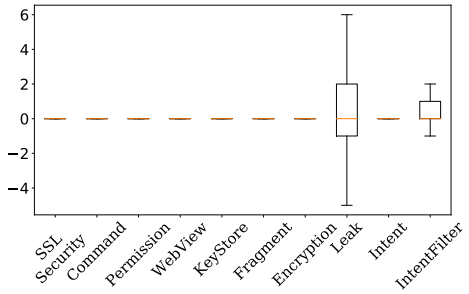
Fig. 12: Distribution of Added and Removed in a Numbers of Vulnerabilities between Consecutive APK Versions. Numbers represent p-values from MWW tests on the statistical significance of the differences.

We now investigate the general trend in vulnerability evolutions, comparing the impact of updates between consecutive pairs and the impact of all updates between the beginning and the end of a lineage. We expect to better highlight the overall evolution of vulnerabilities as several changes have been applied. The box plots in Figures 13 highlight a simple reality: commonly vulnerabilities are neither removed nor introduced during app updates (i.e., all median values equal to 0 in Figure 13a) 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

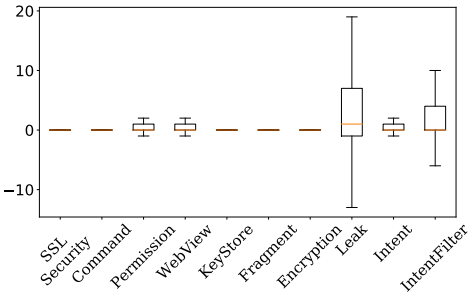
17. A given apk is considered to be vulnerable if it includes any case of our selected vulnerability types.

18. https://www.android.com/play-protect/

shown in Figure 13b, the major pattern stays similarly (i.e., all median values are still 0 only except for *Leak* which is 1 and for most of the mean values, they increased slightly but still between around 0.5 to 0. the exceptions are *Leak* and *IntentFilter* which are 3.8 and 2.3 respectively), but observable differences are exhibited as well. Several vulnerabilities expand in size and the scale increases obviously. The main parts of all boxes are on the positive side of y-axis, which indicates that there are more cases of adding vulnerabilities than removing.



(a) Between Consecutive APK Pairs.



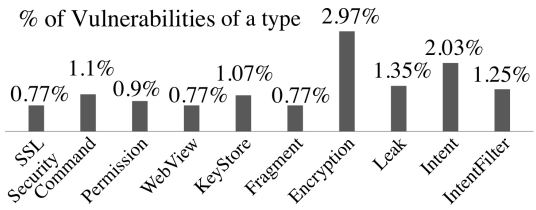
(b) Between Initial and Latest Versions.

Fig. 13: Variations in # of Vulnerabilities Following Updates.

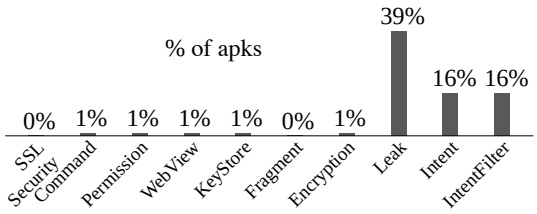
Insights from RQ2: As more than 50% of vulnerabilities stay untouched during 1 update and the possibility of fixing and introducing vulnerabilities during updates does not show a significant difference, app updates indeed do not make apps safer. Moreover, vulnerabilities can normally survive 3 updates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

3.3 Vulnerability Reintroductions

To answer RQ3: ***Do fixed vulnerabilities reappear later in app lineages?*** We track all vulnerability alerts (associated with their locations) and cross-check throughout the lineages. We found 342,809 distinct cases of location-based vulnerability reintroductions (i.e., vulnerable code removed and reappeared in the same method of the same class of an app, as specified in Section 2.1) for 15,375 distinct apps. On average, a given app is affected by 6.7 vulnerability reintroductions. Figure 14a further breaks down reintroduction cases and their proportions among all vulnerability alerts. *Encryption*-related vulnerabilities (2.97%) are the most likely to be reintroduced, in contrast to *SSL Security*-related vulnerabilities (0.77%).



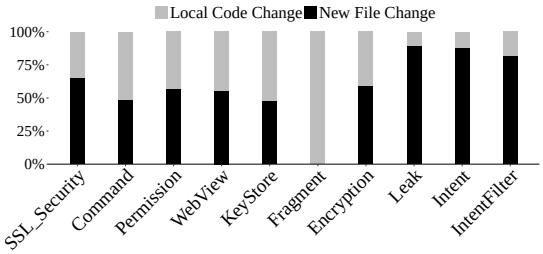
(a) Vulnerability Reintroduction



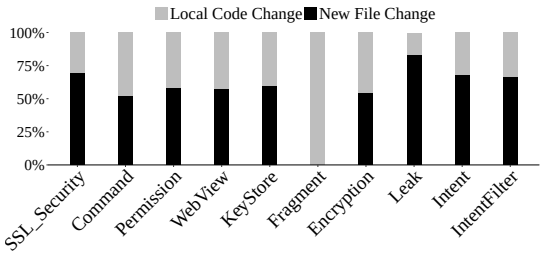
(b) Vulnerability Type Reintroduction

Fig. 14: Statistics on Reintroduction Occurrences.

We investigate whether, in some lineages, a **vulnerability type** may completely disappear at some point and later re-appear. Figure 14b 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 the paper, without specification, the reintroduction should be location-based).



(a) Vulnerability Removal



(b) Vulnerability Introduction.

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).

Figure 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%.

3.4 Vulnerability Introduction Vehicle

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

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

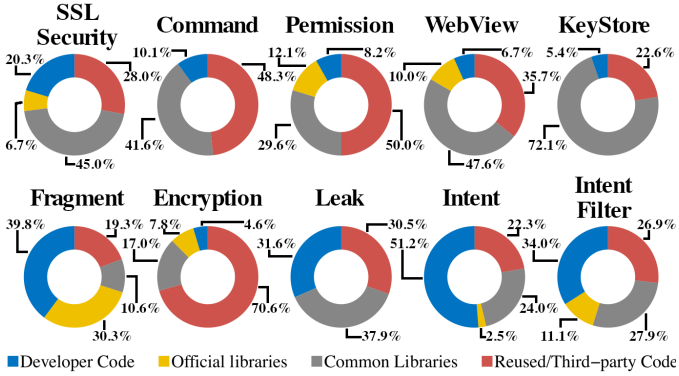


Fig. 16: Distribution of Vulnerable Code in *Developer Code*, *Official Libraries*, *Common Libraries* and *Reused/Third Party Code*.

Figure 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. 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].

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 3 provides Spearman correlation computation results. All correlation appear to be 'Negligible'. *IntentFilter* shows the highest correlation close to being categorized as 'Moderate' w.r.t. the size of the apps.

TABLE 3: Spearman Correlation Coefficient (ρ) Values. With experiments **Exp.1:** # of packages vs. # of vulns. per apk; **Exp.2:** # of new packages vs. # of vulns. per update.

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

Interestingly, computation of LOESS regression [80] shown in Figure 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.

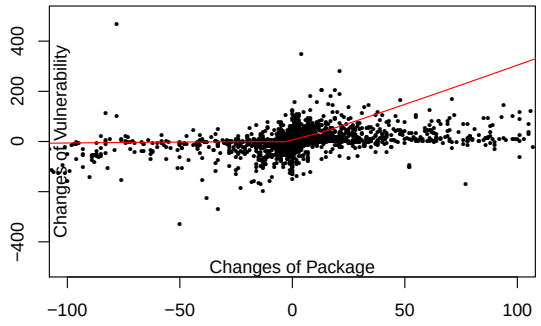


Fig. 17: Overall Regression.

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

Insights from RQ4: Although third-party libraries are the main contributor of vulnerabilities, it is quite possible that the major contribution is only from part of these libraries. Therefore more focus should be given on the analysis of libraries and market maintainers could draw policies rejecting apps using *non-vetted* libraries. Moreover, the claim made in [71] that more code and libraries imply more vulnerabilities may not be always true.

3.5 Vulnerability and Malware

To answer RQ5: ***Do vulnerabilities foreshadow malware?*** We investigate relationships between app vulnerabilities

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

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 anti-virus(AV) flagging reports. AndroZoo provides these reports [20] as metadata for all its apks and in this study, we treat an apk as malware as long as one or more AVs gave positive reports.

TABLE 4: Benign and Malware in Vulnerable APK Sets

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

Table 4 reports the proportion of benign and malware which are detected as vulnerable for each vulnerability type. Malware are not more likely to contain a given vulnerability than benign apps. We further perform a correlation study on these malware to assess whether the number of vulnerabilities in an apk can be correlated to the number of AVs that flag it. This is important since AVs are 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.

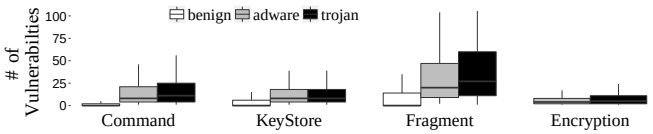


Fig. 18: Vulnerability VS Malware[21]

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 non-flagged predecessors in the lineage and count cumulatively how many vulnerabilities were included in them. For each lineage where all apks are benign we also count the number of vulnerabilities per vulnerability types. We then perform the MWW test, for each vulnerability type, to assess whether the difference between, on the one hand, the median number of vulnerabilities for malware of a given type, and, on the other hand, the median number of vulnerabilities in benign apps, is statistically significant. In most cases, this difference is not statistically significant, suggesting that most types of Android malware cannot be readily characterized by vulnerabilities within the malware itself. Nevertheless, we find four interesting cases of vulnerability types (namely, *Command*, *KeyStore*, *Fragment* and *Encryption*), where vulnerabilities are suggestive of malicious behavior. Figure 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 3 types except *Encryption*, while the absence of *benign* of type *Encryption* in the figure reflects that *benign* apks does not contain any of such vulnerability.

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

4 Discussion

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

4.1 Implication and Future Work

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

- **Understanding the genesis of mobile app vulnerabilities.** Since app lineages represent the evolution of apps, they could contain the information about "when" and "how" is a given vulnerability initially introduced. This information could then be leveraged to understand the genesis of the vulnerability and thereby help researchers and developers invent better means to locate and defend such vulnerabilities.
- **Tools to address vulnerability infections.** By leveraging app lineages, the corrected pieces of code of a vulnerability happened in a certain app version could be spotted and extracted from its subsequent app versions with the fixes. Indeed, the vulnerable code snippets disclosed in this work could be leveraged to mine fix patterns for certain vulnerabilities and subsequently enable the possibility of automated vulnerability fixes.
- **Reintroduction analysis for app updates.** As revealed in the answer to RQ3, the fact that vulnerabilities can be reintroduced into apps during their updates, it is essential to perform reintroduction analysis (either statically or dynamically) for Android apps. These analyses later could be immediately adopted by app markets to guarantee that app updates do not introduce more (known) vulnerabilities.
- **Library screening strategies.** As concluded in Section 3.4, third-party libraries are the main contributor of vulnerabilities in an app. Thus, when libraries containing serious vulnerabilities get to be popular, the aftermath will be difficult to estimate. Such incident happened once on August 21, 2017, Bauer and Hebeisen [85], from the Lookout Security Intelligence team, have reported that their investigation of a suspicious ad SDK (i.e., ad library) revealed a vulnerability that could allow the SDK maintainer to introduce malicious spyware into apps. After it was alerted, Google

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

21. 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.

has then removed from the market over 500 apps containing the affected SDK: those apps were unfortunately already downloaded over 100 million times across the Android ecosystem. Therefore, strategies of selective screening of libraries could be investigated to clean app markets with apps which unnecessarily ship vulnerable libraries.

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

4.2 Threats to Validity

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

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

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

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

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

Furthermore, the experimental results may be impacted by the validity of the results of the selected vulnerability detection tools. Given that these are static analysis tools, it is known that they may yield false positives. We attempt to mitigate this impact by performing a manual verification to some of the randomly selected vulnerabilities yielded by the three analyzers. As discussed in Section 4.1, the naive verification process does not spot any clearly false positive results (i.e., the vulnerabilities are at least in conformance with the definition of vulnerabilities as proposed in the tools documentation). However, since the verification was implemented by 2 PhD students, their experiences could have a direct impact on the verification result. Thus, lack of proof of the authenticity of the vulnerabilities is the main threats to the validity of this study. Moreover, during the manual verification, vulnerabilities of non-HTTPS links are not considered. The main reasons are: 1) they are quite straightforward to be identified, and 2) these links can be changed over time and thereby are difficult to be verified (e.g., for an HTTP link, if an HTTPS page and redirect were added just before the verification, should we consider it as a false positive?). The pervasiveness of such vulnerabilities also makes it hard to be manually checked. But we still have to be aware of the possibility of the impact on the results.

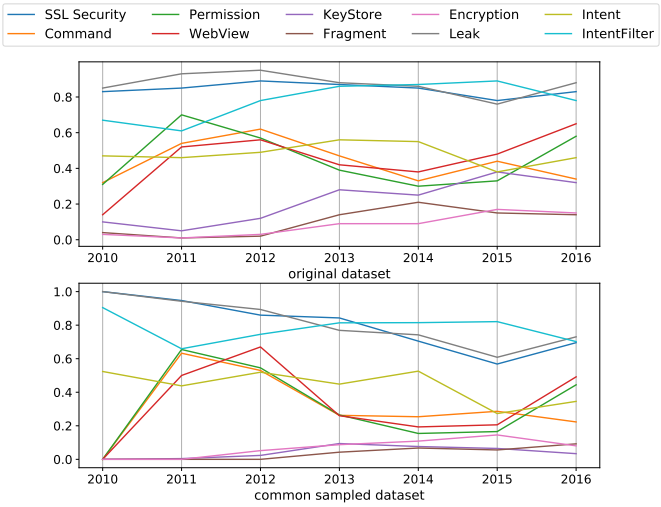


Fig. 19: Trends Comparison between the Original Dataset (imbalanced) and the Common Sampled Dataset (balanced).

Finally, due to constraints such as time budgets and computation resources, the results yielded by the three selected static analyzers are for a different number of apps. Since the results obtained by the static analyzers are not always from the same samples, different vulnerabilities could be collected from different datasets. Therefore, our empirical observations could have been impacted by such inconsistent datasets. However, as we have not attempted to compare the results between different static analyzers, we believe that such an impact should be negligible Nevertheless, to empirically demonstrate this, we go one step deeper to

revisit the aforementioned studies with a common corpus. Specifically, we conduct our revisit study on 356 app lineages, which correspond to 3984 APKs, having all these apps successfully analyzed by the three tools. Our revisit study reveals that the empirical findings observed from a common app lineage set are more or less similar to that of imbalanced datasets. For example, regarding the evolving patterns of vulnerable apps, as illustrated in Figure 19, the results observed from the imbalanced dataset (above sub-figure) and the common corpus (bottom sub-figure) more or less follow similar trends, indicating that the empirical results observed will unlikely be impacted by the dataset chosen in this study.

5 Related Work

Our work is related to several contributions in the literature. In previous sections, we have discussed the case of data leaks vulnerability in Android investigated by the authors of TaintDroid [76], FlowDroid [13] and IccTA [14]. Other analyzers have been proposed based on static analysis [6], [53], [86], dynamic analysis [87], [88], [89], [90], [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 three main topics:

5.0.0.1 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 re-delegation attacks while Grace *et al.* [5] focuses on capability leaks. They unveiled vulnerabilities related to permissions. While this paper 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.

5.0.0.2 Vulnerability studies: Vulnerabilities, also known as security-sensitive bugs, have been extensively studied in the literature [92] for different systems [93], [94], [95], [96], [97] and languages [44], [98], [99], [100]. Camilo et [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: 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 6 until-then unknown vulnerabilities in three common services and 2 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 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 5: 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

Table 5 lists the works which are similar to this study. It is noteworthy that the number of APKs considered in these reported studies is much less (by an order of magnitude) than the number of APKs considered in this paper. Moreover, most of the studies focused on one specific vulnerability type. Although, the latest two works studied Android vul-

nerabilities 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.

5.0.0.3 Software evolution: The software engineering literature includes a large body of work on software maintenance and evolution [111], [112], [113], [114], [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 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.

6 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 addresses these challenges by re-constructing 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 is so far the largest scale of this kind studies. Moreover, investigating Android vulnerabilities from app lineage point of view is the major novelty of this study, which allows us to yield several newly spotted findings: 1) most vulnerabilities can survival at less 3 updates; 2) part of third-party libraries are the major contributors of the most vulnerabilities; 3) vulnerability reintroduction occurs for all kinds of vulnerabilities while *Encryption*-related vulnerabilities are the most reintroduced within all types of this study; and 4) some vulnerabilities may foreshadow malware. In addition to new findings, this large scale study also confirms most of the conclusions from previous studies with relatively small datasets. However, the result of this study also suggests that the recent claim made by Watanabe *et al.* [71] that more code and libraries imply more vulnerabilities may not be always true. Finally, 3 valuable artifacts produced by this study: 1) **app lineages**, 2) the complete dataset of **vulnerability scanning reports**, 3) recorded **vulnerable pieces of code**, are shared.

Acknowledgment

This work is supported by the Luxembourg National Research Fund (FNR) through grant PRIDE15/10621687/SPsquared and project CHARACTERIZE C17/IS/1169386.

References

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

[2] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *IEEE Transactions on Software Engineering*, 40(6):617–632, 2014.

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

[4] Timothy Vidas, Daniel Votipka, and Nicolas Christin. All your droid are belong to us: A survey of current android attacks.

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

[6] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proc. of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

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

[8] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proc. of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

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

[10] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proc. of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[11] Jiang Xuxian and Yajin Zhou. Detecting passive content leaks and pollution in android applications. In *Proc. of the 20th Network and Distributed System Security Symposium*, 2013.

[12] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *In Proc. of the 21st Annual Network and Distributed System Security Symposium (NDSS14)*. Citeseer, 2014.

[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[14] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proc. of the 37th Intl. Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] AndroZoo. Androzoo website. http://androzoo.uni.lu.

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

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

[31] Johnathon Burket, Lori Flynn, Will Klieber, Jonathan Lim, and William Snavely. Making didfail succeed: Enhancing the cert static taint analyzer for android app sets. 2015.

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

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

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

[35] AndroBugs. Hall of fame. https://www.androbugs.com/\#hof.

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

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

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

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

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

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

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

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

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

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

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

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

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

[49] Chaoshun Zuo, Jianliang Wu, and Shanqing Guo. Automatically detecting ssl error-handling vulnerabilities in hybrid mobile web apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 591–596, New York, NY, USA, 2015. ACM.

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

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

[52] Dragos Sbîrlea, Michael G Burke, Salvatore Guarnieri, Marco Pistoia, and Vivek Sarkar. Automatic detection of inter-application permission leaks in android applications. *IBM Journal of Research and Development*, 57(6):10–1, 2013.

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

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

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

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

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

[58] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, 2017.

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

[60] L. Li, T. Bissyand, and J. Klein. Moonlightbox: Mining android api histories for uncovering release-time inconsistencies. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 212–223, Oct 2018.

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

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

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

[64] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE transactions on software engineering*, 43(9):817–847, 2017.

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

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

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

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

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

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

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

[72] Anonymous. Double blind review policy.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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*, 2016.

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

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

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

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

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

[109] daniel r thomas, alastair r beresford, and andrew rice. security metrics for the android ecosystem. In *proc. of the 5th annual acm ccs workshop on security and privacy in smartphones and mobile devices*, pages 87–98, 2015.

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

[111] Evelyn J Barry, Chris F Kemerer, and Sandra A Slaughter. On the uniformity of software evolution patterns. In *Software Engineering, 2003. Proc.. 25th Intl. Conference on*, pages 106–113. IEEE, 2003.

[112] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proc. of the 33rd Intl. Conference on Software Engineering*, pages 151–160. ACM, 2011.

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

[114] Tom Mens. Introduction and roadmap: History and challenges of software evolution. In *Software evolution*, pages 1–11. Springer, 2008.

[115] Qiang Tu et al. Evolution in open source software: A case study. In *Software Maintenance, 2000. Proc.. Intl. Conference on*, pages 131–142. IEEE, 2000.

[116] R. Coppola, M. Morisio, and M. Torchiano. Mobile gui testing fragility: A study on open-source android applications. *IEEE Transactions on Reliability*, 68(1):67–90, March 2019.

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